

# Feedback-Controlled Resource Sharing for Predictable eScience

Sang-Min Park and Marty Humphrey  
Department of Computer Science  
University of Virginia  
Charlottesville, VA 22904  
{sp2kn | humphrey}@cs.virginia.edu

**Abstract**— The emerging class of adaptive, real-time, data-driven applications are a significant problem for today’s HPC systems. In general, it is extremely difficult for queuing-system-controlled HPC resources to make and guarantee a tightly-bounded prediction regarding the time at which a newly-submitted application will execute. While a reservation-based approach partially addresses the problem, it can create severe resource under-utilization (unused reservations, necessary scheduled idle slots, underutilized reservations, etc.) that resource providers are eager to avoid. In contrast, this paper presents a fundamentally different approach to guarantee predictable execution. By creating a virtualized application layer called the performance container, and opportunistically multiplexing concurrent performance containers through the application of formal feedback control theory, we regulate the 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 disturbances. Our evaluation using two widely-used applications, WRF and BLAST, on an 8-core server show our approach is predictable and meets deadlines with 3.4 % of errors on average while achieving high overall utilization.

## I. INTRODUCTION

Increasingly, HPC applications are requiring predictable execution, both in regard to their start time as well as their duration. For example, many new applications are being developed that are required to adaptively process data from environmental sensors in real-time so that discoveries/predictions such as tornado warnings can be produced in time. Adaptive HPC applications have a potentially complex set of discrete processing points, from sensor measurement to visualization, to respond to real-time events as the needs arise. While parts of the application chain have been successfully transitioned toward this goal, such as by deploying configurable sensors in the field, one fundamental challenge that continues is the transitioning of the HPC compute infrastructure to an adaptive one that guarantees predictable job execution. The batch-mode job processing that continues to prevail on most supercomputers (i.e., the queuing systems) cannot generally respond to adaptive computing demand in real-time. The excellent recent work on predicting execution start times in certain queuing systems ([1]) only partially addresses this problem, and cannot provide as tight a bound as some applications need. In addition, [1] is orthogonal

to the need of an application to begin executing, predictably, immediately.

Historically, if an application needs to execute according to an explicit or implied deadline, resource capacity was explicitly reserved with the owner of the supercomputer on an exclusive basis. This approach could achieve the goal but comes at too high a cost and is not generally applicable. Reservations could be made and not actually used for whatever reason, which at worst can result in an idle, expensive resource. Scheduling a particular reservation might also require planning explicit idle times immediately prior to the planned reservation, if for example applications running prior to the planned reservation do not have support for checkpointing. A planned reservation could be actually used, but perhaps at significantly less consumption than requested (sometimes a worst-case reservation is made). All of these situations result in utilization far less than the overall capacity.

This paper presents a new approach, called the performance container and the compute throttling framework, which addresses the problem of HPC application predictability in fundamentally different ways. Unlike the existing approaches that attempt to control wait time of queued jobs, we control the job’s 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 systems resources to regulate the job’s progress. Throttling applies formal control theory to design and implement a feedback controller that enables deadline-based job execution. Our closed-loop control ensures that jobs meet deadlines even when significant “disturbances” might otherwise affect their progress. In contrast to existing approaches that require exclusive access to compute resources, our virtualized approach allows non-deadline-based applications to share the physical resources of the compute platform with one or more performance containers – such non-deadline-based activity is directly accounted for as just another “disturbance” in our control-theoretic approach.

Evaluation results with two well-known applications, Weather Research and Forecasting model (WRF) [2], and BLAST [3] indicate that, across a variety of representative

situations, the job's deadlines are met with 3.4 % errors on average. Furthermore, compared to a priority enforcement scheme based on batch-mode operation, compute throttling achieves nine times higher accuracy on deadline-guarantee. Overall, we believe performance containers and its associated compute throttling is an important first step toward support for the emerging class of adaptive, real-time, data-driven applications, while still achieving high overall resource utilization.

The rest of this paper is organized as follows: In section 2, we present related work. Section 3 defines the problem, requirements for a successful approach, and presents the overview of performance containers and compute throttling. Section 4 and 5 presents the details of the control-theoretic approach. The evaluation is presented in Section 6, followed by the conclusion in Section 7.

## II. RELATED WORK

From the early research on the Grid as a shared, large-scale infrastructure, immediate or advance reservation has been the most widely-discussed mechanism to make the infrastructure predictable. In General-purpose Architecture for Reservation and Allocation (GARA) [4], distributed computational and communication resources are assumed to provide interfaces to reserve them immediately or for some future time span in advance. This allows one to circumvent unpredictably long wait time in resource's queue and make resources available as soon as they are needed. Although modern queue managers such as PBS and LSF implement reservations and there were some follow-up works in the literature [5][6], the reservation-based approach is not widely accepted in Grid, in part because of its complexity and because it can result in resource under-utilization. Resource under-utilization in turn leads to increased wait time of best-effort jobs. Smith et al. reports that if 20% of jobs are reservations, wait-time of best-effort jobs are increased by 37 % [7]. People attempted to address the problem via enforcing penalties to "no-show" cases [8] or creating policy system with human in-the-loop so as to limit the use of reservation only for exceptionally urgent events [9]. Unlike a reservation, which requires exclusive allocation of resources, our approach of the performance container and compute throttling ensures predictable execution while unrelated activities concurrently share the resources. This reduces a resource provider's burden as it allows best-effort jobs to run concurrently with deadline-based jobs on the shared resources. We effectively transform batch-mode resources to time-shared resources with deadline control.

Another widely discussed mechanism for building predictable infrastructure is to periodically monitor resource's state and predict future state using statistical method. Network Weather Service (NWS) [10] measures end-to-end bandwidth and host CPU's availability, and predicts future state with statistical guarantee on its accuracy. The work has been extended to predict queue wait times of large supercomputing centers [1]. The fundamental limitation of the monitoring-prediction approach is that it alone does not allow users to change resource's behavior according to their needs. Though such a prediction can help users adapt to resources' changing

states, it does not help if users have a deadline requirement that is more urgent than possible with the current/future states. Our mechanism allows users to set application's deadline directly according to their needs.

Our work relies on O/S-level virtual machines to isolate performances among concurrent applications, and dynamically adjusts application performance to meet deadlines. Modern virtualization technology such as Windows Hyper-V [11] and Xen [12] recently gained huge attention in business computing environment for its server consolidation effect. The VMs also have been actively discussed in the Grid community as it has the potential to provide (almost) ideal security isolation along with other benefits including O/S customization and portability [13][14][15][16]. People advocate that VM technology can be used to dynamically deploy virtual runtime environments in physical hardware maintained in distributed computing centers, separating a resource provider's management concern from the client's resource requirement [14][15]. Thanks to the suspension/resumption capability provided by virtualization, leasing-based resource provisioning has been actively discussed in the HPC community recently [17][18]. Although the virtualized leasing can potentially solve the underutilization problem of reservation, it is not yet clear how closely the leasing-a form of "space sharing"- can meet the explicit application deadlines. This paper presents a case for the benefit of "controlled time-sharing" to meeting deadlines precisely. To the best of our knowledge, this is the first research on the use of performance isolation and online resource reconfiguration provided by virtualization for building predictable HPC infrastructure.

Control theory is one of the most widely used mathematical frameworks to control the behavior of linear dynamic systems in engineering and mathematics (e.g., cruise control) [19][20]. A closed-loop controller uses observations from the target system to determine the next value of target system's actuation variable(s), thereby changing the target system's output value in the subsequent cycles. Since the loop is closed, the controller keeps changing the system's input variable until the system reaches to the desired state, even in the presence of unexpected disturbances. Feedback control has been successfully applied to various applications of computing systems such as QoS control of e-mail [19] and web servers [21], real-time scheduling [22], and enterprise storage systems [23]. We use control theory to design closed-loop control logic that regulates application's progress in order to meet application's deadline. The formal theory allow us to design the closed loop control systematically and accurately even in the presence of unexpected, non-controllable disturbances such as disk I/O. To the best of our knowledge, this is the first research to apply control theory for predictable execution of HPC applications.

## III. PREDICTABLE ESCIENCE – REQUIREMENTS AND OUR SOLUTION

Recently adaptive, dynamic data-driven applications have emerged as an important class of HPC applications. One pioneering project is Linked Environments for Atmospheric Discovery (LEAD) [24]. Traditionally, weather forecasting has

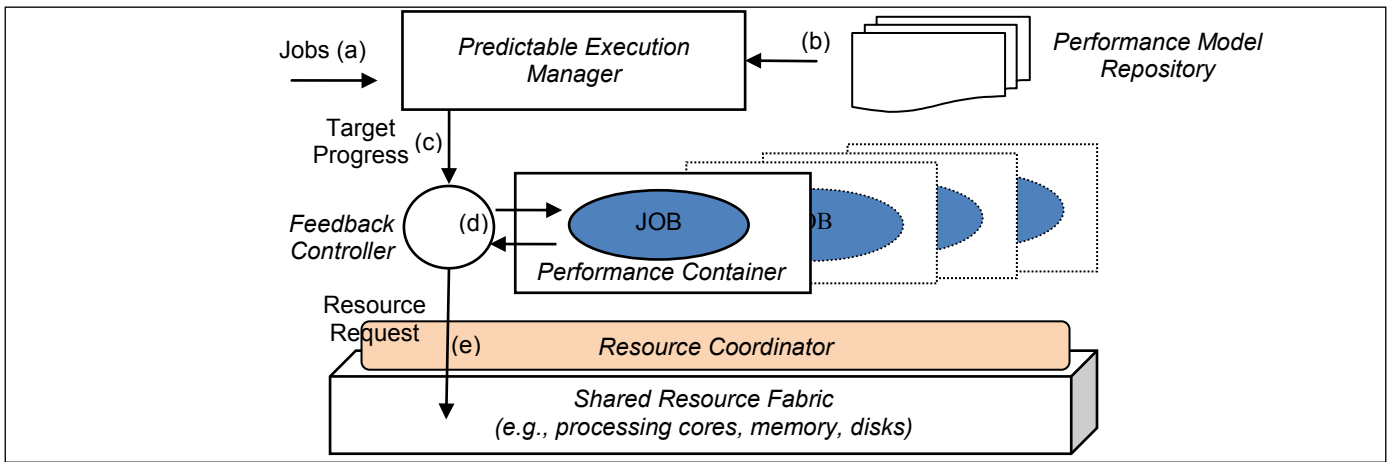


Figure 1: Performance Container Framework

been carried out by a prefixed chain of application component, from measurement to visualization. This static chain has been operated in essentially “batch-mode”: measurement, data collection, processing, and visualization are all carried out according to the pre-defined schedule with a fixed refresh cycle. The goal of LEAD is to transform the “batch-mode” flow to adaptive one such that the forecasting infrastructure responds to real-time, mesoscale weather events as “the needs arise”. As an illustration, during the normal period the infrastructure operates at a usual schedule (e.g., a radar scans large regions at relatively longer interval and workflow processes data at low processing rate). However, as soon as an interesting weather event, such as the one that could be the precursor of tornado, is sensed in a region, the infrastructure would like to adapt to the events, for example by fixing the radar on the small region, scanning it at fine interval and processing the event data at higher rate for more accurate prediction. This adaptive application scenario is not particular to only atmospheric researches. As applications are increasingly integrated with real-time, environmental sensors, similar use cases are found as well as in costal ocean observing and prediction [25], medical image modeling [26], and earthquake analysis [27].

The fundamental challenge that these adaptive applications are facing is that the current computing infrastructure cannot support adaptive data processing at the level they desire. The ideal solution in fact requires a predictable computing infrastructure, which means not only urgent processing but also the ability to opportunistically slow down certain processing if necessary to achieve performance goals of time-dependent other computations. Predictable application execution has the following key requirements:

- 1) *The ability to change completion time of jobs at fine-granularity*
- 2) *The ability to cope with unanticipated disturbances that affect a time-dependent job's performance*

The existing approaches [4][5][6][7][8][9] have attempted to address the first requirement by controlling the wait time of jobs in a resource's queue. However, the approach can suffer from potentially severe resource underutilization [7][8]. Furthermore, wait time could not be controlled at the desired

granularity since job's waiting time depends on the jobs currently holding resource, and most modern O/S lacks support of checkpoint/resumption facility for running jobs [9]. The second requirement is important as well since job's completion time is not only affected by provisioning (that is presumably controllable) computing cycles, but also other, hard-to-control elements such as I/O latency and potentially inaccurate performance models. Existing approaches often ignore the disturbances and allocate resources statically, hence resulting in failure to meet deadlines.

We address the two requirements in the framework called performance containers and compute throttling. We address the first requirement via a novel use of VM that provides fine-grained control over CPU time allocation. Unlike the previous works that mostly focused on job's waiting time, we control job's running time by creating virtualized resources with fine-grained CPU allocation. This virtualized resources throttle up/down the job's progress as the needs change dynamically. We address the second requirement through adaptive resource provisioning using the formal, feedback control theory. The feedback-controlled resource provisioning guarantees that application's progress sustains at the desired level such that application completes within deadlines, yet no earlier than it should be. The Performance Container framework is illustrated in Figure 1.

When new job arrives (a) and if it requires execution before a particular deadline, the Predictable Execution Manager consults the performance model repository (b) to determine the set of “milestones” of the job, given the attributes of the job such as job parameters and size of input data. The milestone is a quantitative metric that determines how much computation must be performed until the job terminates. Another metric, called “progress”, dictates how fast the job performs computation during a fixed interval. As a concrete example, the milestone can be the number of total floating point (FP) instructions that must be executed for the job to terminate, and the progress can be the number of FP instructions per second. Note that by accumulating the progress, we can estimate at which point of milestone the job currently run. The deadline has relationship with milestone and progress as dictated by the following simple equation:

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

Thus, once we know milestone and deadline of the requested job, we expect that the job will meet the deadline if the job sustains at the desired progress. Note that we do not directly address on the issue of how deadline is determined in this paper; we rather focus on the mechanism that control the resource’s behavior to meet the deadline after the deadline is established. “Call admission” is performed based on currently-available capacity by the Predictable Execution Manager. The job with target progress (derived using equation (1)) is then submitted to Performance Container (c) that runs the job on the resources that it shares with other containers. The performance container is under the control of the feedback controller (d), which ensures that progress of the job running inside a performance container sustains the target progress. It achieves the goal by allocating sufficient amount of resources from the resource fabric. The resource fabric consists of processing cores, memory partitions, I/O channels, and provides performance containers with aggregate pool of underlying resources that can be allocated and released at job’s run time. Since multiple performance containers share the single resource pool and some of them dynamically allocate and release resources as feedback controller acts on the job, there can be conflicts when multiple containers allocate resources simultaneously. It is the resource coordinator’s role to balance the resource allocation to concurrent containers such that the containers running deadline-guaranteed jobs are allocated sufficient amount of resources to increase the rate of progress (“throttle up”) as necessary, while best-effort jobs are allocated fair amount of resources by taking idle resources that are not used by deadline-guaranteed jobs. Section 4 presents the feedback-controlled performance container in more detail, followed by presentation of the predictable execution manager in Section 5.

#### IV. FEEDBACK-CONTROLLED APPLICATION PROGRESS

Even if we have correct milestone and there is a model that relates resource allocation to progress level, fixing the level of resource provision at one presumably-correct value at modeling time can result in failure to meet the deadline in real execution due to unforeseen disturbances, which we define as any unanticipated source of influence that affects job’s progress at run time. Examples of disturbance include intensive I/O, inaccurate performance model, changes in resource configuration, and increased network processing load. We use adaptive control to dynamically change provisioned resources if the actual, measured progress is below or over the desired value. Feedback control theory is a mathematical framework that can be used to design adaptive controller. Since the loop is closed, the control errors introduced by disturbances are eventually removed, keeping the target system, the application’s progress, at the desired state. The formal theory allows us to design the adaptive control via analysis of the system’s behavior before the system is actually implemented and deployed. In this section, we present the control-theoretic methodologies for designing a feedback-loop for controlling job’s progress.

##### A. Ensuring Deadline-based Application Progress: Sensing and Actuation

We believe that many applications can provide application-specific descriptions of behavior in the form equivalent to a collection of arbitrary (application-milestone, percentage-of-application-completed) pairs. Our feedback controller would thus invoke such a supplied “sensing” capability periodically in order to measure and actuate behavior (e.g., grant more resources to the computation in order to ensure completion on-time).

Our approach also allows and supports such measurement of application progress based on source code instrumentation. To aid this, we define a sensor library that users can embed into application codes. The sensor library is currently implemented as a proof-of-concept as a simple application-specific counter strategically placed in the critical control-flow of the application (such as in “hot spots” of application codes). This metric is exported and made available to the control system (e.g., via shared memory, the file system, etc.) For example, in the applications used to evaluate this approach (BLAST and WRF), it was sufficient to insert the sensing calls into three and one source code locations, respectively. An alternative way to implement the sensing function, especially when source code is not available for target application, is to profile hardware performance counter using the tools such as PAPI [28] and Oprofile/Xenoprof [29]. We plan to pursue this approach in the future.

We use virtualization technology as an abstraction to implement performance container. Both Hyper-V and Xen implement credit-based CPU scheduling algorithms. The VM’s CPU scheduling allows one to control CPU usage per VM accurately and at fine granularity. This, in effect, creates a virtualized resource with configurable clock speed for each VM. Hyper-V and Xen similarly expose three interfaces regarding CPU allocation: Cap, Reserve, and Relative Weight. The cap and reserve sets upper/lower limits of CPU time that VMs can consume, and the relative weight implements proportional sharing among VMs. The interfaces can be configured dynamically while a VM is active. We implement performance container using Hyper-V, and use its scheduling cap interface, via a library call, to control a job’s progress.

##### B. Mapping Resource Consumption to Application Progress: System Identification

Given the target progress capability as described above, the feedback controller’s goal is to make sure that job’s measured sensing rate sustains at the target rate. The controller regulates the sensing rate through the actuation mechanism, Hyper-V’s scheduling cap. This requires the controller know the relationship between the actuation value (i.e., CPU cap) and the sensing rate, so as to choose the right actuation value that drives the job’s progress to the target level. We use a dynamic model that describes the mathematical relationship between the control input and the output of the system using the linear difference equation. We use system identification [19], the standard technique to establish a linear model in control theory, to derive linear equation that model the relationship. The generalized form of model is as follows:

$$S(k) = \sum_{i=1}^n a_i S(k-i) + \sum_{j=1}^m b_j C(k-j) \quad (2)$$

In the model,  $S(k)$  and  $C(k)$  represent the sensing rate and Hyper-V's schedule cap at time  $k$ , respectively. This model tells that the measurement at time  $k$  is affected by the measurements at time  $(k-1)$ ,  $(k-2)$ , ...,  $(k-n)$ , and the actuation value at time  $(k-1)$ ,  $(k-2)$ , ...,  $(k-m)$ . The coefficients,  $a_i$ ,  $b_j$  describe how the current measurement is determined by the previous measurements and the actuation. Note the previous measurement affects the current measurement since the changes in actuation value at previous cycle may not be fully reflected to the current measurement. As an illustration, even if the scheduling cap is raised at time  $(k-1)$ , the job's progress may not be increased at time  $k$  if the job is idling on I/O requests during  $\{k, k-1\}$ . From the general model expressed in (2), 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 over-fitting, representing that a model cannot be used in variations of contexts (e.g., hardware upgrades). Thus, we choose the first-order model as represented in the form:

$$S(k) = aS(k-1) + bC(k-1) \quad (3)$$

System identification is a black-box modeling approach to derive the model parameter  $a$ ,  $b$  of equation (3). We simulate the dynamics of the target system, Hyper-V running the job, with varying actuation values and use least square estimation [30][19] to determine the model parameters  $a$ ,  $b$ . We denote the measured and predicted value of  $S$  at time  $k$  by  $\tilde{S}(k)$ ,  $\hat{S}(k)$ , respectively. The predicted value  $\hat{S}(k)$  can be expressed as:

$$\hat{S}(k) = aS(k-1) + bC(k-1) \quad (4)$$

The  $k$ -th residual, also known as prediction error, is  $e(k) = \tilde{S}(k) - \hat{S}(k)$ . 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 [30]). Figure 2 is a graphical representation of system identification for BLAST. We stimulate the system (job's progress) with varying actuation values (scheduling cap), following a discrete sine wave as illustrated in Figure 2(a), and the real measurement follows the similar trends as shown in Figure 2(b). We used 500 ms as a sampling time for the system identification, and the same interval is used when controlling job's progress.

The model parameter that we obtained from BLAST and WRF, with Hyper-V as the actuation system, is as follows:

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

Note that BLAST exhibits 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.

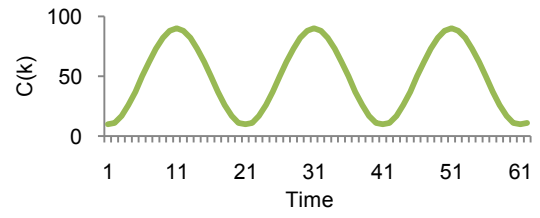


Figure 2(a): Actuation Values

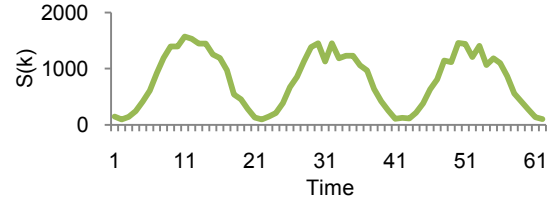


Figure 2(b): Sensor measurements from BLAST

In general, this formal process of mapping the potential results of changing resource availability/consumption to the rate of application-specific progress is a necessary aspect of our approach but need be performed only once, offline. The system identification is not necessarily performed for all different resource configurations (e.g., hardware upgrades) since the feedback controller treats the modeling error as one of disturbances and adapts to it. We note in passing that applications may consist of multiple phases in which the sensors are called at different granularity (e.g., I/O and compute intensive phase). In this case, the system identification may result in multiple models. Since a single model is sufficient for BLAST and WRF, we do not discuss on the issue in this paper. Lu et al. discuss on the possible solutions in [31].

### C. Controller Design

We use the Proportional-Integral (PI) control, the most widely-adopted control logic in mechanical and software systems. Figure 3 illustrates the closed loop with PI controller and the target system.

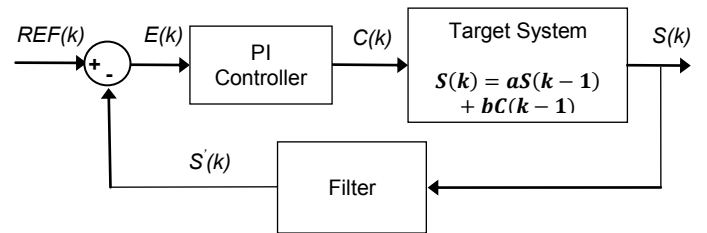


Figure 3: Closed Loop with PI-Controller

The goal of controller is to ensure that the measured progress from the target system (i.e., the job's progress), expressed as a linear difference equation, is equal to the reference progress that it has received from the external commander (human or software entity). The Predictable Execution Manager (as illustrated in Figure 1) sets the reference, which has been derived from equation (1). 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:

$$C(k) = C(k-1) + (K_p + K_I)E(k) - K_p E(k-1) \quad (5)$$

The equations (2)-(5) are the time-domain representation of the system, and the classic control theory provides different, more convenient way to encode the values and to describe systems, called Z-transformations. Z-transform allows one to easily combine multiple components of system (e.g., controller, target system, filter) and describe integrated system as a unified equation. Due to lack of space, we do not provide the mathematical definition and the detailed properties of Z-transformations in this paper; the details can be found in control textbooks [19][20]. The target system, expressed in equation (3), and the PI controller (5) can be modeled as transfer functions in Z-domain as follows:

- $G(z) = \frac{S(z)}{C(z)} = \frac{b}{z-a}$ , where  $S(z)$ ,  $C(z)$  are z-transforms of  $S(k)$ ,  $C(k)$  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 (5) 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  $S(k)$ ,  $S'(k)$  as input and output of the filter, the filter has the form:

$$S'(k) = C \cdot S(k) + (1 - C)S(k) \quad (6)$$

In equation (6),  $C$  determines the desired level of smoothing and we found  $C=0.9$  results in stable system for both BLAST and WRF. We denote,  $H(z)$ , as the z-transform of filter (6). The closed-loop transfer function, representing the overall system in Figure 3, with reference as input variable and the measurement as output is constructed by integrating the controller, target system, and filter into a unified function:

$$FR(z) = \frac{S(z)}{REF(z)} = \frac{D(z)G(z)}{1+D(z)G(z)H(z)} \quad (7)$$

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

- 1) **Accuracy:** the controller must guarantee that the measurement converges to the reference, even when disturbances affect the target system
- 2) **Stability:** the controller must guarantee that output of the target system is bounded within small thresholds
- 3) **Settling time:** the controller must quickly settle to steady-state value (i.e., reference) in the event of reference changes and disturbances

The PI Control law ((5)) guarantees accuracy of the control loop since the equation uses previous error history and regulates the target system's actuation until the accumulated errors are eliminated. The final aspect of the the control design is thus to choose the right values for controller parameters,  $K_p$  and  $K_I$  in equation (5) that satisfies the stability and settling time requirements. The stability and settling time of the closed-loop system are determined by the poles and zeros of

the transfer function (7), which are in turn determined by values of  $K_p$  and  $K_I$ . In general, there is a trade-off between the stability and settling time of system. The faster response to disturbances or reference changes can result in unstable system. Thus, one must carefully choose the right parameters that satisfy both the stability and settling time requirements simultaneously. 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. For BLAST and WRF, desired values of  $K_p$  and  $K_I$  that we found using Root Locus method (using Matlab) are:

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

## V. APPLICATION MODELING

In Section 4, we presented the control-theoretic methodology to dynamically provision resources to performance containers such that a job's progress is sustained at the desired level. Our Predictable Execution Manager (as illustrated in Figure 1) determines the reference progress that is sent to the feedback controller by referring to the application model and the deadline for the requested job. The model determines the milestone of the requested job and the reference progress is derived using equation (1). The model uses application-specific information to estimate the milestone. We establish a simple linear model that relates application information such as file size, input parameters to total sensor count. As an illustration, for BLAST and WRF, we derive the total sensor count using the size of input protein query, or the requested period (e.g., hours/days) for which weather forecast model runs, respectively. After profiling, the model is constructed using the standard linear regression method. Figure 4 presents the profiling results for BLAST and WRF, and the derived model equation along with  $R^2$  values for the verification of model's correctness. While WRF shows a perfect fit to model, BLAST has a moderate fit. Since these results are intrinsic to the applications, we do not present more detailed analysis of the results. Note that the model established in this way is only relevant to an application's semantic, not the resources on which it runs. Thus, a model can be used for different resource configurations. We acknowledge the possibilities that an application's computation quantity is determined by more than one parameter. For example, WRF has another user-configurable parameter, called the *time\_step*, which determines the granularity of the forecast prediction over the period. It is part of our future work to address multi-parameter modeling issues.

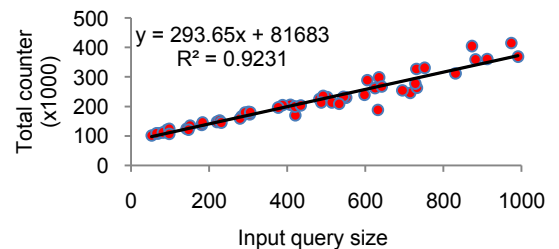


Figure 4 (a): Profiling results for BLAST

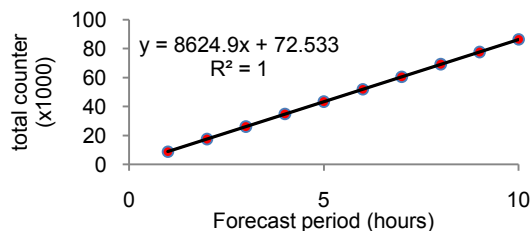


Figure 4 (b): Profiling results for WRF

## VI. EVALUATION

This section presents the evaluation of the Performance Container and Compute Throttling Framework in three ways. We first evaluate the correctness of feedback control in Section 6.1, and then present the correctness of deadline guarantee on low-end resource in Section 6.2. Section 6.3 present results from more complex, realistic use cases on high-end server (8-core system).

### A. Correctness of Feedback Control

The two applications by which we evaluate the framework are BLAST and WRF, the widely used applications in biology and atmospheric research, respectively. We chose the two applications since they represent different types of eScience applications. While WRF job is compute-intensive, running many hours with tens of MB of input/output/intermediary data, BLAST is both a compute- and data-intensive task that can scan more than 1GB sequence database over few minutes of running time. TABLE I and II summarize the characteristics of applications and hardware configuration for the evaluation, respectively. Note that we use a desktop system for evaluations in subsection A and B since it is sufficient to evaluate the correctness of feedback control. We use higher-end server (AMD 8-core) in subsection C to evaluate more complex use case, which argue is readily application to more traditional supercomputers.

TABLE I. APPLICATION CHARACTERISTICS

	<i>WRF</i>	<i>BLAST</i>
Running Time	Hours	Minutes
Data to Process	< 100 MB	≈ 2 GB (nr DB)

TABLE II. RESOURCE CONFIGURATION

CPU	Intel Dual Core 2 2.13 Ghz
MEMORY	3 GB
DISK – I/O	10,000 RPM SATA / PCI-E Bus
O/S	Windows Server 2008 with Hyper-V

Figure 5 illustrates how the choice of controller parameter determines the behavior of BLAST (we omit the results from WRF due to lack of space; similar results are found in WRF as well). We run the feedback-controlled BLAST as reference progress is changed at 70 and 140th seconds. The straight, dotted line represents the reference progress, which is the

target that the feedback control 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 4 (Matlab is used to produce the simulated results). The solid line represents the result that we obtain from real execution. 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 experiment exhibit similar trends to the simulation. This indicates that the controller design that we present in Section 4 is effective in predicting the behavior of real system, and therefore we can choose correct control parameters with high confidence.

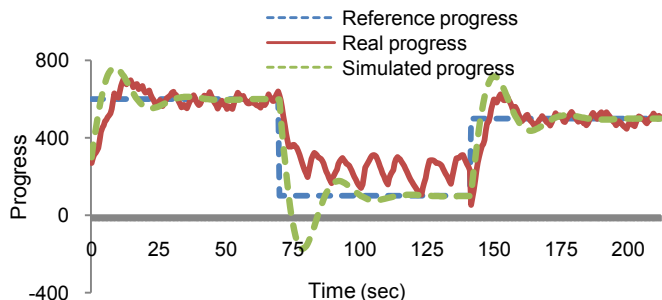


Figure 5(a): Aggressive Controller ( $K_p=0.012$ ,  $K_f=0.018$ )

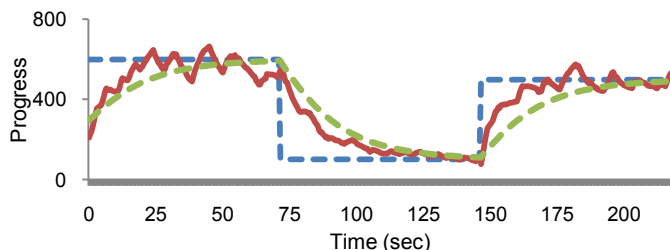


Figure 5(b): Slow Controller ( $K_p=0.0016$ ,  $K_f=0.0024$ )

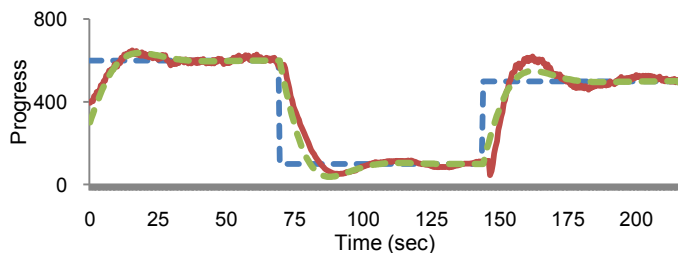


Figure 5(c): Good Controller ( $K_p=0.005$ ,  $K_f=0.007$ )

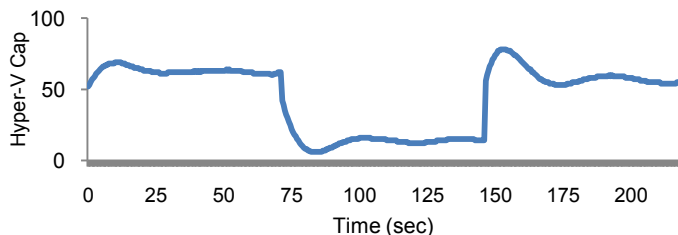


Figure 5(d): Changes in Hyper-V's Scheduling Cap

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 the reference 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 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 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 (3) with BLAST’s model parameter. This corresponds to the *open loop controller* that fixes Hyper-V’s scheduling cap at one value that is supposed to be correct at modeling time. Figure 6 (a) illustrate results when experiments run without disturbances. As we see in the figure, both the open-loop and feedback control tracks the reference progress equally well. However Fig 6 (b) shows that the open-loop fails to track the reference progress 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 Hyper-V cap when it sees errors while tracking. With open-loop control, failure in tracking the reference increases errors in meeting deadlines as it is presented in the following subsection.

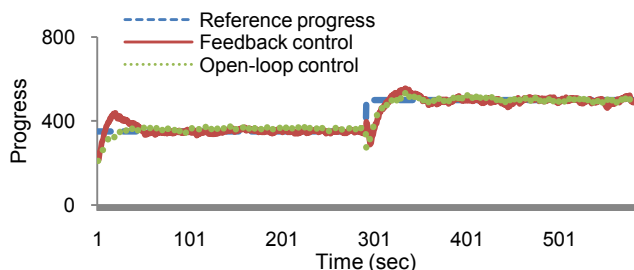


Figure 6(a): Control results without disturbances

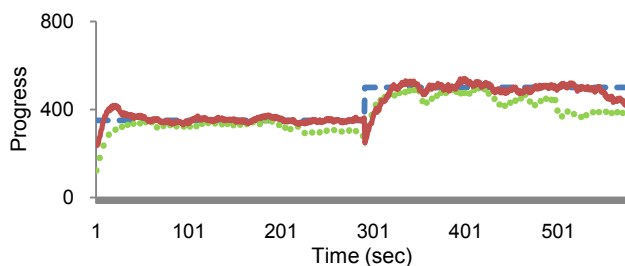


Figure 6(b): Control results with disturbances

### B. Meeting Application Deadline

We measured how closely the feedback-controlled performance container meets job deadlines. For BLAST, we run 100 randomly chosen queries with varying size (1-1000 bytes). Note that input query size is the factor that determines milestone of BLAST job. For WRF, we perform 4 runs with one forecasting period as input parameter. For each application, we run experiments with two deadlines: a faster deadline and a slow one. The faster deadline equals the estimated running time of job if 2/3 share of a CPU would be allocated on servicing the job. The slow deadline is similarly chosen with 1/3 of estimated CPU allocation. When the fast deadline is the reference, it takes approximately 6 hours to process 100 random BLAST queries and 1 hour for single run of WRF with the given forecasting period. To evaluate the framework under disturbances, we run two more VM instances on the same Hyper-V host with BLAST running as disturbance generator. Note that in a real scenario, the disturbance in fact corresponds to best-effort jobs concurrently running on the shared resource.

TABLE III presents the result for each application. In the table, *error* is defined as,  $abs(\frac{running\ time - deadline}{deadline}) \times 100$ , and used for evaluating the correctness of deadline guarantee. Note that we measure the absolute error since our goal is not only to meet deadline, but also to do it as close as possible. In the table, we also present the result when applications run with open-loop control. The numbers in parenthesis are what we obtained using open-loop control.

TABLE III. CORRECTNESS OF DEADLINE GUARANTEE

	<b>BLAST</b>		<b>WRF</b>	
	Fast Deadline	Slow Deadline	Fast Deadline	Slow Deadline
Average Error	7.7 (13.1) %	4.7 (16.4) %	1.1 (6.9) %	1.8 (2.6) %
Standard Deviation of Error	5.5 (10.2) %	3.9 (12.2) %	0.1 (1) %	0.05 (0.06) %

The result strongly indicates that feedback control achieves better accuracy than open-loop case, especially when application’s progress is highly influenced by disturbances (BLAST). The result also indicates that BLAST shows relatively high errors as compared to WRF. The main reason is due to high variability in predicting the application’s milestone

with respect to input query size (Fig 4(a)). If we assume BLAST accepts many input queries as a unit of computation and deadline-guarantee, the errors eventually average out. In other words, the late completion time at the  $i$ -th query is compensated by faster completion time at the  $(i+1)$ -th query. Our result indicates that if more than 10 queries are submitted to the BLAST server at once, the error diminishes to less than 1 %. However, open-loop control does not compensate errors since every execution exhibits the completion time later than deadline. The accumulated errors are more than 10 % if the unit of computation is more than 10 queries.

### C. Evaluation on High-end Server

This subsection presents the evaluation results for more complex, realistic use cases on 8-core server. The purpose of the experiment is to evaluate the correctness of the deadline guarantee when best-effort jobs run on the server concurrently. We run WRF as a use case application for the experiment. TABLE IV and V illustrate the server’s hardware specification and summary of experiments, respectively. We run 8 guest VMs as performance containers for best-effort jobs and additional 4 guest VMs as feedback controlled performance container for deadline-guaranteed jobs. Throughout the experiments, all 8 best-effort VMs continuously run best-effort jobs whose running time ranges from 10 to 30 minutes when the server is idle. Every 10 minutes, we submit a job that needs a deadline guarantee. The same deadline job, whose running time is approximately 10 minutes when server is idle, is submitted to the server throughout the experiment. The deadline for the job is randomly requested from the range {10-30} minutes. This experimental parameter is carefully chosen such that the sum of resources requested by deadline-guaranteed jobs do not exceed the server’s capacity at all time. In real system, simple policy such as limiting the number of deadline-guaranteed jobs equal to or less than the number of processors can ensure that deadline-guaranteed jobs do not compete each other.

TABLE IV. HIGH-END SERVER CONFIGURATION

CPU	2 Quad-core AMD Opteron - 1.7Ghz (8 cores in total)
RAM	8 GB (384 MB per guest VM)
DISK – I/O	10,000 RPM SATA / PCI-E Bus
O/S	Windows Server 2008 with Hyper-V

TABLE V. EXPERIMENTAL PARAMETERS

Application	WRF
# of Guest VMs	8 : best-effort job, 4 : deadline-guaranteed job
Duration	8 hours
Best-effort Job’s Execution Time	10-30 minutes / job
Requested deadline	10-30 minutes

We compare compute throttling with the two baseline cases which represent priority elevation schemes in batch mode resource sharing:

- **BASELINE 1 - BATCH-RANDOM:** For the deadline-requested job, we randomly choose one VM (out of 8 best-effort VMs) and run the deadline-guaranteed job after the job running on the chosen VM terminates.
- **BASELINE 2 - BATCH-BEST:** We assume that a job’s running time can be estimated accurately. For the deadline-requested job, we choose the best VM by which the sum of remaining time of running job and the running time of deadline-guaranteed job is close to the requested deadline.

Baseline 1 corresponds to the popular priority elevation scheme where high priority job is placed on the front of the resource’s queue, but does not suspend or halt the running job (e.g., TeraGrid’s Spruce [9]). Baseline 2 represents the best case that we can expect with batch-mode resource sharing where best-effort jobs cannot be suspended or halted in favor of deadline-guaranteed jobs. Unlike the two baseline cases, compute throttling immediately submits the deadline-requested job to one of four feedback controlled performance containers and run the job concurrently with eight best-effort jobs. For each, the experiment runs for 8 hours and 48 deadline-guaranteed jobs are processed.

The results of the experiments are presented in TABLE VI and Figure 7. TABLE VI shows the average error, as defined in the previous subsection, for the three methods. Figure 7 presents a scatter plot in which x-axis represent the guaranteed deadline and the y-axis corresponds to the real execution time of jobs. The results clearly indicate that compute throttling achieves much higher accuracy in deadline guarantee than the two priority elevation schemes based on batch mode resource sharing. Compared to the best possible scheme based on batch-mode resource sharing, compute throttling achieves about 9 times higher accuracy in deadline-guarantee. Figure 7 indicates that, under throttling, running time of all 48 jobs show no significant deviation from the guaranteed deadlines.

TABLE VI also present WRF’s forecast hours processed by best-effort jobs, as a metric to evaluate how each method affects the performance, in terms of throughput, of best-effort jobs. The compute throttling shows slightly less throughput than the other two methods (about 90% of other two). This is due to resource coordinator’s operation that allocates sufficient amount of resources to feedback-controlled performance containers such that the controller throttles up the job’s progress rapidly. Although, resource coordinator detects feedback controller’s resource release due to throttling down the job’s progress, and provides the released resources to the best-effort containers, in the long-term resource coordinator slightly over-provisions the resources to feedback-controlled containers for controller’s rapid operation, resulting in about 10% reduction in best-effort job’s throughput.

TABLE VI. COMPARISON OF THREE METHODS

	COMPUTE THROTTLING	BATCH BEST	BATCH RANDOM
Average Deadline-Guarantee Error	3.4 %	30.8 %	65.8 %
WRF Forecast Hours by Best-effort Jobs	1358 h	1470 h	1516 h

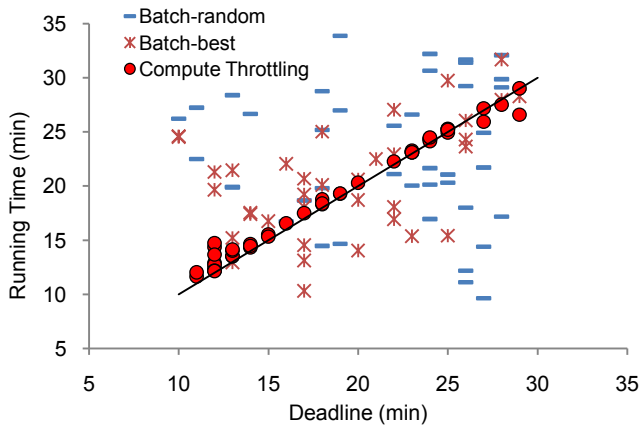


Figure 7: Scatter plot of deadline and real execution time

## VII. CONCLUSION AND FUTURE WORK

We present the performance container and compute throttling framework as a promising approach to build predictable HPC infrastructure. Via novel use of virtualization technology, we run jobs with fine-grained resource configuration during the job's runtime. We use formal control theory to build feedback controllers that meet a job's deadlines even in the presence of severe disturbances. Our evaluation results indicate that our framework achieves high accuracy in deadline-based processing for both desktop and high-end server system.

Our future work includes studying how to integrate this mechanism with more traditional queuing systems (we do *not* believe that this approach should supplant existing queuing systems, which have in general shown to be very effective for non-time-driven processing). We also plan to pursue the more direct integration of compute throttling with data throttling, which we have recently created for predictable data transfer over Internet [32]. We believe the combination of predictable compute execution and predictable data movement can create predictable end-to-end systems such as workflows and lead to a more predictable environment for adaptive, real-time, data-driven applications in general.

## REFERENCES

- [1] J. Brevik, D. Nurmi, R. Wolski. Predicting Bounds on Queuing Delay for Batch-scheduled Parallel Machines. Proceedings of ACM Principles and Practices of Parallel Programming (PPoPP), March 2006.
- [2] J. Michalakos, S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middleco, W. Skamarock. Development of a Next-generation Regional Weather Research and Forecast Model. ECMWF Workshop on the use of Parallel Processors in Meteorology, Reading, U.K., November 2000.
- [3] BLAST: Basic Local Alignment and Search Tool (<http://www.ncbi.nlm.nih.gov/blast/>)
- [4] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. International Workshop on Quality of Service, 1999.
- [5] R.J. Al-ali, K. Amin, G.V. Laszewski, O.F. Lana, D.W. Walker, M. Hategan, N. Zaluzec. Analysis and Provision of QoS for Distributed Grid Applications. Journal of Grid Computing, 2004.

- [6] S. McGough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, L. Young. Making the Grid Predictable through Reservations and Performance Modeling. The Computer Journal, 48(3):358--368, 2005.
- [7] W. Smith, I. Foster, V. Taylor. Scheduling with Advanced Reservations. IEEE International Parallel and Distributed Processing Symposium, 2000.
- [8] G. Singh, C. Kesselman, E. Deelman. Adaptive Pricing for Resource Reservations in Shared Environments. IEEE/ACM International Conference on Grid Computing, 2007.
- [9] P. Beckman, S. Nadella, N. Trebon, I. Beschastnikh. SPRUCE: A System for Supporting Urgent High-Performance Computing. Pg 295-316 in Grid-Based Problem Solving Environments by Springer Press, 2007.
- [10] M. Swamy and R. Wolski. Multivariate Resource Performance Forecasting in the Network Weather Service. Proceedings of Supercomputing, 2002.
- [11] Windows Server 2008 Hyper-V. <http://www.microsoft.com/windowsserver2008/en/us/virtualization-on-consolidation.aspx>
- [12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T.L. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. ACM Symposium on Operating Systems Principles, 2003.
- [13] R.J. Figueiredo, P. Dinda, J. Fortes. A Case for Grid Computing on Virtual Machines. International Conference on Distributed Computing Systems (ICDCS), April 2003.
- [14] T. Freeman, K. Keahey, I. Foster, A. Rana, B. Sotomayor, F. Wuerthwein. Division of Labor: Tools for Growth and Scalability of Grids. International Conference on Service Oriented Computing, Chicago, IL. December 2006.
- [15] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, X. Zhang. Virtual Clusters for Grid Communities. IEEE International Symposium on Cluster Computing and the Grid, May 2006.
- [16] L. Youseff, R. Wolski, B. Gorda, C. Krintz. Paravirtualization for HPC Systems. Workshop on XEN in HPC Cluster and Grid Computing Environments (XHPC), held in conjunction with ISPA, December 2006.
- [17] B. Sotomayor, K. Keahey, I. Foster. Combining Batch Execution and Leasing Using Virtual Machines. ACM International Symposium on High Performance Distributed Computing (HPDC), June 2008.
- [18] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, K. Yocum. Sharing Networked Resources with Brokered Leases. USENIX Technical Conference, June 2006.
- [19] J.L. Hellerstein, Y. Diao, S. Parekh, D.M. Tilbury. Feedback Control of Computing Systems. Wiley-IEEE Press, August 2004.
- [20] G.F. Franklin, J.D. Powell, M. Workman. Digital Control of Dynamic Systems. Addison Wesley, 1998.
- [21] C. Lu, Y. Lu, T.F. Abdelzaher, J.A. Stankovic, S.H. Son. Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. IEEE Transactions on Parallel and Distributed Systems, 17(9): 1014-1027, September 2006.
- [22] C. Lu, J.A. Stankovic, G. Tao, S.H. Son. 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.
- [23] M. Karlsson, C. Karamanolis, X. Zhu. Triage: Performance Differentiation for Storage Systems Using Adaptive Control. ACM Transactions on Storage, 2005.
- [24] B. Plale, D. Gannon, D. Reed, S. Graves, K. Droegemeier, B. Wilhelmson, M. Ramamurthy. Towards Dynamically Adaptive Weather Analysis and Forecasting in LEAD. ICCS workshop on Dynamic Data Driven Applications, Atlanta, Georgia, May 2005.

- [25] SURF Coastal Ocean Observing and Prediction (SCOOP): <http://www.scoop.lsu.edu/gridsphere/gridsphere>
- [26] Grid Enabled Neurosurgical Imaging Using Simulation: <http://wiki.realitygrid.org/wiki/GENIUS>
- [27] ShakeMovie: Caltech's Near Real Time Simulation of Southern California Seismic Events Portal. <http://shakemovie.caltech.edu/>
- [28] S. Browne, J. Dongarra, N. Garner, K. London, P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. IEEE/ACM Conference on Supercomputing, vol., no., pp. 42-42, 04-10 Nov. 2000.
- [29] A. Menon, J.R. Santos, Y. Turner, G. (John) Janakiraman, W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment, ACM/Usenix Conference on Virtual Execution Environments (VEE'05), Chicago, Illinois, 2005.
- [30] J.S. Milton, J.C. Arnold. Introduction to Probability and Statistics. 4th ed. McGrawHill.
- [31] C. Lu and D. A. Reed. Compact Application Signatures for Parallel and Distributed Scientific Codes. IEEE/ACM Supercomputing, 2000.
- [32] S-M. Park and M. Humphrey. Data Throttling for Data-Intensive Workflows. IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008). April 14-18, 2008.