

## Autonomic Virtualized Environments

Daniel A. Menascé and Mohamed N. Bennani  
Dept. of Computer Science, MS 4A5, George Mason University  
Fairfax, VA 22030, USA  
{menasce,mbennani}@cs.gmu.edu

### Abstract

*Virtualization was invented more than thirty years ago to allow large expensive mainframes to be easily shared among different application environments. As hardware prices went down, the need for virtualization faded away. More recently, virtualization at all levels (system, storage, and network) became important again as a way to improve system security, reliability and availability, reduce costs, and provide greater flexibility. Virtualization is being used to support server consolidation efforts. In that case, many virtual machines running different application environments share the same hardware resources. This paper shows how autonomic computing techniques can be used to dynamically allocate processing resources to various virtual machines as the workload varies. The goal of the autonomic controller is to optimize a utility function for the virtualized environment. The paper considers dynamic CPU priority allocation and the allocation of CPU shares to the various virtual machines. Results obtained through simulation show that the autonomic controller is capable of achieving its goal.*

### 1. Introduction

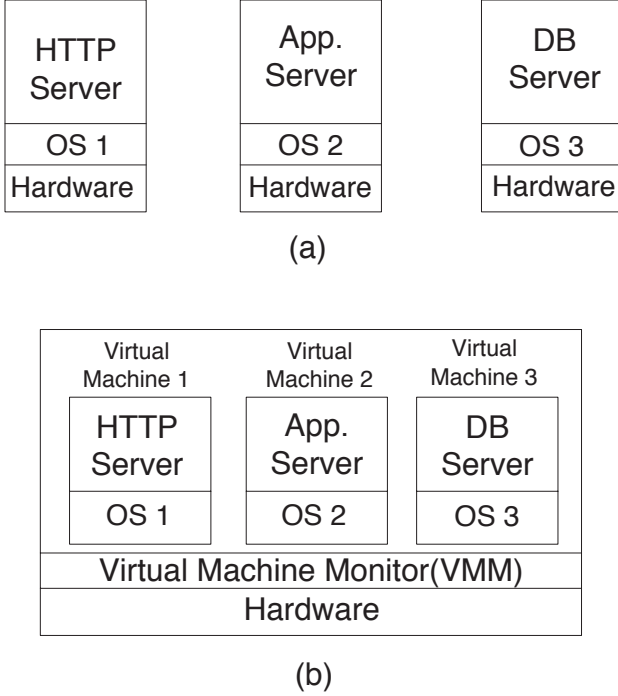
Thirty years ago, system virtualization was a hot topic of academic and industrial research [10, 15], which saw the light of day in products such as IBM's VM/370 [6]. System virtualization adds a hardware abstraction layer, called the Virtual Machine Monitor (VMM), on top of the bare hardware. This layer provides an interface that is functionally equivalent to the actual hardware to a number of *virtual machines*. These virtual machines may then run regular operating systems, which would normally run directly on top of the actual hardware. There are various virtualization techniques as well as requirements for architectures to be virtualizable [15]. The main motivation for virtualization in the early 70's was to increase the level of sharing and utilization of expensive computing resources such as the mainframes.

The 80's saw a decrease in hardware costs that caused a significant portion of the computing needs of an organization to be moved away from large centralized mainframes to a collection of departmental minicomputers. The main motivation for virtualization disappeared and with it their commercial embodiments.

The advent of microcomputers in the late 80's and their widespread adoption during the 90's along with ubiquitous networking brought the distribution of computing to new grounds. Large number of client machines connected to numerous servers of various types gave rise to new computational paradigms such as client-server and peer-to-peer systems. These new environments brought with them several challenges and problems including reliability, security, increased administration cost and complexity, increased floor space, power consumption, and thermal dissipation requirements. The recent rebirth of the use of virtualization techniques in commodity, inexpensive servers and client machines is poised to address these problems [9, 17] in a very elegant way.

Virtualization may be used for server consolidation. In that case, each Virtual Machine supports the operating system and application environments of a server being consolidated in the virtualized environment. Figure 1(a) shows the typical architecture of an e-commerce site with separate machines for Web servers, applications servers, and database servers. Figure 1(b) shows these servers being consolidated on a single, albeit more powerful, server.

Autonomic computing systems, also known as self-\* systems, can regulate and maintain themselves without human intervention. Such systems are able to adapt to changing environments (such as changes in the workload or failures) in a way that preserves given operational goals (e.g., performance goals). There has been significant research and attention to autonomic computing in the recent years [1, 3, 4, 5, 8, 11, 12, 13]. In our previous work [3, 11, 12, 13], we developed a technique to design self-organizing and self-tuning computer systems. This technique is based on the combined use of combinatorial search techniques and analytic queuing network models.



**Figure 1. Server consolidation scenario.**

In that work we defined the cost function to be optimized as a weighted average of the deviations of response time, throughput, and probability of rejection metrics relative to their Service Level Agreements (SLAs). Others have used a control-theoretic approach to design self-configuring systems [7] while others have used machine-learning methods [18].

In this paper, we show how our autonomic computing techniques can be used to dynamically allocate CPU resources in virtualized environments with the goal of optimizing a global utility function,  $U_g$ , under varying workload levels. We consider priority-based allocation and CPU-share based allocation. This paper investigates through simulation how an autonomic controller may achieve the goal of optimizing  $U_g$  or finding a close-to-optimal solution. The results of the paper are quite encouraging and pave the way to implementations of the ideas reported here in actual virtual machine monitors such as Xen and VMWare.

The ideas presented in this paper can also be used to implement autonomic schedulers for operating systems, since the issue of scheduling virtual machines at the virtual machine monitor level is akin to scheduling processes by an operating system. There is a vast literature on scheduling algorithms for operating systems, however, what makes our work different from previous scheduling work is that it presents a method for automatically changing the scheduler parameters (e.g., priority assignment or CPU share allocation) in a way that dynamically optimizes a global utility

function in an environment in which the workloads change dynamically and unpredictably.

The rest of this paper is organized as follows. Section two provides some background and formally defines the problem. Section three discusses the analytic models used by the controller. The following section describes the approach taken by the controller. Section five presents the results of the experiments. Finally, section six presents some concluding remarks.

## 2. Background

We consider a virtualized environment with  $M$  virtual machines (VM). Each VM  $i$  may run  $S_i$  different workload classes. We consider in this paper the problem of dynamically allocating the CPU to the various virtual machines in response to changes in the workload intensity. The workload intensity level for workload class  $s$  at VM  $i$  is specified by the average arrival rate  $\lambda_{i,s}$  of transactions of class  $s$  at VM  $i$ . We define a workload vector  $\vec{w}_i$  for VM  $i$  as  $\vec{w}_i = (\lambda_{i,1}, \dots, \lambda_{i,S_i})$ .

We consider in this study that the relevant performance metric for the workload classes are the average response times  $R_{i,s}$  for each class  $s$  of VM  $i$ . Including throughput as a performance metric of interest is straightforward. We define a response time vector,  $\vec{R}_i$ , for VM  $i$  as  $\vec{R}_i = (R_{i,1}, \dots, R_{i,S_i})$ .

These performance metrics can be obtained by solving an analytic performance model,  $\mathcal{M}_i$ , for VM  $i$ . The value of these metrics is a function of the workload  $\vec{w}_i$  and of the CPU allocation,  $a_i$ , to VM  $i$ . Thus,  $\vec{R}_i = \mathcal{M}_i(\vec{w}_i, a_i)$ . We investigate two approaches for assigning the CPU allocation  $a_i$  to each VM.

- *Priority allocation:* Each VM  $i$  has a dispatching priority,  $p_i$  ( $p_i = 1, \dots, P$ ), at the CPU. We assume preemptive resume priorities with priority 1 being the highest and priority  $P$  being the lowest. The priority changes dynamically according to the autonomic controller.
- *CPU share allocation:* Each VM  $i$  is allocated a share  $f_i$  of the CPU ( $\sum_{i=1}^M f_i = 1$ ). VMWare allows CPU shares to be allocated to different virtual machines in order to influence their processing rate (www.vmware.com). For example, if VM1 has 1,000 CPU shares and VM2 and VM3 have 500 CPU shares each, then VM1 will execute at a rate twice as high as VM2 and VM3, which will execute at the same rate. In the case of VMWare, the allocation of CPU shares can be changed by an administrator. In this paper we show how this can be done by the autonomic controller.

Each VM  $i$  has a utility function  $U_i$  that is a function of the set of performance metrics for the classes of that VM.

So,  $U_i = f(\vec{R}_i)$ . The global utility function,  $U_g$ , of the entire virtualized environment is a function of the utility functions of each VM. Thus,

$$U_g = h(U_1, \dots, U_M). \quad (1)$$

The utility function,  $U_{i,s}$ , for class  $s$  at VM  $i$  is defined as the relative deviation of the response time  $R_{i,s}$  of that class with respect to its service level agreement (SLA),  $\beta_{i,s}$ . Hence,

$$U_{i,s} = \frac{\beta_{i,s} - R_{i,s}}{\max\{\beta_{i,s}, R_{i,s}\}}. \quad (2)$$

If the average response time  $R_{i,s}$  meets the SLA,  $U_{i,s}$  is equal to zero. If  $R_{i,s} > \beta_{i,s}$ , then  $-1 < U_{i,s} < 0$ . If  $R_{i,s} < \beta_{i,s}$ , then  $0 < U_{i,s} < 1$ . Thus, the utility function  $U_{i,s}$  is a dimensionless number in the  $(-1, 1)$  interval. Other utility functions can be defined. See [2, 19] for examples.

The utility function for VM  $i$ ,  $U_i$ , is a weighted sum of the class utility functions. So,

$$U_i = \sum_{s=1}^{S_i} \alpha_{i,s} \times U_{i,s} \quad (3)$$

where  $0 < \alpha_{i,s} < 1$  and  $\sum_{s=1}^{S_i} \alpha_{i,s} = 1$ .  $U_i$  is also a number in the  $(-1, 1)$  interval.

The resource allocation problem in question is how to dynamically allocate the CPU among VMs with the goal of maximizing the global utility function  $U_g$ . The dynamic CPU allocation is achieved through an autonomic controller of the *white box* type; i.e., a controller that knows the internal details of the system being controlled and can use this knowledge to build a model that relates the output with the input for a given value of the controllable parameter. Our previous work [2, 3, 11, 12, 13] falls in the category of white box controllers. We developed a technique to design self-organizing and self-tuning computer systems based on the combined use of combinatorial search techniques and analytic queuing network models [14]. In the current paper, we adapt these techniques to the design of an autonomic controller for virtualized environments.

### 3. The Autonomic Controller

The goal of the autonomic controller is to find an optimal CPU allocation  $\vec{a} = (a_1, \dots, a_M)$  to the set of  $M$  virtual machines that optimizes the global utilization function  $U_g$ . Let  $\mathcal{S}$  be the space of all possible allocations. In the case in which the CPU allocation is priority-based, the cardinality of  $\mathcal{S}$  is  $P^M$ . If  $P = M = 10$ , there are 10 billion possible CPU allocations.

In the case of CPU-share allocations, we discretize  $\mathcal{S}$  by assuming that the total number of shares is  $T$  and that a VM is given a number of shares  $n_i$ ,  $n_i = 1, 2, \dots, T$  such that

$\sum_{i=1}^M n_i = T$ . Thus  $f_i = n_i/T$ . It is not difficult to see, using combinatorial arguments, that the cardinality of  $\mathcal{S}$  in the CPU share case is given by

$$|\mathcal{S}| = \binom{T+M-1}{M-1} - \sum_{j=T+1}^{T+M-2} j \quad (4)$$

which can be written as

$$\binom{T+M-1}{M-1} - \frac{(M-2)}{2} (2T+M-1). \quad (5)$$

For  $T = 100$  and  $M = 10$ , the number of possible allocations is a very large number in the order of  $10^{13}$ .

As it can be seen, the space of possible allocations  $\mathcal{S}$  can be extremely large depending on the values of  $P$ ,  $M$ , and  $T$ . The autonomic controller uses heuristic combinatorial search techniques [16]—beam-search in our case—to explore a subset of  $\mathcal{S}$  in which a solution close to the optimal may be found. Clearly, if the size of  $\mathcal{S}$  is small enough, an exhaustive search should be conducted by the controller. The utility function associated with each point in  $\mathcal{S}$  is computed using the performance models discussed in Section 4.

At regular time intervals, called *controller intervals (CI)*, the controller runs the controller algorithm, which searches through a subset of  $\mathcal{S}$ , as explained above, and determines the best allocation  $\vec{a}$ , which is communicated to the CPU scheduler module of the VMM.

More specifically, the controller algorithm searches the space of CPU allocations  $\vec{a} = (a_1, \dots, a_M)$  using a beam-search algorithm. Beam search is a combinatorial search procedure that works as follows [16]. Starting from the current allocation  $\vec{a}_0$ , the global utility function,  $U_g$ , is computed for all the “neighbors” of that allocation. The  $k$  allocations with the highest values of  $U_g$  are kept to continue the search. The value of  $k$  is called the *beam*. Then, the “neighbors” of each of the  $k$  selected points are evaluated and, again, the  $k$  highest values among all these points are kept for further consideration. This process repeats itself until a given number of levels,  $d$ , is reached.

The following definitions are in order to more formally define the concept of neighbor of an allocation. Let  $\vec{v} = (v_1, v_2, \dots, v_M)$  be a *neighbor* of an allocation vector  $\vec{a} = (a_1, a_2, \dots, a_M)$ . In the priority case,  $\vec{v}$  and  $\vec{a}$  represent priorities of the  $M$  VMs in the range  $1, \dots, P$ . The allocation vector  $\vec{v}$  is a neighbor of  $\vec{a}$  if the priority of one of the VMs in  $\vec{v}$  differs from the priority of the same machine in  $\vec{a}$  by  $\pm 1$  in the sequence  $1, \dots, P$ . Adding one to priority  $P$  should result in priority 1 and subtracting one from priority 1 should result in priority  $P$ .

In the case of CPU-share allocations, the allocation vector  $\vec{v}$  is a neighbor of  $\vec{a}$  if  $v_i = a_i + 1/T$  and  $v_j = a_j - 1/T$  for some  $i \neq j$  and  $v_k = a_k$  for all  $k \neq i$  and  $k \neq j$ .

The controller algorithm is specified more precisely in Figure 2. The following notation is in order:

- $\mathcal{V}(\vec{a})$ : set of neighbors of allocation vector  $\vec{a}$ .
- LevelList<sub>*i*</sub>: set of allocation vectors examined at level *i* of the beam search tree.
- CandidateList: set of all allocation vectors selected as the *k* best at all levels of the beam search tree.
- Top(*k*,  $\mathcal{L}$ ): set of allocation vectors with the *k* highest utility function values from the set  $\mathcal{L}$ .
- $\vec{a}_0$ : current allocation vector.

## 4. The Analytic Models

The queuing network models used here fall into the category of open multiclass queuing network models [14]. Each queue in the queuing network (QN) represents a resource (e.g., CPU, disk). There may be multiple classes of transactions depending on their use of the resources and their workload intensities. The total time spent by a transaction class *s* using a resource *k* is called the *service demand*,  $D_{k,s}$ . Note that the service demand does not include any queuing time spent at the resource. We consider in this paper that all workloads are open workloads whose intensities are specified by their arrival rates.

The response time,  $R_{i,s}$ , for class *s* of VM *i* is given by [14]:

$$R_{i,s} = \sum_{k=1}^K \frac{D_{k,s}}{1 - \sum_{s=1}^{S_i} \lambda_{i,s} \times D_{k,s}} \quad (6)$$

where *K* is the number of resources used by transaction of VM *i*.

Equation (6) does not cover the case of priority-based scheduling nor the case of CPU shares. In the next subsections we show how we adapt the results of Eq. (6) to these situations.

```

LevelList0 ←  $\vec{a}_0$ ;
CandidateList ← LevelList0;
For i = 1 to d Do
  Begin
    LevelListi ← ∅;
    For each  $\vec{a} \in \text{LevelList}_{i-1}$  Do
      LevelListi ← LevelListi ∪  $\mathcal{V}(\vec{a})$ ;
      LevelListi ← Top(k, LevelListi);
      CandidateList ← CandidateList ∪ LevelListi;
    End;
   $\vec{a}_{opt}$  ← max (CandidateList)

```

**Figure 2. Controller Algorithm.**

## 4.1. Priority Based Model

To model CPU preemptive resume dispatching priorities, we use the shadow CPU approximation in [14]. All the workloads of the same VM have the same priority in the model, since the VMM assigns dispatching priorities to the VM as a whole and not to its individual workloads.

In this approach, the open QN model is solved incrementally in *P* steps, one per priority class, from the highest priority class 1 to the lowest priority class *P*. At step *p*, only classes of priority  $q \leq p$  are included and the open QN model is solved. This partial model at step *p* has *p* shadow CPUs instead of one. Each shadow CPU is dedicated to all workloads that have the same priority. Let  $D_{\text{CPU},s}^p$  be the CPU service demand for workload class *s* at the shadow CPU associated to priority *p*. For a workload *s* of priority  $q \neq p$ ,  $D_{\text{CPU},s}^p = 0$  since workload *s* only uses the CPU associated to priority *q*. For a workload *s* of priority *p*, one has to inflate the service demand at the dedicated CPU for that class to account for the fact that class *s* only sees the amount of CPU not used by all workloads of higher priority, i.e., priorities  $q < p$ . Thus,

$$D_{\text{CPU},s}^p = \frac{D_{\text{CPU},s}}{1 - \sum_{q=1}^{p-1} \sum_{r \in \Omega(q)} \lambda_r \times D_{\text{CPU},r}} \quad (7)$$

where  $D_{\text{CPU},s}^p$  is the inflated CPU service demand for class *s* at the shadow CPU associated to priority *p*,  $D_{\text{CPU},s}$  is the original CPU service demand of workload *s*, and  $\Omega(q)$  is the set of workloads with priority *q*. The summation in the denominator of Eq. (7) is the sum of the CPU utilizations due to all workloads of priority  $q < p$ . Even though Eq. (7) does not explicitly indicate a VM, the workload designation *s* is assumed to be associated with a specific VM.

The disk service demands do not need to be changed because priorities only apply to the CPU. The final response time is obtained with the solution of the complete QN model at step *P*.

## 4.2. CPU Share Model

The CPU share model is built by having *M* shadow CPUs, one per VM. The CPU service demands have to be adjusted to account for the share allocation of each VM. This is done by setting

$$D_{i,\text{CPU}_s}^e = D_{i,\text{CPU}_s} / f_i \quad (8)$$

where  $D_{i,\text{CPU}_s}^e$  is the elongated CPU demand of class *s* at the shadow CPU for VM *i* and  $D_{i,\text{CPU}_s}$  is the original CPU demand of class *s* of VM *i*. Equation (8) can be understood by interpreting  $f_i$  as the rate at which workloads of VM *i* execute. For example, if  $f_i = 0.25$ , then any workload of that VM will execute at a rate equal to 0.25 seconds of

work per second of real time, assuming that a workload that receives a 100% allocation of the CPU executes at a rate of 1 second of work per second of real time. Thus, the average amount of time needed to execute one second of work is  $1/0.25 = 4$  seconds in this example.

## 5. Results

We used simulation to evaluate the controller for the two different cases discussed before: priority and CPU shares. A CSIM ([www.mesquite.com](http://www.mesquite.com)) model was built to simulate the virtualized environment in each case. The controller was implemented in a separate machine from the one used to run the simulation and used a controller interval equal to 2 minutes. For the simulation of the priority based case, we used a preemptive resume scheduling discipline at the CPU with a time slice equal to 0.001 sec. For the CPU share case, we used a round-robin scheduling discipline at the CPU with a time slice of 0.001 sec. In both cases, the disk scheduling discipline is FCFS.

The experiments reported in what follows assume two virtual machines 1 and 2 with one workload class each. The methodology described above can be easily applied to a large number of virtual machines and workloads given that the utility function is computed using a very fast analytic model and the beam-search algorithm can be tuned by setting its breadth and depth parameters in order to limit the subset of  $\mathcal{S}$  examined at each control interval.

The service demands assumed in the experiments reported here are given in Table 1. As it can be seen, the I/O service demands were set at a very low value since the controller is acting on CPU allocation only. For this reason, we did not want the I/O activity to mask the results obtained in the experiments. The values shown in Table 1 are the average values used to generate the resource consumption in the experiments. However, the controller was implemented in the simulation in the same way as it would have been implemented in an actual system. In other words, CPU and disk utilizations as well as workload throughputs were measured during each control interval and used to compute the service demand values, according to the service demand law, and used by the performance model.

**Table 1. Service demands (in sec)**

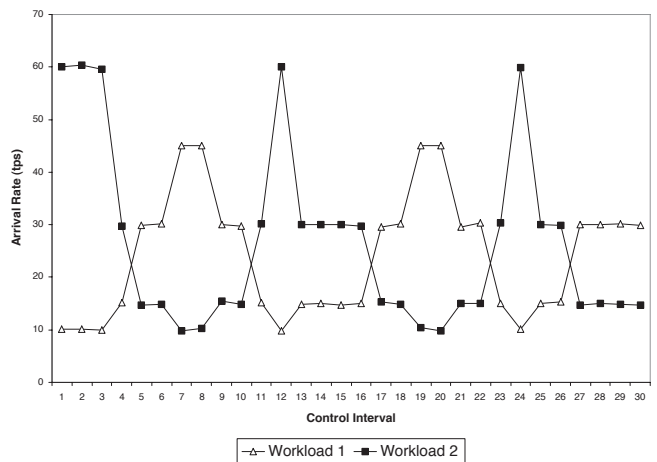
	Class	
	1	2
CPU	0.020	0.010
IO	0.0005	0.0005

We assume in the simulation experiments that CPU shares can only vary in increments of 5%. The global util-

ity function for the virtualized environment is computed as  $U_g = 0.4 \times U_1 + 0.6 \times U_2$ . All experiments reported here include a 95% confidence interval illustrated by the bars in each point in the graphs.

### 5.1. Results for the Priority Allocation

The variation of the workload intensity, in transactions per second (tps), for each of the workloads in each VM is shown in Fig. 3. The workload intensities show peaks and valleys, which are out of phase for the two VMs to force an exchange of CPU allocation between the two VMs. Workload 1 has its peaks at CIs 7 and 8 and then again at 19 and 20. Workload 2 has three peaks: at 1-3, 12, and 24.



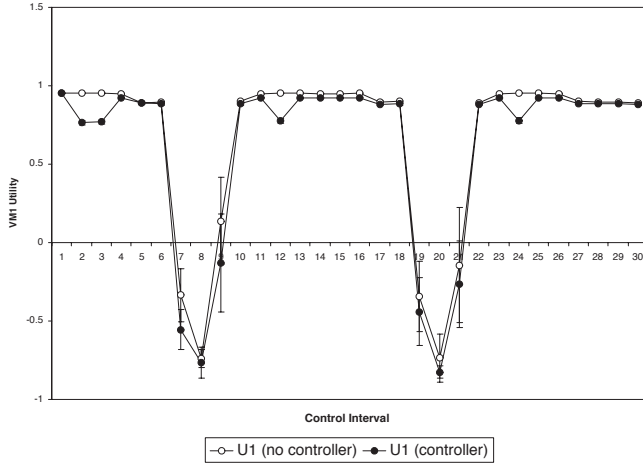
**Figure 3. Variation of the workload intensity, in tps, for the two workloads as a function of time, in CIs, for the priority case.**

Figures 4 and 5 show the variation of the utility functions  $U_1$  and  $U_2$  during the 60 minute experiment, using the variation of the workload intensity shown in Fig. 3. Each curve shows the variation of the utility function when the autonomic controller is used and when it is not used.

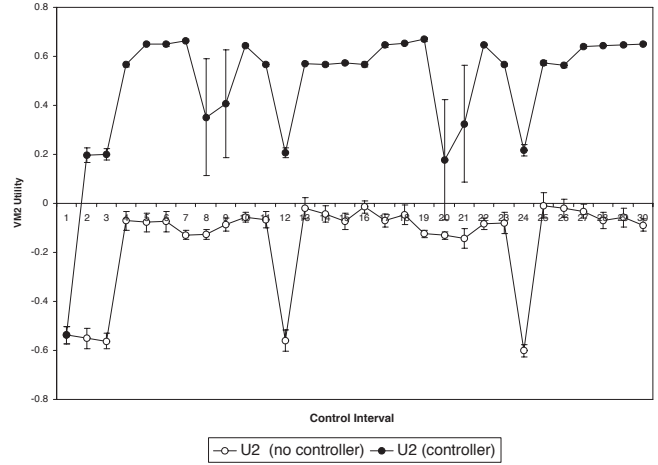
The graph of Fig 4 shows that (1) there is very little difference between the controller and non-controller results, (2) the non-controller results seem to be slightly better in some cases especially when the workload of VM 2 has its peaks; this is due to the fact that the controller is providing more CPU resources to workload 2, which has a higher weight in the global utility function  $U_g$ .

Figure 5 shows a marked difference between the controller and non-controller cases. The utility function for  $U_2$  remains in the positive territory from CI = 2 onwards.

Figure 6 shows the the variation of the global utility function  $U_g$  during the 60 minute experiment. The figure



**Figure 4. Variation of  $U_1$  as a function of time, measured in CIs, for the priority case.**



**Figure 5. Variation of  $U_2$  as a function of time, measured in CIs, for the priority case.**

shows that the controller is able to maintain a significantly higher global utility for the virtualized environment even at periods in which the two workloads go through their peaks in workload intensity.

Figure 7 shows the variation of the priorities of VMs 1 and 2 during the 60-minute experiment. Both VMs start with the same priority (i.e., priority 2). The priority of VM1 remains unchanged throughout the experiment. However, VM2 receives an immediate priority boost to priority 1 (the highest) because this workload is at its workload peak. However, the controller reduces VM 2's priority to 2 (i.e., now both VMs have the same priority) when VM 1 goes through workload peaks. This demonstrates that the autonomic controller is able to react to changes in workload intensity in order to improve the global utility function.

Figures 8 and 9 show the variation of the response time for VMs 1 and 2, respectively, for the controller and non-controller cases, as well as the SLA (dashed line) for each case. For VM1, the response time for both cases (controller and non-controller) stays below the SLA of 0.8 sec except when the workload for VM 1 goes through its peak period. In these intervals, the controller and non-controller results have very little statistical difference as can be seen by the confidence intervals.

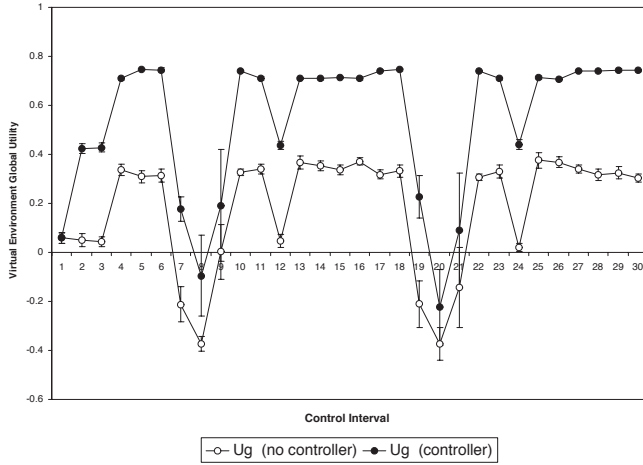
For the case of VM 2, Fig. 9 shows that as soon as the controller starts to operate (at CI = 2), the response time goes below its SLA of 0.035 sec because of the immediate boost in the priority level. The figure also shows that without the controller, this workload always exceed its SLA target.

## 5.2. Results for the CPU Share Allocation

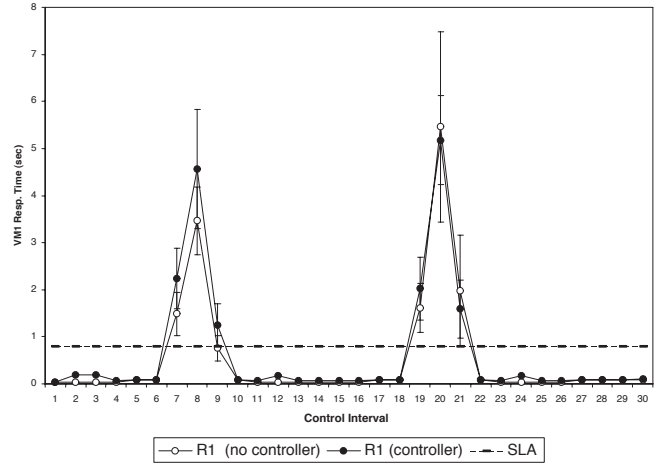
The variation of the workload intensity, in tps, for each of the workloads in each VM is shown in Fig. 10. As before, the workload intensities show peaks and valleys, which are out of phase for the two VMs to force an exchange of CPU allocation between them. Workload 1 has its peaks at CIs 7–8, 19–20, and then again at 27–30. Workload 2 has three peaks: at 1–3, 12, and 24. Please note that although the shape of the curves in Fig. 10 is similar to those of Fig. 3, the workload intensity values in this former case is lower than in the latter.

Figures 11 and 12 show the variation of the utility functions  $U_1$  and  $U_2$  during the 60-minute experiment, using the variation of the workload intensity shown in Fig. 10. Each curve shows the variation of the utility function when the autonomic controller is used and when it is disabled. Figure 11 shows that  $U_1$  is higher for VM 1 when the controller is not used. This is due to the fact the controller forces a higher level for  $U_2$ , as seen in Fig. 12, because VM 2 has a higher weight (0.6 versus 0.4) in the global utility function  $U_g$ .

Figure 13 shows the the variation of the global utility function  $U_g$  during the 60-minute experiment. The figure shows that in most cases the global utility function for the controller is significantly higher when the controller is enabled than when it is not. It should be noted that  $U_g$  for the controller case remains positive for CI > 2, i.e., as soon as the controller's effects start to be felt, while  $U_g$  goes to negative territory at CI = 12 and CI = 24 for the non-controller case, when VM 2's workload has high peaks in workload intensity.



**Figure 6. Variation of  $U_g$  as a function of time, measured in CIs, for the priority case.**

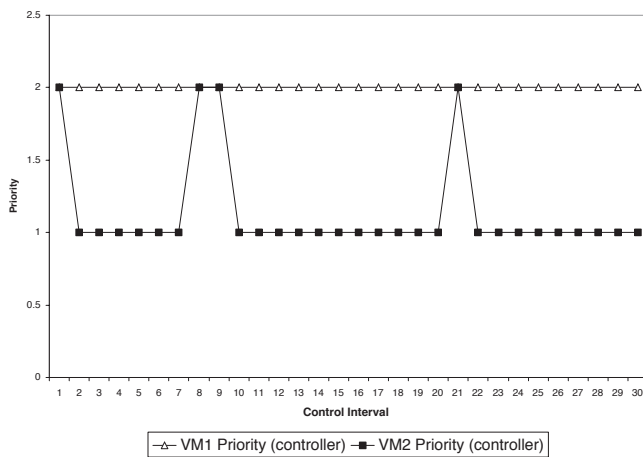


**Figure 8. Variation of response time for VM 1 as a function of time, in CIs, for the priority case.**

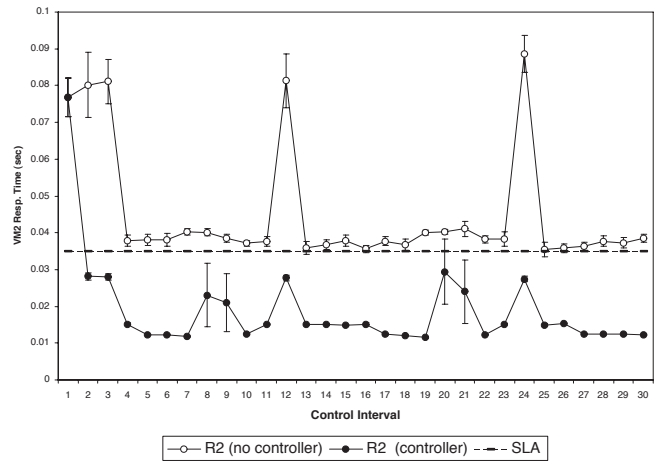
Figure 14 shows the variation of the CPU shares of VMs 1 and 2 during the 60-minute experiment. They both start with the same number of CPU shares each, i.e., 50%. As CPU resources need to be shifted between VMs, the controller changes the share allocation automatically. For example, at CI = 7, VM 1 sees a peak in the number of shares and VM 2 a valley, which coincides with the peaks and valleys seen in the workload intensity of Fig. 10. The same happens at CI = 19.

Figures 15 and 16 show the variation of the response

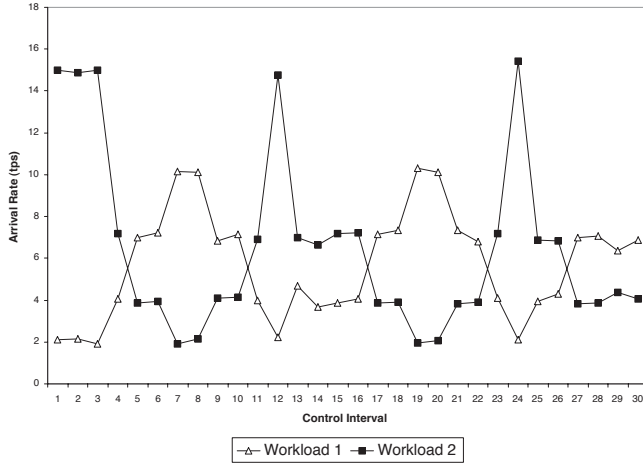
time for VMs 1 and 2, respectively, for the controller and non-controller cases, as well as the SLA (dashed line) for each case. For VM1, the response time stays below the SLA of 0.5 sec in both cases, controller and non-controller, except when the workload for VM 1 goes through its peak period. In these intervals, the response time under the controller case is significantly higher than when the controller is not used. The fact that VM 1 has a lower weight in the



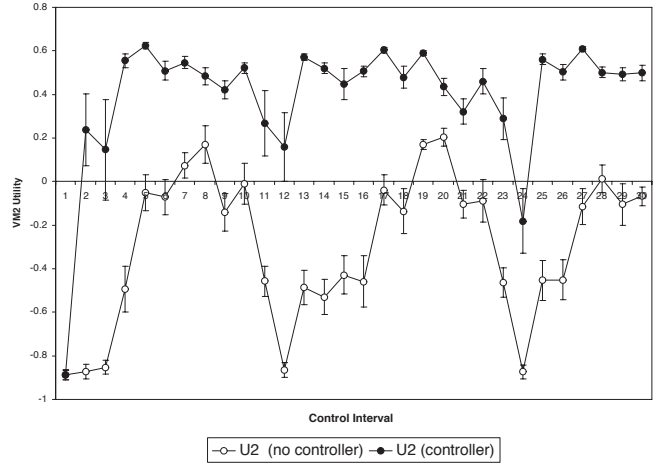
**Figure 7. Variation of priorities of VMs 1 and 2 as a function of time, measured in CIs.**



**Figure 9. Variation of response time for VM 2 as a function of time, in CIs, for the priority case.**



**Figure 10. Variation of the workload intensity, in tps, for the two workloads as a function of time, in CIs, for the CPU share case.**



**Figure 12. Variation of  $U_2$  as a function of time, in CIs, for the CPU shares case.**

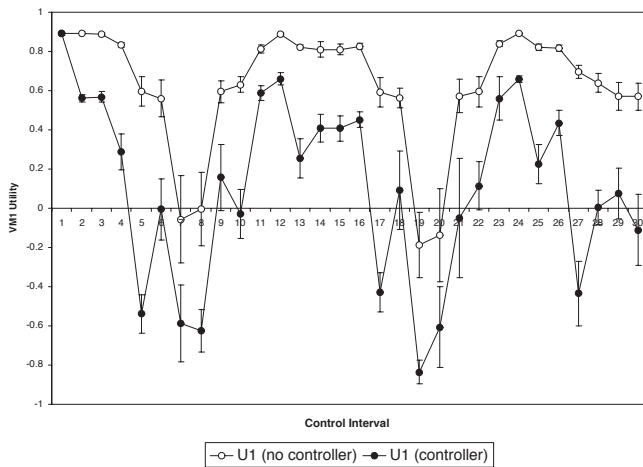
global utility function  $U_g$  causes the controller to tolerate temporary violations of SLA for VM 1. It should be noted that VM 1 received more shares during these peak periods. If that had not been done, the peaks in response time would have been higher. It should be noted also that in the non-controller case, VM 1 uses 50% of the CPU all the time while with the controller it never goes above 40%.

For the case of VM 2, Fig. 16 shows that as soon as the controller starts to operate (i.e., at CI = 2), the response

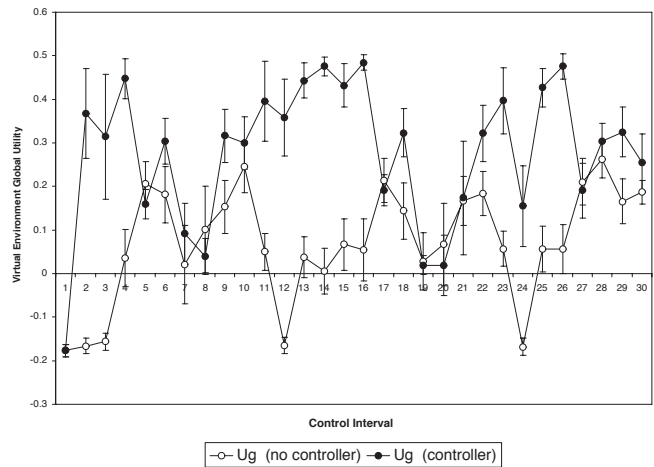
time goes below its SLA of 0.030 sec and stays at that value throughout the entire experiment, except at CI = 24, when it goes only slightly above. The figure also shows that without the controller, this workload exceeds its SLA target by a significant margin at the peak periods for VM 2.

## 6. Concluding Remarks

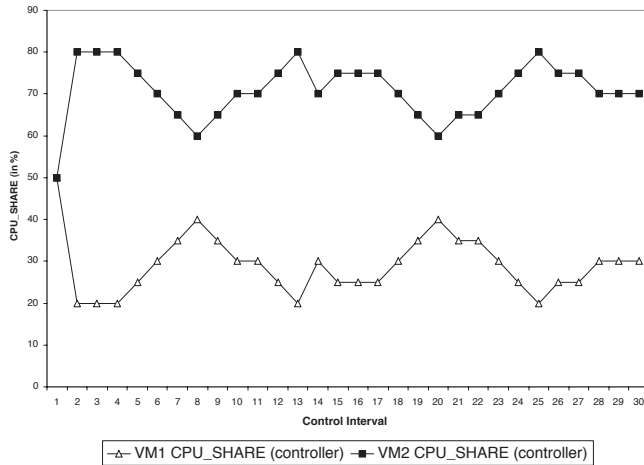
Virtualized environments can bring interesting benefits to data centers that want to consolidate smaller servers into



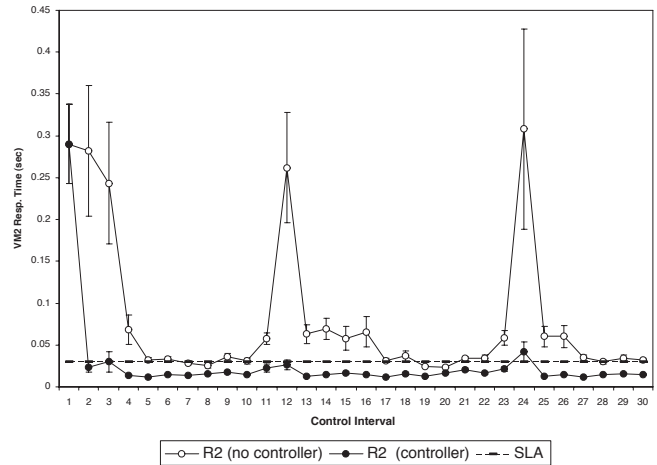
**Figure 11. Variation of  $U_1$  as a function of time, in CIs, for the CPU shares case.**



**Figure 13. Variation of  $U_g$  as a function of time, in CIs, for the CPU shares case.**



**Figure 14. Variation of the CPU shares of VMs 1 and 2 as a function of time, in CIs.**

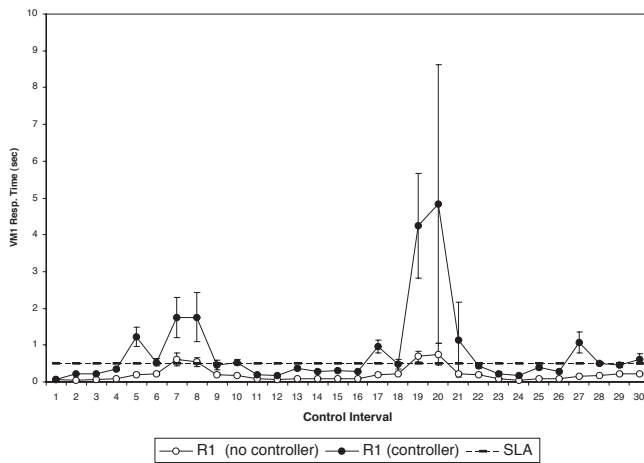


**Figure 16. Variation of response time for VM 2 as a function of time, in CIs, for the CPU shares case.**

higher capacity ones. An important challenge is that of managing the physical resources so that each virtual environment receives an appropriate share of these resources as the workload varies with time. This paper examined the issue of autonomic allocation of the CPU to several virtual machines. Two approaches were studied: dynamic priority setting and dynamic allocation of CPU shares. The former approach provides a less granular way of differentiating among the several virtual machines. Allocation of CPU

shares is a mechanism that allows a finer granularity for resource allocation.

The paper showed through simulation experiments that the autonomic controller is capable to dynamically shift CPU resources among virtual machines in a way that optimizes a global utility function for the virtualized environment. The results reported here indicate that carrying out these ideas to existing VMMs (e.g., Xen and VMWare) would be quite attractive.



**Figure 15. Variation of response time for VM 1 as a function of time, in CIs, for the CPU shares case.**

## Acknowledgements

This work is partially supported by grant NMA501-03-1-2022 from the US National Geospatial-Intelligence Agency.

## References

- [1] Babaoglu, O., Jelasity, M., Montresor, A.: Grassroots Approach to Self-Management in Large-Scale Distributed Systems. In Proc. EU-NSF Strategic Research Workshop on Unconventional Programming Paradigms, Mont Saint-Michel, France, 15-17 September (2004)
- [2] M.N. Bennani and D.A. Menascé, "Resource Allocation for Autonomic Data Centers Using Analytic Performance Models," *Proc. 2005 IEEE International Conference on Autonomic Computing*, Seattle, WA, June 13-16, 2005.
- [3] M.N. Bennani and D.A. Menascé, "Assessing the Robustness of Self-managing Computer Systems under Variable Workloads, *Proc. IEEE Intl. Conf. Autonomic*

- Computing (ICAC'04)*, New York, NY, May 17–18, 2004.
- [4] J. Chase, M. Goldszmidt, and J. Kephart, eds., Proc. First ACM Workshop on Algorithms and Architectures for Self-Managing Systems, San Diego, CA, June 11, 2003.
- [5] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, “Managing Energy and Server Resources in Hosting Centers,” 18th Symp. Operating Systems Principles, Oct. 2001.
- [6] R.J. Creasy, “The Origin of the VM/370 Time-Sharing System,” *IBM J. Research and Development*, Sept. 1981, pp. 483–490.
- [7] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, “Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics With Application to the Apache Web Server,” Proc. IEEE/IFIP Network Operations and Management Symp., Florence, Italy, April 15–19, 2002.
- [8] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat, “Model-Based Resource Provisioning in a Web Service Utility,” Fourth USENIX Symposium on Internet Technologies and Systems, March 2003.
- [9] R. Figueiredo, P.A. Dinda, and J. Fortes, “Resource Virtualization Renaissance,” *IEEE Internet Computing*, May 2005, Vol. 38, No. 5.
- [10] R.P. Goldberg, “Survey of Virtual Machine Research,” *IEEE Computer*, June 1974, pp.34–45.
- [11] D. A. Menascé and M.N. Bennani, “On the Use of Performance Models to Design Self-Managing Computer Systems,” Proc. 2003 Computer Measurement Group Conf., Dallas, TX, Dec. 7–12, 2003.
- [12] D. A. Menascé, “Automatic QoS Control,” *IEEE Internet Computing*, Jan./Febr. 2003, Vol. 7, No. 1.
- [13] D. A. Menascé, R. Dodge, and D. Barbará, “Preserving QoS of E-commerce Sites through Self-Tuning: A Performance Model Approach,” Proc. 2001 ACM Conf. E-commerce, Tampa, FL, Oct. 14–17, 2001.
- [14] Menascé, D. A., V. A. F. Almeida, and L. W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*, Prentice Hall, Upper Saddle River, NJ, 2004.
- [15] G.J. Popek and R.P. Goldberg, “Formal Requirements for Virtualizable Third-Generation Architectures,” *Comm. ACM*, July 1974, pp. 412–421.
- [16] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, eds, *Modern Heuristic Search Methods*, John Wiley & Sons, Dec. 1996.
- [17] M. Rosenblum and T. Garfinkel, “Virtual Machine Monitors: Current Technology and Future Trends,” *IEEE Internet Computing*, May 2005, Vol. 38, No. 5.
- [18] G. Tesauro, “Online Resource Allocation Using Decompositional Reinforcement Learning,” *Proc. AAAI-05*, Pittsburgh, PA, July 9–13, 2005.
- [19] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das, “Utility Functions in Autonomic Computing,” *Proc. IEEE International Conf. Autonomic Computing (ICAC'04)*, New York, NY, May 17–18, 2004.