

Predictable High Performance Computing using Feedback Control and Admission Control

Sang-Min Park, Student Member, IEEE, and Marty Humphrey, Member, IEEE

Abstract— Historically, batch scheduling has dominated the management of High Performance Computing (HPC) resources. One of the most significant limitations using this approach is an inability to predict both the start time and end time of jobs. Although existing research such as resource reservation and queue-time-prediction partially address this issue, a more predictable HPC system is needed, particularly for an emerging class of adaptive real-time HPC applications. This paper presents a design and implementation of a predictable HPC system using feedback control and admission control. By creating a virtualized application layer and opportunistically multiplexing concurrent applications through the application of formal control theory, we regulate a job's progress such that the job meets its deadline without requiring exclusive access to resources even in the presence of a wide class of unexpected events. Admission control regulates access to resources when oversubscribed. Our experimental results using five widely used applications show the feedback and admission controller achieves highly predictable HPC system. The designed feedback controller regulates the HPC job's progress accurately, close to the prediction by theory, thereby showing the successful application of classic control theory to HPC workloads. In week-long experiments, over 90% of jobs met deadlines and the jobs missing deadlines still finished close to the requested deadlines (12.4% error).

Index Terms— Multiprocessor Systems, Parallel Systems, Scheduling, Control Theory

1 INTRODUCTION

In recent years, adaptive, real-time data-driven applications have emerged, often using scientific modeling and high performance computing (HPC) to directly support mission-critical decision making. For example, people explore if weather forecasting infrastructure can be transformed to an adaptive one that can predict mesoscale weather events such as tornadoes in real-time [1]. Similarly, coastal hazard prediction attempts to predict the impact of storm surge in real-time to dynamically facilitate life-saving decisions such as evacuation orders [2]. In medical applications, clinicians would use simulation results on patient-specific data to diagnose and treat patient's medical problems [3]. Their common properties include that 1) there is a chain of application components where, 2) the in-field sensors generate real-time data at unpredictable cycles, 3) the high-bandwidth networks transport sensor data to computational sites, 4) the long-running simulation codes are invoked with the data, and 5) optionally the simulation's results guide the next course of actions to the sensors. It is an adaptive application since the entire loop including sensors and simulations should react to unpredictable changes in the field. They require deadline-guaranteed execution of long-running simulation codes, which is a challenging task to traditional HPC centers.

The conventional queuing system operation of supercomputers and clusters are problematic for such real-time adaptive applications. In the absence of a special "high-priority" queue for use exactly by such applications, jobs can often sit in queues for an unpredictable amount of time, sometimes only beginning to execute long past their

useful execution time. The high-priority queue, which effectively preempts currently-executing jobs, while perhaps effective for the high-priority jobs themselves, can result in significant dissatisfaction for those users of affected jobs. Dedicating the entire resource to real-time data-driven applications is usually cost-prohibitive, because of the sporadic nature of such jobs (the resource would be idle too much of the time). Advance reservation in queuing systems has been an active area of research to eliminate a job's wait time and thus solve unpredictability. However, in practice, reservation requires complex planning that involves time-consuming interactions between users and resource providers (e.g., a week in TeraGrid). For many types of these real-time data-driven applications (e.g., real-time tornado prediction), a job must begin processing the data within minutes or hours of the appearance of the data. In short, existing approaches for managing HPC resources cannot broadly support this new class of applications, which can require commencing executing almost immediately, can be required to deliver results in a timely fashion, should not significantly disrupt existing execution of applications, and should not require resources dedicated to such applications.

This paper presents a new approach for managing HPC infrastructure. Unlike attempting to achieve predictability by controlling the wait time of queued jobs, we control the job's waiting time and running time (and behavior in general) by creating virtualized resources with fine-grained performance configuration at run time. Via novel use of virtual machine abstractions, we run jobs inside a performance container and "throttle up" or "throttle down" that performance container by carefully controlling the virtualized application's access to real system resources to regulate the job's progress. Our throttling methodology uses formal control theory to design

• The authors are with the Department of Computer Science in the University of Virginia, Charlottesville, VA 22904. E-mail: {sp2kn, humphrey}@cs.virginia.edu

and implement a feedback controller that enables deadline-sensitive regulation. The controller ensures that jobs sustain progress to meet the deadlines, even when significant “disturbances” might otherwise affect their progress.

In addition to run-time regulation of a job’s progress, an admission controller running in HPC infrastructure makes accept/reject decision regarding a job’s deadline instantly at submission time. Our design of the admission controller is based on a resource utilization test and adaptive modeling and control that react to run-time characteristics of applications. While a feedback controller locally adjust the job’s access to shared system resources, the admission controller globally regulates the overall capacity of the physical resources.

The experimental results conducted on our prototype implementation with challenging HPC workloads [4][5][6][7][8] are promising:

- The designed feedback controller regulates the HPC job’s progress accurately, close to the prediction by theory, thereby showing the successful application of classic control theory to HPC workloads.
- In week-long experiments, over 90% of jobs met deadlines and the jobs missing deadlines still finished close to the requested deadlines (12.4% error).

The rest of this paper is organized as follows: Section 2 presents related work. In Section 3 we introduce the performance modeling of HPC applications. Section 4 presents the admission control, followed by memory scheduling to handle large workloads in Section 5. Section 6 presents feedback-control CPU scheduling and Section 7 discuss adaptive techniques to enhance admission and feedback control in spite of severe disturbances. The evaluation is presented in Section 8, followed by the discussion and conclusion in Section 9.

2 RELATED WORK

Advance reservation has been the most widely discussed mechanism to build more predictable HPC system [9][10]. Although modern queuing systems such as PBS and LSF implement reservation, reservation has not been widely accepted due in part to its potential for significant underutilization [11] and managerial complexity. While enforcing penalties to “no-show” cases [12] or requiring a human to authorize reservation manually [13] might partially solve the problem, it is not clear to what extent reservation could deliver predictability to emerging, adaptive applications. For example, the TeraGrid requires users to submit a reservation request a week in advance, which would be prohibitively long for scheduling mesoscale weather forecasting codes.

Another widely discussed mechanism for building predictable infrastructure is to periodically monitor a resource’s state and predict the future state using statistical methods. For example, Network Weather Service measures end-to-end bandwidth and the host CPU’s availability and predicts the future state with statistical guarantee on its accuracy [14]. Similarly, the statistical approach has been applied to predict queue wait times of large supercomputing centers [15][16]. The fundamental limitation of the monitoring-prediction approach is that it alone does

not allow users to change resource’s behavior according the needs, for example, when users have a deadline requirement that is more urgent than possible with the available resources. Our mechanism allows users to set an application’s deadline directly.

As system virtualization (VM) [17][18] continue to gain attention across industry and academia, leasing has been actively discussed as a promising abstraction for sharing resources securely and predictably [19][20]. The VM’s checkpoint enables preemption for enforcing priority orders among jobs. Although our approach uses checkpointing, our design principle is controlled time-sharing, in contrast to leasing’s space-sharing that allocates jobs exclusively to resources while some jobs wait in a queue. We argue and experimentally justify that controlled time-sharing achieves better predictability in response to disturbances such as disk I/O.

There is a large body of real-time systems research that addresses performance-guaranteed execution of computational tasks, ranging from static scheduling algorithms such as Rate Monotonic [21], to algorithms and implementations targeted to more dynamic environments such as EDF [22], admission-control-based algorithms [23], and more recently control-theoretic algorithms [24]. Our work differs from them as we design a system to address HPC workloads. Unlike classic real-time schedulers where the periodicity, worst-case execution time, and deadline defines the task model of embedded computer systems [21][22], we target compute and I/O intensive applications, such as weather simulation and gene-sequence analysis, that typically run for long duration (hours or even days) without interruption. The critical research goal is to design a framework that allows one to model and control the behaviors of such HPC workloads as thoroughly as the classic real-time systems. We believe we are one of the first to address the question.

Control theory, one of the most widely used mathematical frameworks to control behavior of linear, dynamic systems [25][26], is at the core of our application modeling and scheduling. The theory has been used in a variety of software services including real-time scheduling [24][27], QoS for web servers [28], storage systems [29], and QoS control in virtualized environments [30][31][32]. Our earlier works showed the basic control-theoretic methodology to solve unpredictability of HPC infrastructures [33][34]. Our work departs from the existing works by addressing diverse, concurrent applications within a single framework. The existing works are shown to work for one application, typically a web-based, enterprise application [28][30][31][32]. However, our requirement, driven from the broad HPC community, is to support the existing and future HPC applications from the domains that exhibit diverse performance properties. The key research question is how to create a model-based scheduler that delivers predictable computations to an unpredictable mix of concurrent HPC jobs. To the best of our knowledge, no existing works address this question directly.

3 MODELING HPC APPLICATIONS

In this work, we assume a job has a single-thread of ex-

ecution that is run on a single CPU, typically scheduled by a queuing system of a cluster. A set of such sequential jobs can be grouped together to form a loosely-coupled, distributed applications such as parameter sweep experiments or workflows with precedence constraints. It is known that such sequential workloads constitute a large fraction of the experiments running on production Grids (e.g., 75% of batch jobs in Grid'5000) [35]. There are certain types of shared-memory multi-threaded jobs that the work can be easily applicable, though we do not present the evaluation results for this type. The support for distributed memory multi-threaded jobs (e.g., MPI) is the subject of future research. We discuss on the possible extensions to support parallel jobs in Section 9.

A performance model represents a job's resource requirements during execution. In making admission decision, the admission controller (AC hereafter) relies on a prediction about a job's resource requirements derived from the model. Also, the job's feedback-control CPU scheduler (FC hereafter) is customized using the model. We use a linear difference equation in the time domain to represent a job's progress with respect to its resource consumption. We model an HPC job using the two quantitative metrics *milestone* and *progress*. "Milestone" represents a job's overall computation requirements and/or desired performance, and "progress" measures how fast a job performs computation during a fixed interval. There are many possible sources for determining a job's milestone, including application's semantics, source code, and linear estimation from the job's quantifiable problem size. From the milestone, the progress can be derived by defining a sampling period. Note that we do not define a specific unit and scale for milestone and progress; rather we leave them on users' deliberate choice, considering the specific properties of a particular application. For example, for an application we evaluated (Montage), we used a number of input files as a unit of milestone (all files to process) and progress (number of files to process in a minute). For other applications, we used an application specific counter. For certain HPC systems with the support of system-level instrumentation, FLOPS would be a good unit for the reason described below.

The following simple equation expresses the relationship between a job's milestone, progress, and deadline:

$$\frac{\text{Milestone}}{\text{Deadline}} = \text{Progress} \quad (1)$$

If a milestone and deadline (more precisely, remaining time to deadline) are known for a job, we expect the job will meet the deadline if its run-time performance is equal to or above the expected progress. As the term milestone implies, the relationship can be extended to model a job that has multiple phases. For example if a job first processes input files and then performs computation, the model can be extended as:

$$\{\text{Progress}_{io}, \text{Progress}_{cpu}\} = \left\{ \frac{\text{Milestone}_{io}}{\text{Deadline}_{io}}, \frac{\text{Milestone}_{cpu}}{\text{Deadline}_{cpu}} \right\} \quad (2)$$

The above relationship assumes that a job's progress is observable during execution. The current approach to obtain the progress measurement is source code instrumentation. We built a sensor library, as a collection of application specific counters that allows users to strategically place code in the critical path of application source

code (e.g., "hot spots"). While the job runs, sensor values are incremented and reported to AC and FC periodically. Although the code instrumentation requires an extra effort from HPC developers, we found this effort is often minimal because many applications already implement similar routines to measure job's progress, for example, for debugging purposes. An alternative approach to measure progress with no developer support is profiling hardware counter using the system-level instrumentation tools such as PAPI [36] and Xenoprof [37]. We plan to pursue this direction in the future.

A job's progress is modeled using a linear difference equation expressed in the following generalized form:

$$P(k) = \sum_{i=1}^n a_i P(k-i) + \sum_{j=1}^m b_j C(k-j) \quad (3)$$

In (3), $P(k)$ and $C(k)$ represent the job's progress and the allocated CPU fraction in k -th sampling time. The linear model in the time-domain shows that a job's progress measured in time k is determined by progress in previous n cycles ($k-1, k-2, \dots, k-n$) and CPU allocation in previous m cycles ($k-1, k-2, \dots, k-m$). Note the progress in previous cycles can affect the current progress because there can be a delay between the time CPU allocation has changed and the time progress is affected by the change. I/O latency is one example that causes the delay. From the general model expressed in (3), we determine the model structure by choosing values for n, m . In general, a better model is obtained as we choose higher value for n, m , since more history is reflected on the current measurement. However, a higher order model requires a complex controller design and is vulnerable to overfitting, representing that a model cannot be used in variations of contexts (e.g., hardware upgrades). Throughout experiments with various HPC applications, we found the following first-order model is often sufficient:

$$P(k) = aP(k-1) + bC(k-1) \quad (4)$$

We use virtualization technology as the resource provisioning abstraction. In both Hyper-V and Xen, there are two different interfaces to schedule CPUs to concurrent VMs. The first is to assign a particular number of virtual cores to a VM. The second is to allocate a particular fraction of the assigned cores to a VM via the credit-based scheduling algorithms. While the first method is a static binding made before a VM goes live, the second is a dynamic allocation while a VM is live. Since we consider only single-thread jobs in this paper, we assume a single core is bound to a VM statically and use only the credit interface for the control purpose. The VM's credit-based scheduler allows one to control CPU usage per VM accurately and at fine granularity, as if the VM is run at configurable clock speed. We implement the resource provisioning abstraction using Hyper-V, and use its credit cap interface, via a library call, to control a job's progress. The credit cap sets the upper limits of VM's CPU usage.

We use system identification [25], the black-box modeling approach to establish a linear model in control theory, to derive the linear equation that models the relationship between provisioned resources and job's progress. The jobs instrumented with sensor library are run with varying credit values, and the changes in CPU

allocation ($C(k)$) and job's measured progress ($P(k)$) are used to create linear model via least-square regression. We denote the measured and predicted value of P at time k by $\tilde{P}(k)$, $\hat{P}(k)$, respectively. The predicted value $\hat{P}(k)$ can be expressed as:

$$\hat{P}(k) = aP(k-1) + bC(k-1) \quad (5)$$

The k -th residual, also known as prediction error, is $e(k) = \tilde{P}(k) - \hat{P}(k)$. Least square estimator chooses the right value of a , b so as to minimize the sum of squared errors (more detail on this can be found in any introductory statistics book such as [38]). Figure 1 is a graphical representation of system identification for BLAST [7]. We stimulate the system (job's progress) with varying actuation values (VM scheduling cap), following a discrete sine wave as illustrated in Figure 1(a), and the real measurement follows the similar trends as shown in Figure 1(b).

As a concrete illustration, the first-order parameters that we obtained from BLAST and WRF [6] are as follows:

- BLAST: $a=0.32$ $b=6.38$
- WRF: $a=0.13$ $b=1.28$

Note that BLAST exhibits a larger value of a than WRF. This is since BLAST is more I/O intensive than WRF and thus takes longer time for the actuation, determined by b , to take effect on the measurement. It also shows a larger value of b since the sensor is more frequently called than WRF. After system identification, the application-specific performance models are stored in model repository which is built on a file system in the current prototype.

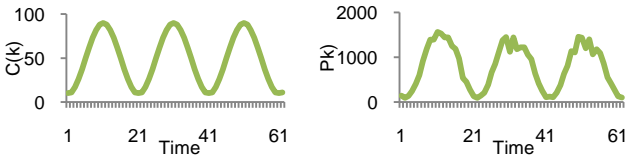


Figure 1(a): VM credit

Figure 1(b): Progress of BLAST

4 ADMISSION CONTROL

We model a HPC server's computational capacity using a virtual machine's fine-grained CPU scheduler capacity and interfaces. Since the granularity of credit expression is implementation dependent (e.g., [1-100] per core in Hyper-V), we denote [1-N] as a range of CPU credit for a single core. If a server has M cores, the server's typical configuration will express $(M \cdot N)$ as the server's total computational capacity. For instance, a Hyper-V server running on 2-way quad core processors has a capacity of 800 credits. The server's another parameter, utilization bounds $U \leq 1.0$, expresses the server's capacity reserved for implementing AC. If a Hyper-V server on an 8-core system has $U=1.0$, the server reserves all 800 credits for AC. If the server is configured with $U=(1.0 - r)$, the server reserves a fraction r of its capacity as extra resources to compensate for a job's performance losses due to disturbances. In our experimental setting on 8-core AMD server, we have found it sufficient in practice to reserve 1 core (i.e., $r=1/8$), to handle any such disturbances.

A job submitted to a server has an associated performance model expressed in (3). Additionally, the job's model of a milestone is obtained from application-specific sources, including a software component that estimates a

milestone from the job's quantifiable problem size (e.g., WRF [6], BLAST [7]) or a component that directly reads the job's internal variable (e.g., ADCIRC [4], OpenLB [5]), or direct request by users (e.g., Montage [8]). From the deadline and milestone of a job, we derive a reference progress, P_{ref} , the job's computational requirements in unit time using (1). From the target progress and the job's performance model, we derive a job's loading factor that determines server's estimated utilization committed to keep the job's progress at the target. To derive an estimated utilization, we first convert a job's performance model (3) to a steady state form. Note the linear difference model expressed in (3) represents input-output relationships in temporal order. The conversion to a steady-state form approximates the output from a constant input (e.g., estimated progress at 100% CPU consumption).

$$P_{ss} = \sum_{i=1}^n a_i P_{ss} + \sum_{j=1}^m b_j C_{ss}, \text{ where } P_{ss} \text{ and } C_{ss} \text{ represent } P \text{ and } C \text{ in steady state} \quad (6)$$

Note in (6), a_i and b_j are obtained from a job's performance model retrieved from the repository. For instance a first-order model ($m=n=1$) can be substituted to a steady-state form:

$$P_{ss} = a \cdot P_{ss} + b \cdot C_{ss} \quad (7)$$

From the steady state form, we derive the estimated scheduling credit for keeping the job's progress at the target:

$$P_{ref} = \sum_{i=1}^n a_i P_{ref} + \sum_{j=1}^m b_j C_{est} \quad (8)$$

$$C_{est} = \frac{(1-a_1-a_2-\dots-a_n)}{b_1+b_2+\dots+b_m} \cdot P_{ref} \quad (9)$$

In a first-order model, the scheduling credit can be approximated as:

$$C_{est} = \frac{(1-a)}{b} \cdot P_{ref} \quad (10)$$

The AC makes an Accept/Reject decision using the estimated scheduling credit, C_{est} , subject to the following constraint:

$$\mathbf{C}(1): \frac{C_{est}}{M \cdot N} + \sum U_i \leq U$$

where U_i is estimated utilization of a job in the server, and U is the server's utilization bounds, and M is a number of reserved cores

The constraint, $C(1)$, specifies that aggregate utilization of all accepted jobs plus the estimated utilization of a new job should be less than the server's utilization bounds. Since we measure the progress of running jobs via our sensor library, we can continuously update the remaining milestones of all jobs (i) in the server and derive their estimated utilization at every instant, k :

$$Milestone_i(k) = Milestone_i - \sum Progress_i(k) \quad (11)$$

$$Deadline_i(k) = Deadline_i - RunTime_i(k) \quad (12)$$

$$P_{ref}^i(k) = \frac{Milestone_i(k)}{Deadline_i(k)} \quad (13)$$

$$U_i(k) = \frac{C_{est}^i(k)}{M \cdot N}, \text{ where } C_{est}^i(k) \text{ is an estimated scheduler credit derived from (9) and (13)} \quad (14)$$

5 MEMORY SCHEDULING VIA VM CHECKPOINT

After a job passes admission control and is accepted to a server, it is assigned to one of worker VMs that are either in the off or checkpointed state. Since there is a potentially large number of jobs admitted to a server with limited physical memory, we regulate a job's memory allocation

according to its urgency. The worker scheduler schedules limited memory using VM's checkpoint capability.

In the current implementation, we have found it effective to use a simple, equal-sharing policy by which memory partition is equally distributed among concurrent worker VMs. The scheduler has a maximum number of concurrent worker VMs as a configurable parameter. The upper bound on a concurrent worker is determined by the systems' available memory and job's working set. While there is no strict lower bound, the typical setting will allow workers more than the number of server's cores to run concurrently.

In our approach, a worker begins executing a job sufficiently earlier than the deadline in order to give the job's feedback controller enough time to cope with disturbances. As an illustration, assume that a job should process 1,000 files and the job's performance model predicts 10 files/sec rate of processing at 100% utilization. If the job begins execution exactly 100 seconds before the deadline, the job can finish within the deadline *only if* there is no disturbance affecting the job's performance. However disturbances are inevitable elements of shared system. Therefore, we start the job earlier than projected execution time and let the feedback controllers cope with disturbances at runtime.

Every job admitted to a server is maintained in three job queues that sort jobs in priority order. The ready-queue holds accepted jobs that have not been executed. The suspend-queue contains jobs that were checkpointed as a result of priority enforcement. Finally, run-queue maintains jobs that are currently running in worker VMs. A job's priority is used for enforcing the order of worker VM's memory allocation (i.e., execution). As an illustration, if 8-core server supports 16 VMs running concurrently, some jobs may wait in ready or suspend queue if the number of accepted jobs is more than 16.

The basic algorithm to determine a job's priority uses the job's estimated utilization. At every schedulable instant, a job's utilization can be estimated using (14) and the jobs with higher utilization demands are assigned higher priority. That is, for two jobs i and j , the worker scheduler enforces the constraint:

$$C(2): Priority_i \geq Priority_j \text{ if and only if } U_i(k) \geq U_j(k)$$

The worker scheduler maintains the invariants:

$$C(3): State_i = RUNNING \ \&\& \ State_j = \{READY|SUSPENDED\} \text{ if and only if } Priority_i \geq Priority_j$$

If an update to a job's estimated utilization, followed by a sensor's progress report, breaks the above constraints, the worker scheduler performs preemption by which a worker VM running the lower priority job is checkpointed, and the higher priority jobs are resumed or started.

6 FEEDBACK CONTROLLED CPU SCHEDULING

After a worker VM begins executing the assigned job or resumed from a checkpointed state, the job's CPU consumption is strictly controlled by feedback controller daemon running inside the worker. From the viewpoint of a job, the feedback controller is equivalent to a virtual CPU that dynamically throttles its clock speed to keep the

job's progress at a target goal. As expressed in (13), the job's target progress is derived from the remaining milestone divided by job's relative deadline.

We use formal control theory to design feedback controllers to ensure the controller has desirable properties in terms of predictability when tracking the target progress. The control theory allows one to design a feedback controller with mathematically provable accuracy and speed. Our framework uses Proportional-Integral (PI) controllers. Figure 2 illustrates the closed loop with PI controller and the target system expressed in a first-order model. The input to the controller is the error, $E(k)$, (reference - measurement), and the output of the controller is $C(k)$, the scheduling cap of Hyper-V. The PI controller law has the mathematical form:

$$C(k) = C(k-1) + (K_p + K_i)E(k) - K_p E(k-1) \quad (15)$$

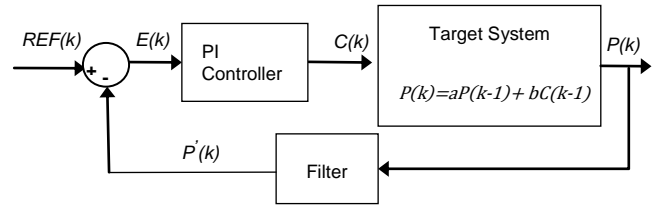


Figure 2: Closed Loop with PI-Controller

The jobs' model (3) and the PI control law are in the time-domain expression of the system. The classic control theory provides different, more convenient way to encode time-series and to describe systems, called Z-transformations. Z-transform uses the variable z to indicate time delays and encode time-domain representation of a signal as a sum of the coefficients of z -term. If a z -transform is used to describe a system's component such as PI-controller, job's performance model, we call it transfer function that describes how an input signal is transformed into an output. For example, a transfer function of job's performance model relates how Hyper-V's scheduling cap (input signal $C(k)$) affects the job's progress (output signal $P(k)$). By using a transfer function, the system's discrete components can be combined via simple algebraic manipulations. Due to space limitations, we do not provide a more rigorous definition of z -transform and proofs of properties that we present hereafter. Interested readers are referred to control textbooks [25][26].

The target system, expressed in the first-order equation (4) and the PI controller (15) can be modeled as transfer functions in Z-domain as follows:

- $G(z) = \frac{P(z)}{C(z)} = \frac{b}{z-a}$, where $P(z)$, $C(z)$ are z -transforms of $P(k)$, $C(k)$ and a, b are the model parameters of target system that we obtained from system identification.
- $D(z) = \frac{C(z)}{E(z)} = K_p + \frac{K_i z}{z-1}$, where $D(z)$ is the z -transform of equation (15) and $E(z)$ is the z -transform of error, $E(k)$

In the control loop, we place a filter in between the measurement and controller input in order to eliminate measurement noises. If we denote $P(k)$, $P'(k)$ as input and output of the filter, the filter has the form:

$$P'(k+1) = \alpha \cdot P'(k) + (1-\alpha)P(k) \quad (16)$$

In equation (16), α determines the desired level of smoothing and we found $\alpha=[0.8-0.9]$ shows stable outputs for the tested applications in Section 8. We denote, $H(z)$,

as the z -transform of filter (16). The closed-loop transfer function, representing the overall system in Figure 2, with reference as input variable and the measurement as output is constructed by integrating the controller, target system, and filter into a unified function:

$$FR(z) = \frac{P(z)}{REF(z)} = \frac{D(z)G(z)}{1+D(z)G(z)H(z)} \quad (17)$$

The goal of controller design is to ensure that the behavior of target system is within the constraints that satisfy the following requirements:

- **Accuracy:** the controller must guarantee that the measurement converges to the reference, even when disturbances affect the target system
- **Settling time:** the controller must quickly settle to steady-state value (i.e., reference) in the event of reference changes and disturbances

The PI law (15) guarantees the accuracy of the control loop since the equation uses the previous error history and regulates the target system’s actuation until the accumulated errors are eliminated. The final aspect of the control design is thus to choose the right values for controller parameters, K_p and K_i , in equation (15) that satisfies the settling time requirements. The settling time of the closed-loop system are determined by the poles and zeros of the transfer function (17), which are in turn determined by values of K_p and K_i . In general, there is a trade-off between the stability¹ and settling time of system. When a system is unstable, the output of the system, job’s progress, can oscillate between two extreme values. The faster response to disturbances or reference changes often causes unstable condition because of significant actuation changes in short period. Thus, one must carefully choose the right parameters that show fast settling time without causing unstable condition. Root Locus is the most common, graphical technique that plots the traces of poles and zeros of the closed-loop system, as control parameters, K_p and K_i , vary. As an illustration, for BLAST and WRF, desired values of K_p and K_i that we manually found using Root Locus method (using Matlab) are:

- BLAST: $K_p = 0.005$ $K_i = 0.007$
- WRF: $K_p = 0.028$ $K_i = 0.07$

Furthermore, we developed the automated tuning algorithm, presented in the Appendix, which computes the near-optimal controller parameters in terms of the settling time and stability. The automated design is particularly important to the following adaptive heuristics.

7 ADAPTIVE MODELING AND CONTROL

7.1. Real-Time Modeling and Scheduler Design

A drawback of model-based utilization test and scheduling comes from the difference between the model and reality. We found the real performance can drift from model when jobs run concurrently with many other jobs, causing disturbances to each other. There are many poss-

ible forms of disturbances in a shared HPC system, including shared disk I/O, network load, and hardware upgrade/replacement, to name a few. Among those, we found the two disturbances are particularly problematic:

- **Variable Workloads:** A HPC server may run different, unpredictable mix of jobs. We found a job’s performance depends significantly on the type of jobs concurrently running on a server. For instance, while BLAST-I/O-intensive job-performs well when others are mostly compute-intensive, its performance degrades rapidly when other I/O-intensive jobs share a disk.
- **CPU Loading Factor:** besides I/O intensity, aggregate CPU utilization affects job’s performance. In other words, on 8-core VMM server, average running time of 8 single-threaded jobs is longer than 4 jobs, due to increased overhead in VMM kernel’s address translation, paging, and system call virtualization [18].

As HPC system becomes powerful with an increased number of cores, the disturbances would become more severe. The classic approach to address model inaccuracy in real-time systems is to use a worst-case model; however in open systems it is difficult to guarantee a model is indeed a worst case, and the system’s utilization can be extremely low as the job’s average performance is frequently better than the worst-case.

A benefit of feedback controller is its ability to cope with disturbances. The controller can throttle up CPU allocation if the measured progress is below target due to disturbances. While this feedback mechanism can address run-time disturbances locally to a job, it may fail when the server’s limited capacity is saturated by a large number of feedback controllers. When the server is fully overloaded, feedback controllers see increasing errors in meeting progress requirements and keep requesting more cycles. This can lead to a state in which every job tries to get more CPU without ever releasing its allocation. The server then degenerates to an uncontrolled time-sharing system.

Another, perhaps more catastrophic, consequence of disturbances is an incorrect admission decision made by AC. The utilization tests based on optimistic performance model can result in ‘Accept’ decision in the system that has no more remaining capacity to serve the new job. The job would wait in the ready queue too long or be allocated less CPU than necessary to complete before deadline. After the system transits into an unpredictably loaded state, it is difficult to revert it back to a normal state since AC keeps accepting new jobs based on inaccurate performance models that computes the aggregate utilization of a server to be less than reality.

Our design and implementation addresses the problem using adaptive modeling and controller design as illustrated in Figure 3. While the PI controller changes its VM scheduling credit ($C(k)$) to achieve the desired state ($REF(k)=P(k)$), the allocated scheduling credit, $C(k)$, and sensed progress, $P(k)$, are exported to the *Model Estimator*. In the current implementation, the *Model Estimator* and *Control Tuner* are running as a separate service daemons, outside the worker VMs, and communicate with the controller daemons. A single adaptation server provides services to all controller daemons in a physical server.

¹ In control theory, there is a theorem that guarantees a system’s stability using the poles of the closed-loop. However, we do not offer a stability guarantee because the result is limited to single closed-loop systems and not directly applicable to distributed systems including our work. In this paper, the term stability is used in a generic sense and does not mean the rigorous result from the formal analysis.

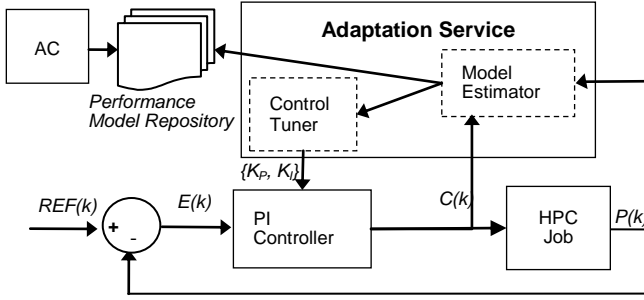


Figure 3. Adaptive Modeling and Controller Design

The message from a controller daemon contains the pair $\{C(k), P(k)\}$. If enough messages are received (i.e., enough history has been obtained), the model estimator runs least square regression and produces a performance model as expressed in (3). For each job, the model estimator continues to perform regression and produce a new model using the most recent history. A newly constructed model must pass a validation test that consists of the three constraints:

(C4): The model is verified using the standard R^2 tests as defined by
$$R^2 = 1 - \frac{\text{var}(y - \hat{y})}{\text{var}(y)}$$
 where $\text{var}(y)$ is the variance of y and \hat{y} is predicted output by the model. The R^2 of the new model applied to the most recent history must exceed the predefined thresholds (e.g., 0.7).

(C5): In the model expressed in (3), a_i for all i must be less than 1.0. If $a_i \geq 1.0$, the PI controller causes instability of the closed loop (see the Appendix).

(C6): Similarly in (3), b_j for all j must be positive because throttling-up CPU allocation cannot result in decreased progress.

Throughout extensive tests with real HPC workloads, we found the three validation steps correctly filter out an inaccurate performance model that might cause unstable feedback controllers or wrong utilization tests. The model that passes the tests is updated to the repository and therefore the AC's utilization test uses the most recent model. If the constructed model does not pass the test and thus is discarded for a long period of time, the system may not capture the accurate performance in real-time. However, we found, throughout the extensive tests, this filtering approach works well in practice.

The successful model is also given to the control tuner to construct new control parameters. Our controller design heuristics search for the PI parameters (K_p, K_i) using the controller's mathematical properties regarding transient performances. The Appendix presents the control tuning heuristics in more detail.

One implementation issue concerns variability in the input, $C(k)$, and outputs, $P(k)$, of the closed-loop system. If the closed-loop is in steady-state, there is not enough variability in inputs and outputs to capture a relationship between them. For example, if the controller has been wound up because a job's measured progress never reaches the target, the most recent scheduler credit ($C(k), C(k-1) \dots$), requested by controller contains only the maximum in the ranges of VM scheduler credit. Least square regression cannot be performed on data without variability. We address the issue by occasionally randomizing $C(k)$ requested by controllers. More specifically, in some fixed cycles, we randomly choose a positive or negative three-

holds that is added to a controller's output, $C(k)$. Thus, we introduce variability forcefully to support accurate adaptive modeling at run-time. The parameters for randomization should be chosen carefully to not to interfere feedback controllers too significantly.

7.2. Overload Protection via Loop-Switching

Loop switching protects the server from overloading. As feedback controllers keep changing VM scheduling credit autonomously, the aggregate requests can exceed the server's preconfigured reservation (e.g., 800 on 8-core server). When this overload condition occurs, each feedback controller sees more degradation in a job's progress and requests even more scheduling credit. In PI controller, this creates an *integral wind-up*, a situation in which accumulated errors in the controller drive the system into an unmanageable state. While *feedback* helps the scheduler to cope with disturbances, the *feedback* can also become a source of unpredictability when overloaded. It is therefore necessary to "remove" feedback in some jobs to help the server avoid overloading and thus feedback controllers of other jobs can perform normally. Figure 4 illustrates the open-loop CPU scheduler compared to closed-loop controller in Figure 2.

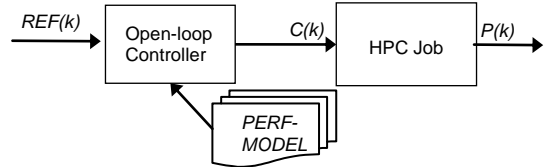


Figure 4. Open-loop scheduler

The open-loop scheduler uses a performance model (i.e., the most recent model updated to the repository) to find the right scheduling credit by solving (9). If there are no disturbances affecting a job's progress, the open loop scheduler's output is equivalent to the feedback controller's (see Figure 6). In a real deployment with non-trivial disturbances, the open-loop controller cannot achieve 100% accuracy, because a job's real performance degrades from the model's prediction. Therefore, switching to open-loop *does not* guarantee that the jobs in open-loop will sustain the target progress and meet the job's deadline. However, when many controllers are run independently, it helps the system to avoid overloading in the following sense:

- When a server is overloaded, the sum of VM's scheduler credit exceeds the utilization bounds.
- A feedback controller with accumulated positive errors ($REF(k) - P(k) > 0$) issues increasing CPU credits.
- By turning a feedback-loop to an open-loop, the increased CPU credits due to accumulated errors are removed from the system.

When our framework detects overloading, it picks one of running jobs and switches its control logic to open-loop until the server transits back to a normal state. A simple heuristic is to pick the lowest priority job - later, when the server is less loaded, the job can be switched back to closed-loop and the PI controller can address accumulated errors. However, we found the most useful heuristic is to pick the lowest priority, I/O jobs first.

Table 1. Prototype Components

	Sensor Library	Feedback Controller Daemon	Modeling& Controller Design Tools	AC/ Worker Scheduler	VM Management	Total
Language	C, Fortran	Java	Java, Matlab	C# (.NET)	C# (.NET)	
LoC	1.2 K	5.2 K	4.5 K (Matlab: 0.6k)	7.3 K	6.9 K	25 K

Table 2. HPC Applications Used for Evaluation

	ADCIRC [4]	BLAST [7]	mProject(Montage) [8]	OpenLB [5]	WRF [6]
Domain	Costal flood modeling	Biology	Astronomy	Fluid-flow modeling	Weather Forecast
Compute/Data Intensive	Compute-Intensive	Data-Intensive	Data-Intensive	Compute-Intensive	Compute-Intensive
Milestone	Source code	Linear estimation	Program manual	Source code	Linear estimation
Number of Sensor Calls	1	3	1	1	1

Table 3. HPC Server Configuration

	CPU	RAM	Disk	O/S
Desktop configuration	Intel Dual Core 2-2.13 Ghz	3 GB	10,000 RPM, SATA	Microsoft
HPC Server configuration	2 Quad Core Opteron-1.7Ghz	8 GB (256 MB/VM)	7,200 RPM, SATA	Hyper-V Server

In a workload of mixed CPU and I/O-intensive jobs, the I/O-intensive jobs perform significantly worse than prediction due to competition on shared medium. When the feedback controllers for I/O-intensive jobs consume too much credit to compensate for degraded performance, the CPU-intensive jobs suffer as well due to large aggregate credits requested for a limited capacity. It is more prudent to protect the CPU-intensive jobs from unpredictable behavior of I/O-intensive jobs by removing “aggressive feedback” in I/O-intensive jobs first.

Note that a loop’s switching may cause instability of controllers, because sudden, significant changes in one control loop may cause cascading changes in other controllers. Therefore, the heuristics should be performed infrequently and should ensure it does not reduce/increase server’s load too much at a time. In our experiments on an 8-core server, the switching is performed once every few minutes and the server’s load is changed by 1/8 at most. Between the switching, feedback controllers readjust to the graceful changes in server’s load.

7.3. Discussion on the adaptive heuristics

The heuristics presented in this section are by no means verifiable through the theory. Rather they are developed incrementally after observing the drawback of the direct application of control theory to HPC workloads. There are works in which the adaptive control theory is applied to cope with application’s changing performance [29][31]. However, our heuristics approach is necessary as we support multiple, concurrent HPC jobs that exhibit extremely variable performance characteristics. Unlike the existing works that explicitly models the interactions between the known workload types (e.g., QoS classes), it is hard to model such interactions between the unpredictable mix of HPC jobs. Furthermore, the framework should accommodate the unknown, future applications without significant changes.

In summary, the control-theoretic modeling and scheduler offers the verifiable properties such as control speed for individual jobs. The admission controller and the adaptive heuristics create an environment where the scheduler works as the off-line prediction by the theory.

8 PERFORMANCE EVALUATION

8.1. Implementation on Hyper-V

We implemented HPC admission and feedback control

framework on top of Hyper-V as a virtualization platform. The major components of the prototype include a sensor library, modeling/controller design tools, feedback controller daemon, admission controller, worker scheduler, and VM management layer that interacts with Hyper-V via management APIs. Table 1 illustrates the implementation language and lines of code (LoC) for each component.

8.2. Workloads and HPC Server

The workloads used for evaluation consist of five widely used applications from diverse domains, as illustrated in Table 2. The milestones of each application are derived from three application-specific sources. In *ADCIRC* and *OpenLB*, the milestone is directly read from the source code as the variables containing the bounds of outer-most loop corresponded to job’s overall computation. In *Montage’s mProject*, the milestone simply corresponds to number of raw image files to process. The *BLAST* and *WRF* required a linear estimation from job’s quantifiable problem size (number of protein queries in *BLAST* and forecast hours in *WRF*). The last row of Table 2 presents the number of sensor call placements in each application. Only a small number of placements were sufficient for each application.

The hardware and software configuration of the server used for evaluation are described in Table 3. We use different hardware configuration in the following subsections. For the evaluation in subsection 8.3, where performance of feedback controllers is measured in small-scale, we use a desktop configuration. In subsection 8.4 and 8.5, we use more powerful 8-core HPC server as we measure performance of many concurrent jobs.

8.3. Feedback Control Performance

This subsection evaluates the applicability of control theory to predict and regulate the behavior of HPC applications. Figure 5 illustrates how the choice of controller parameter determines the behavior of *BLAST* (we omit the results from other applications due to lack of space; similar results appear in our earlier study [34]). We run the feedback-controlled *BLAST* as reference is changed at 70 and 140th seconds. The straight, dotted line represents the reference progress, which is the target that the controller aim to track, and the curved, dotted line represents the simulated progress that we can obtain when controller is designed using the procedures introduced in Section

6 (Matlab is used to produce the simulated results). The solid line represents the result that we obtain from real execution. As discussed previously, we assume a user has an application-specific knowledge about the scale of milestone and progress. Thus, the scale of progress in y-axis is specific to BLAST. The three graphs (Fig 5 (a)-(c)) present results when different controller parameters are chosen. The parameters in Fig 5(a), (b), and (c) represent aggressive, slow, and good controller, respectively.

In all figures, we can see that the results from real execution exhibit similar trends to the simulation. This indicates that the controller design that we present in Section 6 is effective in predicting the real behavior of a job, and therefore we can choose correct control parameters with high confidence.

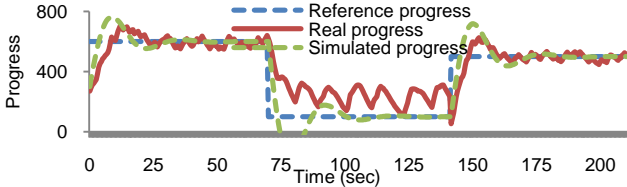


Figure 5(a): Aggressive Controller ($K_p=0.012$, $K_i=0.018$)

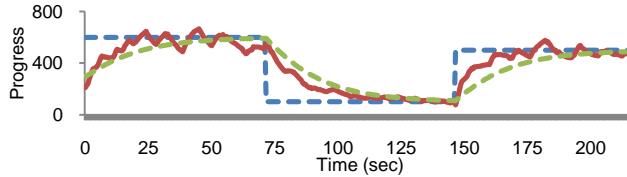


Figure 5(b): Slow Controller ($K_p=0.0016$, $K_i=0.0024$)

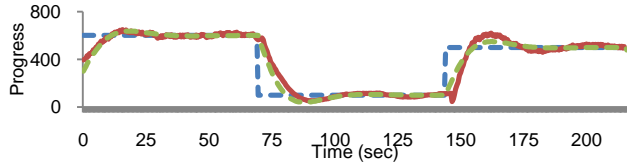


Figure 5(c): Good Controller ($K_p=0.005$, $K_i=0.007$)

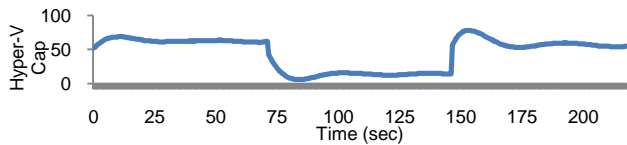


Figure 5(d): Changes in Hyper-V's Scheduling Cap

As we see in Fig 5(a), the aggressive controller (high-gain controller) responds to reference changes faster than the others. However, it fails to track accurately as it becomes unstable at {70-140} seconds. The slow controller (low-gain controller) shows the opposite. While the controller is slow in responding to reference changes, the real progress eventually converges to the target and stays at the reference stably. In terms of control actuation, the aggressive controller changes Hyper-V's scheduling cap as soon as it sees errors (reference-measurement) while the slow controller does not change the value immediately and waits until enough errors are accumulated. The controller needs to react to reference changes rapidly especially when computation is relatively short (e.g., BLAST). Also, a fast response means that the controller rejects disturbances rapidly. Thus, the controller parameter must be chosen after considering the trade-offs between controller speed and stability. The good controller, as illustrated in

Fig 5(c) has parameter values that are intermediates of the aggressive and slow one. The results from the good controller and the corresponding Hyper-V scheduling cap changes are presented in Figure 5(c-d).

Figure 6 illustrates how effectively the feedback controller accounts for disturbances. We run two VM instances on Hyper-V: one is the performance container that we aim to control and another instance acts as a disturbance generator. For simplicity, the application that we illustrate in Figure 6 is BLAST and the disturbance generator runs BLAST as well since it generates a huge amount of read requests to the disk that it shares with the controlled VM. The figure contains two results: baseline (dotted) and feedback control (solid). We run the experiment first on baseline case where the VM's scheduling cap is fixed at the value that is derived from the equation (9) with BLAST's model parameter. This corresponds to the open loop controller that we discussed in 7.2.

Figure 6 (a) illustrate the results when experiments run without disturbances. As we see in the figure, both the open-loop and feedback control tracks the reference equally well. However Fig 6 (b) shows that the open-loop fails to track accurately since the disk disturbance results in less progress than what open-loop control expects with the fixed Hyper-V scheduling cap. In contrast to open-loop, the feedback controller successfully rejects disturbances since the controller raises cap when it sees errors while tracking. With open-loop control, failure in tracking the reference increases errors in meeting deadlines.

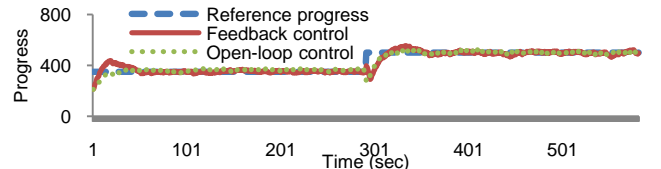


Figure 6(a): Control results without disturbances

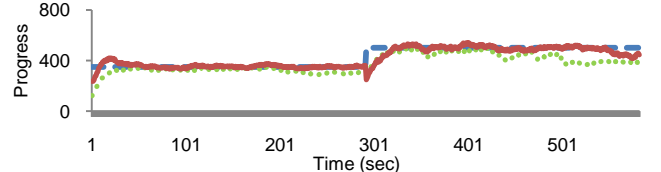


Figure 6(b): Control results with disturbances

8.4. Deadline-guarantee via Admission Control

In the previous subsection, we established that a feedback controller tracks a job's reference progress accurately and stably. This subsection evaluates the behavior of our system when a large number of jobs are admitted to AC and the feedback controllers operate on jobs autonomously, potentially causing conflicting resource consumption. We use an 8-core AMD server for this experiment. Throughout the test, the server reserved 7 cores to support AC-enabled job execution. Thus, the scheduling credit representing a server's capacity was 700. Also, the server is configured with 14 worker VMs as a maximum number of concurrent VMs. For the test, we set up a client program that reads synthetic workloads from a workload file. The workload file contains job's descriptions along with deadlines. The client attempts to submit a job

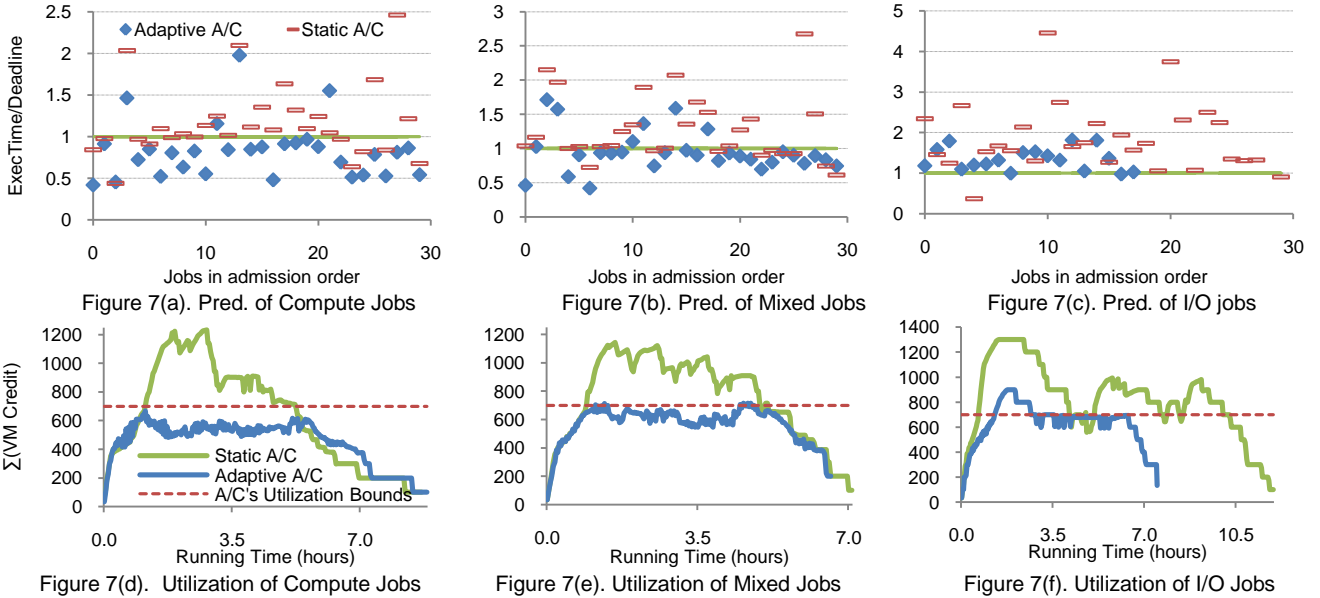


Figure 7: Static vs. adaptive admission control

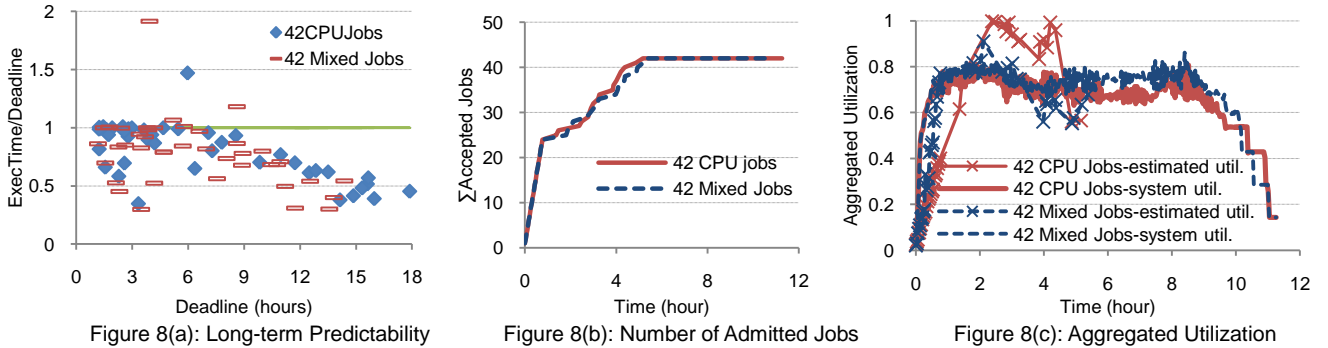


Figure 8: Achieving long-term predictability

until the job is accepted by the server and terminates after all jobs specified in the workload file are submitted. We constructed three different types of workloads:

- *Compute-intensive jobs*: jobs constructed from ADCIRC, OpenLB, and WRF.
- *I/O-intensive jobs*: jobs from BLAST and Montage
- *Mixed jobs*: jobs from all applications with ratio of compute and I/O-intensive jobs, 50:50.

A job's deadline is chosen randomly from a range described in the workload file. The result is collected from extensive tests throughout a week. It took about 10 hours to complete one synthetic workload (30-42 jobs), and we run 10 such tests throughout the experiment. A job's running time on idle server was in the range of 0.5-2 hours.

Static vs. adaptive admission control

The first evaluation compares the static modeling to the adaptive one as we discussed in Section 7. Unlike static modeling with predefined models, the AC based on adaptive modeling uses a variable performance model that was being evolved throughout the experiments. We run all three types of workloads, each consisting of 30 jobs. Figure 7(a-c) illustrates the results comparing the two approaches. In the figure, x-axis represents jobs sorted in the order of admission time and y-axis represent the ratio of real execution time to job's requested deadline. The ratio below 1.0 indicates the job's meeting deadlines.

The graphs clearly indicate that the adaptive modeling outperforms static modeling in terms of meeting a deadline. For all three different workloads, a large fraction of jobs fails to meet the deadline with static modeling. In contrast, the adaptive AC achieves good predictability in the workload of compute-intensive jobs and mixed jobs. In compute-intensive workloads, only 4 out of 30 jobs fail to meet the deadline (86% success). In mixed workloads where 50% of jobs were I/O-intensive, 7 out of 30 jobs fail to meet the deadline (76% success). However, both approaches result in a large failure ratio for the I/O-only workload. This is due to the workload-dependent disturbances that we discussed earlier. The I/O-intensive jobs compete for the limited bandwidth of shared disks and cause significant performance degradation. Even adaptive AC cannot predict the model correctly if the job's real performance drifts from the model too significantly. To solve this problem, the more conservative approach would be necessary. For instance, if the I/O-only workloads are frequent to the system, the administrator would manually adjust job's performance model such that the adjustment predicts performance only the half of the original model (because all jobs could finish in less than the twice of the requested deadline).

The graphs in Figure 7(d-f) illustrate the causes behind the different performances. The y-axis in the graphs represents the sum of VM scheduler credit requested by

feedback controllers. As we reserved 7 cores, the utilization bounds were 700 (the dotted line in the figures). We can see the aggregate credits in the static AC exceed the utilization bound. Too many jobs are admitted to the server due to the optimistic performance model and feedback controllers with relatively high reference progress saturated their possible credit range. However, we can see throughout the workloads that adaptive AC can predict variable workloads very well and the feedback controllers do not overload the server.

Achieving Long-Term Predictability

So far, a synthetic deadline specified in the workloads is relatively close to the real execution time of the job. For the previous evaluations, it was randomly chosen from the range [1.0 - 4.0] multiplied to real execution time of the job (we obtained the real execution time prior to the test). The deadline close to real execution time results in higher loading factor accumulated in AC and therefore only a small number of jobs were accepted to the server. In this section, we evaluate the AC's long-term predictability by choosing deadlines from more broad ranges. More specifically the workload client chose random deadline from the range [2.0 - 10.0] multiplied to real execution time of the job. Figure 8 presents the results from running two types of workload, compute-intensive jobs and mixed jobs. Each workload consists of 42 jobs respectively. Figure 8(a) illustrates the measured predictability in terms of ratio between real execution time and requested deadline. The x-axis represents the requested deadline of each job (Note a job's running time was 0.5-2 hours). Figure 8(b) presents the accumulated number of jobs accepted to the server. Finally, Figure 8(c) presents the total utilization of jobs that are accepted to the server. As shown in Figure 8(b), 25 jobs were accepted to the server within an hour. The jobs were no longer accepted by the server since there were no more worker VM images to assign the job (we had 25 worker VM images). The remaining 17 jobs are accepted to the server within next 4 hours. The experiment took about 12 hours to complete. As shown in Figure 8(c), the server's estimated utilization ($\sum U_i$ for job i) and the real system utilization (sum of scheduling credit) was kept high throughout the experiment. In the steady-state, the system's utilization remains above 75%. The result is obtained when adaptive modeling and loop switching was activated.

Figure 8(a) clearly indicates AC achieves high accuracy in meeting deadlines for all accepted jobs. Regardless of the relative deadline, as shown on the x-axis, the server was kept highly predictable throughout 12-hour run. If the server were unpredictably overloaded for any period of time, the jobs terminating after that period would have experienced significant deviation from the requested deadline. The result does not indicate any such unpredictable load. Overall, 90.4% of jobs in compute-intensive workloads are completed before the deadlines. The average deadline error, as defined by, $\frac{ExecTime - Deadline}{Deadline}$, of late jobs were only 12.4%. For mixed workloads, 80.9% jobs were completed before deadlines, and the average deadline error of late jobs were 14.7%.

8.5. Comparison to VM-Leasing

Our design principle for building predictable HPC systems is a controlled-time sharing in which jobs are under strict control by feedback schedulers that govern a job's access to CPUs. So far in the experiment we run 14 worker VMs concurrently on reserved 7 cores. Thanks to the checkpoint capability that enables a job's preemption, people have recently began pursuing VM-leasing abstraction to schedule HPC resources to multiple users [19][20]. A possible extension to enhance predictability is to schedule jobs exclusively on available cores in a priority order. As more urgent jobs arrive in a server, the jobs running on a core can be checkpointed and resumed later. There are many differences between our design principle and VM leasing if extended to support predictability:

- *Modeling performances*: while our approach uses formal expression to model job's performances, it is not clear how leasing would express a job's performance. A simplistic assumption would be to rely on user's estimation (e.g., wall-clock-time option in $qsub$).
- *Agile priority enforcement*: we enforce priority of jobs by throttling job's CPU consumption. Compared to checkpoint, throttling allows rapid priority enforcement which is essential to cope with disturbances.
- *Priority enforcement on parallel machines*: one limitation of checkpoint as a priority enforcement mechanism is the difficulty of supporting parallel jobs. Checkpointing a low-priority task, which is a part of a job running across machines, implies not only technical challenges to support synchronized checkpoint, but also complicates deadline scheduling algorithms as they should scale to parallel jobs. Compared to checkpoint, throttling can be an effective tool to enforce priority across machines.
- *Memory pressure*: a limitation of time-sharing approach is increased memory pressure due to larger number of jobs running concurrently.

Additionally, we present a quantitative evaluation comparing the two approaches. We created a simple leasing abstraction by disabling feedback controller daemons and setting concurrent VMs equals to reserved cores. We implemented an EDF scheduler, as a worker scheduler, that schedules a job's worker VM according to the EDF priority rule. The admission controller rejects jobs if the total utilization ($\sum \frac{ExecTime}{Deadline}$) of all accepted jobs plus the new job exceeds utilization bounds of 1.0. The client submits a workload of 42 mixed-jobs whose deadline range is [1-3] of job's estimated execution time. In fact, as we used the real execution time of jobs we obtained before experiments, we were assuming the best cases in EDF where job's performance model is 100% accurate. Table 4 presents the results comparing the two approaches.

Table 4. Comparison of Two Predictable Schedulers

	Percent of jobs meeting deadline		Average deadline error		Overall execution time
	Comp. Jobs	I/O Jobs	Comp. Jobs	I/O Jobs	
Controlled time-sharing	80.9 % (17/21)	47.6 % (10/21)	13.7 %	21.5 %	10 h 40 m
VM-leasing (EDF)	66.6 % (14/21)	42.8 % (9/21)	12.9 %	39.3 %	11 h 30 m

Our approach (controlled time-sharing) achieves better predictability for both types of jobs. Over 80% of compute-intensive jobs meet the deadlines compared to 66% obtained from VM-leasing with EDF. Although our approach did not meet the deadlines of many I/O-intensive jobs (47% success), still the average deadline error of late I/O jobs are small (21%) compared to VM-leasing (39%). The reasoning behind the difference is that controlled time-sharing is more effective in handling large disturbances. In VM-leasing with EDF, if I/O-intensive jobs suffer from significant performance degradation and miss their deadline, the jobs in ready or suspended queue wait too long to be allocated to a worker VM. This causes a cascaded failure of meeting deadlines. However, our approach uses loop switching to protect compute-intensive jobs from overloading by I/O-intensive jobs, and resulted in more compute-intensive jobs meeting deadlines.

A surprising result appears in average deadline error of late I/O jobs. Although we use loop switching to limit I/O job’s CPU consumption, the average error of late I/O jobs is significantly less than with VM-leasing. In other words, limiting I/O job’s CPU consumption could actually result in better performance of I/O jobs! In VM-leasing, many I/O jobs may be allocated exclusively on fast cores, and each job generates huge I/O request that saturates disk’s bandwidth. Faster CPU in fact caused too much disturbance and eventually caused worse performance.

Another unexpected result appearing in overall execution time can be explained along the same reasoning. Our approach completed execution of all 42 jobs almost 1 hour earlier. In general, throttling VM credit causes overhead to VMM’s CPU scheduler and we observed the overhead could be potentially significant to support many feedback controllers simultaneously. However, as we address large I/O disturbances effectively via throttling down I/O job’s CPU access, the server’s overall performance in fact improved! We believe this experimental result illustrates a compelling case for building predictable HPC system on controlled time-sharing principle.

9 DISCUSSION

So far, we showed that the control-theoretic scheduler achieves good predictability for the challenging HPC workloads. There are remaining issues that potentially limit the wide deployment of this approach: *usability*, support for *parallel jobs*, *VM performance overhead*, and *control stability*.

In open HPC systems (e.g., Grid), the clusters do not in general require a sophisticated description about job’s properties. However, building a model-based scheduler assumes that a system has profiled jobs previously or there is an extra profiling step at job’s start-up time. Based on our experiences, though, the usability can be improved:

- The modeling and controller design can be automated without requiring extra tuning by users. The evaluation results are based on the algorithm in the Appendix.
- As shown in Table 2, adding sensing function to the existing source code can be trivial for many applications. One promising, transparent approach is to use hardware counters for profiling.

- While this paper addressed the use case that all jobs are deadline sensitive, there are jobs that do not require such firm guarantee. The best-effort scheduling that uses only the remaining cycles from deadline-scheduler can be used to support such batch jobs. Only the deadline-sensitive application would require an extra effort.

Nevertheless, the usability issue calls for future research.

The second issue concerns the support for parallel jobs which represent a large share of HPC workloads. For shared-memory, multi-threaded jobs, we believe the current framework does not require major changes because VM’s vcpu binding works together with throttling; we can throttle 8 vcpus assigned to a VM. In this case, the admission policy can be slightly changed as follows:

$$C(1) : \frac{P \cdot C_{est}}{M \cdot N} + \sum U_i \leq U \quad \text{where } P = \text{job's number of threads}$$

The more problematic issue that we leave as a future work is the increased complexity to model jobs that have variable number of threads. The significant future research is necessary to support distributed memory multi-threaded jobs (e.g., MPI). We need not only a synchronization construct to checkpoint and throttle parallel tasks simultaneously, but also more sophisticated admission control policy and deadline scheduling algorithms. We leave them as future works.

The next issue concerns the performance loss due to the use of VM as an actuation mechanism. Although the recent studies argue that performance loss of HPC applications on modern VMs is tolerable [39][40], still many in HPC communities remain skeptical about embracing virtualization. To address the concern, we measured the actual performance loss in our experimental setting. We ran the five applications on both guest O/S atop Hyper-V and stock O/S atop hardware (8-core server). Table 5 presents the performance overhead for each application. In the test, all five single-threaded applications were run concurrently. As a result, the compute-intensive jobs and data-intensive jobs showed about 15% and 22% of overhead, respectively, due to Hyper-V VMM. We believe that this slowdown, while significant, is a reasonable trade-off for this predictability and resource-sharing capability. This overhead could be reduced with further optimizations within our control software, as the subject of future research.

Table 5. Performance Overhead due to Virtualization

	WRF	ADCIRC	OpenLB	Blast	Montage
Overhead	14%	15%	17%	22%	22%

The final issue concerns the stability of distributed controllers. In our current approach, the stability analysis for a single feedback controller does not guarantee the stability of overall systems where many such controllers may interact with each other. This would become the more serious issue when we extend the work to support parallel jobs. This design is a result of practical considerations for HPC systems where there is an unknown mix of concurrent applications. It would be very difficult, if not impossible, to model the interactions among the wide variety of existing and future HPC applications. However, we believe still the stability analysis for a single controller serve as a valuable guideline in achieving stability of

overall systems because it can significantly reduce chances that a controller chooses wrong actuation. For example, as a baseline, if we are to design a feedback controller without consulting to stability analysis, it would be hard to determine to what extent to react to 20% performance degradation at a previous cycle. Too much actuation would lead to instability yet too small actuation causes inaccurate and slow responses. While we offer no guarantee regarding stability, we present a useful framework to reason about it. The admission controller and adaptive heuristics are presented as a practical approach to relieve instability. Based on our empirical evaluation, we present the conditions that stability of a controller is largely applicable to stability of distributed controllers:

- The fraction of data-intensive jobs in a concurrent workload should be kept small. While we could meet 90.4% of deadlines for compute-intensive jobs, the success rate is reduced to 80.9% when half jobs were disk intensive.
- A controller's actuation should be kept small compared to overall capacity of the system. This can help other controllers to adjust to the system changes smoothly. Also, it is necessary to avoid big system-wise changes in a short period of time. For example, loop switching should be carefully scheduled to avoid "double tumble".

In conclusion, we present the design and implementation of feedback and admission controller as a promising approach to building predictable HPC infrastructures. We use formal control theory to design a feedback controller that regulates a job's access to system resources in non-exclusive manner. The admission controller based on utilization test and adaptive modeling and control is designed to handle large workloads and support long-term predictability. Our evaluation results indicate that a feedback controller achieves high predictability in tracking progress of a job even in the presence of a wide variety of disturbances. In addition, the adaptive modeling/control allows a large number of feedback-controlled jobs run harmonious way, thereby achieving a highly predictable HPC server in scalable manner.

Appendix - online controller-design algorithm

In the on-line controller design, we draw requirements from the following properties of closed loop:

- **Accuracy:** PI-control law achieves zero steady state error since the integral term (K_i) accounts for the errors in previous history. Thus, accuracy does not add a constraint.
- **Settling Time:** The setting time and maximum overshoot define the transient behavior of a closed-loop system. The transient behavior refers to system's reaction when there is a change in reference or disturbances. The input signal, reference, to our closed-loop is a type of step, and the control theory offers a theorem that approximate the settling time, K_s , for a step input signal, as follows:

$$K_s \approx \frac{\log 100}{\log a}, \text{ where } a \text{ is the largest pole of the closed loop } (FR(z)) \quad (18)$$

- **Maximum Overshoot:** The maximum overshoot is defined as the maximum amount by which the transient value exceeds the steady-state value divided by the steady-state output. Smaller overshoot is desirable not only because the overshoot is a transient error, but also can leads to output oscillation in the following cycles. Since the closed-loop equation (17) is in higher-order having multiple poles, the poles of the loop can be either real or complex. If all poles are real, the maximum overshoot can be computed as:

$$M_p = -a, \text{ if largest pole of closed loop is negative.} \\ M_p = 0, \text{ otherwise} \quad (19)$$

For complex poles, we assume the largest complex poles, $p_1=c+dj$ and $p_2=c-dj$ (note roots of quadratic polynomials have a real part, c , and two imaginary parts with imaginary number j). Then, the maximum overshoot can be approximated as:

$$M_p \approx r^{|\theta|}, \text{ where } r = \sqrt{c^2 + d^2} \text{ and } \theta = \tan^{-1}\left(\frac{d}{c}\right) \quad (20)$$

We transform PI controller equation (15) to an equivalent form:

$$D(z) = \frac{(K_p+K_i)z-K_p}{z-1} = (K_p+K_i) \left\{ \frac{z^{\frac{K_p}{K_p+K_i}}}{z-1} \right\} \quad (21)$$

(K_p+K_i) and $\left(\frac{z^{\frac{K_p}{K_p+K_i}}}{z-1}\right)$ represent overall gain and zero of the PI controller, respectively. The goal of controller design is to select values for the gain and zero, whereby subsequently K_p and K_i are obtained by solving the equation (21). However, as gain and zero are real, there are infinite possible values for them. We use bounded search as a basic strategy, testing candidates to 1) see if the poles of the closed-loop are all within unit circle (to avoid instability), 2) estimate the settling time and overshoot, and 3) apply a rank function to choose a combination of zero and gain that minimizes an objective function. Figure 9 illustrates the pseudo-code of the heuristic. The arguments to the function are maximum numbers of candidates for zero (M) and gain (N), and the transfer function of a model (given by Model Estimator of Figure 3). The return values from the algorithm are near-optimal gain and zero, from which K_p and K_i are solved.

The algorithm picks candidates of zero and gain evenly distributed by M , N , within their valid range (line 5, 9). Since $K_p > 0$ and $K_i > 0$, the zero $\left(\frac{z^{\frac{K_p}{K_p+K_i}}}{z-1}\right)$ must be between 0 and 1. M and N must be limited to certain thresholds since routines to find settling time (line 11) and maximum overshoot (line 12) on z -transform equations are computationally expensive. In our Matlab implementation, the algorithm takes about 10 seconds for $M=10$, $N=20$. Thus, there is a trade-off between the algorithm's running time and the quality of output which is controlled by M and N . For a given constraint on the running time (e.g., 10 seconds), a good heuristic is to limit the search space for zero and gain to where it is more likely to produce better results. Line 2 is one such heuristic.

We set the smallest of zero candidates at the minimum pole of the application model (G_z), as the zero location with respect to the model's minimum pole has pivotal influence to the settling time of the closed-loop.

```

1 function [opt_gain, opt_zero] = ControllerDesigner (M, N, Gz)
2 zero_min = min (pole(Gz));
3 zero_max = 1.0;
4 zero_unit = (zero_max-zero_min) / N;
5 zero_values = zero_min:zero_unit:zero_max;
6 for i=1 to N
7   gain_max = stability_analysis(Gz, zero_values(i));
8   gain_unit = gain_max/M;
9   gain_values = 0:gain_unit:gain_max;
10  for j=1 to M
11    Ks(i,j) = compute_Ks(zero_values(i), gain_values(j), Gz);
12    Mp(i,j) = compute_Mp(zero_values(i), gain_values(j), Gz);
13  end;
14 end;
15 [i, j]= rank (Ks, Mp);
16 opt_gain=gain_values(i,j);
17 opt_zero=zero_values(i,j);

```

Figure 9: Controller Design Heuristic

Figure 10 illustrates a root locus of the closed-loop that shows the effects of the heuristic. In the figure, the solid line draws the branches of root locus (locations of closed-loop pole), stemming from the three poles of open-loop components (application model, filter, controller). The controller's zero is a small circle on x-axis and the three small dots are the poles of the closed-loop that moves along the solid lines. As we explained with equation (18), the settling time is proportional to the largest pole of closed-loop. As we see in the figure, zero location with respect to model's minimum pole (0.4) has significant influences on the possible locations of closed loop poles: zero location at the

right of minimum pole (c) produces the pole locations that moves toward circle's center (smaller poles), resulting in shorter settling time.

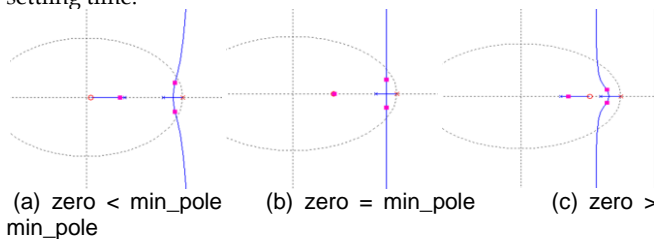


Figure 10. Effects of zero location with respect to min. pole

At line 7, the range of gain test is reduced as well using the stability analysis. According to the stability theorem, the close-loop can be stable *iff* all poles of closed-loop ($FR(z)$ in eq. (17)) are inside the unit circle (note this does not imply we guarantee stability of overall system because the theorem for single controller is not directly applicable to the distributed controllers; rather we use the theorem to avoid obviously unstable condition). Using the fixed zero candidate ($zero_values(i)$), we quickly test different gain candidates to see if the resulting largest closed-loop poles lie close to unit circle ($0.95 < \text{largest pole} < 1$). The stability analysis terminates after the gain candidate satisfying the condition is found. Using binary search, the gain location is found typically within few tests, and set as a maximum of possible gain range (a gain candidate larger than this is likely to result in unstable system).

After determining the ranges, the algorithm computes settling time (line 11) and maximum overshoot (line 12) using the equations (18)-(20) and stores the computed values to tables. Finally, the rank function (line 15) chooses the best candidates for gain and zero by examining the tables. In our implementation, we use simple rank function that finds the zero-gain pair achieving the shortest settling time while maximum overshoot is subject to a fixed thresholds (e.g., 0.2).

References

- [1] B. Plale, *et al.* Towards Dynamically Adaptive Weather Analysis and Forecasting in LEAD. ICCS workshop on Dynamic Data Driven Applications, Atlanta, Georgia, May 2005.
- [2] SURA Coastal Ocean Observing and Prediction (SCOOP): <http://scoop.sura.org>
- [3] S. Manos, *et al.* Life or Death Decision-making: The Medical Case for Large-scale, On-demand Grid Computing. CTWatch Quarterly, Volume 4, Number 1, March 2008.
- [4] Coastal Circulation and Storm Surge Model: <http://adcirc.org>
- [5] Open source lattice Boltzmann code: <http://www.openlb.org>
- [6] The Weather Research and Forecasting Model: <http://www.wrf-model.org>
- [7] BLAST: Basic Local Alignment and Search Tool (<http://www.ncbi.nlm.nih.gov/blast/>)
- [8] Montage-An Astronomical Image Mosaic Engine: <http://montage.ipac.caltech.edu>
- [9] I. Foster, *et al.* A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. International Workshop on Quality of Service, 1999.
- [10] R.J. Al-ali, *et al.* Analysis and Provision of QoS for Distributed Grid Applications. Journal of Grid Computing, 2(2), June 2004.
- [11] W. Smith, *et al.* Scheduling with Advanced Reservations. IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2000.
- [12] G. Singh, C. Kesselman, E. Deelman. Adaptive Pricing for Resource Reservations in Shared Environments. IEEE/ACM International Conference on Grid Computing, 2007.
- [13] P. Beckman, *et al.* SPRUCE: A System for Supporting Urgent High-Performance Computing. Pg 295-316 in Grid-Based Problem Solving Environments by Springer Press, 2007.
- [14] R. Wolski, *et al.* The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Journal of Future Generation Computing Systems, 15(5-6), pp. 757-768, October, 1999.
- [15] J. Brevik, *et al.* Predicting Bounds on Queuing Delay for Batch-scheduled Parallel Machines. Proceedings of ACM Principles and Practices of Parallel Programming (PPoPP), March 2006.
- [16] W. Smith, *et al.* Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. Proceedings of the Job Scheduling Strategies for Parallel Processing, 1999.
- [17] Windows Server 2008 Hyper-V. www.microsoft.com/windowsserver2008/en/us/hyperv.aspx
- [18] P. Barham, *et al.* Xen and the Art of Virtualization. ACM Symposium on Operating Systems Principles, 2003.
- [19] B. Sotomayor, *et al.* Combining Batch Execution and Leasing Using Virtual Machines. ACM International Symposium on High Performance Distributed Computing (HPDC), June 2008.
- [20] D. Irwin, *et al.* Sharing Networked Resources with Brokered Leases. USENIX Technical Conference, June 2006.
- [21] I. Lee, *et al.* Handbook of Real-Time and Embedded Systems. CRC Press, 2007.
- [22] J. Stankovic, *et al.* Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms. Kluwer Academic Publishers, 1999.
- [23] J. Stankovic, *et al.* The Spring kernel: a new paradigm for real-time operating systems. ACM SIGOPS Operating Systems Review, vol 23, Issue 3, July 1989.
- [24] C. Lu, *et al.* Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms, Real-Time Systems, Special Issue on Control-theoretical Approaches to Real-Time Computing, 23(1/2): 85-126, July/September 2002.
- [25] J.L. Hellerstein, *et al.* Feedback Control of Computing Systems. Wiley-IEEE Press, August 2004.
- [26] K. J. Astrom, B. Wittenmark. Adaptive Control. Addison-Wesley, 1994.
- [27] C. Lu, *et al.* "Feedback Utilization Control in Distributed Real-Time Systems with End-to-End Tasks," IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 6, June 2005.
- [28] C. Lu, *et al.* Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. IEEE Transactions on Parallel and Distributed Systems, September 2006.
- [29] M. Karlsson, *et al.* Triage: Performance differentiation for storage systems using adaptive control. ACM Transactions on Storage, volume 1, issue 4, November 2005.
- [30] Y. Zhang, *et al.* Friendly Virtual Machines Leveraging a Feedback-Control Model for Application Adaptation. ACM/Usenix International Conference on Virtual Execution Environments (VEE), 2005.
- [31] P. Padala, *et al.* Adaptive control of virtualized resources in utility computing environments. EuroSys 2007: 289-302
- [32] J. Xu, *et al.* On the Use of Fuzzy Modeling in Virtualized Data Center Management. International Conference on Autonomic Computing (ICAC), 2007.
- [33] S-M. Park, M. Humphrey. Feedback-Controlled Resource Sharing for Predictable eScience. IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC08), Austin, Texas, Nov 2008.
- [34] S-M. Park, M. Humphrey. Self-Tuning Virtual Machines for Predictable eScience. IEEE International Symposium on Cluster Computing and the Grid (CCGRID'09), Shanghai, China, May 2009.
- [35] A. Iosup, *et al.* The Characteristics and Performance of Groups of Jobs in Grids. International European Conference on Parallel and Distributed Computing (EuroPar), 2007.
- [36] S. Browne, *et al.* A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. IEEE/ACM Conference on Supercomputing, 2000.
- [37] A. Menon, *et al.* Diagnosing Performance Overheads in the Xen Virtual Machine Environment. ACM/Usenix Conference on Virtual Execution Environments, Chicago, Illinois, 2005.
- [38] J.S. Milton, J.C. Arnold. Introduction to Probability and Statistics. 4th ed. McGrawHill.
- [39] Huang, *et al.* A case for high performance computing with virtual machines. International Conference on Supercomputing, Cairns, Queensland, Australia, 2006.
- [40] Youseff, *et al.* Paravirtualization For HPC Systems. Proceedings of Workshop on XEN in HPC Cluster and Grid Computing Environments (XHPC), Dec 2006.

Sang-Min Park is a Ph.D. candidate in the Department of Computer Science at the University of Virginia. He received B.S. in Computer Science in 2002 from the Ajou University, South Korea, and Master of Computer Science in 2006 from the University of Virginia. His research interests include virtualization, parallel, distributed systems and control-theory applications.

Marty A. Humphrey is an Associate Professor in the Department of Computer Science at the University of Virginia. He received his PhD in Computer Science from the University of Massachusetts.