Preliminary version. Final version appears in *Proceedings of 9th IEEE International Symposium on Cluster Computing and the Grid (ccGrid 2009),* May 18-21, 2009, Shanghai, China.

Self-Tuning Virtual Machines 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*— Unpredictable access to batch-mode HPC resources is a significant problem for emerging dynamic data-driven applications. Although efforts such as reservation or queue-time prediction have attempted to partially address this problem, the approaches strictly based on space-sharing impose fundamental limits on real-time predictability. In contrast, our earlier work investigated the use of feedback-controlled virtual machines (VMs), a time-sharing approach, to deliver predictable execution. However, our earlier work did not fully address usability and implementation efficiency. This paper presents an online, software-only version of feedback controlled VM, called self-tuning VM, which we argue is a practical approach for predictable HPC infrastructure. Our evaluation using five widely-used applications show our approach is both predictable and practical: by simply running time-dependent jobs with our tool, we meet a job's deadline typically within 3% errors, and within 8% errors for the more challenging applications.**

I. INTRODUCTION

Many pioneering projects including real-time mesoscale weather prediction [1], coastal hazard prediction [2], and patient-specific medical modeling [3] have started to explore opportunities and challenges that arise when scientific modeling is used to process environmental, real-time events. This emerging class of HPC jobs must produce results within explicit, possibly evolving, deadlines due to dependence on real-time data. The most difficult challenge today for such applications is that HPC infrastructures are typically operated in shared batch-mode and do not provide predictability both in regard to an HPC job's start time as well as its duration. Most existing research in this area thus attempts to eliminate a job's wait time via advance reservation [4][5][6][7], despite a potentially severe resource underutilization [7]. Moreover, a reservation requires strict planning that can involve time-consuming interactions between users and resource providers (e.g., TeraGrid requires reservations be made at least one week in advance). The sporadic nature of dynamic events may not permit such planning.

Our earlier results [8] introduced a fundamentally different approach to solve HPC unpredictability. In our Compute Throttling Framework, instead of attempting to achieve predictability by controlling a HPC job's wait time and granting exclusive access to a resource, our mechanism controls a job's running time by hosting jobs in virtualized resources, called performance containers, and "throttling" up/down the job's access to resources. We use a feedback controller to dynamically supply/remove system resources to the container(s). We showed that we are able to achieve predictable run-time performance, without requiring exclusive access to resources, and while still being reactive to unexpected events (e.g., new job arrivals, within limits). However, the significant limitation of [8] is that arguably only experts in control theory were realistic candidates for using our system. For example, our run-time system required a broad, quantitative understanding of a target application's behavior in a variety of situations in order to regulate the application progress dynamically. Sophisticated knowledge of control theory was necessary to determine the feedback controller parameters through a manual modeling process (e.g., Matlab).

The research reported in this paper significantly improves the usability of our control theoretic approach while retaining good controller performance that was the result of comprehensive manual modeling by an expert. We achieve usability by creating a self-tuning VM that performs application modeling, controller design, and control, all at runtime with no manual tuning by users. In other words, our goal is to essentially take an off-line and frequently tedious design process and automate it and thus turn it into an on-line process without human intervention. The heuristics we embed in our on-line mechanism to design the feedback controller achieve high performance in terms of controller design metrics (e.g., steady-state, transient behaviors) while attaining good algorithm efficiency. The experimental evaluations across five widely used HPC applications on an 8-core server confirm the viability of our approach: without any tuning effort, we meet a job's deadline with less than 3 % errors for ADCIRC [9], OpenLB [10], WRF [11], and less than 8 % errors for the more challenging BLAST [12] and Montage [13]. Overall, we believe the research reported in this paper is a practical strategy toward building predictable, usable, and cost-effective HPC infrastructure.

The rest of this paper is organized as follows: In section 2, we present related work. Section 3 defines the problem and presents the brief overview of our solution. Section 4 presents our approach for self-tuning control in detail including model estimation and controller design heuristics. The experimental evaluations are presented in Section 5 and we conclude in Section 6.

II. RELATED WORK

Foster et al. presents General-purpose Architecture for Reservation and Allocation (GARA) [4] in which distributed compute and communication resources provide a reservation capability immediately or for some future time span. Although reservation has been implemented in modern queue managers such as PBS and LSF (with additional research, e.g.,[5][6][7]) reservation has not been widely accepted by resource providers, in part because of its managerial complexity and because it can result in severe resource underutilization [6][7]. While enforcing penalties to "no-show" cases [6], or putting humans in charge of authorizing reservations [14] might eventually solve some part of its problems, reservation will still require strict planning which would not be viable to dynamic data driven applications. We believe this is the fundamental limitation of space-sharing that makes it extremely difficult to satisfy real-time requirements. Our research pursues an alternative approach, based on the time-sharing principle, that does not impose significant overhead to resource providers, while facilitating time-dependent applications to run with deadline guarantee.

Our research relies on modern system-level virtualization such as Hyper-V [15] and Xen [16] to isolate performance among concurrent applications and dynamically adjust system resources supplied to each application. As a result, we multiplex compute-intensive threads onto multiple cores of an underlying system, with differentiated resource provisioning at run-time. In the HPC community, there has been research into the use of virtualization to O/S customization and portability, security isolation, and fault tolerance [17][18][19]. We believe we are one of the first to investigate virtualization as an enabler for predictable HPC applications [8].

Control theory is one of the most widely used mathematical frameworks to control the behavior of linear dynamic systems in engineering [20][21]. Feedback control has been previously applied to various applications of computing systems including QoS for web servers [22], real-time scheduling [23], datacenter applications [24][25]. However, most of them present control theory as a methodology to solve problems in particular application domain without sufficiently addressing the usability of complex theory. While, arguably, control-theoretic schemes could be implementable by a small group of experts (e.g., data center administrator) for a small set of applications (e.g., web), the wide spectrum of users and applications in HPC domain makes it difficult to accept the theory as a practical tool. In this paper, we argue and experimentally confirm that a control design process can be programmed as software, and thus can be used as a tool.

III. COMPUTE THROTTLING FRAMEWORK

To support dynamic data-driven applications, a successful resource sharing mechanism must address two requirements:

A. *The ability to dynamically regulate the completion time of jobs at fine-granularity*

B. *The ability to cope with unanticipated "disturbances" that affects a job's performance*

The first requirement is a key for balancing highly prioritized resource provisioning to deadline-guaranteed jobs and fair-share provisioning to best-effort jobs. By ensuring execution finishes *at* the deadline, neither far earlier nor later, a system can not only offer predictability to time-dependent jobs but also provide a fair share of resources to more traditional batch jobs, thereby creating a win-win solution to both users and resource providers. The second requirement is important as well since an application's progress is not only affected by provisioning computing cycles but also by other difficult-to-control elements such as disk I/O and network load.

Milestone and Progress: In our compute throttling framework [8], we model HPC jobs using two quantitative metrics: ***milestone*** and ***progress***. The milestone determines how many computational steps should be executed before the job terminates and the progress dictates the number of steps within a fixed interval. A job's total floating point instructions is an example of milestone and executed floating point per second is an example of progress. Unlike batch-mode resource where implicit running time estimation is used when requesting resources (e.g., wall-clock-time option in *qsub*), we assume that an explicit milestone can represent a job's computational requirement. Possible sources for determining a milestone of a job include:

- *Application's semantic*: Some applications are fairly well-defined in their resource requirements. For example, a job's number of raw files to process or number of input queries can be known to users.
- *Source code*: Many HPC applications have a relatively simple program structure with deeply nested loops. A variable containing the bound of a loop, often the outer-most one, can be the basis of a milestone.
- *Linear estimation*: Profiling (i.e., sample runs) and linear estimation techniques, such as least-square regression [26], can produce a linear model that predicts total processing steps with respect to a quantifiable problem size.

In Section 5, we further discuss on the applications that fall into each category.

Another metric, *progress*, has a relationship with *milestone* and *deadline* as dictated by the following simple equation:

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

Therefore, if we know the milestone and the deadline of a job, we expect that the job will meet the deadline if the job executes *on average* at the desired progress. To measure the application's progress, we created a sensor library that users can embed into an application's source code. The library is implemented as an application-specific counter, which is strategically placed in a critical path of applications (e.g., outer-most loop or "hot spots"). The measured progress by sensor library is exported to the feedback controller that allocates and releases resources as measured progress is more or less than the target.

While our earlier work advocated the benefit of virtualization being controlled by feedback controllers whose property is rooted in the mature field of control theory, the major limitation lies in the use of mathematically complex theory for designing a resource scheduler. Arguably, an

ordinary computer/computational scientists lack the necessary knowledge and skills for designing a feedback controller. In our earlier study, we had to perform application modeling, controller design, test runs in iterative fashion, until we find a set of good control parameters for a particular application. The steps often rely on control designer's intuition, using graphical techniques such as root locus [20][21] for choosing the right control parameters. Lack of an automated, systematic approach resulted in a time-consuming design process, which often took days to create a feedback loop for just one application.

IV. SELF-TUNING VIRTUAL MACHINE

A. Performance Container as Resource Provisioning Abstraction

The resource provisioning abstraction used in our compute throttling is a virtualized resource configurable by users or resource providers. This abstraction is different from the job abstraction used in batch queue systems and the more recent leasing abstraction by which users customize application environment; however, the VM is still tightly coupled with static resources [27]. Throughout the paper, VM reconfiguration refers to changing a wide variety of resources associated with a VM, and throttling specifically refers to reconfiguration on a provisioned share of a processing unit (e.g., 50% of a core) to a VM. In our implementation on Hyper-V, we use Hyper-V's management APIs to dynamically configure a provisioned CPU share to a VM.

If a user's job requires a particular deadline, the user's VM is classified as an ***Active VM*** that can change its resource configuration at any time. The VMs that run best-effort jobs are considered ***Passive VM*** whose resource configuration can be changed by only resource providers. The resource provider's policy determines how many active VMs to be admitted to a system at a given time. In a simple case, it will be limited by host's available processing unit (cores) so as not to create a situation where multiple deadline jobs compete for the limited cores. The active VM in fact corresponds to a self-tuning VM in which a feedback controller regulates resource provisioning. It is the role of Resource Coordinator [8] to monitor the system's provisioned resources to active VMs and dynamically distribute the remaining pool to passive VMs. We implement the equal-sharing of VM scheduling credits as a simple policy for passive VMs.

B. Self-Tuning Controller

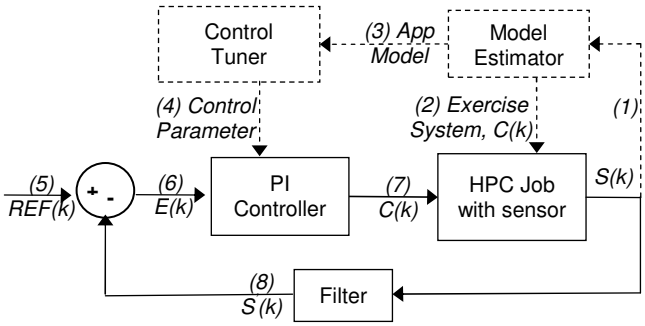


Figure 1: Block Diagram of Self-Tuning Control Loop

Once a VM is deployed on a host and authorized as an active one, the feedback controller can request/release (throttle) its share of system's core. The goal of the controller is to sustain/adapt the progress of the job at the target specified by users (to meet the deadline). To achieve the goal, we perform the three phases: *1) application modeling, 2) controller design, and 3) actual control*. Figure 1 illustrates the block diagram of the self-tuning controller that runs in an active VM.

When a job starts to run, (1) the sensors embedded in the job reports the progress (*S(k)*) to the **Model Estimator,** (2) which then exercises the system by throttling to varying levels and (3) estimates the model that relates the resource consumption to measured progress. After the modeling phase, the **Control Tuner** uses the model to design parameters for PI Controller. It uses the heuristics that we present later to find right control parameters. Once the tuning phase completes, (4) the control parameters are set in the **PI controller,** which periodically throttles to (5) track the reference progress (*REF(k)*) derived to meet the job's deadline. It uses the (6) error (reference-measurement) in previous cycles to determine the (7) throttling at the next cycle. (8) A moving average filter is placed in between the controller and the job being sensed such that measurement noise can be smoothed out. Note that the three phases can be repeated if there's a significant change in the application model. For example, if the application consists of different routines (binaries) executed in series, each routine may invoke the three phases again.

Model Estimator: The progress of a job with respect to provisioned resources is modeled as a first-order linear difference equation: $S(k) = a \cdot S(k-1) + b \cdot C(k-1)$ (2)

In the model, *S(k)* represent the sensed progress and *C(k)* represent the provisioned share of a core (Hyper-V's VM scheduling cap). In the model, the previous outputs, *S(k-1),* affect the current output, *S(k),* because there is an actuation delay due to various disturbances such as I/O latency. When a job is in modeling phase, model estimator directly issues throttling (*C(k)*), following a low-frequency, discrete sine waves whose amplitude is from a minimum to maximum throttling (0-100 for Hyper-V). We found, in practice, a sine wave with frequency=5 and period=2 (i.e., 10 different tests) can exercise the system with sufficient excitation. After progress measurement is obtained, Model Estimator runs least-square regression algorithm [26] which can estimate the linear model (values for a and b) quickly (less than a second).

PI Controller: We use a digital form of Proportional-Integral (PI) controller [20][21] since it strikes the fine balance between control performances and design complexity. In our work, simple design is important criteria since the design process must be programmed/automated. The time-domain representation of PI control law has the form:

$$C(k) = C(k-1) + (K_P + K_I)E(k) - K_P E(k-1) \quad (3)$$

In the equation, the signal *C(k)* refers to a provisioned resource and *E(k)* refers to an error (reference – measured progress). K_P and K_I are the controller parameters that determine how much to react given the errors at previous cycles. The K_P is a proportional term which determines the

actuation (throttling) for the error in a previous cycle. The K_I is an integral term that determines the throttling for accumulated errors in previous cycles. The controller design reduces to choosing the right values for K_P and K_I that has good control performances.

We express the variables of the closed-loop, including reference, measured progress, error, as a signal which is a series of values at different sample cycles. The digital control theory defines a convenient way to encode the signals and system's components, 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 z-transform is used to describe a system's component such as PI-controller, application model, we call it transfer function that describes how an input signal is transformed into an output. By using a transfer function, system's discrete components can be combined via simple algebraic manipulations. Due to space limitation, 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 [20][21].

The closed-loop (the lower part of Figure 1 with solid line) can be integrated into a simplified transfer function as follows. We first define the transfer function of the target system (application model) that has time-domain representation, $S(k) = a \cdot S(k-1) + b \cdot C(k-1)$ where a, b are the model parameters that Model Estimator produces:

$$G(z) = \frac{S(z)}{C(z)} = \frac{b}{z-a} \qquad (4)$$

The PI law (equation 3) can be similarly represented as a transfer function:

$$D(z) = \frac{C(z)}{E(z)} = \frac{(K_P+K_I)z-K_P}{z-1} \qquad (5)$$

We also add a moving average filter which has a time-domain equation, $S'(k) = C \cdot S'(k) + (1-C)S(k)$, where C is a constant determining degree of smoothness. The equivalent transfer function is: $H(z) = \frac{1-c}{z-c}$ *(6)*

Finally, the overall closed-loop is reduced to a unified transfer function:

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

***Control Tuner*:** In the on-line controller design, we draw requirements from the four properties of closed loop:

- *Stability*: Control Tuner must ensure that for bounded input (reference progress) to a closed-loop, the loop's output (measured progress) is bounded as well. The unstable state refers to a situation where controller issues throttling request that is excessively variable. According to the control theorem, the close-loop is stable if and only if all poles of closed-loop (*FR(z)*) are inside the unit circle.
- *Accuracy*: PI-control law achieves zero steady state error since I-term (K_I) accounts for the errors in previous history. Thus, accuracy does not add constraint to controller design.
- *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. In general, we say system is in steady-state if the closed-loop's output reaches within k % of the steady state value. In this paper, we use 10% as a threshold. In self-tuning VM, reference is changed whenever the job's deadline is changed. Thus, shorter settling time is especially important if the job runs relatively short, or frequent deadline changes are expected. Also slow settling time leads to lagging reaction to disturbances such as disk I/O. 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: $Ks \approx \frac{log\frac{k}{100}}{log\, a}, where\ a\ is\ the\ largest\ pole\ of\ the\ closed\ loop\ (FR(z))$ (8)
- *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. We can find an example in Figure 5(d) where measured output exceeds the reference at around 75th cycles. 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 (7) 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:

$$M_P = -a, if\ largest\ pole\ of\ closed\ loop\ is\ negative.$$
$$M_P = 0, otherwise \qquad (9)$$

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:

$$M_P \approx r^{\frac{\pi}{|\theta|}}, where\ r = \sqrt{c^2+d^2}\ and\ \theta = tan^{-1}(\frac{d}{c}) \qquad (10)$$

We transform equation (5) to an equivalent form:

$$D(z) = \frac{(K_P+K_I)z-K_P}{z-1} = (K_P+K_I)\{\frac{z-\frac{Kp}{Kp+KI}}{z-1}\} \qquad (11)$$

(K_P+K_I) and $(\frac{Kp}{Kp+KI})$ represent overall gain and zero of the PI controller, respectively. The goal of Control Tuner is to select values for the gain and zero, whereby subsequently K_P and K_I are obtained by solving the equations (11). 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 guarantee stability), 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 2 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). The return values from the algorithm are near-optimal gain and zero, from which K_P and K_I are solved.

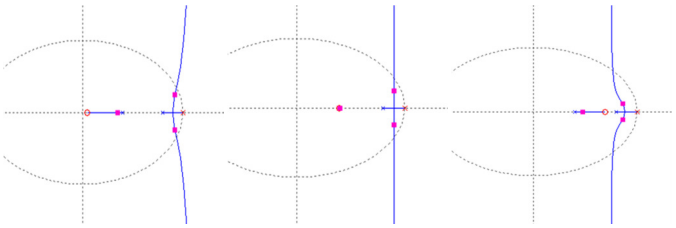
The algorithm picks candidates of zero and gain evenly distributed by M, N, within their valid range (line 5 and 9). Since $K_P > 0$ and $K_I > 0$, the zero $(\frac{K_P}{K_P+K_I})$ must be between 0

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.

```
1   function [opt_gain, opt_zero] = ControlTuner(M, N, G_Z)
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        K_S(i,j) = compute_K_S(zero_values(i), gain_values(j), Gz);
12        M_P(i,j) = compute_Mp(zero_values(i), gain_values(j), Gz);
13      end;
14    end;
15    [i, j]= rank (K_S, M_P);
16    opt_gain=gain_values(i,j);
17    opt_zero=zero_values(i,j);
```

Figure 2: Control Tuner 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 great influence to the settling time of the closed-loop. Figure 3 illustrates a root locus of the closed-loop that shows the effects of the heuristic. Root Locus is the most common, graphical technique that plots the traces of poles and zeros of the closed-loop system as controller's zero and gain vary [20][21]. 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 (8), 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.



(a) zero < min_pole (b) zero = min_pole (c) zero > min_pole

Figure 3. 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, every pole of closed-loop must lie within the unit circle. 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 < 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 will result in unstable system).

After determining the ranges, the algorithm computes settling time (line 11) and maximum overshoot (line 12) using the equations (8)-(10) 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. There are many possible strategies for implementing the rank function. An algorithm may pick the gain/zero that results in minimum settling time, regardless of maximum overshoot associated with the pair, or may put more priority to maximum overshoot. Although, theoretically, a zero-gain pair can produce unbalanced results (e.g., short settling time-large overshoot), we found, in practice, a good zero-gain pair tends to achieve both. Thus in our implementation, we use simple objective 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).

V. EXPERIMENTAL EVALUATOIN

In our implementation, the Model Estimator and Control Tuner are written in Matlab using algebraic tools for Z-transform equations and compiled into C library using Matlab compiler. The sensor libraries are written in C and Fortran supporting the applications written in both languages.

A. Usability of Self-Tuning VM

Once users have an application to run with an explicit deadline, they are expected to place sensor calls in the application's source code, compile the code, and run the self-tuning controller with the two parameters, *milestone* and *deadline*. The reference progress is derived from the two parameters using equation (1). The sensor library reports progress measurement to the self-tuning controller via IPC channel (e.g., file, shared memory). Once the self-tuning controller starts to receive progress from the application, it performs the three phases- application modeling, controller tuning, actual control-in series. There are neither more inputs the user has to give to the controller, nor any tuning to address idiosyncrasy of controller or application.

While running the self-tuning controller with just two inputs are very straightforward, users are still required to undertake two extra efforts: embedding a sensor library in existing source code and estimating the milestone (explicit computational quantity) of the job. We discuss the difficulty of the two efforts based on our experience with the five applications illustrated in Table 1. As we explained earlier, there are many possible sources for estimating the milestone of a job. The simplest case that does not require user effort is to estimate it from job's known semantic such as number of raw

data to process. We found *mProject* [13] belongs to the case. Since the semantic is known, we could easily spot the location where the sensor library should be placed. For *mProject*, only one sensor call is placed after the statement processing a raw data. The next approach requiring slightly more user effort is to directly reading the milestone from variables of program's source code, often the variable containing bounds of outer-most loop. In *OpenLB* [10] and *ADCIRC* [9], we could easily spot the variables as they are already used by program's debugging routines that show progress of job's execution. It is easy to place sensor calls as well; we simply put the sensor calls at the first statement of (outer-most) loop. For both applications, only one sensor call is placed in the source code. The most difficult case to estimate the milestone is to use profiling and linear estimation techniques such as least-square regression. The *BLAST* [12] and *WRF* [11] are examples. We placed sensor calls in some (possibly many) locations throughout the source code and run the jobs with varying problem size (e.g., query size in *BLAST*, forecast hours in *WRF*), and establish a model that relates the problem size to number of sensor calls. At run-time, our automated controller designer uses the model with the requested problem size to estimate the milestone. This requires users to understand the structure of application source code and spots the location where it is likely to be executed constantly over a period. *BLAST* required more effort to identify the locations; we placed three sensor calls in the source code. *WRF* required only one sensor call.

Overall, although the source code instrumentation and milestone estimation requires a little extra effort, we believe the usability of feedback-controlled application is significantly improved via the automation of the controller design process. In our earlier study without automated control, we spent more time on application modeling and controller design than for the sensor placement and the milestone estimation. While the sensor placement is required only once for each application, the controller design had to be repeated not only for each application, but also for varying underlying resources.

B. Correctness of Self Tuning

We measure in small scale the steady-state and transient behaviors of the applications being regulated. The resources used in the experiments are described in Table 2. We start the self-tuning VM which is given the target progress for tracking. It performs model estimation, controller design, and actual control with the sensor signal received from the application. To measure the application's behavior, we modulate the reference progress. For each application, we run the same test 5 times and report average of each evaluation metric. Moreover, to measure the effect of non-trivial disturbance, we run another set of test with a VM that executes BLAST concurrently. The BLAST as a disturbance generator has sustained 6.43 MB/s read rate on the disk that it shares with the self-tuning VM. Figure 5 and Table 3 present the results.

Among the 5 runs without disturbance, Figure 5 presents the result that shows the third best performances in terms of steady-state error. In each figure, the straight, dotted line represents the reference progress, which is the target that the controller aim to track, and the solid line represent the real progress that we obtain from application execution. Finally, the curved, dotted line represents the simulated progress using the application model from Model Estimator and K_P and K_I determined by Control Tuner. This illustrates what Control Tuner predicts as application's behavior when the controller parameters are determined online. In each figure, the first 10 cycles show the estimation phase by Model Estimator that measures the progress at varying levels of resource provisioning. After the modeling phase, PI control parameters are determined shortly and the resulting PI controller regulates application's progress.

Table 3 presents the results more quantitatively. The table's entry contains two values (average of five runs) separated by '/'. The left is the result from the runs without disk disturbance (BLAST) and the right value is obtained when disk disturbance is introduced. The first two rows contain the steady-state error, as defined by, $abs(\frac{measurement-reference}{reference}) \times 100$, for the references in high (60-110th cycles) and low (110-160th cycles) values. The last two rows present the settling time (to reach 10% of new reference), for both low-high and high-low reference transitions. The values in the parenthesis are predicted settling times when Control Tuner determines control parameters (there is no prediction in steady-state error because PI-controller has zero steady state error in theory). The smaller steady-state errors are desired as it result in meeting deadline closely, and shorter settling time is important if computations are short or frequent deadline changes are expected.

Figure 5 and Table 3 confirm that the four applications, ADCIRC, BLAST, OpenLB, WRF, track the reference progress with high accuracy. The real behaviors of application are very close to the predicted behaviors at the time the controller is designed online. This experimentally confirms that the Control Tuner can produce high quality control parameters, and the resulting PI controller ensure that it tracks the application's progress in reality just as predicted in design time. However, *mProject* warrants more explanation. Unlike the other applications, *mProject* exhibits high irregularity. The sensors in *mProject* report progress whenever a raw data is processed, however some raw data occasionally requires much longer processing. This results in periods in which no progress is reported (69-73th cycles in Figure 5(d)). During this period, errors are continuously accumulated within PI controller and the large throttling values are repeatedly requested. However, after the blocking raw data is eventually processed, it results in errors in opposite sign since too much resource are enforced previously. This so-called *integral windup* is often found in controllers with integral term. The output fluctuation eventually leads to more errors in meeting deadline, as we present in the next subsection.

In addition to the major findings above, the additional findings can be summarized as follows. For all applications, the maximum overshoots were negligible. Disk disturbance causes more steady-state errors especially when reference is set higher. While the real settling times during the low-to-high reference transition are very close to what Control Tuner predicts, the high-to-low reference transition takes longer and exhibit more deviation from the prediction.

Table 1. Applications in Experiments

	ADCIRC	BLAST	mProject (Montage)	OpenLB	WRF
Domain	Costal flood modeling	Biology	Astronomy	Fluid-flow modeling	Weather Forecast
Compute/ Data Intensive	Compute Intensive	Compute-Data Intensive	Compute-Data Intensive	Compute Intensive	Compute Intensive
Milestone	Source code	Linear estimation	Program manual	Source code	Linear estimation

Table 2. Resource Configuration

	Low-end server (Subsection B)	High-end server (Subsection C)
CPU	Intel Dual Core 2 2.13 GHz	2 Quad Core Opteron 1.7 Ghz (8 cores)
RAM	2 GB	8 GB
DISK	10,000 RPM SATA	7,200 RPM SATA
O/S	Windows Server 2008 with Hyper-V	Windows Server 2008 with Hyper-V
VMs	1 Self tuning VM + 1 Disturbance generator	5 Self tuning VMs + 8 Best-effort VMs

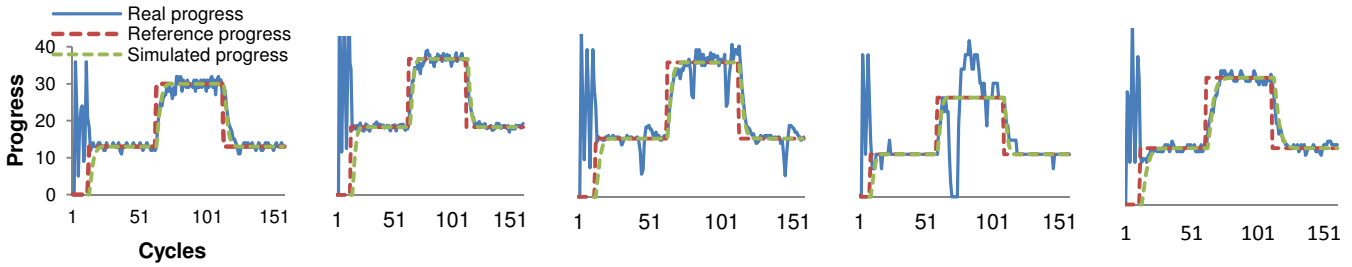


Figure 5(a): ADCIRC (b): BLAST (c): OpenLB (d): mProject (e): WRF

Table 3. Steady-state and Transient Behaviors of Applications

	ADCIRC	BLAST	mProject	OpenLB	WRF
Steady-state Error (high)-%	4.8 / 6.6	2.5 / 5.7	19.5 / 12.6	6.5 / 8.4	3.4 / 6.1
Steady-state Error (low)-%	3.0 / 3.2	2.8 / 3.1	4.4 / 1.7	8.1 / 7.5	4.0 / 4.6
Settling-Time (low to high) - cycles	7.2 (6) / 6.8 (6)	5 (4.8) / 5.8 (8.6)	13.1 (5.6) / 6 (5.1)	5.6 (5.8) / 5.8 (5.2)	6.2 (5.4) / 8.4 (5.6)
Settling-Time (high to low) - cycles	9.6 (7) / 10.6 (7)	7 (5.4) / 6 (6)	12 (7) / 11.1 (6.4)	9.4 (6.8) / 5.8 (5.2)	6.6 (6.6) / 12.2 (6.6)

C. Meeting Deadlines on a High-end Server

We now report how closely the self-tuning VM meets the application's deadlines in realistic environments. For the evaluation, we use 8-core AMD server which is highly overloaded by HPC jobs during the experiments. We run 8 passive VMs that run best-effort jobs (WRF) concurrently with 5 self-tuning VMs (13 VMs in total). In each self-tuning VM, the application is run with varying deadlines presumably requested by users. The self-tuning VM is given the deadline and milestone of a job from which it derives the target progress to track. In this experimental scenario, we assumed users specify deadlines in broad range, but the earliest deadline is chosen such that it can be met with 100% of a system's core. If the deadline is earlier than that, the application will just saturate the core with constant errors (reference > measurement) in controller. Our contribution is to guarantee that the real execution time finishes *at* the deadline, neither earlier nor later, so that we achieve both high predictability and fair-share to best-effort jobs. In shared resources, the jobs finished too earlier than deadline are considered harmful as they take resource's share that could be consumed by other time-dependent or best-effort jobs. We are currently working on admission controller that deterministically accept/reject jobs with deadlines. Note that in this experiment, all 5 applications are run simultaneously with 8 best-effort jobs, with server's total utilization reaching almost 100%. The experiments ran for 14 hours.

Figure 6 presents scatter plots in which the x-axis is the requested deadline and the y-axis corresponds to the real execution time. The linear lines illustrate the ideal results. All applications meet the deadlines very closely. In particular, ADCIRC, OpenLB, and WRF show almost perfect correspondence between the deadlines and the real execution times. Table 4 presents the results quantitatively. The average deadline guarantee error, which is defined as $abs(\frac{running\ time - deadline}{deadline}) \times 100$, for the three applications are less than 3 %. In other words, if the user's requested deadline is 1 hour, the real execution time would have less than 2 minutes in error. The results indicate that BLAST and Montage exhibit more errors compared to others: there could be up to 5 minutes error for 1 hour deadline. We estimate the milestone of BLAST using linear estimation from requested query size. However, the estimation is not 100% accurate in predicting the real milestone at run time. The R^2 values from the least-square regression is 0.92, hence it results in 6.2 % deadline errors despite of accurate progress tracking (Figure 5(b)). The relatively high errors in Montage are due to its irregular behaviors as we explained earlier. Overall, if an application does not exhibit high irregularity and the estimated milestone is accurate, the self-tuning VM can meet the deadline with very small errors (< 3 %), as shown by ADCIRC, OpenLB, and WRF.

One way to evaluate the performance of self-tuning is to compare it against the manual controller tuning by control

experts. Since the manual tuning by people is subject to qualitative judgment, we use our earlier results presented in [8] for comparison, in which measured deadline-guarantee error for WRF on the same 8-core machine. We believe the earlier results can represent almost the best possible case with manual tuning, as performance was our main goal. In [8], we achieved 3.4% errors, and we achieve 2.6% errors in this study, both for WRF. This is an important evidence that show programmed control tuning can possibly achieve better predictability, or to be more conservative, achieve comparable predictability to manual tuning.

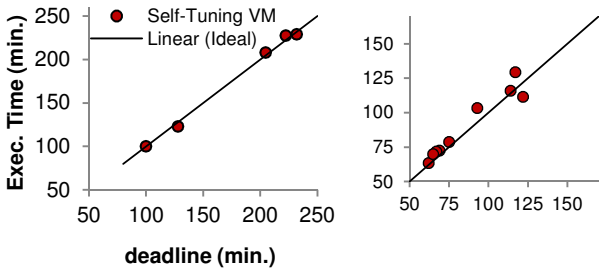
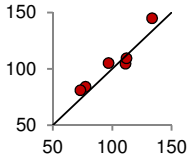
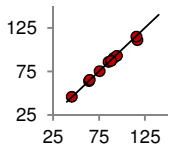


Figure 6(a). ADCIRC

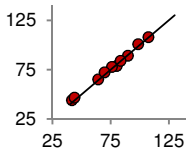
(b). BLAST



(c). mProject



(d). OpenLB



(e). WRF

Table 4. Average Deadline Guarantee Error

	ADCIRC	BLAST	mProject (Montage)	OpenLB	WRF
Error (%)	1.8	6.2	7.6	1.3	2.6

VI. CONCLUSION AND FUTURE WORK

We present self-tuning VMs as a practical approach to build predictable HPC infrastructure. By "programming" the controller design process, we not only achieve better usability, but also enable highly predictable execution service. The online heuristics whose properties are rooted in the mature field of control theory achieves both provable correctness and practical efficiency. Our future work includes studying how to extend the control-theoretic scheduling to different types of HPC applications including MPI parallel programs and loosely coupled science workflows. We also plan to extend Self-Tuning VMs to support admission control. .

Acknowledgements

We would like to thank to ADCIRC team for sharing their source code with us.

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] Manos, S., Zasada, S., Coveney, P. V. Life or Death Decision-making: The Medical Case for Large-scale, On-demand Grid Computing. CTWatch Quarterly, Volume 4, Number 1, March 2008.

[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 QoS, 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] G. Singh, C. Kesselman, E. Deelman. Adaptive Pricing for Resource Reservations in Shared Environments. IEEE/ACM International Conference on Grid Computing, 2007.

[7] W. Smith, I. Foster, V. Taylor. Scheduling with Advanced Reservations. IEEE Int. Par. and Dis. Proc. Symp. 2000.

[8] S-M. Park, M. Humphrey. Feedback-Controlled Resource Sharing for Predictable eScience. IEEE/ACM Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC08), Nov 15-21, 2008, Austin, Texas

[9] Coastal Circulation and Storm Surge Model: http://adcirc.org

[10] Open source lattice Boltzmann code: http://www.openlb.org

[11] The Weather Research and Forecasting Model: http://www.wrf-model.org

[12] BLAST: Basic Local Alignment and Search Tool (http://www.ncbi.nlm.nih.gov/blast/)

[13] Montage-An Astronomical Image Mosaic Engine: http://montage.ipac.caltech.edu

[14] 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.

[15] Windows Server 2008 Hyper-V. http://www.microsoft.com/windowsserver2008/en/us/hyperv.aspx

[16] 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.

[17] R.J. Figueiredo, *et. al.* A Case for Grid Computing on Virtual Machines. Int. Conf. on Dis. Comp. Sys. (ICDCS), April 2003.

[18] I. Foster, *et. al.* Virtual Clusters for Grid Communities. IEEE Int. Symp. on Cluster Comp. and the Grid, May 2006.

[19] A.B. Nagarajan, F. Mueller, C. Engelmann, S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. International Conference on Supercomputing, 2007.

[20] J.L. Hellerstein, *et. al.* Feedback Control of Computing Systems. Wiley-IEEE Press, August 2004.

[21] G.F. Franklin, J.D. Powell, M. Workman. Digital Control of Dynamic Systems. Addison Wesley, 1998.

[22] 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.

[23] 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.

[24] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, R. West. Friendly Virtual Machines Leveraging a Feedback-Control Model for Application Adaptation. ACM/Usenix International Conference On Virtual Execution Environments (VEE), 2005.

[25] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the Use of Fuzzy Modeling in Virtualized Data Center Management. International Conference on Autonomic Computing (ICAC), 2007.

[26] J.S. Milton, J.C. Arnold. Introduction to Probability and Statistics. 4th ed. McGrawHill.
[27] 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.