Pacer: Automated Feedback-Based Vertical Elasticity for Heterogeneous Soft Real-Time Workloads

Yu-An Chen*, Andrew J. Rittenbach†, Geoffrey Phi C. Tran†*, John Paul Walters†, and Stephen P. Crago*†

*Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: {chen116, geoffret}@usc.edu

†Information Sciences Institute
University of Southern California
Arlington, VA 22203
Email: {arittenb, gtran, jwalters, crago}@isi.edu

***Abstract*—Cloud computing can be used to provide a virtualized platform for running various services, including soft real-time applications such as video streaming. To satisfy an application's real-time requirements, CPU resources are often allocated for the worst case, resulting in system under-utilization. To solve this problem, we present Pacer, a framework that allows system or application developers to implement their own resource allocation algorithm and utilize real-time performance feedback from applications that are running in VMs. We then present and compare two example resource allocation algorithms that are based on Additive-Increase-Multiplicative-Decrease and Self-Tuning PID control. We apply Pacer to a video stream object detection application and show that Pacer can save more than 50% CPU utilization while still meeting deadlines. Finally, we present a stride scheduling based sharing algorithm to maintain applications real-time performance when the total resource demands exceed the system's available resources. Compared with static CPU resource allocation, our experiments show that Pacer can reduce deadline misses by more than 50% in an oversubscribed system.**

***Keywords*-soft real-time; vertical elasticity; feedback-based resource allocation; Virtualization; Xen**

I. Introduction

More and more applications are being deployed to the cloud, including real-time applications. Since cloud computing introduces non-deterministic latency[1], resources are usually over-provisioned for real-time applications to ensure performance requirements are met. On top of that, many real-time applications have dynamically varying workloads which result in greater resource under-utilization when the application is running in a lower workload mode. We can solve this problem by monitoring the status of the application and dynamically adjusting its resources. Since cloud computing relies heavily on virtualization, we are developing the Pacer framework in Xen. Pacer monitors real-time applications running in VMs and provides developers a platform on which they can implement their own choice of resource management scheme. The high level view of the Pacer framework is shown in figure 1, where the Pacer resource manager collects performance feedback and then dynamically adjust resource allocations for the VMs.

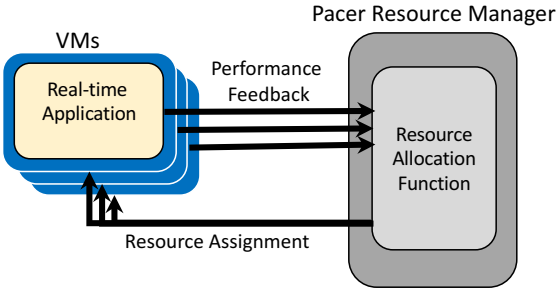


Figure 1: High Level Pacer Framework

A motivating example is a server hosting multiple VMs that run security surveillance applications monitoring different parts of a building. The application runs in a light workload mode when no intruder is detected, and switches to a heavier workload mode when an intruder is detected. While the application runs in the light workload mode, CPU resources can be saved for non real-time tasks such as routine server backups. When one or more cameras detect an intruder and switch to the heavier workload mode, Pacer becomes aware of the increased demand from the application and reassigns resource to the VMs such that all applications can meet their real-time performance requirements.

This paper makes the following contribution:

1) We develop Pacer, a framework that provides developers a platform to implement CPU resource allocation for VMs that run soft real-time applications.
2) We modify Additive-Increase-Multiplicative-

Decrease[2] and Self-Tuning PID Controller[3] and turn them into CPU resource allocation algorithms that utilizes feedback from real-time applications.
3) We present stride sharing, a CPU resource sharing algorithm based on stride scheduling[4] to improve real-time performance under an overloading system.

The remainder of the paper is organized as follows: Section II presents related work. Section III presents background for the Pacer framework. Section IV presents the details of the Pacer framework and its usage. Section V presents resource allocation and sharing algorithms. Section VI presents the experimental results and analysis. Finally, section VII presents our conclusions and future work.

II. Related Work

There is much related work on feedback-based resource control and vertical elasticity in the cloud.

Tran et al. [5] present a dynamic resource allocation framework that calculates the resources needed and adjusts CPU resources before workload changes. This work targets hard real-time applications and requires that the system has prior knowledge about workload changes. Our work emphasizes feedback-based resource allocations that apply to soft real-time applications whose behavior is not known in advance.

Guan et al. [6] design a resource allocation algorithm for Docker containers that uses applications' requirements as one of the optimization variables. Their work focuses on resource placement for containers and assumes all jobs in the application have the same finish time.

Sotiriadis at al. [7] create a dynamic reconfiguration system that implements vertical and horizontal elasticity by eliminating service down-times and communication failures. Their work focuses on load balancing service traffic to the servers.

Hoffmann et al. have much related work[8][9][10][11] that uses feedback-based control and machine learning to optimize soft real-time performance and energy efficiency of embedded or bare-metal systems. One major difference between our work and theirs is that we provide a framework that is applied to virtualized platforms that host multiple applications.

Molto et al. [12] present a framework for a cloud management system that controls live VM migration and adjusts memory resources assigned to the VMs based on memory usage.

III. Background

A. Real-time Application

A real-time application refers to an application that requires tasks to finish before certain deadlines. Based on how critical it is to meet the deadlines, real-time applications can be divided into three categories[13]:

1) Hard Real-time: Any task that misses deadline indicates application failure. An example includes nuclear reactor monitoring.
2) Firm Real-time: Tasks missing deadlines are allowed, but become useless for the application. An example includes database transactions.
3) Soft Real-time: Tasks that miss deadlines are allowed but their usefulness degrades over time. An example includes video surveillance.

Many real-time applications that are running in the cloud are soft real-time because of their tolerance for deadline misses allow the benefits of cloud computing to be exploited. Our work focuses on dynamic resource allocation for soft real-time applications.

B. Heartbeats API

The Heartbeats API[14] is a tool used for monitoring application's status. Many applications perform certain tasks repeatedly, such as video and audio processing. The idea of the Heartbeats API is that the application programmer inserts heartbeat calls between periodic tasks and then the framework can report application performance in terms of heart rate(beats/second). The interpretation of heart rate depends on the application. For example, heart rate for a video application can be interpreted as frame per second (FPS) if heartbeats are recorded for every frame. Pacer uses heart rate as the mechanism for real-time performance feedback. In this work, we also define a deadline miss as when the reported heart rate (based on the time between the current and previous heart beats) is lower than a predetermined value.

C. Virtualization Through Xen

Xen is an open-source hypervisor that supports para-virtualization[15]. Domain0(Dom0) is the privileged domain that manages other guest domains (VMs). In this work, applications are run inside the VMs, while the Pacer resource manager runs inside Dom0. In Xen, physical CPUs(PCPUs) are virtualized for all VMs including Dom0. All VMs are required to be assigned to a PCPU pool. A PCPU Pool is composed of a subset of PCPUs and a scheduler that schedules virtual CPUs (VCPUs) to run on PCPUs. To avoid confusion, CPU refers to VCPU in the remainder of this paper.

The default CPU scheduler in Xen is the Credit scheduler[16]. The Credit scheduler is a proportional

fair share scheduler that allocates CPU utilization based on the weight of the VM and time-slice set by the user. Each VM shares the CPU time proportionally in each time-slice. For example, in a PCPU pool with 2 VMs, if the user want both VMs to have 50% utilization, the user can simply set the same weight values for both VMs.

The Real-Time-Deferrable-Server(RTDS) scheduler [17], a CPU scheduler designed for real-time tasks, is another scheduler option for Xen. In the RTDS scheduler, CPU utilization for each VM is determined by the budget and period set by the user. Each VM is guaranteed to receive the assigned CPU budget time for every period. For example, in a PCPU pool with 2 VMs, if the user want to assign 50% utilization for both VMs, the user can simply set the same budget values for both VMs. Pacer provides an interface to both Credit and RTDS scheduler for developers to assign CPU utilization for VMs.

IV. Pacer Framework

In this section, we will go over the building blocks of Pacer in detail. Pacer is written in Python and can be easily extended to support applications in other languages.

A. Application Heartbeats

In this work, application heartbeats are used as real-time performance feedback. The Heartbeats API has the option of recording instant or window heart rates. We modify the Heartbeats API[14] to provide a Python interface. Application developers can use this interface to insert proper heartbeat calls in their applications.

B. Inter-Domains Communication

For VMs to send their heartbeats feedback to Pacer resource manager in Dom0, we create an interface to handle communication between Dom0 and VMs by utilizing Xenstore[18] and pyxs[19]. Xenstore is a database that supports transactions and atomic operations. All VMs have a unique path to access Xenstore hosted by Dom0 through Xenbus driver or Unix domain socket[20]. Using our interface, the application can specify a list of entries for Pacer's resource manager to watch for any update. Application developers can use this interface to communicate with the Pacer resource manager in their applications.

C. Pacer Resource Manager

The Pacer resource manager runs in Dom0. Its primary function is to gather performance feedback from the VMs and to control dynamic CPU resource allocation. Figure 2 shows the details of a running Pacer framework. The Pacer resource manager creates a thread for every VM it monitors. Each thread watches for its monitoring VM's heartbeats entry to Xenstore and activates a resource allocation function when a new heartbeat is updated by the VM. To avoid assigning invalid CPU utilization to the VMs, the CPU utilization configuration for all VMs is recorded at the start of Pacer resource manager. This data is shared among the threads. Every thread is required to obtain a mutex lock before modifying this data and adjusting CPU resources for VMs. Application or system developers can implement their own CPU resource allocation algorithm inside the resource allocation function. We present example resource allocation algorithms in the next section. Finally, Pacer provides an interface for accessing the LibXenlight API to assign CPU utilization to VMs based on the output from the resource allocation function.

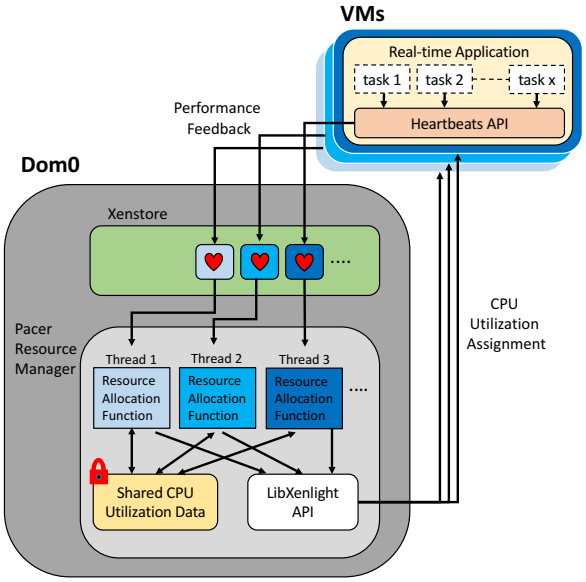


Figure 2: Pacer Framework

V. Resource Allocation and Sharing Algorithms

In this section we present two resource allocation algorithms and one resource sharing algorithm.

A. Additive-Increase-Multiplicative-Decrease(AIMD)

AIMD[2] is a popular feedback-based control algorithm. TCP congestion control is a well known use of AIMD, as shown in equation 1, where r is the sending rate, t is the time slot, m and n are parameters that satisfy $m > 0$ and $1 > n > 0$.

$$r(t+1) = \begin{cases} r(t) + m & \text{if congestion is not detected} \\ r(t) * n & \text{if congestion is detected} \end{cases} \quad (1)$$

The challenges Pacer resource allocation faces are similar to what TCP control faces. First, congestion

in TCP control is analogous to a deadline miss in real-time application. Second, additive increase of the sending rate before congestion occurs in TCP control is analogous to additive increase of spare CPU resources before a deadline miss in the Pacer framework. Finally, multiplicative decrease of transmission rate upon congestion is analogous to multiplicative decrease of spare CPU resources upon a deadline miss.

We modify AIMD to fit the Pacer resource allocation scheme. The modified update rules for AIMD are shown in equation 2, where $u(t)$ and $u(t+1)$ represent the current and the next CPU utilization assignment, $s(t)$ and $s(t+1)$ represent the current and next spared CPU utilization, m and n are parameters that satisfy $1 > m > 0$ and $1 > n > 0$. Operating on spare CPU resources instead of CPU resources directly ensures the demanded CPU utilization, $u(t+1)$, is always valid. In the modified AIMD, we also introduce two new parameters, b_u and b_l, which are the upper bound and lower bound for acceptable heart rate value. The function of b_u and b_l is to define a range such that if a heart rate falls within this range, the CPU utilization demand remains constant, thereby achieves a smoother CPU utilization allocation curve when the algorithm reaches steady state. m and n can be tuned to achieve the desired performance. A larger m causes CPU utilization to decrease faster when heart rates are greater than b_u, but it can cause overshoot and make heart rates drop below the lower bound, b_l. A smaller n causes CPU utilization to increase faster when the heart rates are less than b_l, but it can cause overshoot and make heart rates greater than the upper bound, b_u.

To use the AIMD algorithm, the developer needs to specify m, n, b_l and b_u. The AIMD algorithm will adjust CPU utilization based on heartbeats feedback.

$$
\begin{aligned}
&s(t) = 1 - u(t) \\
&s(t+1) = \begin{cases} s(t) + m & \text{if heart rate} > b_u \\ s(t) & \text{if } b_u \geq \text{heart rate} \geq b_l \\ s(t) * n & \text{if heart rate} < b_l \end{cases} \\
&u(t+1) = 1 - s(t+1)
\end{aligned}
\tag{2}
$$

B. Self-Tuning Proportional-Integral-Derivative Control (STPID)

STPID[3] is another feedback-based control algorithm. STPID's ability to adapt to system changes makes it a suitable resource allocation algorithm for applications with heterogeneous workloads. The block diagram STPID is shown in figure 3, where u is the target heart rate, y_4 is the feedback heart rate, and e is the difference between y_4 and u. γ in figure 3 is the parameter that dictates how sensitive the controller is to the error(e in figure 3). We modify the algorithm such that γ becomes an adaptive variable based on current error, e. The new update rule for γ is shown in equation 3. With the new update rule, a larger e gives a larger γ which leads to a more aggressive response.

To use the STPID controller, the developer just needs to specify u, the target heart rate. The STPID algorithm will adjust CPU utilization based on heartbeat feedback to reach the target heart rate.

$$
\gamma = \frac{log(|e|+1)}{log(u)} \tag{3}
$$

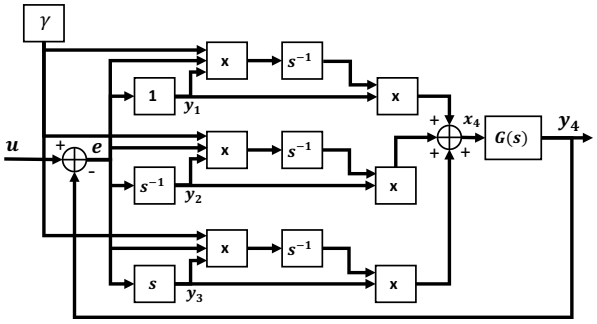


Figure 3: Self-Tuning PID Controller[3]

C. Stride Sharing

We develop a stride sharing algorithm that is inspired by stride scheduling[4], a proportional-share CPU scheduler. Both stride sharing and stride scheduling distribute resources based on stride values. The major difference between stride scheduling and stride sharing is how the CPU resource is used. In stride scheduling, one particular task is selected to run with full CPU utilization for the current quantum while other tasks are put in a waiting queue. In stride sharing, a particular VM is selected to run with the CPU utilization it demands for current quantum while other VMs share the remaining CPU utilization instead of being inactive like the idle tasks in stride scheduling. The stride sharing algorithm is applied when the Pacer resource manager observes the total CPU utilization demand from the VMs exceeds the system's capacity.

To use stride sharing, the developer needs to specify the stride values for the VMs that are sharing the same PCPU pool and the scheduling quantum that specifies the duration for the selected VM to run with the CPU resource it demands. AIMD and STPID can be used with stride sharing because AIMD and STPID calculate the CPU utilization each VM should request and stride sharing manages those requests when the system is oversubscribed.

VI. Evaluations and Discussion

In this section, we first go over the system setup and the overhead introduced by Pacer. We then present the soft real-time application that is used for the experiments. Finally, we evaluate Pacer's ability to save CPU resources and maintain soft real-time performance through three different sets of experiments.

A. System Setup

The system used for the experiments is a SGI Altix XE500 server. The server utilizes two Intel Xeon E5520 processors, each with eight threads running at 2.27 GHz. RT-Xen is installed on this server with three PCPU pools, configured as shown in table I. Dom0 has 24 GB of memory and 6 CPUs with 6 dedicated PCPUs. Every VM for the experiments is created with 4GB of memory and 5 CPUs. All CPU utilization assignments in the experiments are applied to each CPU in a VM. For example, assigning 50% CPU utilization to a VM refers to assigning 50% CPU utilization to all 5 CPUs for that VM. Finally, we use the default 30ms scheduling quantum for Credit and RTDS scheduler.

Table I: PCPU Pools Configuration

PCPU Pool Name	Assigned PCPUs	User	Scheduler	Scheduling Quantum
0	0-5	Dom0	RTDS	30ms (period)
1	6-10	VMs	RTDS	30ms (period)
2	11-15	VMs	Credit	30ms (time-slice)

B. Overhead

Pacer's overhead includes the time it takes to record a heartbeat in the VM and the time it takes to send a heartbeat from the application in VM to Pacer resource manager in Dom0. We measure the overhead introduced by heartbeats as well as the communication latency between applications and Pacer resource manager.

Heartbeats Throughput:

To measure heartbeats throughput, we run a loop inside a VM that performs 10000 heartbeat calls and calculate the average rate(beats/second).

Communication Latency:

To measure communication latency between a VM and Dom0, we run the experiment as follows: a VM creates a counter and passes its value to Dom0, and then Dom0 passes the same value back to the VM. After receiving the original counter value, the VM records the round trip time and increases the counter value by one before sending it to Dom0 again. This process is repeated for 10000 times. The communication latency is estimated by taking the average of all the round trip times and divided by two.

Both heartbeats throughput and communication latency are measured using a VM with one CPU under the RTDS and Credit schedulers. The results are shown in table II. This information is necessary for proper implementation of Pacer to ensure the impact of the overhead is negligible.

Table II: Pacer's Overhead

Scheduler	Heartbeats Throughput (beats/second)	Communication Latency (ms)
RTDS	44189	0.9444
Credit	44111	0.9587

C. Soft Real-time Application: Video Object Detection

The base application used for the experiments is written in Python and it makes use of Pythons OpenCV module. It performs real-time object recognition using MobileNets[21], an efficient neural network model designed for hardware where resources may be limited, such as mobile devices. For the application, the neural network is simply used to identify whether a person is present within a frame, using either a live feed or a video file. To vary the computational resources needed by this multi-threaded application, the rate at which frames are processed by the neural network can be varied. Three different sampling frequencies are used to create heavy, medium and light workloads for this application. The sampling frequency can be adjusted manually or automatically based on if a person is detected in the frame or not.

For this application, since we make heartbeat calls for every frame, the unit for the heart rate is defined as frame per second (FPS). The window heart rate is sent to Pacer resource manager with a window size of 12. We set the deadline to be 10 FPS for this application since any frame rate lower than 10 FPS is too slow to be recognized as video during the experiments.

Controllers' parameters are tuned for both AIMD and STPID to balance between fast convergence and overshoot. The only parameter for STPID is the target heart rate, u. Since the deadline is 10 FPS, target heart rate is set to 11 FPS. The target rate is set a little bit higher than the deadline to avoid deadline misses when STPID is converging toward steady state. As for the AIMD controller, b_u and b_l, upper and lower bound heart rates, are set to 12 FPS and 10 FPS. The additive variable, m, is set to 0.035 and the multiplicative

variable, n, is set to 0.9. The configurations for AIMD and STPID for the experiments are shown in table III and IV.

Table III: AIMD Parameter Configurations

AIMD			
Additive Variable (m)	0.035	Acceptable Upper Bound Heart Rate (b_u)	12 FPS
Multiplicative Variable (n)	0.9	Acceptable Lower Bound Heart Rate (b_l)	10 FPS

Table IV: STPID Parameter Configurations

STPID	
Target Heart Rate (u)	11 FPS

D. Experiment 1: Monitoring VMs with Different Schedulers

In this experiment, we want to evaluate Pacer's versatility through monitoring VMs that are running on different subset of cores with different schedulers.

Application:

In this experiment, we run the object detection application on both VM1 and VM2. The duration of each trial is 120 seconds. In the first 40 seconds, the application is running in heavy workload mode. The workload is switched to light workload at 40 seconds. Finally, at 80 seconds, the workload is switched to medium workload.

VM Configurations:

Pacer requires the hypervisor scheduler to have the ability to assign specific CPU utilization to the VMs. With one VM, under the RTDS scheduler, specific CPU utilization can be allocated by assigning proper period and budget. With the Credit scheduler, which is a proportional share scheduler based on weights, it is required to create a dummy VM in the same PCPU pool so specific CPU utilization for the main VM can be allocated by assigning proper weights to the dummy and the main VM. To have a fair comparison, we also create a dummy VM for RTDS scheduler. Both dummy VMs are assigned 1% CPU utilization before every trial. The complete VM configurations are shown in table V.

Methodology:

Before each trial, all four VMs are configured as shown in table V. Each trial starts with VM1 and VM2 running the object detection application with static CPU resource allocations. The same procedure is then repeated with the AIMD and STPID resource allocation

Table V: VM Configurations for Experiment 1

	PCPU Pool	# of VCPUs	Scheduler	Initial CPU Utilization
VM1	1	5	RTDS	99%
Dummy1	1	5	RTDS	1%
VM2	2	5	Credit	99%
Dummy2	2	5	Credit	1%

algorithms. For each trial, the resulting heart rates and CPU utilization assignments are recorded for analysis.

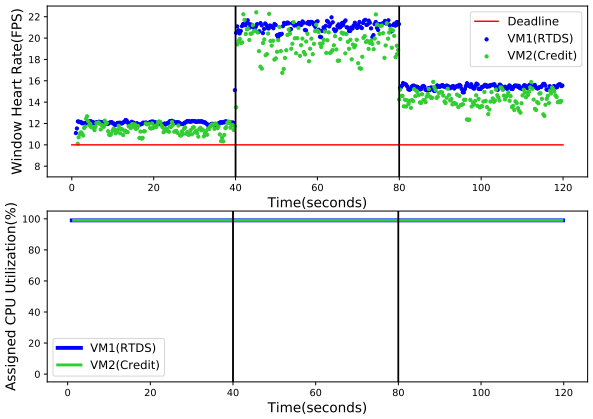


Figure 4: Static Algorithm Result for Experiment 1

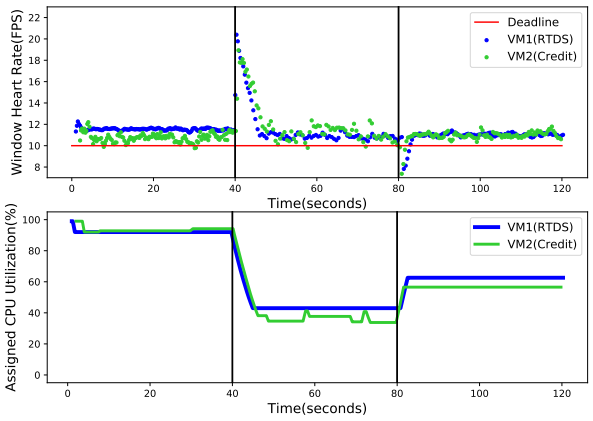


Figure 5: AIMD Algorithm Result for Experiment 1

Result & Analysis:

The experiment results are shown in figure 4, 5 and 6. In these three figures, the first vertical line indicates when the workload is manually changed from heavy to light. The second vertical line indicates when the workload is switched from light to medium. The top subplots show the heart rates, and the bottom subplots show the CPU utilization assignments. For the static allocation results shown in figure 4, although no heart rate drops below the deadline (10 FPS) for the entire 120 seconds, CPU resources can be saved without missing deadlines in the light and medium workload regions

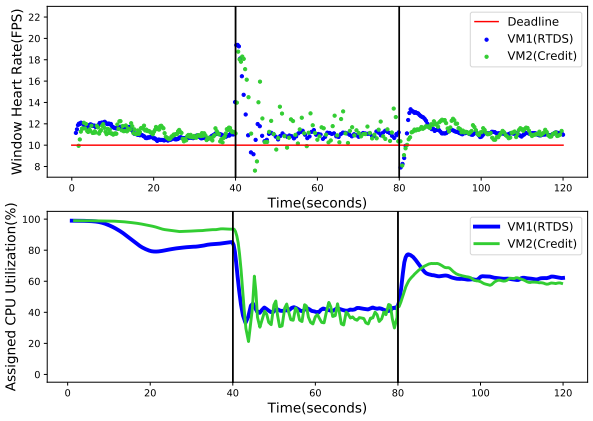


Figure 6: STPID Algorithm Result for Experiment 1

from 40 to 120 seconds. For the AIMD and STPID results shown in figure 5 and 6, both algorithms are able to lower CPU usage (making resources available for other applications) while keeping heart rates above the deadline and able to raise CPU utilization when deadline miss is detected.

Figure 7 shows the performance of the AIMD and STPID algorithms when the VM is running under the RTDS scheduler. Figure 7a shows the real-time performance of all three allocations. In figure 7a, in the first 40 seconds, all three allocations are able to meet the deadlines. Despite being under heavy workload in the first 40 seconds, AIMD and STPID are able to save 6.88% and 11.3% of CPU utilization respectively as shown in figure 7b.

Between 40 to 80 seconds, figure 7a shows that AIMD is still able to maintain perfect real-time performance, but the percentage of heart beats that meet the deadline for STPID drops slightly to 97.37% due to overshoot. Since it is under light workload, we can see from figure7b that both AIMD and STPID can achieve more than 50% of CPU utilization savings.

For the final 80 to 120 seconds shown in figure 5 and 6, the initial switch from light to medium workload causes deadline misses for AIMD and STPID, but both algorithms adjust CPU utilizations based on the feedback and are still able to meet deadlines 96.36% and 96.49% of the time as shown in figure 7a. Under medium workload, AIMD and STPID can save more than 35% CPU utilization as shown in figure 7b.

Similar performance is observed for the Credit scheduler in figure 8. Even though the Credit scheduler achieves more CPU utilization savings, its real-time performance is worse than the RTDS scheduler. Figure 4 also shows that the heart rates reported under the RTDS scheduler have less variance compared with the heart rates reported under Credit scheduler. A similar trend is observed when using other resource allocation algorithms as shown in figure 5 and figure 6.

In this experiment, we show that Pacer can work with VMs that are operating under different PCPU pools and hypervisor schedulers. We also verify that the RTDS scheduler is more suitable for real-time applications than Credit scheduler. From this observation, we use RTDS scheduler as the main hypervisor scheduler for the following two experiments.

E. Experiment 2: Monitoring Multiple VMs

In this experiment, we evaluate Pacer's ability to save CPU resources through a simulated video surveillance application and show how the saved CPU resources can be utilized for doing more work.

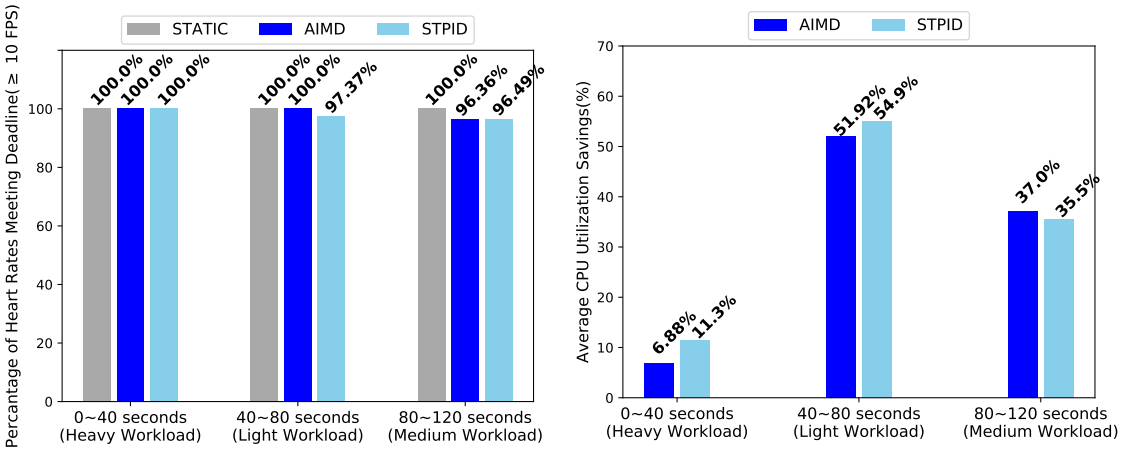
Application:

Three VMs are needed for this experiment. VM1 and VM2 run the real-time application, and VM3 runs a non-real-time application.

- Video Surveillance Application: A simulated real-time surveillance application that runs on VM1 and VM2. This soft real-time application reads in video streams and performs object detection. If a person is detected in the current frame, the sampling frequency is switched from low to medium, shifting the workload from light to medium. If no person is detected in the current frame, the sampling frequency is switched to low frequency, shifting the workload from medium to light. Each trial is run for 120 seconds. The waiting time for a person to enter the frame is drawn from an exponential distribution with $\beta = 30$ seconds, and the time for a person to stay in the frame is drawn from an exponential distribution with $\beta = 15$ seconds. The application exits when the time reaches 120 seconds. To avoid the total CPU resource demands exceeding the system's available resources, video frame size is decreased by 5% to lower the workload for both low and medium sampling frequency.
- Matrix Multiplication: A non real-time application that runs on VM3. This application executes continuous matrix multiplication with a matrix size of 500 by 500.

VM Configuration:

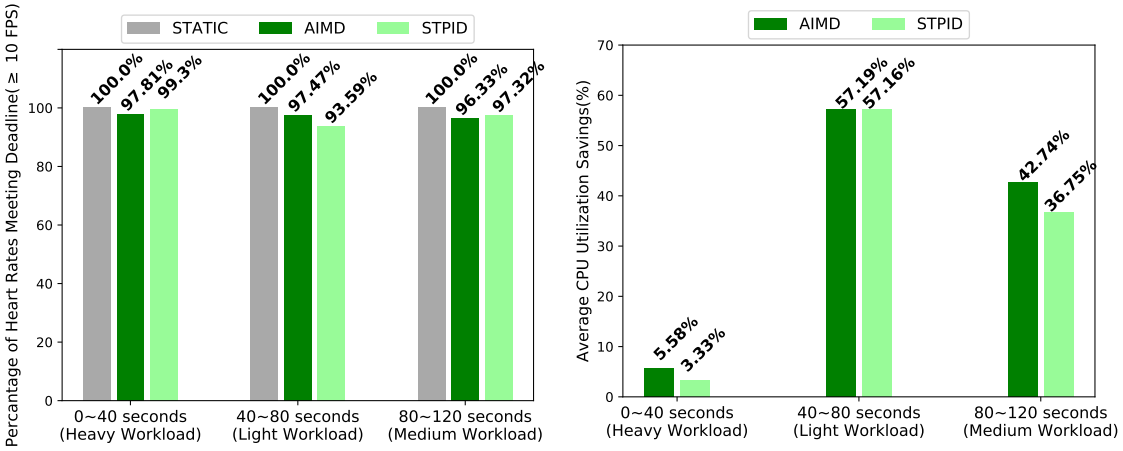
VM1, VM2 and VM3 all run under the same PCPU pool with the RTDS scheduler. To have a fair comparison between static, AIMD and STPID, we first find the static CPU utilization assignment that results in heart rates of 11 FPS for medium sampling frequency. 11 FPS is used because the average of upper and lower bound acceptable heart rates for AIMD and the target heart rate for STPID are both 11 FPS. The resulting static CPU utilization found is 40%, so VM1 and VM2 are initially assigned with 40% CPU utilization. VM3 is



(a) Real-time Performance

(b) CPU Utilization Savings

Figure 7: Performance under RTDS Scheduler



(a) Real-time Performance

(b) CPU Utilization Savings

Figure 8: Performance under Credit Scheduler

assigned with the remaining 20% CPU utilization. The VM configurations are shown in table VI.

Table VI: VM Configurations for Experiment 2

	PCPU Pool	# of VCPUs	Scheduler	Initial CPU Utilization
VM1	1	5	RTDS	40%
VM2	1	5	RTDS	40%
VM3	1	5	RTDS	20%

Methodology:

First, we generate two random sequences of timestamps of a person entering and leaving the frame separately for VM1 and VM2. We run the surveillance application in VM1 and VM2 with these two random sequences of frames. As for VM3, for each trial, it starts running the matrix multiplication application when VM1's or VM2's first heartbeat reaches the Pacer resource manager. VM3 stops and reports the number of computations completed when VM1 and VM2 notify the Pacer resource manager that they have finished running their applications. The experiment starts by running VM1, VM2 and VM3 with static CPU resource allocation. The same procedure is then repeated with the AIMD and STPID resource allocation algorithms. For each trial, the resulting heart rates and CPU utilization assignments are recorded for analysis.

Result & Analysis:

The experiment results are shown in figure 9, 10 and 11. For these three figures, the vertical lines indicate when a person is detected or when a person leaves the frame. The top subplots show the heart rates, and the bottom subplots show the CPU utilization assignments.

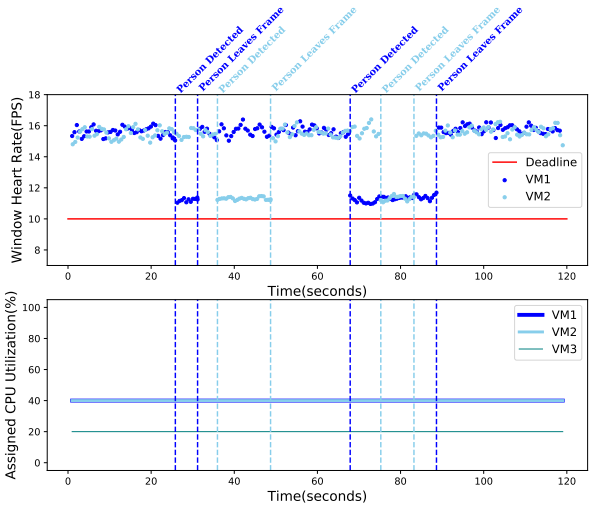


Figure 9: Static Algorithm Result for Experiment 2

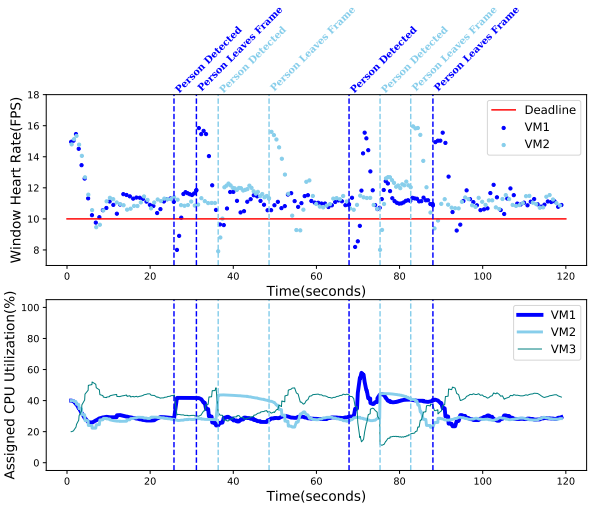


Figure 11: STPID Algorithm Result for Experiment 2

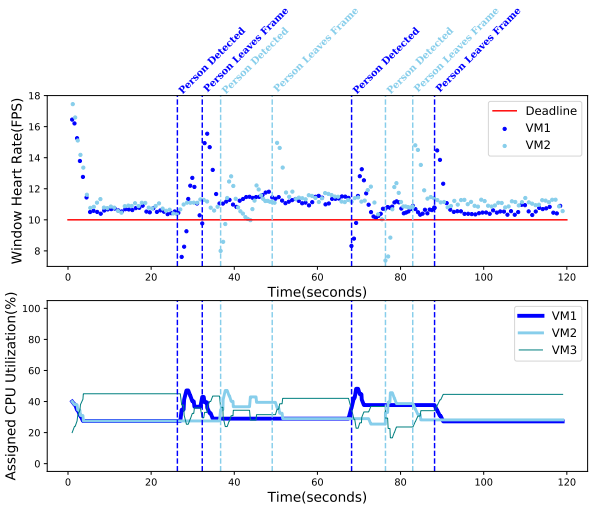


Figure 10: AIMD Algorithm Result for Experiment 2

For the static allocation shown in figure 9, VM1, VM2 and VM3 have constant CPU utilization. As for AIMD and STPID, CPU utilization assigned to VM1, VM2 and VM3 changes based on the heartbeats feedback that reflects workload variation as shown in figure 10 and 11.

Figure 12 compares the real-time performance, and figure 13 shows the computation counts of VM3 under three different resource allocation algorithms. The static algorithm is able to meet the FPS requirement 100% of the time, but has the lowest matrix computation count of 743. Under AIMD, VM3 is able to increase the computation count to 1112, a 49.67% increase compared with static allocation performance, with a cost of less than 5% of the heart rates missing deadline for both VM1 and VM2. Finally, under STPID, VM3 achieves computation counts of 1064 while more than 93% of the heart rates meet their deadline for both VMs.

In this experiment, we show that Pacer is able to handle non-deterministic workload changes and save CPU resources to perform more computations.

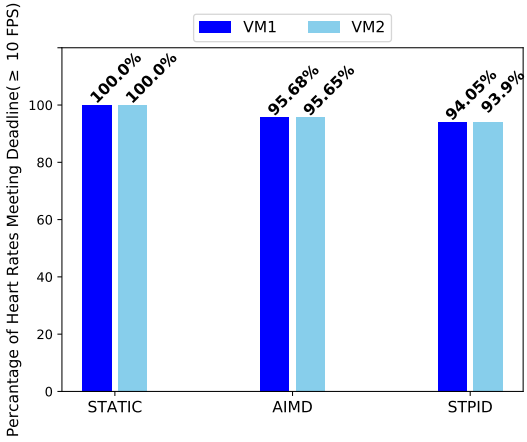


Figure 12: Real-time Performance Comparison for Experiment 2

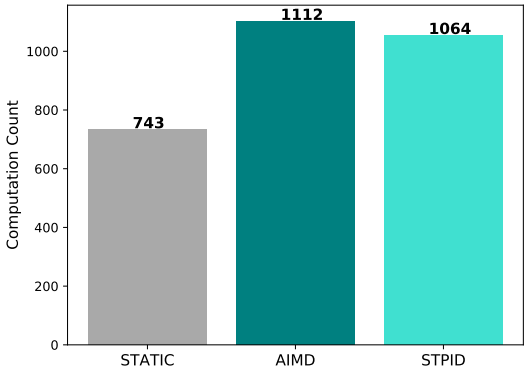


Figure 13: VM3 Computation Counts Comparison

F. Experiment 3: Monitoring VMs in an Oversubscribed System

In this experiment, we show that Pacer is able to satisfy VMs' soft real-time requirements when static allocation fails to do so. We also demonstrate that Pacer can improve real-time performance when the total resource demands from the VMs exceed the system's available resources.

Application:

Both VM1 and VM2 run the same object detection application as in experiment 2 but with deterministic sequence of frames with or without a person present as shown in figure 14. Deterministic sequences of frames are used because we want to emphasize how Pacer performs under various system loads. For this experiment, the application runs at medium sampling frequency when a person is not detected in a frame, and switches to high sampling frequency when a person is detected in a frame. To create an overloading scenario, we increase the overall workload by increasing the frame size by 5%.

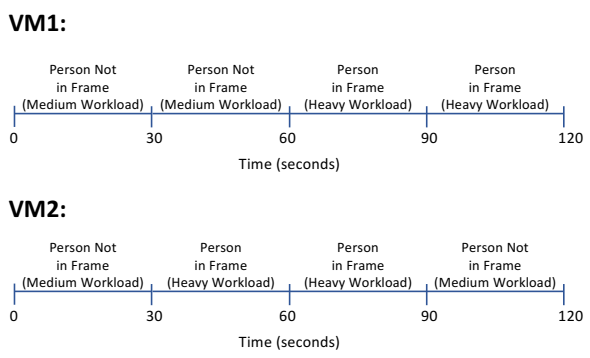


Figure 14: Deterministic Video Frames Sequences for Experiment 3

— VM Configuration:

VM1 and VM2 run in the same PCPU pool. Both VMs are initially assigned with 50% CPU utilization. The configuration is shown in table VII.

Table VII: VM Configurations for Experiment 3

	PCPU Pool	# of VCPUs	Scheduler	Initial CPU Utilization
VM1	1	5	RTDS	50%
VM2	1	5	RTDS	50%

Methodology:

Before each trial, both VMs are configured according to table VII. The experiment starts with VM1 and VM2 running with static CPU utilization assignments. The same procedure is then repeated with the AIMD and STPID resource allocation algorithms and the stride sharing algorithm. For each trial, resulting heart rates and CPU utilization assignments are recorded for analysis.

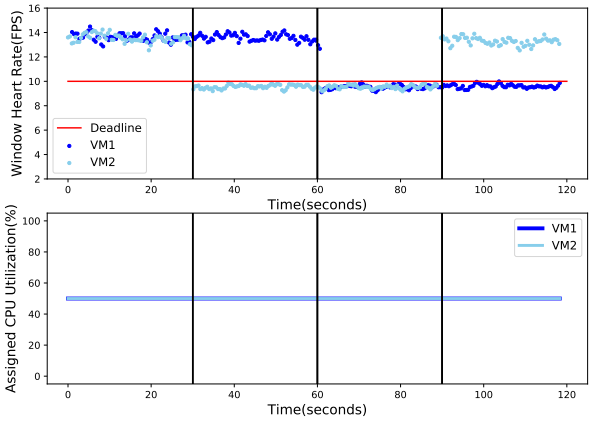


Figure 15: Static Algorithm Result for Experiment 3

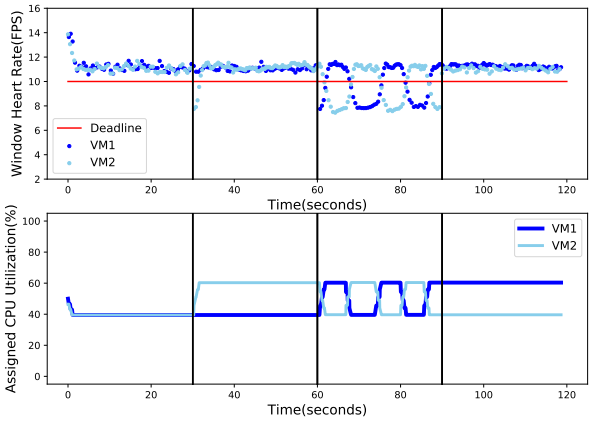


Figure 16: AIMD Algorithm Result for Experiment 3

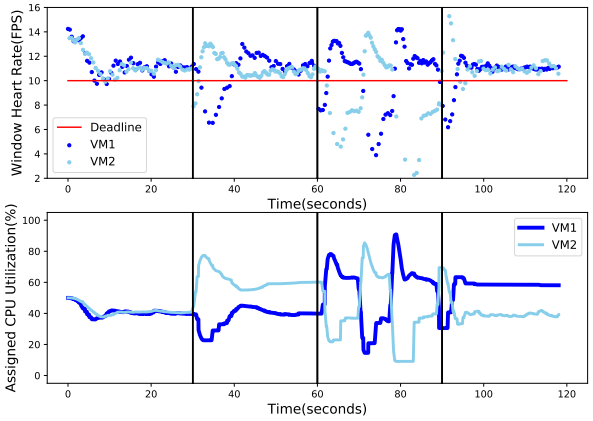
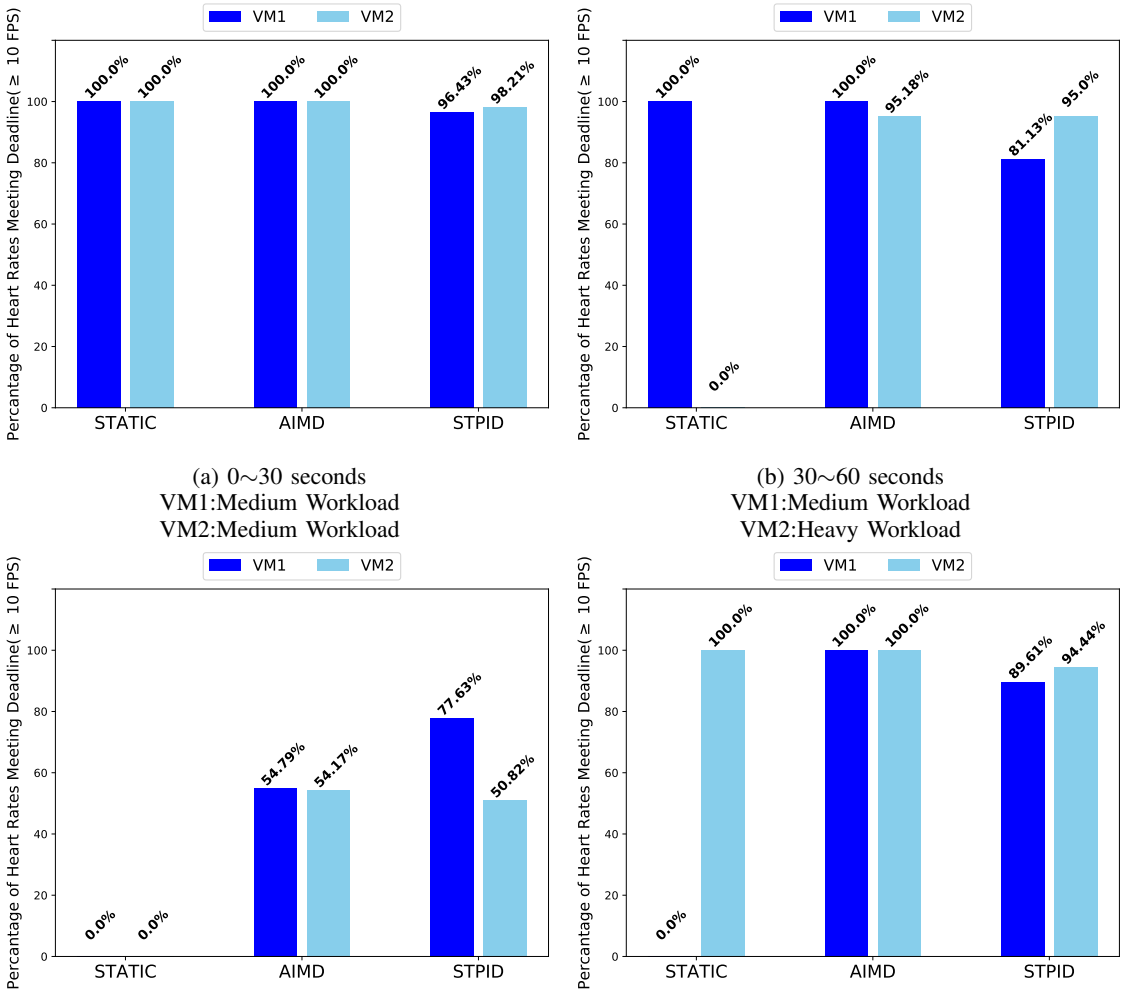


Figure 17: STPID Algorithm Result for Experiment 3

Result & Analysis:

The recorded CPU utilization assignments and heart rates data are shown in figure 15, 16 and 17. For these three figures, the first vertical line indicates when the



(a) 0∼30 seconds
VM1:Medium Workload
VM2:Medium Workload

(b) 30∼60 seconds
VM1:Medium Workload
VM2:Heavy Workload

(c) 60∼90 seconds
VM1:Heavy Workload
VM2:Heavy Workload

(d) 90∼120 seconds
VM1:Heavy Workload
VM2:Medium Workload

Figure 18: Real-time Performance Comparison in an Oversubscribed System

workload for VM2 is switched from medium to heavy. The second vertical line indicates when the workload for VM1 is switched from medium to heavy, and the last vertical line indicates when the workload for VM2 is switched from heavy to medium. The top subplots show the heart rates and the bottom subplot show the CPU utilization assignments.

The real-time performance comparison of static, AIMD and STPID is shown in figure 18. In the first 30 seconds, both VM1 and VM2 run under medium workload. Figure 18a shows that static and AIMD have no deadline misses. STPID has less than 4% missed deadlines due to overshoot as shown in figure 17.

Figure 18b shows the real-time performance comparison from 30 to 60 seconds when VM2 switches to heavy workload mode. Under static allocation, none of the VM2's heart rates can meet the deadline, but with AIMD, VM2's heart rates can meet the deadline 95.18% of the time while enabling VM1' heart rates to meet all deadlines. STPID has a slightly worse real-time performance than AIMD but is still able to keep VM1's and VM2's heart rate under the deadline 81.13% and 95.0% of the time respectively.

System overloading occurs between 60 and 90 seconds when both VMs are in heavy workload mode. Figure 18c shows that all heart rates from both VMs miss their deadlines under static allocation. With a scheduling quantum of 6 seconds and setting equal stride values for both VMs, stride sharing is able improve heart rates for both VMs to meet deadlines more than 50% of the time with AIMD and STPID as shown in figure 18c.

Figure 18d shows the real-time performance compari-

son between 90 and 120 seconds when VM2 switches to medium workload and VM1 remains in heavy workload mode. Similar behavior is observed in figure 18b but with VM1 missing all the deadlines under static allocation. On the other hand, AIMD is able to have both VMs meet deadlines 100% of the time, as shown in figure 18d. STPID has a slightly worse real-time performance than AIMD but still able to meet VM1's and VM2's heart rate deadlines 89.61% and 94.44% of the time respectively.

In this experiment, VM1's and VM2's real-time performance benefit from the Pacer dynamic CPU resource allocation. On the contrary, all the deadlines are missed for the VM running the heavier workload without Pacer. We also shows that stride sharing can improve real-time performance when all deadlines are missed under an overloaded system with static allocation.

VII. Conclusion and Future Work

In this work, we present Pacer, a Xen-based framework that monitors multiple VMS and provides developers a platform to implement their choice of CPU resource allocation algorithms that utilize feedback from soft real-time applications. We also present two resource allocation algorithms that are based on AIMD and STPID to dynamically adjust CPU utilization assignments to the VMs from their application's heartbeats feedback. Finally, we develop stride sharing, a stride scheduling based control algorithm that distributes CPU resource proportionally to improve real-time performance in an overloaded system.

Future work includes expanding Pacer to manage multi-node systems. As of now, Pacer has been implemented on nodes provisioned using OpenStack[22]. Docker containers[23] are another virtualization technology with which Pacer can be implemented. Besides deploying Pacer to distributed systems, we also plan to expand Pacer's ability to manage multiple resources such as memory and network. Finally, learning-based methods such as Q-learning and regression can be used to develop new resource allocation algorithms.

This work was supported by the Office of Naval Research grant #N00014-16-1-2887.

References

[1] M. Garca-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726 – 740, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762114001015

[2] Y. R. Yang and S. S. Lam, "General aimd congestion control," in *Proceedings 2000 International Conference on Network Protocols*, 2000, pp. 187–198.

[3] F. Lin, R. D. Brandt, and G. Saikalis, "Self-tuning of pid controllers by adaptive interaction," in *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334)*, vol. 5, 2000, pp. 3676–3681 vol.5.

[4] C. A. Waldspurger and E. Weihl. W., "Stride scheduling: Deterministic proportional- share resource management," Cambridge, MA, USA, Tech. Rep., 1995.

[5] G. P. C. Tran, Y. A. Chen, D. I. Kang, J. P. Walters, and S. P. Crago, "Automated demand-based vertical elasticity for heterogeneous real-time workloads," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 831–834.

[6] X. Guan, X. Wan, B. Y. Choi, S. Song, and J. Zhu, "Application oriented dynamic resource allocation for data centers using docker containers," *IEEE Communications Letters*, vol. 21, no. 3, pp. 504–507, March 2017.

[7] S. Sotiriadis, N. Bessis, C. Amza, and R. Buyya, "Vertical and horizontal elasticity for dynamic virtual machine reconfiguration," *IEEE Transactions on Services Computing*, pp. 1–1, 2016.

[8] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "Poet: a portable approach to minimizing energy under soft real-time constraints," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2015, pp. 75–86.

[9] C. Imes and H. Hoffmann, "Bard: A unified framework for managing soft timing and power constraints," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 31–38.

[10] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "Portable multicore resource management for applications with performance constraints," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Sept 2016, pp. 305–312.

[11] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "Caloree: Learning control for predictable latency and low energy," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 184–198. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173184

[12] G. Moltó, M. Caballer, and C. de Alfonso, "Automatic memory-based vertical elasticity and oversubscription on cloud platforms," *Future Gener. Comput. Syst.*, vol. 56, no. C, pp. 1–10, Mar. 2016. [Online]. Available: https://doi.org/10.1016/j.future.2015.10.002

[13] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, Jan 1994.

[14] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809065

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*,

vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462

[16] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling i/o in virtual machine monitors," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346258

[17] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11. New York, NY, USA: ACM, 2011, pp. 39–48. [Online]. Available: http://doi.acm.org/10.1145/2038642.2038651

[18] Y. C. Cho and J. W. Jeon, "Sharing data between processes running on different domains in para-virtualized xen," in *2007 International Conference on Control, Automation and Systems*, Oct 2007, pp. 1255–1260.

[19] pyxs. [Online]. Available: http://pyxs.readthedocs.io/

[20] xenbus. [Online]. Available: https://wiki.xen.org/wiki/XenStore_Reference/

[21] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[22] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Article: Openstack: Toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, October 2012, full text available.

[23] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241