# Capacity and Performance Overhead in Dynamic Resource Allocation to Virtual Containers

Zhikui Wang, Xiaoyun Zhu, Pradeep Padala, Sharad Singhal
Enterprise Systems and Software Laboratory
HP Laboratories Palo Alto
HPL-2007-67
April 24, 2007*

overhead,
dynamic control,
virtualization

Today's enterprise data centers are shifting towards a utility computing model where many business critical applications share a common pool of infrastructure resources that offer capacity on demand. Management of such a pool requires having a control system that can dynamically allocate resources to applications in real time. Although this is possible by use of virtualization technologies, capacity overhead or actuation delay may occur due to frequent re-scheduling in the virtualization layer. This paper evaluates the overhead of a dynamic allocation scheme in both system capacity and application-level performance relative to static allocation. We conducted experiments with virtual containers built using *Xen* and *OpenVZ* technologies for hosting both computational and transactional workloads. We present the results of the experiments as well as plausible explanations for them. We also describe implications and guidelines for feedback controller design in a dynamic allocation system based on our observations.

* Internal Accession Date Only
Presented at the 10th IFIP/IEEE Symposium on Integrated Management, 21-25 May 2007, Munich, Germany
Approved for External Publication

# Capacity and Performance Overhead in Dynamic Resource Allocation to Virtual Containers

Zhikui Wang[1] Xiaoyun Zhu[1] Pradeep Padala[2] Sharad Singhal[1]
[1]Hewlettt Packard Laboratories
Palo Alto, CA 94304, USA
{zhikui.wang, xiaoyun.zhu, sharad.singhal} @hp.com
[2]University of Michigan
Ann Arbor, MI 48105, USA
ppadala@eecs.umich.edu

***Abstract*—Today's enterprise data centers are shifting towards a utility computing model where many business critical applications share a common pool of infrastructure resources that offer capacity on demand. Management of such a pool requires having a control system that can dynamically allocate resources to applications in real time. Although this is possible by use of virtualization technologies, capacity overhead or actuation delay may occur due to frequent re-scheduling in the virtualization layer. This paper evaluates the overhead of a dynamic allocation scheme in both system capacity and application-level performance relative to static allocation. We conducted experiments with virtual containers built using *Xen* and *OpenVZ* technologies for hosting both computational and transactional workloads. We present the results of the experiments as well as plausible explanations for them. We also describe implications and guidelines for feedback controller design in a dynamic allocation system based on our observations.**

## I. INTRODUCTION

Utility computing is a new computing paradigm that has attracted a great deal of interest and support across the information technology (IT) industry. As a result, there is a trend in today's enterprise data centers to consolidate business critical applications from individual dedicated servers onto a shared pool of servers that offer capacity on demand, as illustrated in Figure 1. Each physical machine in the pool can consist of a number of virtual containers, each of which can host one or more applications. These containers can be realized using various virtualization technologies, including *HP VSE* [3], *IBM Enterprise Workload Manager* [4], *OpenVZ* [5], *Solaris Zones* [6], *VMware* [7], *Xen* [9], etc. .

On the other hand, enterprise applications typically have resource demands that vary over time due to changes in business conditions and user demands. Figure 2(a) and 2(b) show the CPU consumption of two servers in an enterprise data center for a week. Each server has 8 CPUs, and we can see that the CPU utilization for both servers is lower than 15% most of the time. Server A has a peak demand of 2.3 CPU and a mean demand of 0.2 CPU. Server B has a peak demand of 2.8 CPU and a mean demand of 0.9 CPU. Therefore, we can move the applications running on these two servers into two virtual containers hosted on a single physical machine, where each virtual container can be provisioned with its peak demand. This would require a total of 5.1 CPUs, hence a 6-way machine to host these two virtual containers. However, it is obvious that the peaks in server A and server B are not synchronized. Figure 2(c) shows the sum of the consumptions from both servers, which is always below 4 CPUs. Therefore, if a resource allocation system could allocate and de-allocate resources to each virtual container based on its need in real time, we would only need a 4-way server to host these two virtual containers.

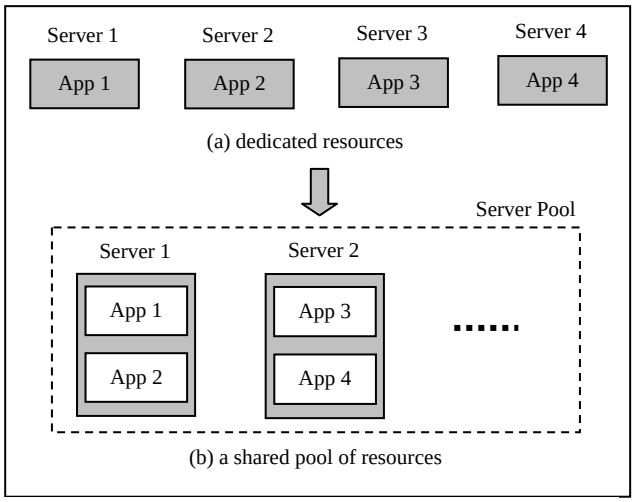


**Figure 1. Data center resource consolidation**

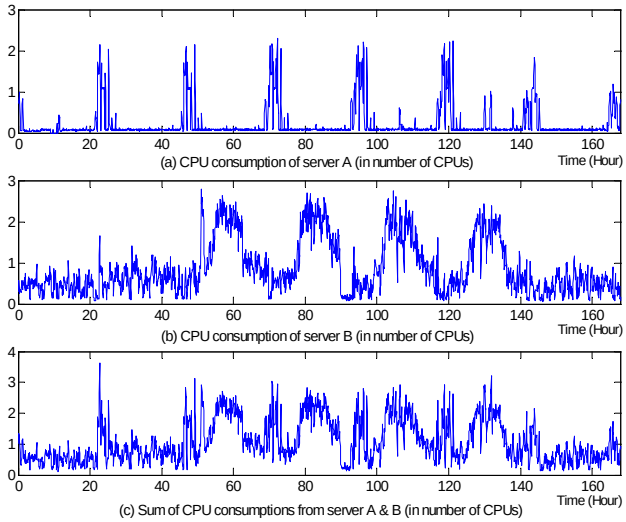


**Figure 2. An example of server CPU consumptions in a data center**

This poses new challenges to systems and application management that did not exist in dedicated environments. Because each of the hosted applications can have a resource demand that changes over short time scales (e.g., seconds or minutes), there needs to be a control system that can dynamically allocate the server's capacity to the virtual containers in real time. The benefit of doing this is that it allows statistical multiplexing between resource demands from co-hosted applications so that shared servers can reach higher resource utilization. At the same time, the control system should be responsive enough to ensure that application level service level objectives (SLOs) can be met. Such a closed-loop control system was presented in [8] to manage resource utilization of a virtual container hosting an Apache Web server.

One important element in this type of control loop is an actuator that can dynamically allocate a portion of the total capacity of certain system resource (CPU, memory, I/O bandwidth) to an individual container. Indeed, most virtualization technologies that exist today do contain a scheduler that implements some form of fair share scheduling (FSS) for CPU capacity, and provide APIs for communication with the scheduler so that resource allocation to a virtual container can be varied at run time. However, the scheduler itself may not be originally designed to handle frequent variation of resource allocation. Because re-allocation involves re-scheduling in the kernel or the hypervisor that requires extra computation, it may have the following impact on the system and the hosted applications:

- *Capacity overhead*: Loss of total capacity in the system.
- *Performance overhead*: Loss of capacity in the virtual container, hence degradation of application-level performance.

To the best of our knowledge, there has been no published work that systematically evaluates the capacity and performance overhead due to dynamic allocation of virtualized resources. The most related work is in [2] where a load balancer was designed to partition the DB2 memory pool dynamically in order to optimize the application performance. In particular, they estimated the cost of memory re-allocation and specifically incorporated the "cost of control" into the design of the controller.

This paper aims to evaluate the potential overhead of a dynamic allocation scheme relative to static allocation. We choose Xen [9] and OpenVZ [5] as representatives of the two main types of virtualization technologies today --- *hypervisor-based* (Xen) and *OS-level* (OpenVZ) virtualization. We use them to set up virtual containers for running both computational and transactional workloads. We compare the achieved system performance using either dynamic or static resource allocation, and present the results of the experiments while answering the following questions:

1. Is there a capacity or performance overhead?
2. How is the overhead related to the switching frequency as well as the switching magnitude?
3. Is the overhead comparable between different types of workloads or virtualization technologies?
4. What are plausible explanations for the overhead?

Finally, we discuss implications for tradeoffs in the feedback controller design in a dynamic allocation system based on our observations. We conclude by describing a number of future research directions.

## II. TESTBED SETUP AND EXPERIMENT DESIGN

In this section, we describe the setup of our testbed and the design of the experiments.

### A. Testbed Architecture

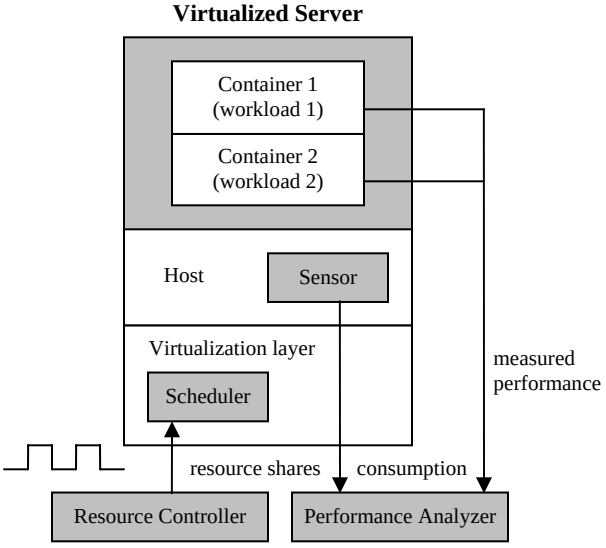


**Figure 3. Testbed architecture**

Figure 3 shows the architecture of our testbed, where a physical server is shared by two or more virtual containers, each running an independent workload. The virtualization layer contains a scheduler that can allocate a portion of the total resource capacity to each virtual container according to a specified share. In addition, a resource counter (or sensor) is running in the host that measures how much of the allocated capacity is consumed by individual containers.

A resource controller running on another computer sends SSH calls to the scheduler periodically to change the resource shares for all the virtual containers. We refer to the period when these changes occur as the *control interval*. At the same time, each workload also has a sensor that measures its performance (in terms of the amount of work done per unit time) during each control interval. A performance analyzer is also running on another computer to collect both the resource consumption and the performance measurements in order to calculate the possible capacity and performance overhead caused by frequent tuning of the resource shares.

In this paper, we focus on CPU capacity as the resource being shared by the two containers. However, the above architecture should be generally applicable to studying overhead in other resources, such as memory and I/O bandwidth.

### B. Two Virtualization Technologies

We have used the following two virtualization technologies to create virtual containers in our experiments.

*1) Xen:* Xen is a hypervisor-based paravirtualization technology [1]. It enables multiple Xen *domains*, i.e., virtual machines, to be created on a single physical server, each having its own guest OS. The Xen hypervisor provides a software virtualization layer between the guest OS and the underlying hardware. It also provides a *credit cheduler* that implements weighted fair sharing of the CPU capacity between multiple domains. This scheduler allows each domain to be allocated certain share of CPU time during a period of time. The scheduler can work in a *capped* (or non-work-conserving) or *non-capped* (or work-conserving) mode. In the former, a domain cannot use more than its share of the total CPU time, even if there are idle CPU cycles available, whereas in the latter, a domain can use extra CPU time beyond its share if other domains do not need it.

*2) OpenVZ:* OpenVZ is a Linux-based OS-level virtualization technology. It can create isolated, secure *virtual private servers* (VPSs) on a single physical server enabling better server utilization and ensuring a certain degree of isolation between applications. Each VPS performs and executes exactly like a stand-alone server. They can be rebooted independently and have root access, users, IP addresses, memory, processes, files, applications, system libraries and configuration files [5]. The OpenVZ CPU scheduler implements the FSS strategy, and each VPS's CPU share can be capped.

We refer to both a Xen domain and an OpenVZ VPS as a *virtual container*. We chose these two virtualization technologies because they are both open source projects that have a lot of developer support. And since hypervisor-based and OS-level virtualization technologies offer different levels of isolation between virtual containers, they may have different overheads associated with dynamic allocation of system resources, which is a subject of our study.

Both Xen and OpenVZ provide a uni-processor kernel as well as an SMP kernel. For our experiments in this paper, we use the SMP kernel for both types of virtualized systems so that we can allocate the full capacity of one CPU to the virtual containers (excluding the host). In addition, the capped mode of the CPU scheduler is used in either case because it provides better performance isolation between different containers.

### C. Two Types of Workloads

We studied two types of workloads in our experiments. One consists of a series of compute-intensive jobs, for which the performance metric is the average number of tasks completed per second. We wrote the job execution program in a multi-threaded fashion such that whenever CPU capacity is available, it is able to utilize it. The other is an online transaction processing workload, where a continuous stream of HTTP requests is submitted to an Apache Web server for processing. The requests are generated using a client workload generator *httperf* (ftp://ftp.hpl.hp.com/pub/httperf). For each request, the Web server executes a CGI script to generate the response and sends it back to the client. The performance metric in this case is the throughput (number of completed requests per second). Both workloads were intentionally designed to be CPU intensive so that CPU is the only potential bottleneck resource in the system. The reason for studying a transactional workload in addition to the simple computational one is that the former often has more variable performance due to the queuing behavior in the server.

### D. Experiment Design

We designed the following experiments to evaluate the potential capacity and performance overhead associated with dynamic resource allocation to virtual containers.

Using the CPU scheduler, the resource controller periodically switches the CPU share of each virtual container between a low value *El* and a high value *Eh*, where *El+Eh=1* in the case of two containers. For example, a container may be allocated 0.3 or 0.7 CPU in alternate control intervals. The shares for the two containers are exactly out of phase such that when one container receives *El*, the other receives *Eh*, and vice versa. This way they are always allocated one CPU's capacity in total. Let *T* be the control interval, then the *switching frequency* is $f = 1/T$. The exact value of the [*El*, *Eh*] pair is referred to as the *switching magnitude*. For example, [0.4, 0.6] has a smaller switching magnitude than [0.2, 0.8]. Each experiment is run for a period of 360 seconds so that short term noise is smoothed out over time. The CPU consumption of each virtual container and the workload performance are measured in every second, and are then fed into the performance analyzer for evaluation.

The experiment is repeated for different control intervals, including $T$ = 6, 10, 15, 20, 30 seconds. For every given *T*, it is also repeated for different switching magnitude, including [*El*, *Eh*] = [0.4, 0.6], [0.3, 0.7], [0.2, 0.8]. For comparison to static allocation, we also test a baseline where each container is statically allocated 0.5 CPU with no switching over time. Because a container whose allocated capacity alternates between *El* and *Eh* has effectively an average capacity of 0.5 CPU over a period of time, this baseline allows us to quantify the capacity loss due to frequent re-scheduling of the CPU. The goal of the experiments is to gain insight into whether dynamic control causes overhead, and if so, how the overhead is related to both the switching frequency and magnitude.

The experiments are run on four HP Proliant DL385 G1 servers, each with two 2.6GHz AMD Opteron processors and 8GB memory. The OpenVZ node uses an OpenVZ-enabled 2.6.9 SMP kernel. The Xen node uses a Xen-enabled 2.6.16 SMP kernel. Both OpenVZ and Xen virtual containers use the stock Fedora core images. On the Xen node, Dom-0 has access to the capacity of both CPUs, and its overhead is measured independently of work done in the virtual containers. We test two modes of operation for the other Xen domains. In the *pinned* mode, we pin all the domains to one physical CPU using an API provided by Xen, and no such restriction is in place in the *unpinned* mode. OpenVZ does not have the pinning capability, therefore only the unpinned mode is used.

## III. EXPERIMENTAL RESULTS

### A. *Scenario I: Computational Workloads in Xen Containers*

In this section, we focus on using Xen virtual containers for running computational workloads. We test four configurations using two options: two vs. four containers, and pinned vs. unpinned mode for the CPU scheduler.

*1) Two-container, unpinned:*

First, we present experimental results from running two Xen containers (Dom-1 and Dom-2) in the unpinned mode. In this case, both containers can use idle cycles from both CPUs up to their caps, but the sum of the caps is maintained at one CPU. The results are shown in Figure 4-8.

Figure 4 shows an example of the results for T = 30 seconds and [El, Eh] = [0.3, 0.7]. Figure 4(a) and 4(b) show the control variable (CPU allocation determined by the controller), CPU consumption and workload performance (number of completed tasks per second) for Dom-1 and Dom-2, respectively. Figure 4(c) shows the CPU consumption for Dom-0. Note that all these metrics are sampled at 1-second intervals, and that the CPU control and consumption numbers are multiplied by 1000 (applicable to similar figures later on). The legend in Figure 4(c) applies to all the three subfigures.

For each container, there is a small delay (1-2 seconds) in switching from low CPU consumption to high consumption when the allocation shifts from 0.3 to 0.7, and vice versa. With some measurement noise, the workload performance oscillates in proportion to the CPU usage. At the same time, we see a clear spike in Dom-0 CPU usage (from roughly 0.04 to 0.25 CPU) whenever switching occurs, indicating a loss in available capacity due to dynamic re-allocation of the CPU resource.

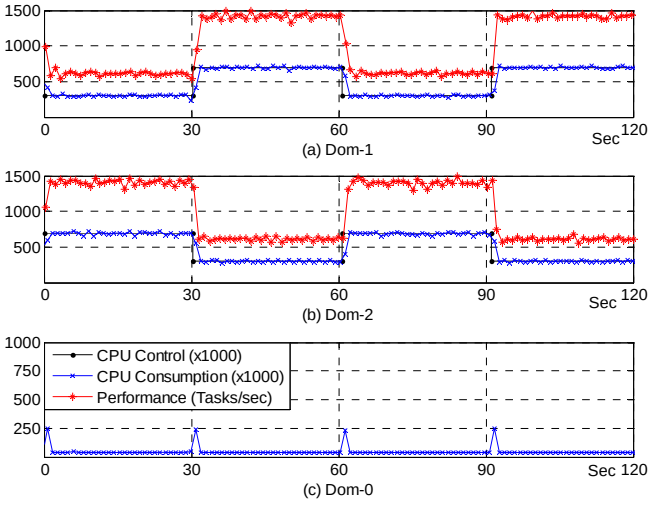


**Figure 4. CPU consumption and workload performance for T = 30s, and [El, Eh] = [0.3, 0.7] (Xen, comp., 2-container, unpinned)**

Another example is shown in Figure 5 for the same switching magnitude but a control interval of 6 seconds. The same observations can be made on the CPU consumption and workload performance for the two containers. The spike in Dom-0 CPU usage has about the same magnitude, but occurs more frequently due to a shorter control interval.

Figure 6(a) and 6(b) show the measured average performance for Dom-1 and Dom-2, respectively, as the switching frequency changes from 1/30 to 1/6. The four curves correspond to the baseline (0.5) and [El, Eh] = [0.4, 0.6], [0.3, 0.7], [0.2, 0.8], respectively. As we can see, except for a few outliers, the switching frequency does not have a significant impact on the workload performance. However, the performance is worse with higher switching magnitude. Figure 6(c) shows the sum of the measured performance from both containers. The performance overhead due to dynamic allocation increases with the switching magnitude and reaches 3% ((2038 – 1980)/2038) of the baseline performance when [El, Eh] = [0.2, 0.8].

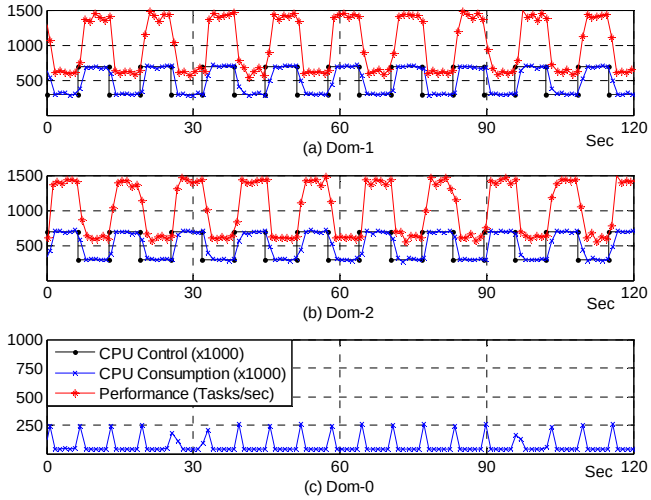


**Figure 5. CPU consumption and workload performance for T = 6s, and [El, Eh] = [0.3, 0.7] (Xen, comp., 2-container, unpinned)**

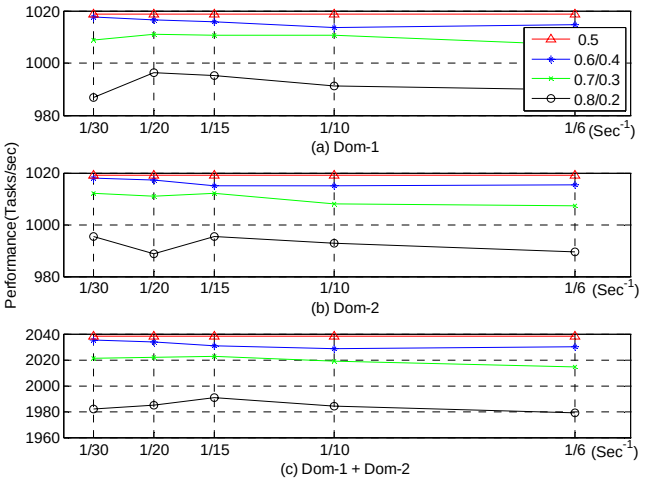


**Figure 6. Workload performance vs. switching frequency for both containers (Xen, comp., 2-container, unpinned)**

Figure 7 shows the corresponding results for the CPU consumption. Figure 7(a) and 7(b) show the average CPU consumption for the two individual containers, and Figure 7(c) shows the total CPU consumption from both containers. Similar to workload performance, we observe that the loss in capacity due to dynamic allocation goes up as the switching magnitude increases, but there is no clear impact by the switching frequency. With static allocation, both containers

together can consume close to 100% of one CPU's capacity. With dynamic allocation switching between 0.2 and 0.8 CPU, the two containers can only consume a total of 97% of one CPU, resulting in a capacity loss of 3%, consistent with the workload performance loss we observed from Figure 6.

In Figure 7(d), Dom-0 CPU usage is shown for different test conditions. As we can see, Dom-0 consumes an average of 0.042 CPU even with static allocation, due to basic virtualization overhead and sensor overhead. With dynamic allocation, as the switching frequency increases from 1/30 to 1/6, Dom-0 consumption grows linearly from 0.049 to 0.077 CPU, reaching a maximum overhead of 0.035 CPU compared to the static case for T = 6 seconds. This result is consistent across different switching magnitudes, which is why we only see one line for all the three magnitudes. This overhead has two potential sources: processing of the SSH call from the controller and processing of the two API calls (one per container) to the scheduler. We ran a separate experiment where the controller submits only SSH calls to Dom-0 without re-allocation. The resulting Dom-0 CPU consumption is also shown in Figure 7(d) (middle line). We can see by comparison that approximately 30% of the capacity loss in dynamic allocation is due to processing of the SSH call, and the remaining 70% is due to re-allocation. In both cases, the overhead seems fairly constant at each occurrence, resulting in a linear relationship between the overall observed overhead for a given period of time and the switching frequency.

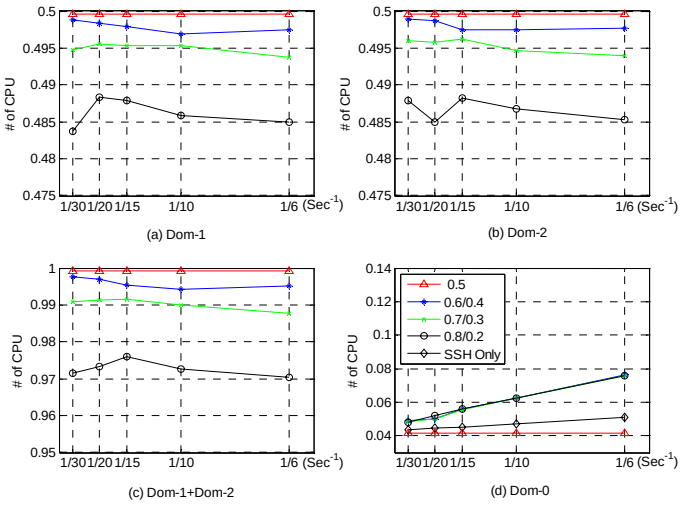


**Figure 7. CPU consumption vs. switching frequency for all the containers (Xen, comp., 2-container, unpinned)**

To explain why performance overhead increases with the switching magnitude, we show in Figure 8 the control variable, CPU consumption and workload performance for Dom-1 for a period of 60 seconds, for different switching magnitudes. It seems that that loss of average capacity to the container (therefore performance degradation) is due to a slower response of the container consumption as the allocation increases compared to when it decreases, shown as asymmetry in the front and back edges of the square-waves in the figure. For example, Figure 8(d) corresponds to the switching magnitude of [0.2, 0.8], where this asymmetry is the most visible. This is consistent with our earlier observation that performance overhead is the highest at this switching magnitude.

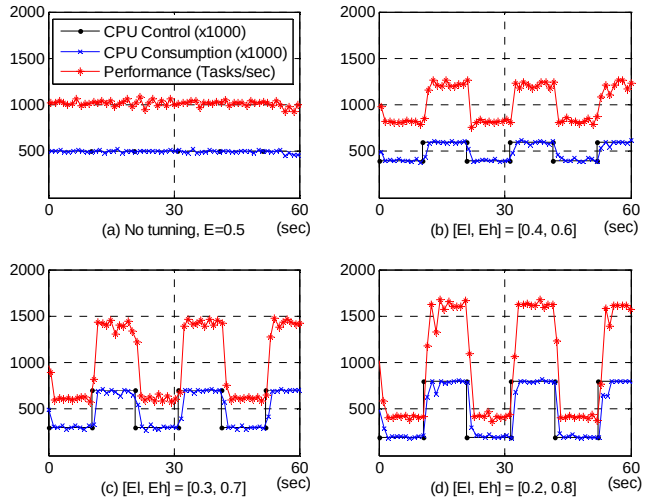


**Figure 8. Dom-1 CPU consumption and workload performance for T=10s at different switching magnitudes (Xen, comp., 2-container, unpinned)**

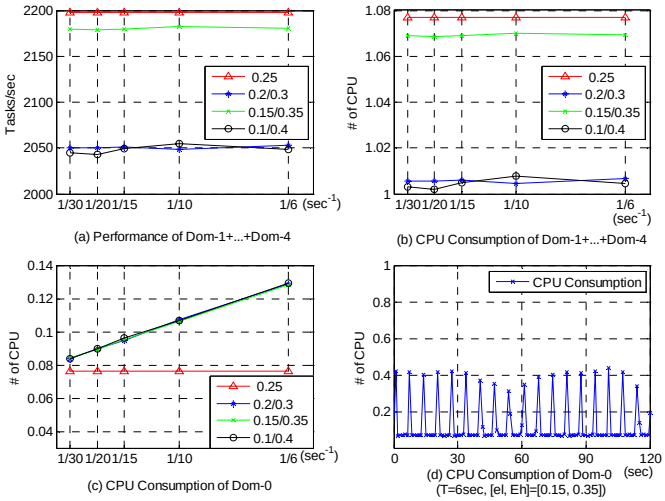


**Figure 9. Workload performance and CPU consumption vs. switching frequency for all the containers (Xen, comp., 4-container, unpinned)**

*2) Four-container, unpinned:*

We run similar experiments using four Xen containers (Dom-1 to Dom-4) in the unpinned mode, and the results are summarized in Figure 9. Note that with four containers, the baseline is when each container is allocated with a fixed capacity of 0.25 CPU, and the three switching magnitudes tested become [0.2, 0.3], [0.15, 0.35], [0.1, 0.4], respectively. Figure 9(a) shows total workload performance from the four containers as a function of switching frequency for different switching magnitudes. The average performance still does not depend on the switching frequency. It does change with the switching magnitude, but the relationship is no longer monotonic, as in the 2-container case. The observed performance overhead is the largest at 6.7% ((2195-2050)/2195) of the baseline performance for a magnitude of either [0.2, 0.3] or [0.1, 0.4]. Figure 9(b) shows the total CPU consumption of all the containers. At the baseline, although each container is statically allocated 0.25 CPU, the total consumption from the four containers exceeds one CPU by almost 8%. This indicates that capping of per-container

consumption is not strictly enforced in the unpinned mode of the CPU scheduler when there are more containers. Similar to workload performance, the capacity loss due to dynamic allocation is the maximum at 6.7% of the baseline consumption for a magnitude of either [0.2, 0.3] or [0.1, 0.4].

Figure 9(c) shows Dom-0 CPU consumption as a function of switching frequency. The behavior is similar to that for the 2-container case, as shown in Figure 7(d). The differences are: (i) Baseline consumption for the 4-container case (0.075 CPU) is higher than that for the 2-container case (0.042 CPU); (ii) Capacity overhead due to dynamic allocation for the 4-container case reaches 0.055 (0.13 – 0.075) CPU for T = 6 seconds, higher than the overhead for the 2-container case (0.035 CPU). Figure 9(d) shows an example of the Dom-0 CPU consumption over time for T = 6 seconds and [El, Eh] = [0.15, 0.35]. We can see that the spikes in Dom-0 consumption during re-allocation have a higher magnitude (from 0.075 to roughly 0.4 CPU) than that for the 2-container case. This is because the controller has to make four instead of two API calls to the CPU scheduler during each re-allocation. These observations are consistent with our expectation that both performance and capacity overheads of dynamic allocation grows with the number of virtual containers.

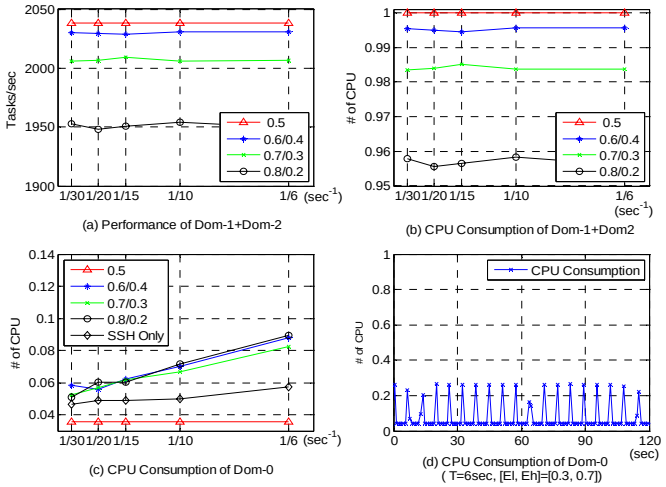


**Figure 10. Workload performance and CPU consumption vs. switching frequency for all the containers (Xen, comp., 2-container, pinned)**

*3) Two-container, pinned:*

We then repeat the 2-container experiments using the pinned mode of the CPU scheduler where both containers are pinned to a particular physical processor. Figure 10 shows the resulting total workload performance and CPU consumption from all the containers as a function of switching frequency for different switching magnitudes as well as an example of Dom-0 CPU consumption for T = 6 seconds and [El, Eh] = [0.3, 0.7]. The results are similar to the unpinned case. The differences are: (i) The average loss in workload performance (Figure 10(a)) or container capacity (Figure 10(b)) reaches a maximum of roughly 4% for [El, Eh] = [0.2, 0.8], slightly higher than the 3% in the unpinned case; (ii) Overhead in Dom-0 CPU consumption (Figure 10(c)) reaches a maximum of roughly 0.054 CPU at T = 6 seconds, higher than the 0.035 CPU overhead in the unpinned case.

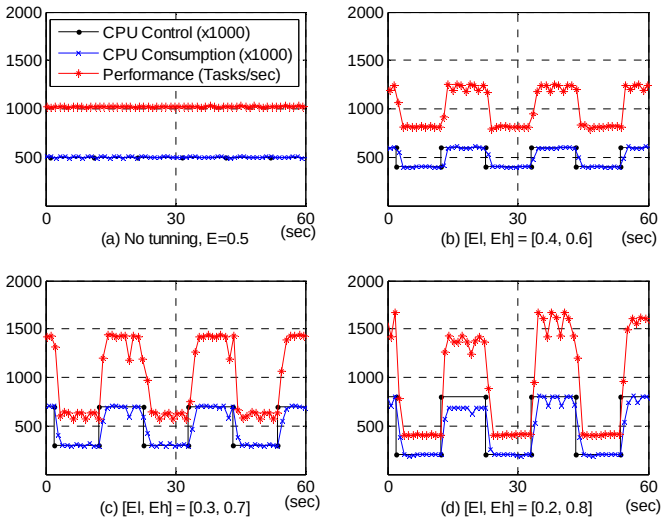


**Figure 11. Dom-1 CPU consumption and workload performance for T = 10s at different switching magnitudes (Xen, comp., 2-container, pinned)**

Similar to Figure 8, Figure 11 provides an explanation for why the loss in container capacity (and workload performance) increases with the switching magnitude. For example, with the pinned mode of the CPU scheduler, when the CPU capacity allocated to a container jumps from 0.2 to 0.8 CPU, it sometimes cannot fully utilize the allocated capacity, as can be seen from the second square-wave in Figure 11(d), which is smaller than the other square-waves. This causes the overall CPU consumption per container with dynamic allocation to be lower than the baseline. By repeating the experiment multiple times and examining the time domain traces for the three switching magnitudes, we observe that this under-utilization occurs more frequently with a higher switching magnitude, therefore resulting in higher loss in workload performance.

*4) Four-container, pinned:*

Finally, we test the configuration with four Xen containers using the pinned mode of the scheduler, and the results are shown in Figure 12. In this case, there is little overhead in either total workload performance or total container CPU consumption due to dynamic allocation, except for [El, Eh] = [0.15, 0.35]. For this particular magnitude, the average loss in workload performance (Figure 12(a)) or container capacity (Figure 12(b)) is roughly 2% of the baseline, independent of the switching frequency. Overhead in Dom-0 CPU consumption (Figure 12(c)) reaches a maximum of 0.06 CPU at T = 6 seconds, only slightly higher than the 2-container, pinned case, or the 4-container, unpinned case.

We provide some intuition behind why the loss in container capacity (and workload performance) is the highest for [El, Eh] = [0.15, 0.35] in the 4-container, pinned case in Figure 13. As can be seen from Figure 13(c), when a container is allocated 0.35 CPU, it is not able to fully utilize the allocated capacity during some intervals, causing loss in workload performance. This behavior is not seen for [El, Eh] = [0.2, 0.3] or [0.1, 0.4]. We have repeated the experiment multiple times, and the results are similar. Finding a source for this anomalous behavior would require further understanding of the scheduler implementation, which is outside the scope of this paper.

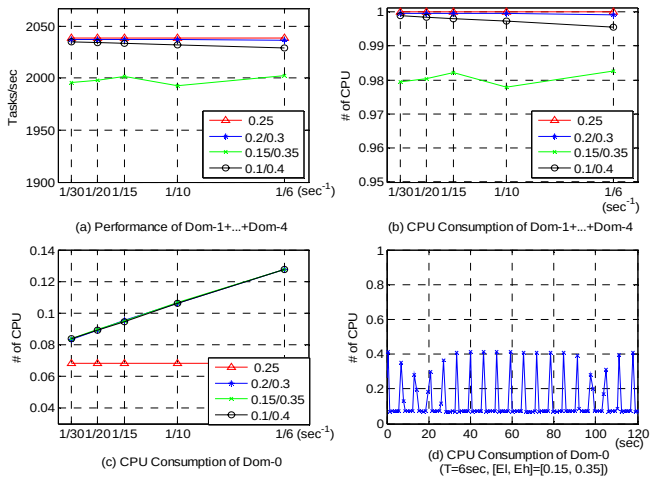


**Figure 12. Workload performance and CPU consumption vs. switching frequency for all the containers (Xen, comp., 4-container, pinned)**

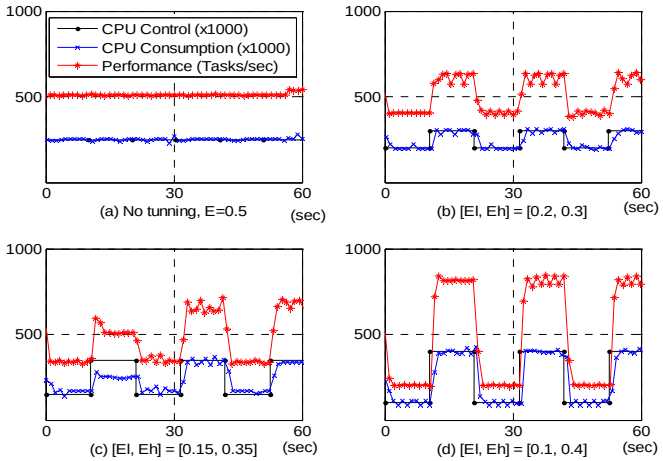


**Figure 13. Dom-1 CPU consumption and workload performance for T = 10s at different switching magnitudes (Xen, comp., 4-container, pinned)**

### *B. Senario II: Transactional Workloads in Xen Containers*

In this section, we present the results from experiments of two Xen virtual containers (Dom-1 and Dom-2), each running a transactional workload, as shown in Figure 14-17.

Figure 14(a) and 14(b) show the total workload throughput (requests per second) and CPU consumption from both containers as a function of switching frequency for different switching magnitudes. We see that losses in both metrics due to dynamic allocation increase more or less with the switching frequency. The total capacity loss also increases with the switching magnitude as shown in Figure 14(b). We do not see the same relationship between the performance loss and the switching magnitude from Figure 14(a). In the most significant case, the total performance loss is 13% (170-148)/170) of the baseline performance for T = 6 seconds and [El, Eh] = [0.2, 0.8], whereas the total capacity loss is only about 3% for the same configuration. It suggests that average performance loss of a transactional workload is no longer proportional to average capacity loss of the virtual container hosting the workload.

Figure 14(c) shows an average consumption of 0.07 CPU in Dom-0 for static allocation. Dom-0 capacity overhead due to dynamic allocation has similar behavior as that for the computational workload, where it is a linear function of the switching frequency, but is independent of the switching magnitude. The maximum overhead observed is roughly 0.04 (0.11 – 0.07) CPU for T = 6 seconds. Figure 14(d) shows spikes in Dom-0 CPU usage when capacity re-allocations occur similar to those observed in Scenario I, with a peak consumption of about 0.3 CPU. Compared to Figure 7(d) for the case of the computational workload, Dom-0 consumes more CPU when the containers are hosting the transactional workloads, for both static and dynamic allocations. This additional capacity overhead is due to more intensive I/O operations in Dom-0 for servicing the transactions.

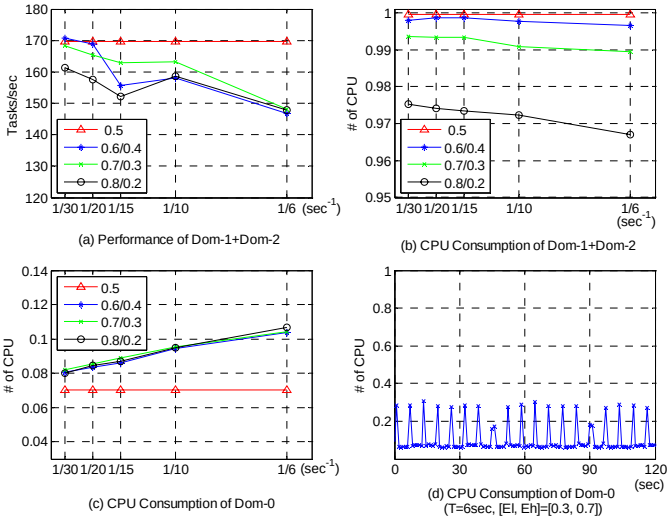


**Figure 14. Workload performance and CPU consumption vs. switching frequency for all the containers (Xen, trans., 2-container, unpinned)**

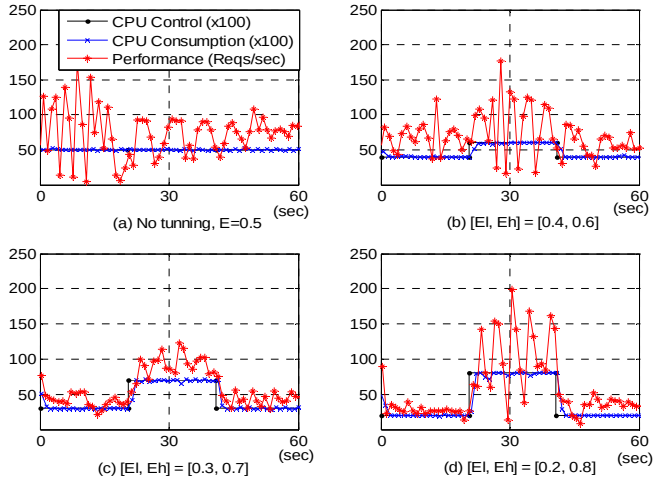


**Figure 15. Dom-1 CPU consumption and workload performance for T = 20s at different switching magnitudes (Xen, trans., 2-container, unpinned)**

Figure 15 offers some explanation for the discrepancy observed between Figure 14(a) and 14(b), by showing the time series of the CPU allocation, CPU consumption and workload performance for one container for different switching magnitudes. We see much higher variability in the throughput of the transactional workload compared to that of the computational workload, especially when the allocated capacity is above 0.5 CPU. Even for a fixed capacity as in

Figure 15(a), the throughout is oscillating heavily, yet the CPU consumption is much more stable and matches the CPU allocation most of the time. As a result, charactering workload performance for a transactional workload requires more metrics (e.g., variance) in addition to its mean.

Figure 16 shows the performance and capacity losses when the two Xen containers are pinned to one physical CPU. Compared with the unpinned case shown in Figure 14, both losses are increasing with the switching magnitudes, but neither has an explicit dependency on the switching frequency. In the most significant case where T = 10 seconds and [El, Eh] = [0.2, 0.8], the performance loss due to dynamic allocation is about 24% (170-130)/170) of the baseline performance, as shown in Figure 16(a). This is consistent with the capacity loss of about 23% (1-0.77) as seen in Figure 16(b). The Dom-0 CPU overhead shown in Figure 16(c) and 16(d) are comparable to that for the pinned case.

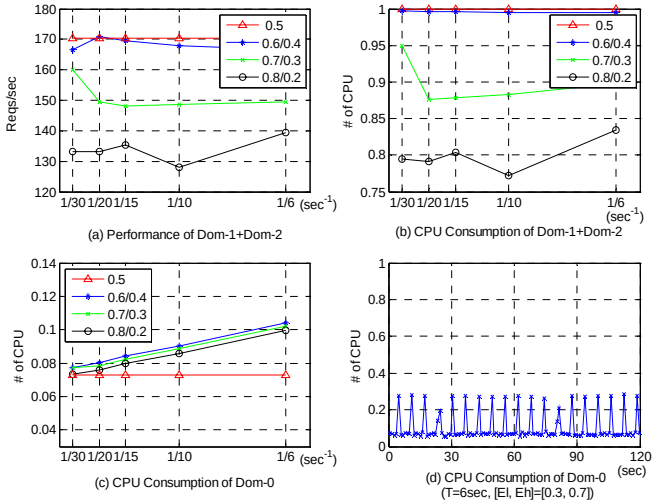


**Figure 16. Workload performance and CPU consumption vs. switching frequency for all the containers (Xen, trans., 2-container, pinned)**

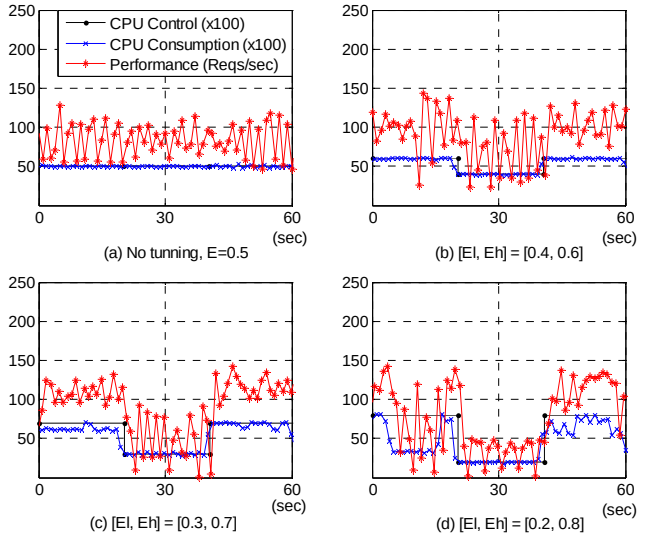


**Figure 17. Dom-1 CPU consumption and workload performance for T = 20s at different switching magnitudes (Xen, trans., 2-container, pinned)**

Figure 17 shows why the performance loss is more consistent with the capacity loss in the pinned case than in the unpinned case when the switching magnitude changes. For fixed allocation (Figure 17(a)) or a small switching magnitude (Figure 17(b)), the CPU consumption of a container matches the CPU allocation almost exactly. However, for larger switching magnitudes as in Figure 17(c) and 17(d), a container is not always able to consume all of its allocated capacity. This issue is consistently observed when the containers are pinned to one physical CPU and each container's capacity is capped, either with the computational workload (Figure 11(d)) or the transactional workload (Figure 17(d)). This behavior has to do with the specific implementation of the fair share scheduler. With the capped and pinned mode of the credit scheduler, when the capacity of the shared CPU is used up by all the containers, it may be difficult for the scheduler to add capacity to one container. We have experimented with different CPU schedulers in Xen. With the SEDF scheduler included in earlier Xen releases, we notice that total CPU consumption by all the virtual containers is capped at 95% whenever the capped mode is enabled. The 5% of CPU capacity might have been reserved for handling extra overhead, in which case we do not see the under-utilization phenomenon as seen with the credit scheduler that does not reserve this 5% capacity. More experimentation and further analyses are needed to validate these arguments.

### *C. Scenario III: Computational Workloads in OpenVZ Containers*

In this scenario, we test two OpenVZ containers (VPS1 and VPS2), each running a computational workload. The results are shown in Figure 18-19. Note that we use the capped and unpinned mode for the CPU scheduler.

Figure 18(a) shows the sum of the workload performance from the two containers as a function of switching frequency. We observe some loss in performance with dynamic allocation relative to static allocation. For dynamic allocation, the total performance goes up as either the switching frequency or the switching magnitude increases. This is the opposite of what we expect in either case, and is different from what we observe in the Xen system. The maximum loss in performance is observed for T = 30 seconds and [El, Eh] = [0.4, 0.6], where it is only 0.5% ((2072.5-2062.5)/2072.5) of the baseline, which is much smaller than that in the Xen case.

Figure 18(b) shows the total CPU consumption from the two containers. In all cases, the two VPSs together can consume slightly more than 100% of the one CPU's capacity, possibly due to noise in the sensor or inaccuracy of the scheduler. The total consumption goes up linearly with the switching frequency, and decreases slightly as the switching magnitude increases. For most conditions tested, the total CPU consumption is higher with dynamic allocation than in the static case. This behavior is not totally consistent with what we see in workload performance, suggesting that not all of the consumed CPU capacity is used for workload processing.

Figure 18(c) shows CPU consumption by system processes (measured by subtracting consumption of the two containers from total consumption of the server). Similar to the Xen case, the loss of capacity due to dynamic allocation goes up linearly with the switching frequency, and is slightly higher for a larger switching magnitude. The line marked "SSH only" indicates

that roughly half of that lost capacity is used for processing SSH calls from the controller, and the remaining half is used for re-allocating CPU capacity between the two containers.

Figure 18(d) provides some intuition for why the total CPU consumption by both containers increases linearly with the switching frequency. For T = 6 seconds and [El, Eh] = [0.3, 0.7], we see a spike in the total consumption whenever re-allocation occurs. And this is observed for all switching frequencies and magnitudes. Similar to our observation from Figure 18(b), it seems to imply that a small portion of the container-consumed capacity is used for processing related to re-allocation of CPU capacity.

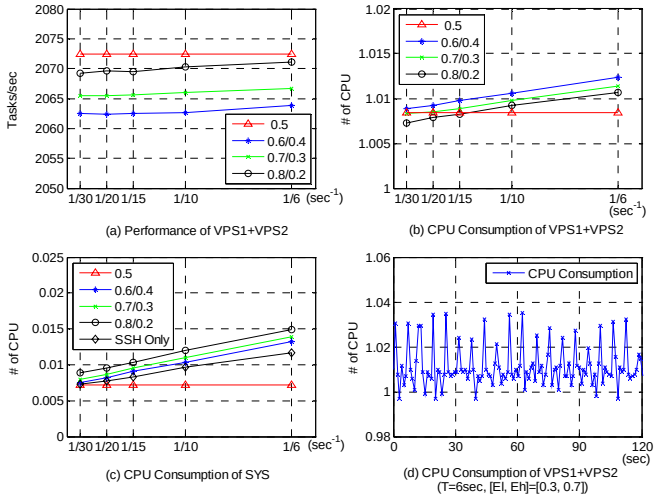


**Figure 18. Workload performance and CPU consumption vs. switching frequency for all the containers (OpenVZ, comp., 2-container)**

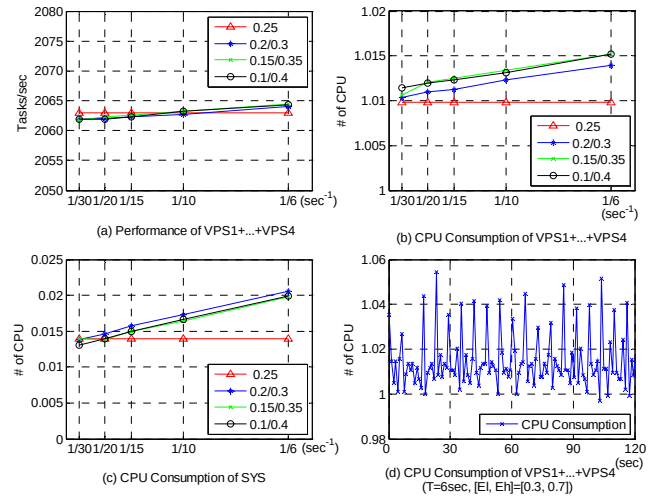


**Figure 19. Workload performance and CPU consumption vs. switching frequency for all the containers (OpenVZ, comp., 4-container)**

Figure 19 shows similar results from running four OpenVZ containers with computational workloads. Compared to the 2-container case, the differences are: (i) With both static and dynamic allocations, the total workload performance from all the containers (Figure 19(a)) is lower in the 4-container case, and the system-level CPU consumption (Figure 19(c)) is higher in the 4-container case, both indicating a higher virtualization overhead with more containers; (ii) We no longer see performance overhead due to dynamic allocation in the 4-container case (Figure 19(a)); (iii) The relationship between the total CPU consumption by the virtual containers and the switching magnitude is no longer monotonic, and the extra consumption in the containers due to re-allocation is slightly higher in the 4-container case (Figure 19(b and 19(d))); (iv) For dynamic allocation, the system-level CPU consumption is almost constant across different switching magnitudes for a given control interval (Figure 19(c)). We do not yet have explanations for all of the observed differences. However, for OpenVZ, neither performance overhead nor capacity overhead due to dynamic allocation is significant to be of major concern.

### *D. Scenario IV: Transactional Workloads in OpenVZ Containers*

We repeat the same experiments with two OpenVZ virtual containers, each running a transactional workload. The results are summarized in Figure 20.

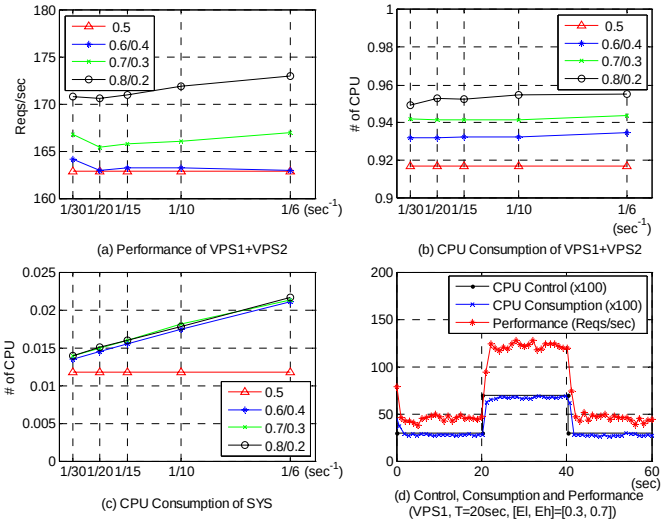


**Figure 20. Workload performance and CPU consumption vs. switching frequency for all the containers (OpenVZ, trans., 2-container)**

Figure 20(a) shows the total throughput from the two Web servers in VPS1 and VPS2. It is an increasing function of the switching magnitude, but it does not depend explicitly on the switching frequency. Figure 20(b) shows the total CPU consumption of the two containers, which shows similar dependencies on the switching frequency and magnitude. For all the switching frequencies and magnitudes tested, the maximum CPU consumption of both containers is about 0.955 CPU, 4.5% lower than the allocate capacity of one CPU. To explain why this happens, Figure 20(d) shows the time series of CPU allocation, consumption and workload throughput for T = 6 seconds and [El, Eh] = [0.3, 0.7]. We can see that the CPU consumption is slightly below the allocation most of the time. The containers with static allocation consume the least capacity (0.915 CPU). For some reason, larger switching magnitudes allow the containers to use more CPU capacity, resulting in higher workload throughput. In the most significant case where T = 6 seconds and [El, Eh] = [0.2, 0.8], the two containers together consume 4% more CPU compared to the static case, while the total workload throughput increases by 6% ((173-163)/163) over the baseline.

Compared to the case of the computational workload shown in Figure 18(c), OpenVZ system processes consume an additional 0.005 (0.0125 – 0.0075) CPU for the transactional workload as shown in Figure 20(c). This is not surprising due to more I/O operations for the transactional workload. Compared to the results from the Xen system as shown in Figure 14, the system processes in the OpenVZ case consume much less capacity (0.01-0.02 CPU) than Dom-0 does in the Xen case (0.07-0.11 CPU).

## IV. CONSLUSION AND IMPLICATION FOR CONTROLLER DESIGN

In conclusion, we have made the following observations on the impact of dynamic resource allocation in a virtualized server, as well as on their implications for the design of a resource control system.

1. We have observed both degradation of workload performance as well as loss of system capacity due to dynamic allocation of CPU capacity.
2. Both performance and capacity overheads are higher in the Xen system than in the OpenVZ system. The overheads in the OpenVZ system (below 1% over the static case) are not significant to be of major concern.
3. In the Xen system, both performance and capacity overheads are higher for transactional workloads than for computational workloads. The former is due to higher variability in performance of a transactional workload for a given capacity. The latter is because of more intensive network I/O operations for servicing the transactions.
4. Performance overhead in the Xen system is a function of the switching magnitude for the computational workload, and is a function of the switching frequency for the transactional workload. It ranges between 0-13% of the baseline performance for all the frequencies and magnitudes tested.
5. Capacity overhead in either Xen Dom-0 or OpenVZ kernel grows linearly with the switching frequency, and is below 7% of a CPU for all the configurations tested. Note that this is on top of the existing virtualization overhead, and we have seen that it goes up with the number of containers.

Both performance and capacity overheads result in tradeoffs between quicker controller response and less performance or capacity loss. A dynamic resource controller needs to be aware of these tradeoffs and to choose design parameters (e.g., control interval, actuator bounds) accordingly.

## V. FUTURE WORK

In general, our experience with the virtualization technologies for data center consolidation indicates that there is a lot to be improved in these technologies. We would like more flexible and powerful knobs to control resource allocation to each virtual container. More work needs to be done to improve accuracy and reduce overheads in sensors and actuators. Our dynamic control paradigm requires the designers of these technologies to re-think certain code-paths (e.g., scheduling code) to optimize for control at a higher frequency.

Moreover, the following problems are of particular interest to us in our future research.

First, as indicated in Section IV, we currently do not have the explanations for all the behavior we have observed in the experiments. More experiments to gain insights, more advanced system monitoring tools to provide finer details on resource usage in various parts of the system, as well as better understanding of the scheduler implementation will be of great help.

Second, we have focused on the CPU resource in this paper. We are interested in designing similar experiments to evaluate the impact of dynamic allocation of other resources, including memory, network I/O and disk I/O bandwidth.

Third, we would like to include other virtualization technologies into our evaluation study, such as VMware, Solaris Zones, and new emerging virtualization platforms.

Finally, using control theoretical approaches including the one in [2], we will specifically incorporate the tradeoffs we concluded in Section IV into the design of a closed-loop controller that is part of the management system for virtualized resources in data centers.

## REFERENCES


[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Nergebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), October, 2003.

[2] Y. Diao, J.L. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh and C. Garcia-Arellano, "Incorporating cost of control into the design of a load balancing controller," invited paper, *Real-Time and Embedded Technology and Application Systems Symposium*, 2004.

[3] D. Herington and B. Jacquot, *The HP Virtual Server Environment: Making the Adaptive Enterprise Vision a Reality in Your Datacenter*, Prentice Hall, September, 2005.

[4] IBM Enterprise Workload Manager, http://www-03.ibm.com/servers/eserver/zseries/zos/ewlm/

[5] OpenVZ, http://openvz.org/

[6] D. Price and A. Tucker, "Solaris Zones: Operating system support for consolidating commercial workloads," in *Proceedings of 18th Large Installation System Administration Conference (LISA)*, November, 2004.

[7] VMware, http://www.vmware.com

[8] Z. Wang, X. Zhu, and S. Singhal, "Utilization and SLO-based control for dynamic sizing of resource partitions," 16th IFIP/IEEE Distributed Systems: Operations and Management, October, 2005.

[9] Xen Virtual Machine, http://www.xensource.com