Memory Overbooking and Dynamic Control of Xen Virtual Machines in Consolidated Environments

Jin Heo
University of Illinois at Urbana-Champaign*
jinheo@cs.uiuc.edu

Xiaoyun Zhu†
VMware, Inc.
xzhu@vmware.com

Pradeep Padala
University of Michigan, Ann Arbor
ppadala@eecs.umich.edu

Zhikui Wang
Hewlett-Packard Laboratories
zhikui.wang@hp.com

***Abstract*— The newly emergent cloud computing environments host hundreds to thousands of services on a shared resource pool. The sharing is enhanced by virtualization technologies allowing multiple services to run in different virtual machines (VMs) on a single physical node. Resource overbooking allows more services with time-varying demands to be consolidated reducing operational costs. In the past, researchers have studied dynamic control mechanisms for allocating CPU to virtual machines, when CPU is overbooked with respect to the sum of the peak demands from all the VMs. However, runtime re-allocation of memory among multiple VMs has not been widely studied, except on VMware platforms. In this paper, we present a case study where feedback control is used for dynamic memory allocation to Xen virtual machines in a consolidated environment. We illustrate how memory behaves differently from CPU in terms of its relationship to application-level performance, such as response times. We have built a prototype of a joint resource control system for allocating both CPU and memory resources to co-located VMs in real time. Experimental results show that our solution allows all the hosted applications to achieve the desired performance in spite of their time-varying CPU and memory demands, whereas a solution without memory control incurs significant service level violations.**

***Keywords*—virtualization, consolidation, resource overbooking, dynamic control, application performance**

I. INTRODUCTION

THE newly emergent cloud computing environments such as Amazon's EC2 [2] host hundreds to thousands of services on a shared resource pool. The sharing is enhanced by virtualization technologies such as Xen [3] and VMware [17] allowing multiple services to run in different virtual machines (VMs) in a single physical node. The same technologies have been used by enterprises to consolidate servers in order to improve resource efficiency, reduce data center footprint, and to reduce power consumption and environmental impact of IT organizations. Although the average resource utilization in traditional data centers range between 5-20%, servers in consolidated data centers or cloud computing environments are likely to run at much higher utilization levels, exposing applications to possible resource shortages [18].

Researchers in both academia and industry have studied mainly three resource management strategies for consolidated environments – capacity planning [16], virtual machine (VM) migration [6], and dynamic resource allocation. These techniques are complementary to one another because they typically operate at different time scales and different scopes of a data center [23]. We discuss their relationship in more details in the next section. In this paper, we focus on dynamic resource allocation in a virtualized server that hosts multiple services. Runtime allocation of server resources to individual VMs enables *resource overbooking*, where the total capacity of a resource is below the sum of the peak demands of all the VMs sharing this resource. This allows each physical server to achieve much higher resource utilization while still maintaining performance isolation among the co-hosted services.

In the past, researchers have mostly studied overbooking of the CPU resource on virtualized servers and dynamic control mechanisms for allocating CPU to the individual virtual machines [15][20][22]. At the same time, memory overbooking and runtime re-allocation of memory among multiple VMs has not been widely studied, except in the VMware ESX Server [19]. However, the dynamic memory management policies in ESX do not directly support application-level performance assurance. The availability of memory balloon driver in recent Xen Server releases [5] provides a mechanism for memory sharing, but no dynamic policies have been implemented.

We have built an experimental testbed using Xen virtualized servers and performed a set of experiments to understand the relationship between memory allocation to a VM and performance of the hosted application. We observe that this relationship is different from the relationship between CPU allocation and application performance. Based on these observations, we make the following two contributions in this paper. First, we have developed a dynamic memory allocation controller that ensures that each VM gets sufficient memory to achieve desired application performance. We validate this controller against time-varying memory demands and demonstrate that memory overbooking on Xen can be achieved using our controller. Second, we have built a prototype of a joint CPU and memory control system for a consolidated environment. We evaluate the performance of this control system by driving multiple test applications with either synthetic or real world demand traces, and demonstrate that all the hosted applications can achieve their service level objectives (SLOs) without creating CPU or memory bottlenecks.

* This work is sponsored in part by NSF CNS 06-15301.

† Xiaoyun Zhu worked on this project while employed at HP Labs.

978-1-4244-3487-9/09/$25.00 © 2009 IEEE

The remainder of the paper is organized as follows. Section II provides an overview of related work. Section III describes the setup of our experimental testbed and the architecture of our resource control system. In Section IV we present the results of our modeling experiments. We describe the design and performance evaluation results for the dynamic memory controller in Section V, and for the joint CPU and memory controller in Section VI. In Section VII, we conclude the paper and discuss future research directions.

II. RELATED WORK

Capacity planning is a common practice in consolidated environments. For example, the techniques in [16] determine whether a given resource pool has enough capacity to support a given set of services by pre-computing an optimal placement of services onto the physical nodes based on historical resource usage traces of these services. However, this does not mean that runtime capacity management is no longer needed, because no capacity planning tools can deal with unpredicted workload surges that might happen on production systems.

Other tools have used live VM migration to handle dynamic workload changes and resource overloads in production systems to avoid application performance degradation [12] [21]. However, migration of stateful applications (e.g. databases) might take too long causing service level violations during the migration. In addition, security concerns in VM migration [14] may cause the vendors to add security features that will make migration much slower. Finally, in a heavily-consolidated data center where most of the nodes are highly utilized, migration may not be viable.

VMware's dynamic memory management policies include page sharing and memory overbooking [19], where the latter allows one VM to borrow memory from the balloon driver within another VM. However, these policies are not directly associated with application-level performance metrics, such as response time and throughput. Memory overcommit was also studied in [13] using the balloon driver provided by Xen. That work has focused on how many VMs can be launched with a given memory capacity, and there is no discussion of the impact of such an exercise on application performance.

Control theoretic approaches have been applied before to resource management and software performance control (see [9] and the references therein). For example, in [1], a Proportional-Integral (PI) controller was used for an Apache Web server to adapt the quality of content in order to meet the performance target. In [7], a MIMO controller was designed to automatically adjust two application parameters to manage CPU and memory utilization of a Web server. Control theory has also been applied to develop admission control policies for 3-tier web sites [10] and storage systems [11]. These admission control schemes are complementary to our dynamic resource allocation approach, because the former shapes the resource demand into a server system, whereas the latter adjusts the supply of resources in real time for handling the demand.

III. SYSTEM SETUP AND ARCHITECTURE

We have built a testbed for experimentation, modeling, development, and performance evaluation of dynamic memory allocation schemes in a consolidated environment. In this section, we first briefly describe the setup of our testbed, and then present the architecture of a feedback-based resource control system we developed.

A. Testbed setup

Our experimental testbed consists of 4 HP Proliant DL 320 G4 servers. Each server has dual 3.2GHz Pentium D processors with 2MB L2 cache and 4GB main memory. The servers run SLES 10.1 or SLES 10.2 with a Xen-enabled 2.6.16 kernel. One server is used to run a control application that monitors and controls the resource sharing of the virtual machines. A second server is configured to have 1-4 production VMs, each running an Apache Web server (version 2.2.3). Each of the last two servers hosts two client VMs, where *httperf* (ftp://ftp.hpl.hp.com/pub/httperf), a scalable client workload generator, is used to send concurrent HTTP requests to the Apache Web servers running on the production VMs. The virtual machines run the same kernel image, SLES 10.1, and have the same /usr file system. The virtual machines and the servers are located on the same network, interconnected via a Gigabit Ethernet.

We have developed an application called *MemAccess* that drives the memory usage and CPU consumption of a VM from trace files. We employ an Apache module to implement *MemAccess* on the Apache Web server. When a new HTTP request arrives, this Apache module is invoked to execute some computation (e.g. public key encryption) to consume a certain amount of CPU time, while randomly accessing a portion of the heap memory of the *MemAccess* application.

Memory usage patterns in a trace file are recreated by properly adjusting the size of the allocated heap in the Apache module over time. The heap memory is created when the Apache process starts and it consists of memory chunks of 5MB each. Memory chunks are allocated or de-allocated based on the memory load in the trace. The heap is randomly accessed when requests are received as explained formerly, to actively utilize the entire heap region.

CPU consumption of the application is adjusted by properly changing the average rate of HTTP requests generated from an *httperf* client. We calibrated the CPU consumption per request in order to compute the needed request rate for a VM's CPU consumption to match the CPU consumption in the trace. The calibration process is performed in one VM, since all the VMs running the Apache server are configured the same.

B. Control system architecture

Figure 1 shows the architecture of our resource control system, where two or more applications running on virtual machines share the resources in a physical node. Actuators provided by the virtualization layer allow us to allocate portions of the CPU and the physical memory to VMs to achieve desired performance. A resource controller gathers information on the

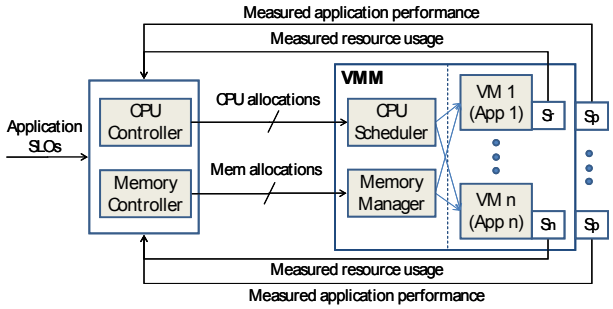


Figure 1. Resource control system architecture

resource usage of VMs and application performance from a set of sensors, as well as the service-level objectives (SLOs) for the applications, and determines in real time the CPU and memory allocations for all the VMs in the next control interval. Details about the actuators, sensors and controller are described below.

We use the Xen credit scheduler as the actuator for setting the CPU shares for VMs. The credit scheduler allows each VM to be assigned a *cap*, which limits the amount of CPU a VM can use. This non-work-conserving mode of CPU scheduling allows better performance isolation among multiple VMs, preventing a poorly-behaving application from draining the CPU capacity.

We use the *balloon driver* in Xen to dynamically change the memory allocations for the virtual machines. Each VM is configured with a maximum entitled memory (*maxmem*). The value of *maxmem* is the sum of the balloon value and the allocation value. If one VM does not need all of its entitled memory, the unused memory can be transferred to another VM that needs it. This is done by inflating the balloon (reducing allocation) in the first VM and deflating the balloon (increasing allocation) in the second VM. This mechanism allows multiple VMs to be configured with a higher amount of total memory than the size of the physical memory.

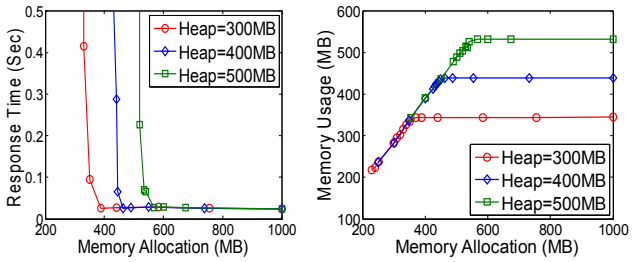
Our sensors periodically collect resource consumption and application performance statistics. CPU consumption is measured using Xen's `xentop` command. Memory allocation and usage are measured from the balloon in the `/proc` file system. We also monitor page fault rates of individual VMs using the `/proc` interface. Application performance is measured by an interposing proxy between the client and the server.

Our resource controller consists of two parallel controllers for CPU and memory. Each controller periodically changes the CPU or the memory allocations for the individual VMs in every control interval. The control decisions are made based on a quantitative model inferred from black-box testing of the system, described in the next section.

IV. Memory Usage and Performance

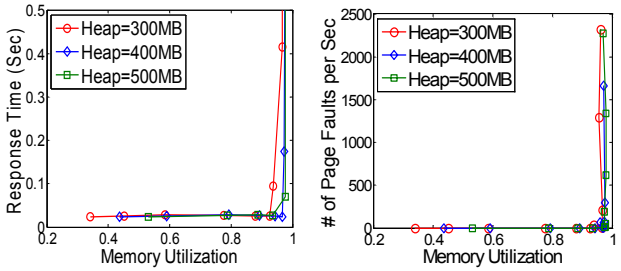
In order to develop the memory controller shown in Figure 1, we need to first understand the quantitative relationship between an application's performance and its memory allocation. To this end, we performed the following experiments using our Xen-based testbed and the *MemAccess* application described in Section III.

We ran the *MemAccess* application in one of the production VMs. An *httperf* client generated a workload with exponentially distributed inter-arrival times and a mean request rate of 30 requests/s. We allocated certain heap size for the application, and then varied the memory allocation to this VM from 230 MB to 1 GB. For each memory allocation setting, we ran the experiment for 50 seconds to allow the system to settle down to a steady state. In the meantime, the credit scheduler was running in the work-conserving mode so no CPU cap was imposed on the VM. The following metrics were collected during that 50-seconds interval: average memory usage, mean response time (MRT) of the application, and rate of major page faults. The experiment was repeated three times, for a different application heap size of 300, 400, and 500 MB, respectively.



(a) (b)

Figure 2. Mean response time (a) and memory usage (b) as a function of memory allocation



(a) (b)

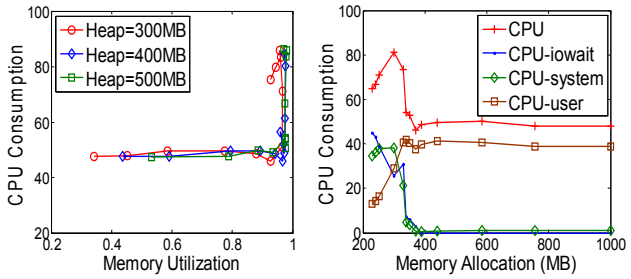
Figure 3. Mean response time (a) and rate of major page faults (b) as a function of memory utilization

Figure 2(a) displays the measured MRT as a function of the memory allocation for the three different heap size configurations. For each configuration, the MRT of the application remains at around 20 ms for most values of the memory allocation until the latter drops below a certain threshold. To better understand this behavior, we also show the measured memory usage versus the memory allocation in Figure 2(b). We can see that the memory usage is constant when enough memory is allocated, and starts to decrease linearly with the memory allocation when the latter becomes small.

In Figure 3(a), we demonstrate the MRT as a function of the average memory utilization for the three configurations, where memory utilization of a VM is computed as the ratio of memory usage to memory allocation. It is easy to see that the MRT increases sharply as the memory utilization goes beyond 90%.

This is consistent with our expectation, because when memory utilization is high, the guest operating system experiences significant memory pressure and starts reclaiming memory by paging a portion of the application memory to disk [4]. This will lead to a higher number of page faults when the application tries to access the main memory, resulting in higher latency for the application. To validate this, we also plot the number of major page faults per second as a function of the memory utilization in Figure 3(b). The sudden surge in the page fault rate when the memory utilization is above 90% confirms our explanation for the relationship we see in Figure 3(a).

Figure 4(a) shows the average CPU consumption of the VM as a function of the memory utilization. We can see that the VM's CPU consumption remains below 50% for a memory utilization less than 90%, independent of the heap size configuration. However, as the allocated memory is near saturation, we observe significant CPU overhead where the CPU consumption becomes much higher. To further illustrate this, we also show a breakdown of the VM's CPU consumption in Figure 4(b), as a function of the memory allocation, for a heap size of 300 MB. As the memory allocation becomes smaller, we first see the total CPU consumption goes up, mainly due to the extra paging activities by the guest OS, as shown by the "CPU-system" line in Figure 4(b). However, if the memory allocation is further reduced, the VM's CPU usage starts to decrease, mainly because the CPU spends a higher portion of time waiting for IO, as the "CPU-iowait" line indicates.



(a) (b)

Figure 4. VM CPU consumption vs. memory utilization for three different heap sizes (a) and breakdown of VM CPU consumption for heap size of 300MB (b)

In order to understand whether the dynamic CPU control techniques developed in [20][22] are applicable to controlling memory, we needed to compare the relationship between application performance and memory utilization to the similar relationship for CPU. To this end, we performed a similar set of experiments where the CPU credit scheduler was used to vary the CPU allocation to the VM while using a constant heap size of 300 MB and a memory allocation of 512 MB. In this case, the memory utilization of the VM remained constant, whereas CPU utilization of the VM varied between roughly 20% and 100%. The same experiment was repeated three times, for a different workload intensity of 10, 20, and 30 requests/s, respectively.

Figure 5 shows the mean response time as a function of the CPU utilization for the three different workload conditions. By comparing Figure 3(a) and Figure 5, we can see that these two relationships are similar in that the application's MRT is a monotonically increasing function of either the CPU utilization or the memory utilization. At the same time, we also observe two differences: First, for CPU, the curve that represents the relationship varies with the workload, whereas for memory, the relationship remains almost the same when the application's memory demand changes; Second, the function that represents the relationship is smoother and more differentiable for CPU, but is almost a binary function for memory. This means, when CPU allocation is reduced for an application, there is more graceful degradation in its performance. But we should never allow a VM's memory utilization to go beyond a certain threshold in order to ensure good application performance. From Figure 3, we observe that a reasonable threshold to use is 90% for memory utilization.

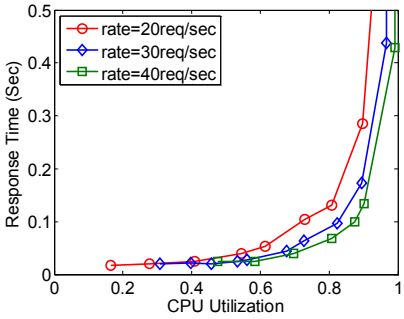


Figure 5. Mean response time vs. CPU utilization

V. DYNAMIC MEMORY ALLOCATION

In this section, we present a dynamic memory controller that periodically adjusts the memory allocation to individual VMs such that the application running in each VM can meet its SLO.

A. Memory controller design

Based on the discussion at the end of Section IV, it is desirable to maintain the utilization of a VM's allocated memory below a critical threshold. Therefore, we apply the design of the utilization controller previously developed for CPU allocation [20] to dynamic memory allocation.

Let $u_{mem}(k)$ and $v_{mem}(k)$ be the memory allocation and usage for a VM during the kth control interval, and let $r_{mem}(k)= v_{mem}(k)/u_{mem}(k)$ be the VM's average memory utilization for the same interval. Then the *Memory utilization controller* uses the following equation to compute the desired memory allocation for the next, or the $(k+1)$th interval:

$$u_{mem}(k+1) = u_{mem}(k) - \lambda_{mem} v_{mem}(k)(r_{mem}^{ref} - r_{mem}(k))/r_{mem}^{ref}. \quad (1)$$

This control law has two configurable parameters, where r_{mem}^{ref} is the desired level of memory utilization for the VM, and λ_{mem} is a constant for fine tuning the aggressiveness of the controller. In [20], we have shown that the above controller is stable as long as $\lambda_{mem} < 2$. For the experiments in this paper, we set $r_{mem}^{ref} = 90\%$ and $\lambda_{mem} = 1$. The design of the control law in (1)

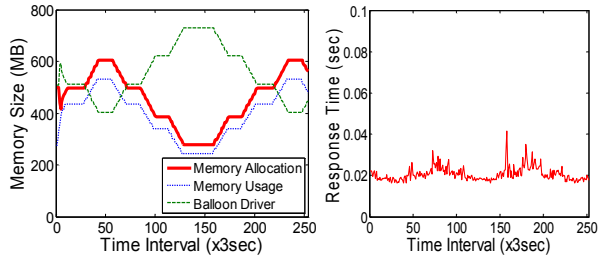
was driven by the bimodal behavior of the system, as shown in Figure 3(a). The controller aggressively allocates more memory when the previously allocated memory is close to saturation, and slowly decreases memory allocation in the underload region. This memory utilization controller is fairly easy to implement where the only measurement needed is the memory usage of the VM. The controller also ensures that the value of the memory allocation, $u_{mem}(k+1)$, remains within a specified range of $[U_{mem}^{\min}, U_{mem}^{\max}]$, where $U_{mem}^{\max} = mem\max$, the maximum amount of memory the VM is entitled to, and $U_{mem}^{\min}$ is the minimum amount of memory to keep the VM running. It is also important to choose a proper control interval, T_{mem}, for the memory controller. We chose the control interval to be as short as possible so that the controller can respond quickly when there is a spike in the VM's memory usage, while taking into consideration the latency in adding and removing memory.

B. Memory controller evaluation

We ran the following experiment to test the performance of the memory controller. We used the *MemAccess* application to emulate a synthetic memory usage trace that has two peaks and one valley and that varies between 256 and 512 MB. Table 1 lists the values of the memory controller parameters used in the experiment, and Figure 6 shows the results.

Table 1. Parameter values of the memory controller

r_{mem}^{ref}	λ_{mem}	$U_{mem}^{\min}$	$U_{mem}^{\max}$	T_{mem}
0.9	1	230 MB	1024 MB	3 s



(a) (b)

Figure 6. Memory allocation, usage, and balloon size for the VM (a) and measured response time (b) as a result of the memory controller

The red line in Figure 6(a) represents the measured memory usage, and the blue line represents the memory allocation computed by the controller. As we can see, our memory controller does a good job in providing enough memory to the VM, in spite of the varying memory demand from the application. The green line in the figure shows the size of the balloon. This is the amount of memory that can be "borrowed" by another VM when needed to handle load spikes.

The resulting average response time of the application is shown in Figure 6(b). It is evident that the application performance was good throughout the duration of the experiment, where the response time remained below 50ms.

VI. Combined CPU and Memory Control

A. Nested CPU controller

We use the nested controllers developed in [22] for CPU allocation to each VM: a *CPU utilization controller* running in the inner loop, which periodically adjusts the CPU allocation to maintain utilization of the allocated capacity at a given target, and a *Response Time controller* in the outer loop, which sets the target utilization in real time based on the observed error between the response time reference and its measurement. Figure 11 shows a block diagram of the nested control loops.

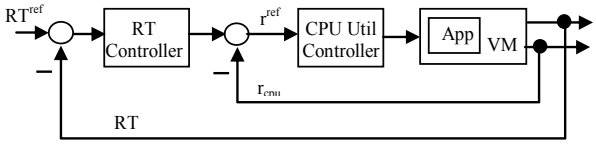


Figure 7. Nested CPU utilization controller and response time controller

The CPU utilization of an application is a common metric monitored in production systems to determine whether more or less CPU resource should be allocated to the application. Similar to the notation for memory, let $u_{cpu}(k)$ and $v_{cpu}(k)$ be the CPU allocation and consumption for a VM during the kth control interval, and $r_{cpu}(k) = v_{cpu}(k)/u_{cpu}(k)$ be the VM's average CPU utilization for the same interval. Then the controller uses the following equation to compute the desired CPU allocation for the next, or the $(k+1)th$ interval:

$$u_{cpu}(k+1) = u_{cpu}(k) - \lambda_{cpu} v_{cpu}(k)(r_{cpu}^{ref} - r_{cpu}(k))/r_{cpu}^{ref}. \quad (2)$$

In this controller, r_{cpu}^{ref} is the desired level of CPU utilization for the VM. The parameter λ_{cpu} is a constant again for fine tuning of the aggressiveness of control actions. The controller is stable for $\lambda_{cpu} < 2$ (see [22]). For the experiments in this paper, we set $\lambda_{cpu} = 1.5$, allowing the CPU controller to be slightly more aggressive than the memory controller. The output of the integral controller is then bounded in the range $[U_{cpu}^{\min}, U_{cpu}^{\max}]$.

We see from Figure 5 that the application's MRT is a monotonically increasing function of the CPU utilization. For a given response time target, RT^{ref}, the corresponding CPU utilization target, r_{cpu}^{ref}, is different for different workloads. For example, the same figure shows that, for $RT^{ref} = 0.1$s, the ideal CPU utilization is 0.73, 0.82, and 0.87 for our workload at 10, 20, and 30 requests/s, respectively. Given that both workload mix and intensities can change at runtime, a second controller is required to translate the value of the RT target to the value of the CPU utilization target automatically in real time. We refer to this controller as the RT controller.

As in [22], we use the following integral control law for the RT Controller:

$$r_{cpu}^{ref}(i+1) = r_{cpu}^{ref}(i) + \beta(RT^{ref} - RT(i))/RT^{ref}, \quad (3)$$

where *RT(i)* is the measured average response time for the *i*th control interval. The value of the integral gain, β, can be computed based on the approximate slope of the response time curve in Figure 5 at the expected operation point, RT^{ref}. For example, if the maximum estimated slope is α, then we need to have $\beta < 1/\alpha$ to ensure stability of the RT controller. The controller also ensures that the r_{cpu}^{ref} value fed to the CPU utilization controller is bounded to an interval $[R_{cpu}^{\min}, R_{cpu}^{\max}]$.

Note that two different time indices are used in (2) and (3) to indicate that the utilization controller and the RT controller use different control intervals. In a nested design, typically the inner loop has faster dynamics and therefore uses a shorter control interval than that of the outer loop. Let T_{cpu}^{UC} and T_{cpu}^{RT} be the control intervals for the CPU Utilization controller and the RT controller, respectively. Then $T_{cpu}^{UC} < T_{cpu}^{RT}$. The difference between the two should depend on how many intervals are required for the inner controller to converge before its reference setting is changed by the outer controller. (See more discussion of this nested design in [22]).

B. Joint CPU and memory controller

As we have seen from many resource utilization traces of real world applications, both the CPU and the memory demands of an application can vary over time. Therefore, our resource controller runs the CPU controller and the memory controller side by side, as shown in Figure 1, to ensure that each application running in a VM can obtain enough of both CPU and memory resources such that its SLO can be met.

Note that, unlike the CPU controller, the memory controller is not driven by the response time target in real time due to the sharply different behavior of memory as shown in Figure 3(a). Instead, we use 90% as the target for the memory utilization controller to ensure that there is always some amount of free memory available within each VM to avoid memory-related performance degradation in the hosted applications.

Both the CPU and the memory controllers include an arbiter to handle overload situations, where the total computed allocation for the next interval exceeds the resource capacity. In this case, the arbiter reduces the CPU or memory allocation to each VM in proportion to the requested allocation.

C. Performance evaluation results

Table 2. Parameter values of the CPU controller

λ_{cpu}	$U_{cpu}^{\min}$	$U_{cpu}^{\max}$	T_{cpu}^{UC}	RT^{ref}	β	$R_{cpu}^{\min}$	$R_{cpu}^{\max}$	T_{cpu}^{RT}
1.5	0.1	0.9	3 s	0.1s	0.1	0.5	0.95	12 s

We have run performance evaluation experiments of the joint CPU and memory controller on our testbed, described in Section III. In this subsection, we present the results of two such experiments, one using synthetic workload traces, and the other using utilization traces collected from real systems. In these experiments, the parameter values for the memory controller are the same as shown in Table 1, and the parameter values for the nested CPU controller are displayed in Table 2.

1) Results from using synthetic workloads

In the first experiment, we ran two Xen virtual machines, VM1 and VM2, on the same physical node, each hosting a *MemAccess* application. The two guest VMs were pinned to one of the two processors while Dom-0 was pinned to the other processor. In addition, 3GB out of the 4 GB of memory was allocated to Dom-0, and the remaining 1 GB was shared by VM1 and VM2. For each application, we varied both the CPU and the memory loads according to synthetically generated time-varying patterns. For either CPU or memory, the patterns for the two applications are complementary to each other such that the sum of the two peaks is above the capacity of that resource, but the sum of the two instantaneous values is much lower. These workload patterns and resource configurations were specifically chosen to test the performance of our joint CPU and memory controller when both resources are overbooked.

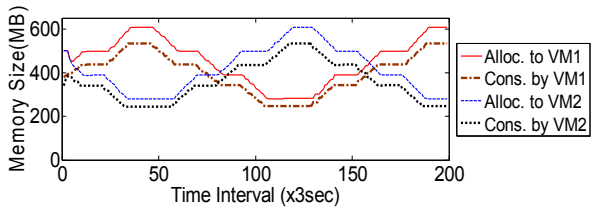
We consider two controller configurations in our experiment. One is the joint CPU and memory controller described earlier, and the other is the CPU controller only without the memory controller. In the latter case, the memory allocation for each VM is statically configured at 512 MB. For both cases, the response time target, RT^{ref}, was set at 100 ms (or 0.1 s) for both applications. The experiment was run for 10 minutes for each controller configuration. The results are shown in Figures 8-10.

Figure 8 shows the memory allocation and measured memory usage for both VM1 and VM2 under the two controller configurations. We can see that the memory allocations for the two VMs perfectly track the time-varying memory demands of both applications, when the memory controller was used, as shown in Figure 8(a). In contrast, Figure 8(b) shows that when no dynamic memory allocation was used, both VMs had their peaks capped by the fixed allocation.

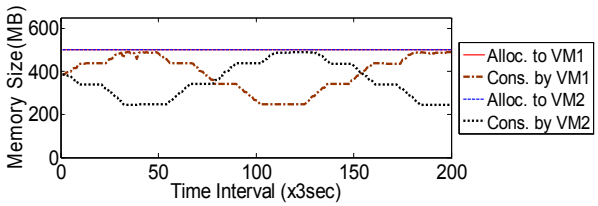
Figure 9(a) shows the CPU allocations and consumptions for the two VMs with the joint CPU and memory controller, where the allocations track the consumptions well. Figure 9(b) shows the same metrics when only CPU controller was used. We observe that the peak CPU consumption by VM1 was increased from 50% to 70% around the 40th and the 180th time intervals as a result of the memory shortage in this VM. This is similar to what we observed in Figure 4. Consequently, the required CPU allocation to VM1 was also increased, causing the shared processor to be overloaded. The arbiter in the CPU controller was applied to reduce CPU allocations to both VMs.

The resulting mean response times of the two applications running in VM1 and VM2 are shown in Figure 10. When the joint CPU and memory controller was used, the MRTs of both applications were maintained around their target value of 100ms. When no memory controller was used, both applications experienced significant service level violations where the peak response times were up to two orders of magnitude higher, due

to both memory shortage and CPU overload conditions.

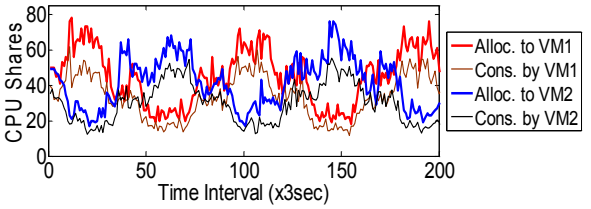


(a) Joint CPU and memory control

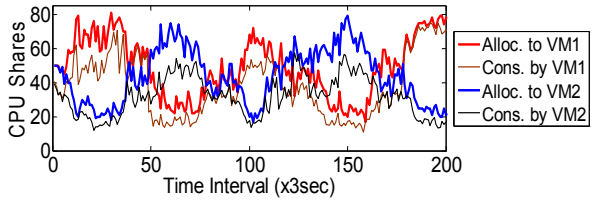


(b) CPU control only

Figure 8. Memory allocation and usage for VM1 and VM2 under two controller configurations

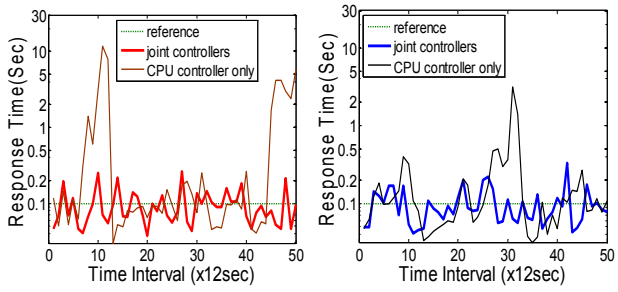


(a) Joint CPU and memory control



(b) CPU control only

Figure 9. CPU allocation and consumption for VM1 and VM2 under two controller configurations



(a) VM1 (b) VM2

Figure 10. Mean response times (in log10 scale) under two controller configurations for the two applications running in VM1 and VM2

2) Results from using real world traces

In the second experiment, we emulate a scenario where four virtual desktops are consolidated onto a single physical node. To this end, we ran four *MemAccess* applications in four different Xen VMs on the same node. The workloads used for these applications were based on real CPU and memory demand traces collected on four different desktop machines at HP Labs. Each trace contained average CPU and memory consumption for every 1 minute interval during the 9am-2pm window on the same day. This window was chosen such that all the machines were reasonably active where both CPU and memory consumptions demonstrate dynamic behavior. We consider this a more challenging scenario for our resource controller compared to other time periods where either the CPU load was mostly below 10% or the memory usage was flat.

As described in Section 3.A, we reproduced both the CPU and the memory demand patterns in each of the applications by dynamically varying both the mean request rate in the workload as well as the heap size allocation to the application once every minute. We also did a capacity planning exercise in advance where we estimated the peak of the total CPU load to be around 85%, and the peak of the total memory load to be around 3.1 GB. Therefore, we allocated one processor to Dom-0 and VM1, and the other processor to the remaining VMs (VM2, VM3, and VM4). We also allocated 512 MB of memory to Dom-0, and the remaining 3.5 GB of memory to be shared by the four guest VMs. The experiment was run for 5 hours, and the results are shown in Figures 11-13.

Figure 11 shows the memory allocation and usage for the four VMs, and Figure 12 shows the CPU allocation and measured consumption for the four VMs. Both sets of metrics were collected over the 3-second control intervals.

The measured mean response time for every 12-second interval is shown as a function of the interval number in Figure 13(a) for the four applications running in the four VMs. To better assess the overall performance, we also show a cumulative distribution function (CDF) of the MRT for the four VMs in Figure 13(b). Interestingly, the CDFs for the four applications are almost identical, although their CPU and memory demands are different. This is likely due to the large number of samples in the data. All the four applications were able to achieve an MRT of 100 ms 60% of the time, and an MRT of 300 ms more than 98% of the time. Such performance is fairly good considering that both CPU and memory are shared among the five VMs, and overall resource utilization of this system is much higher than that of typical enterprise servers.

VII. CONCLUSION

In this paper, we have presented a case study for dynamic memory allocation to virtual machines on Xen-based platforms. Based on experimental results that characterize the quantitative relationship between memory allocation to a VM and performance of the application running inside the VM, we have designed a joint resource controller combining a utilization-based memory controller and a performance-driven

CPU controller. Experimental results on our testbed validate that memory overbooking can be achieved using our memory controller, and our joint controller can ensure that all of the consolidated applications can have access to sufficient CPU and memory resources to achieve desirable performance.

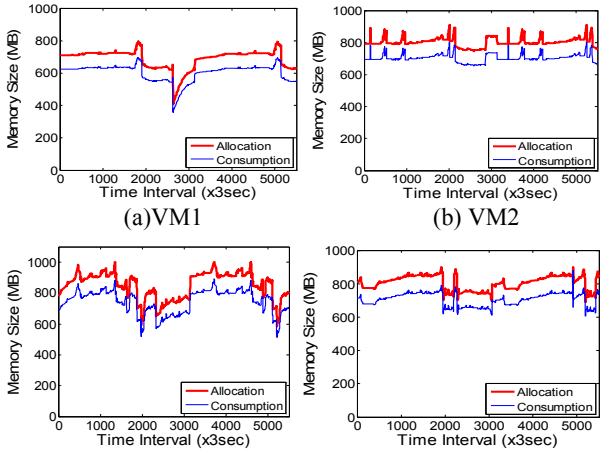


Figure 11. Memory allocation and usage for four VMs

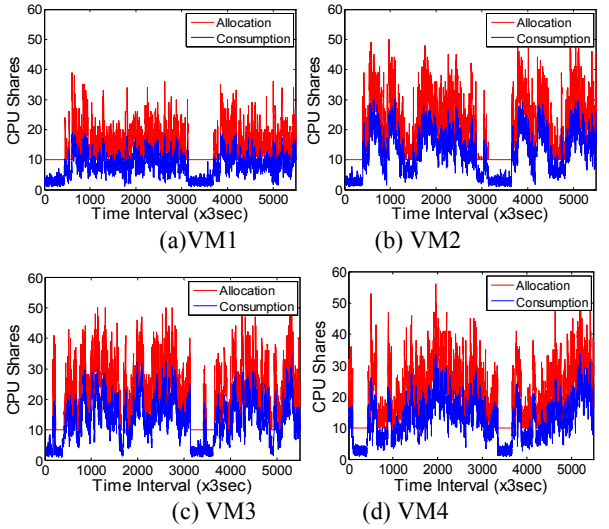


Figure 12. CPU allocation and consumption for four VMs

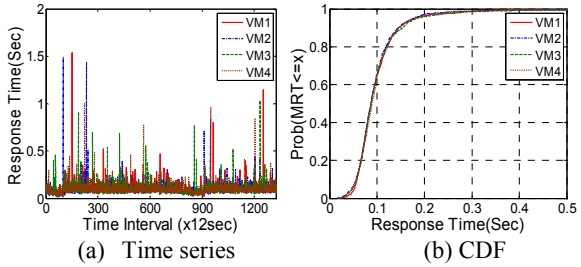


Figure 13. Measured MRT of the four applications

In our next steps, we will expand our testbed infrastructure and apply the joint CPU and memory controller to a larger scale environment with many more physical nodes and a higher number of applications, including multi-tiered applications that run across multiple nodes. We will also integrate our controller with controllers using other resource management schemes, such as disk I/O scheduling [8] and live VM migration [6] to provide a more comprehensive resource control solution to managing applications in consolidated environments.

REFERENCES

[1] T.F. Abdelzaher, K.G. Shin, and N. Bhatti, "Performance guarantees for Web server end-systems: A control-theorectical approach," *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[2] Amazon Elastic Computing Cloud (EC2), http://aws.amazon.com/ec2

[3] 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'03), October, 2003.

[4] D.P. Bovet and M. Cassetti, *Unerstanding the Linux Kernel, Third Edition,* O'Reilly & Associates, Inc, November, 2005.

[5] Citrix XenServer, http://www.citrixxenserver.com

[6] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield, "Live migration of virtual machines," *the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05),* May 2005.

[7] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury, "MI MO control of an Apache Web server: Modeling and controller design," *American Control Conference*, 2002.

[8] A. Gulati, A. Merchant, M. Uysal, and P. Varman, "Efficient and adaptive proportional share I/O scheduling," *Technical Report HPL-2007-186, HP Labs*, November, 2007.

[9] J.L. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*, Wiley-Interscience, 2004.

[10] A. Kamra, V. Misra, and E. Nahum, "A self-tuning controller for managing the performance of 3-tiered web sites," in *Proc. of the International Workshop on Quality of Service (IWQoS),* June, 2004.

[11] M. Karlsson, C. Karamanolis, and X. Zhu.Triage, "Performance isolation and differentiation for storage systems," in *Proc. of IEEE Int. Workshop on Quality of Service (IWQoS),* June, 2004.

[12] G. Khana, K. Beaty, G. Kar and A. Kochut, "Application performance management in virtualized server environments," *IEEE/IFIP Network Operations & Management Symposium (NOMS),* April, 2006.

[13] D. Magenheimer, "Memory overcommit… without the commitment," extended abstract at the *Xen Summit Boston 2008*, June, 2008.

[14] J. Oberheide, E. Cooke, and F. Jahanian. "Empirical exploitation of live virtual machine migration". In *Proc. of BlackHat DC convention*, 2008.

[15] P. Padala, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, and K. Shin, "Adaptive control of virtualized resources in utility computing environments," *EuroSys2007*, March 2007.

[16] J. Rolia, L. Cherkasova, M. Arlitt and A. Andrzejak. "A Capacity Management Service for Resource Pools," *the 5th Intl. Workshop on Software and Performance (WOSP'05)*, Spain, 2005.

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

[18] W. Vogels "Beyond server consolidation," *ACM Queue*, 6(1):20–26, 2008.

[19] C.A. Waldspurger, "Memory resource management in VMware ESX Server," in *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December, 2002.

[20] 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.

[21] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proc. 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI),* April, 2007.

[22] X. Zhu, Z. Wang, and S. Singhal, "Utility-Driven workload management using nested control design," American Control Conference (ACC), 2006

[23] X. Zhu, D. Young, B.J. Watson, *et al.*, "1000 Islands: Integrated capacity and workload management for the next generation data center," 5th *IEEE International Conference on Autonomic Computing (ICAC)*, June, 2008.