Agile Resource Management in a Virtualized Data Center [*]

Wei Zhang[1], Hangwei Qian[2], Craig E. Wills[1], Michael Rabinovich[2]
[1]CS, Worcester Polytechnic Institute, Worcester, MA 01609
[2]EE&CS, Case Western Reserve University, Cleveland, OH 44106
[1]{weizhang,cew}@cs.wpi.edu, [2]{hangwei.qian,michael.rabinovich}@cwru.edu

ABSTRACT

In this paper we develop, implement and evaluate an approach to quickly reassign resources for a virtualized utility computing platform. The approach provides this platform *agility* using *ghost* virtual machines (VMs), which participate in application clusters, but do not handle client requests until needed. We show that our approach is applicable to and can benefit different virtualization technologies.

We tested an implementation of our approach on two virtualization platforms with agility results showing that a sudden increase in application load could be detected and a ghost VM activated handling client load in 18 seconds. In comparison with legacy systems needing to resume VMs in the face of sharply increased demand, our approach exhibits much better performance across a set of metrics. We also found that it demonstrates competitive performance when compared with scripted resource changes based on a known workload. Finally the approach performs well when used with multiple applications exhibiting periodic workload changes.

Categories and Subject Descriptors

C.5 [**Computer System Implementation**]: Servers

General Terms

Experimentation, Measurement, Performance

Keywords

Virtualization, Utility Computing, Agility

1. INTRODUCTION

Utility computing delivers computing resources to multiple Web applications across one or more data centers. A central problem in utility computing is managing the resources to support the varying demand for these applications. One approach is to simply over-provision the resources needed by each application, but this approach wastes resources. An alternate approach is to maintain a pool of spare resource capacity that is allocated to applications as they experience increased load and retrieved from applications that no longer need additional resources.

The problem of dynamic allocation of resources is to do it in a timely manner relative to changes in load. Load increases can be gradual making the detection and reallocation of resources relatively straightforward or increases can be sudden, such as what occurs with flash crowds [2], in which case timely detection and reallocation of resources is critical. We refer to the capability of a utility computing platform to quickly reassign resources as the *agility* of the platform [10].

A promising approach for achieving agility is to build utility computing platforms using virtual machines (VMs). Virtualization technologies typically offer control APIs that allow virtual machines to be programmatically stopped and booted, or suspended and resumed; this provides a foundation for an obvious approach for dynamic management of active VMs running a particular application.

However, these actions can take a long time when the virtual machine to be activated has been previously swapped out to disk. Furthermore, modern Internet applications typically run within application servers that often operate in an application cluster. The application servers on a suspended machine will be considered inoperable by the cluster and, as we found in initial work [10], take a long time to re-integrate into the cluster upon reactivation. One could attempt to compensate for this delay by activating a machine early, using an artificially low utilization threshold that triggers the activation. However, this leads to an increased overall number of active machines, which, as we will see in Figure 1, can be detrimental to performance.

As an alternate, we proposed the idea of *ghost* virtual machines [10]. Ghost VMs are spare VMs maintained on each physical machine within a utility computing platform that remain active, but detached from the Internet. These VMs do not service client requests, but their application servers do participate in their respective clusters so that when needed these ghost VMs can be activated by making their existence known to the front-end content switch receiv-

ing and redirecting client requests. We call these *ghost* VMs because their existence is "invisible" to the content switch and hence to the Internet. A small number of these ghost VMs on each physical machine are allocated a slice of the physical memory, but their CPU consumption is negligible. Once the decision is made to activate a ghost VM, resource allocation simply involves reconfiguring the content switch, which we found takes on the order of a few seconds.

While promising, our previous work only proposed and made a preliminary case for ghost VMs [10]. In the present work, we implement the ghost VM concept into a fully functioning utility computing prototype and evaluate the resulting platform. We further show that the ghost concept is general enough to apply across different virtualization technologies. To demonstrate this aspect, we implemented and deployed our architecture with two virtualization technologies at the opposite ends of the degree of resource isolation they provide. Specifically, we used the VMWare ESX technology, which allows VMs to be allocated hard slices of physical resources, and VMWare Server, which only guarantees memory allocation. Each technology also allows controls on the number of cores assigned to a VM.

Our focus on the agility is in contrast to related algorithmic work concerning resource allocation in shared data centers [7, 11]. While the latter seeks to find high-quality solutions for a required number and placement of application instances on physical machines, our goal is to reduce the time for enacting those decisions.

This work makes a number of contributions:

1. We introduce a utility computing platform, based on the use of ghost VMs, that provides agility in quickly reacting to changes in application demand.

2. By instantiating our platform on top of two distinct virtualization products, we show that our approach, including both the architecture and the resource allocation algorithm, is applicable across different virtualization technologies.

3. We evaluate our approach by testing a fully functioning prototype using an external benchmark application— TPC-W. Our results show that our approach effectively reassigns resources among applications as dictated by their changing demand and significantly improves the agility in reacting to demand changes.

In the remainder of the paper, Section 2 presents background on VM agility measures we have performed while Section 3 discusses the approach used to manage resources in a virtualized utility computing platform along with the reallocation approach. In Section 4 we discuss our implementation of our architecture and in Section 5 describe how its performance is studied in two utility computing platforms. Results from this study are shown and discussed in Section 6. We conclude the paper with a presentation of related work in Section 7 along with a summary and future work in Section 8.

2. BACKGROUND

Agility, the capability to quickly reassign resources in a data center, is a desirable property of a virtualized utility computing. As background for this work we revisit the relative costs of different approaches for reassigning resources. These costs are based on preliminary measurements reported in [10] as well as additional experiments we have performed for this work.

We consider four current approaches for reallocation of resources as well a new approach to resource reassignment— the deployment of ghost VMs. Each of these alternatives along with ghost VMs is described below, along with costs each incurs for reallocation.

1. **Stop/Boot of VM**. One approach for reassigning resources among applications is to stop some instances of VMs running an application with excess capacity and boot more instances of VMs running an application with insufficient capacity. In [10] we found the average startup time for a VM took on the order of 45-65s depending on how much of a VM's memory was already in place from previous use. Furthermore, we found that starting the cluster agent can take around 20s. We measured the time to start an application server to be about 97s on average when done for the first time after a VM reboot. Thus, the total time it takes before the new VM can serve client requests to be on the order of 180s—not a good result for agility.

2. **Redeployment of Application Servers On Demand**. Instead of stopping and rebooting VMs across hosts, in this method we consider pre-deploying VMs across the set of hosts and use them for different applications as dictated by the demand. In [10] we measured 95s to stop a cluster member, 19s to start a new cluster member and 97s to start the new application as a cluster member for an average time on the order of 210s. Again this approach does not achieve good agility.

3. **Live VM Migration**. One could add capacity to an application by migrating a VM with this application from a physical server that is highly utilized with competing applications to a physical server with more spare capacity. Recently, techniques have been developed for live migration of a VM, where the migration occurs without stopping the source VM, and the updates to the source VM state are forwarded to the target host during migration [5, 14]. This method dramatically reduces the downtime during migration but does not by itself improve the agility of resource reassignment. Results in [14] show that live VM migration can move a 256MB VM from one physical machine to another in 20s. However this measurement does not account for any disk space and VMs are increasing in size (e.g., we use sizes of 512MB and 1GB in our current work). Thus higher migration delays are more realistic. In addition, the source machine will not be relieved of the overload due to the operation of the source VM until the migration is complete.

4. **Suspend/Resume VMs**. This method also uses a number of pre-deployed VMs across all hosts, but each VM always runs its assigned application. With some number of VMs left alive for serving client requests in the initial stage, all other VMs are suspended. We then suspend or resume VMs on different hosts depending on the observed demand for their corresponding applications. In [10], we measured the time to resume a suspended VM with one competing active VM to be 14s on average. However, these experiments did not have the suspended VM memory pushed out of RAM and

only accounted for when the resume API completes, not when the application is ready to process requests. In subsequent experiments we have found the resumption time to be between 40 and 80s depending on the number of competing VMs and the degree to which VM memory has been pushed out of RAM.

5. **Active/Ghost VMs**. In this new approach, first proposed in [10], we pre-deploy VMs, each with its own application(s), on hosts in the data center. In contrast to previous methods, all VMs are alive using this approach. However, the switch at each data center is configured so that it is not aware of all of these VMs and consequently not all are used to service client requests. We refer to VMs that are known by the switch and can handle client requests as *active* VMs and the others as *ghost* VMs[1]. While hidden from the public Internet behind the switch, the ghost VMs can still communicate with each other using either the hosts' second network card, available on most modern servers, or through the L2 functionality of the switch. All VMs (both active and ghosts) use this "private" network for managing application clusters. Thus, the application servers on ghost VMs fully participate in their application clusters and maintain both their membership in these clusters and the current application state. When another active VM for an application is needed, if a ghost VM for the application is available then we found in [10] that it takes as little as 2s to reconfigure the switch before the activated VM is handling requests.

Given the relative performance for each of these reconfiguration approaches, this work seeks to exploit the agility of the ghost VM approach for rapid resource reallocation. The architecture for this approach and the algorithm we used to manage its resources are described in the following section.

3. APPROACH

Our work focuses on the problem of resource reallocation providing agility within a data center for a virtualized utility computing platform. We use a typical model that each data center is deployed using a three-tiered model with a front-end content switch redistributing client requests to a set of application servers backed by one or more underlying database servers. Each VM runs only one application server and each application server runs only one instance of an application. We assume that all physical machines running VMs with application servers are identical and each can support the execution of multiple VMs. Multiple instances of an application can exist if multiple VMs for that application are active.

In conjunction with our resource reallocation algorithm is the request distributed algorithm used by the front-end switch to load balance incoming client requests amongst the set of active VMs for an application. In the model for our algorithm and its deployment we configure the switch to use a weighted round-robin algorithm to distribute requests

[1]In principle, the same effect could be achieved if the switch was aware of the ghost VMs but configured not to send any requests to it. Unfortunately, with at least one of the switches we use (the Alteon switch - see Section 5) the minimum load one can specify for any active machine is one concurrent connection. Thus, the only way to shield a machine from load is to "hide" it from the switch.

in proportion to the relative capacity of each active VM for an application. However, due to variations in request complexity and competing VMs on each physical machine, the load amongst the active VMs for an application may exhibit some imbalance requiring ongoing monitoring of the VMs.

Given this model of a data center, in the remainder of this section we define measures of capacity and utilization, show how reconfiguration changes affect these measures and present the state transition diagram used by the resource reallocation algorithm we developed for such a platform. The algorithm was developed with a number of key ideas:

1. Ghost VMs are used as a "stepping stone" for rapid promotion to and demotion from active VMs running an instance of an application. Because these ghost VMs are already participating in the cluster management of their respective applications, deployment of these ghosts simply requires reconfiguring the content switch.

2. Extra capacity within a data center is not deployed to VMs until needed and some amount of this extra capacity is dedicated to ghost VMs so deployment can be done quickly. The alternative is to dedicate all capacity of a data center to active VMs, but our preliminary experiment, with results shown in Figure 1, indicates that needlessly activating VMs, versus leaving them in the ghost state, does not improve reply rate and response time for incoming requests. In this experiment, we used *httperf* to send a total of 1700 requests/sec to a sample WebSphere application deployed on seven VMs within the same physical machine. Each VM is either in active or ghost state, and the requests are load-balanced among the active machines. The VMs are all configured to have 0.5GB memory and run on a server with 4G physical memory, to avoid VM memory swapping. The results show that, even when all VMs together fit into physical memory, unneeded active VMs do not improve, and actually degrade, the overall system performance. We attribute this to the scheduling overhead of VMs, such as context switching.
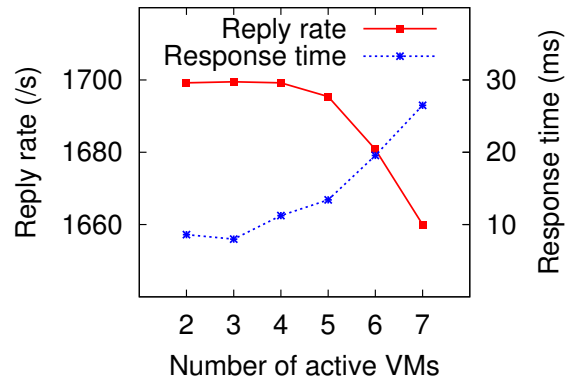

Figure 1: Impact of Active/Ghost VM Tradeoff on Reply Rate and Response Time

3. The resource reallocation algorithm is controlled by high and low watermarks, which define when a VM is overloaded or underloaded. The goal of the algorithm

is to allocate additional resources and deallocate excess resources so VMs are neither over nor underloaded.

4. Additional resources can be assigned to an application either by increasing the capacity of existing instances of an active VM running the application or by promoting a ghost VM of the application. When possible, overloaded conditions are alleviated by increasing the capacity of an existing VM so as to minimize the number of active VMs. Similarly, resources can be deallocating by reducing the capacity of a VM or demoting an active VM to a ghost VM. No more than one VM (active or ghost) is allocated to an application on a physical machine—if additional capacity exists on the physical machine then the capacity of an existing active VM should be increased rather than creating an additional VM on the machine.

5. The algorithm is intended for deployment on a variety of virtualization products with the only requirement that each provides a means to bound the amount of memory that each VM consumes on a physical machine. The algorithm works with products supporting this minimal feature, but is also able to exploit features of other products that allow proportional allocation of CPU and network resources to each VM.

6. Because virtualization products do not allow the amount of allocated memory to be increased once a VM is created, we focus capacity and utilization measures on CPU consumption, although the algorithm could be easily extended to account for a weighted combination of memory, CPU and network consumption.

3.1 Capacity and Utilization

An important aspect of our approach is that it formulates a general framework that applies to different virtualization technologies. This approach simplifies the construction of large utility computing platforms from heterogeneous server clusters. The basic notions behind our general framework are observed and projected capacity and utilization. These notions are defined based on one key notion—the capacity of a virtual machine. We re-target the framework by simply redefining this one notion for a particular virtualization technology. For virtualization technologies that provide strong resource isolation by allowing one to configure virtual machines with hard slices of physical resources, such as ESX, the VM capacity is determined by the actual slice. For the virtualization technologies that do not provide this capability, such as VMWare Server, our framework treats the VM capacity as a *desired capacity*. The desired capacity is a soft limit that is not enforced by VMWare Server, but is adhered to in our algorithm. The algorithm adjusts the "soft slice" capacity for each VM over time and because it runs periodically, the duration a VM will be overloaded is limited. Note that with soft slices, the observed VM utilization (since it is computed relative to the desired capacity) can exceed 100%.

3.1.1 Observed Capacity and Utilization

Let $c(P_i)$ be the capacity of physical machine (PM) P_i and $c(V_{i,j})$ be the capacity of the jth VM on P_i. Intuitively, the capacity of a PM is greater than or equals to the total capacity of its VMs. Thus, we assume that VM capacities are assigned to satisfy the following constraint where m_i is the number of VMs on PM P_i:

$$c(P_i) \geq \sum_{j=1}^{m_i} c(V_{i,j}) \qquad (1)$$

Let $u(V_{i,j})$ be the resource utilization on PM P_i due to running VM $V_{i,j}$. Let $u_r(V_{i,j})$ be the relative utilization of VM $V_{i,j}$, then

$$u_r(V_{i,j}) = u(V_{i,j}) \times \frac{c(P_i)}{c(V_{i,j})} \qquad (2)$$

For example, suppose the capacity of $V_{i,j}$ is 30% of P_i. If $V_{i,j}$ consumes 24% of the resources on P_i, i.e., $u(V_{i,j}) = 24\%$, then the relative utilization $u_r(V_{i,j}) = 80\%$. That is to say, $V_{i,j}$ consumes 80% of its assigned resources.

Let $u(P_i)$ be the overall resource utilization of PM P_i. Intuitively, the overall utilization equals to the sum utilization of its VMs (we ignore the small overhead for the host OS or hypervisor), i.e.,

$$u(P_i) = \sum_{j=1}^{m_i} u(V_{i,j}) = \frac{\sum_{j=1}^{m_i}(u_r(V_{i,j}) \times c(V_{i,j}))}{c(P_i)} \qquad (3)$$

Let $c(A_k)$ be the total capacity of application A_k. It is the sum capacity of VMs running application A_k, i.e.,

$$c(A_k) = \sum c(V_{i,j}), \quad \forall \, app(V_{i,j}) = A_k \qquad (4)$$

Let $u(A_k)$ be the average server utilization of application A_k, then

$$u(A_k) = \frac{\sum(u_r(V_{i,j}) \times c(V_{i,j}))}{c(A_k)}, \quad \forall \, app(V_{i,j}) = A_k \qquad (5)$$

We introduce two load watermarks into our platform, *high-watermark* (HW) and *low-watermark* (LW). We define the overloaded/underloaded situation for the servers and applications as follows:

If $u_r(V_{i,j}) > HW$, $V_{i,j}$ is overloaded. If $u_r(V_{i,j}) < LW$, $V_{i,j}$ is underloaded. If $u(P_i) > HW$, P_i is overloaded. If $u(P_i) < LW$, P_i is underloaded. If $u(A_k) > HW$, A_k is overloaded. If $u(A_k) < LW$, A_k is underloaded.

For example, suppose $HW = 90\%$, then any VM which consumes more than 90% of its assigned resource, or any application which has an average utilization over 90%, is considered overloaded.

The goal of our resource reallocation algorithm is reassigning resources among VMs, PMs, and applications, so that none of them is overloaded, i.e., $\forall i, j, k$, none of $u_r(V_{i,j})$, $u(P_i)$, or $u(A_k)$ is greater than HW. In doing so we focus on keeping the VMs from being overloaded.

If a VM $V_{i,j}$ running A_k is overloaded, then the maximum CPU usage for $V_{i,j}$ could be increased if such spare capacity exists on P_i . Alternately we could try to bring up one or more VMs for A_k elsewhere to share the load, i.e., increase the capacity for the application rather than for the VM through the promotion of a ghost VM. Since the total capacity of the platform will not change, increasing the capacity for one application may require decreasing the capacity of others, so that the constraint (1) remains satisfied.

Let $c'(A_k)$ be the new capacity of application A_k after resource reallocation. The goal of our algorithm is to find capacities for all active VMs such that none will be overloaded.

3.1.2 Projected Capacity and Utilization

In the following we express how the change of capacity affects the load distribution, and in turn affects the utilization of VMs. Let $u'(A_k)$ be the projected average server utilization of application A_k after reallocation, then

$$u'(A_k) = \frac{u(A_k) \times c(A_k)}{c'(A_k)} \qquad (6)$$

Intuitively, if we choose a proper $c'(A_k)$, we are able to keep the projected average utilization $u'(A_k)$ under the *high-watermark*. However, switch load balancing may not be perfect and individual VMs for the application may still be overloaded.

Let $u'_r(V_{i,j})$ be the projected relative utilization of $V_{i,j}$ after reallocation. Our goal is to keep the utilization of every $V_{i,j}$ under HW. We assume that the change of utilization is in proportion to the change of capacity. For example, suppose we have two VMs $V_{1,2}$ and $V_{3,4}$ with the same capacity, but for some reasons, their utilization is not the same ($u_r(V_{1,2}) = 80\%$ and $u_r(V_{3,4}) = 60\%$). If we bring up two more VMs (double the total capacity), we assume that the projected utilization will drop in proportion to the current utilization ($u'_r(V_{1,2}) = 40\%$ and $u'_r(V_{3,4}) = 30\%$). That is,

$$u'_r(V_{i,j}) = \frac{u_r(V_{i,j}) \times c(A_k)}{c'(A_k)}, \quad \forall\, app(V_{i,j}) = A_k \quad (7)$$

To bring utilization below HW, we need $u'_r(V_{i,j}) < HW$ i.e., from Equation (7),

$$c'(A_k) > \frac{u_r(V_{i,j}) \times c(A_k)}{HW}, \quad \forall\, app(V_{i,j}) = A_k \quad (8)$$

So, we need at least this amount of capacity to guarantee that $V_{i,j}$ will not be overloaded. If we calculate $c'(A_k)$ for all $V_{i,j}$ running application A_k then get the maximum value. It is the value that keeps the utilization of all its VMs in bound.

We define $\Delta c(A_k) = c'(A_k) - c(A_k)$ as the change of capacity. Note that for some applications, Δc might be negative which indicates a decreasing demand. This indicates that the current capacity is such that the maximum utilization of any VM running this application is below HW. We use LW to decide we can reduce the capacity of an application as follows.

By Equation (7) and (8), we define $c'(A_k)$, the minimum capacity of application A_k, which guarantee A_k will not be overloaded. Similarly, we define $c''(A_k)$ as the maximum capacity of application A_k, guaranteeing that it will not be underloaded.

$$u''_r(V_{i,j}) = \frac{u_r(V_{i,j}) \times c(A_k)}{c''(A_k)} > LW, \quad \forall\, app(V_{i,j}) = A_k \quad (9)$$

and

$$c''(A_k) < \frac{u_r(V_{i,j}) \times c(A_k)}{LW}, \quad \forall\, app(V_{i,j}) = A_k \quad (10)$$

Again, because load balancing may not be perfect, we choose the minimum value as the final $c''(A_k)$ for application A_k which keeps all of its VMs from being underloaded. for any application A_k, if $c(A_k) > c''(A_k)$, we can safely decrease its capacity.

Thus, our resource reallocation algorithm is transformed to a problem of satisfying all positive $\Delta c(A_k) = c'(A_k) -$

$c(A_k)$ by using negative $\Delta c(A_k) = c''(A_k) - c(A_k)$ or residual PM capacities.

3.2 Resource Reallocation Algorithm

Given these specifications, the resource reallocation algorithm is periodically executed within a data center using data gathered about the current status of PMs, VMs and applications. The multi-stage algorithm iterates over each of these entities. A diagram showing the transition between states for an application within a data center is given in Figure 2.

The diagram illustrates a number of distinguishing aspects of our approach. First, each active VM executing the application is represented as a different state. Once at least one instance of an application has been activated then one or more ghost VMs may be in existence between each of the active VM states.

Second, the activation of the first active VM within a data center is under the control of a global decision manager, which determines the placement of applications amongst a set of geographically dispersed data centers along with the assignment of user requests to specific data centers. While not all applications are active at each data center, we do assume that suspended instances of an application exist in all data centers. If not, then the initial activation of an application requires that a VM be created and the application to be started, which takes much longer than resumption. Once one instance of an application has been activated at a data center then management of that application is controlled by a local decision manager, which performs the algorithm described in this work. Details of the global resource manager and how it interacts with the local decision manager are current work, but not discussed in this paper. The focus of this paper is how the local decision maker manages available resources within a single data center.

Third, once an VM is active for an application within a data center, the operations of the local decision manager can be divided into three stages: decreasing the capacity for those applications with too much, increasing the capacity for those applications with too little, and managing the availability of ghost VMs so they will be available to the applications that are most likely to see an increase in demand. More details on each of these stages is described below.

The first stage of the algorithm seeks to remove extra capacity from underloaded applications with operations D1, D2 and D3 shown in Figure 2. If the utilized capacity of any application drops below the LW threshold then a VM with a relatively low capacity compared to the application is demoted to a ghost VM as shown with operation D1. If its capacity is relatively higher then its capacity simply reduced as shown with operation D2. The algorithm will not demote the last VM for an application in order to ensure there is some capacity in the data center for the application. Only in case that the global decision manager decides to de-activate the application at this data center is operation D3 invoked, which causes the last active VM to be suspended.

Now that any extra capacity for underloaded applications has been reclaimed, the next stage of the algorithm seeks to reallocate residual capacity if applications are overloaded. This increase in application capacity is accomplished via one of three operations. If available capacity exists on one of the physical machines currently running an active VM of the application then the capacity of that VM is increased as shown
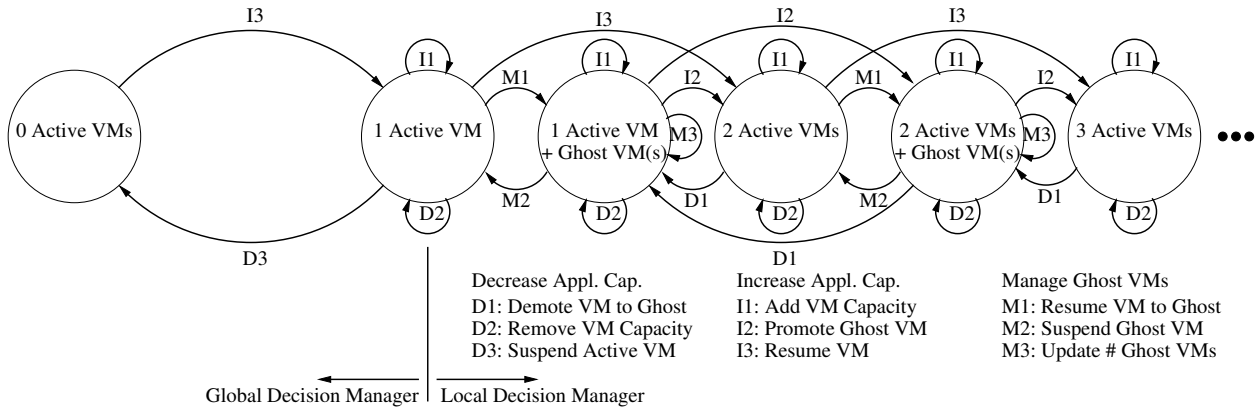
Figure 2: Application State Transition Diagram

with operation I1. If increasing the capacity of an active VM does not meet the application's demands and a ghost VM is available then the ghost VM is promoted as indicated with operation I2. Finally, if a ghost VM is not available then a suspended VM for the application must be resumed directly to the active state via operation I3. This is an undesirable situation from the standpoint of agility as resumption of an active VM takes much longer than promotion from a ghost VM. This stage of the algorithm continues to iterate until all overloaded applications are allocated sufficient capacity or there exists no more available capacity.

The final stage of the algorithm detects and handles management of ghost VMs. This function is implemented through a ghost manager, which runs periodically to reallocate ghosts among applications, and is also invoked explicitly any time the ghost allocation changes through ghost promotion or suspension. We assume that the amount of additional capacity that an application might need quickly is proportional to its current load, thus the ghost manager allocates the available number of ghosts among applications in proportion to their observed load. However, this correlation may not always exist as a heavily loaded application may exhibit relatively constant demand while a lightly loaded application may have much potential for a sharp increase in demand. We leave more elaborate ghost management algorithms for future work.

When a ghost VM for an application is allocated, the ghost manager is likely to create a new ghost instance by resuming a suspended VM as shown with operation M1. Similarly if an application currently has a ghost VM determined to be no longer needed, operation M2 suspends the ghost VM. Finally, operation M3 in Figure 2 shows that once at least one ghost VM exists for an application, the number of ghost VMs can be updated as application conditions and available capacity exist. Because ghost VMs are specific to an application, it is possible for a ghost VM to be switched from one application to another, but only after stopping the current application on the VM then starting the new application. As we found in [10] the time to create/resume a VM, start an application and join an application cluster can be on the order of minutes so it is important to keep ghost VMs available.

4. IMPLEMENTATION

We implemented our approach in a testbed that contains two data centers, representing the two major types of virtu-

alization technology. One data center uses VMWare Server, which operates VMs on top of a regular host operating system and relies on the host OS for the VM scheduling (thus giving us little control over resource sharing within the same physical machine). The other data center utilizes VMWare ESX Server, which uses its own VM monitor in place of the host OS and allows precise allocation of various resource to individual VMs on a physical machine. Employing these distinct virtualization technologies allows us to demonstrate the flexibility of our approach and the resource allocation algorithm at its heart.

Each data center uses the same architecture shown in Figure 3. As shown, the Resource Manager has three components: Resource Collector, Resource Reallocator and the Ghost Manager. The Resource Collector communicates with agents on each of the PMs to gather average physical CPU utilization due to each VM. In the setup we used for our experiments, the agent on each PM makes periodic instantaneous utilization measurements (every 2sec using the UNIX *top* command for VMWare Server and every 1sec using an equivalent command for ESX) and keeps the moving average of the last three of these measurements. The Resource Collector obtains the current values of these averages from each agent every 10sec and uses them as the utilization metric in the algorithm. The Resource Collector also implements the portion of our approach dealing with removal of excess capacity shown in operations D1-D3 in Figure 2.
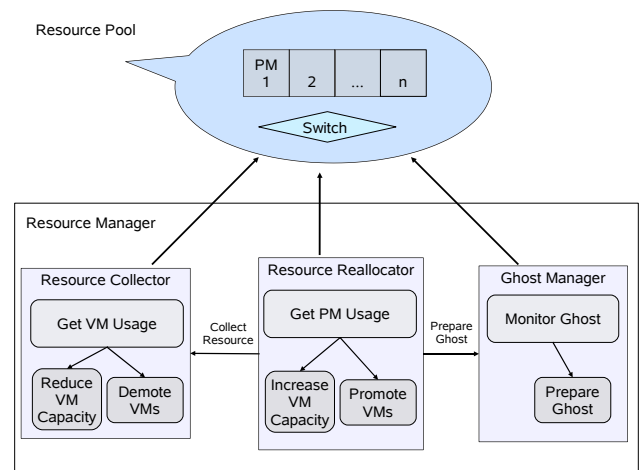

Figure 3: Architecture of Local Resource Manager.

The Resource Reallocator performs the resource reallocation portion of our algorithm (operations I1-I3 in Figure 2) in reevaluating whether overloaded applications are detected based on the collected data. It is currently configured to execute every 10sec. It modifies the capacity of existing VMs and promotes ghost VMs. The Ghost Manager runs every 120sec monitoring the set of ghost VMs in the system and resuming new ghost VMs if an application is more likely to need additional capacity, and suspending existing ghost VMs if an application is less likely to need additional capacity as shown in operations M1-M3 of Figure 2.

Our approach was deployed using fixed size memory allocations for VMs on each platform. As mentioned, the distinction in the deployment between the two platforms is how the assigned CPU capacity for each VM is mapped to the features of the platform. In the case of VMWare ESX, a Min_CPU and Max_CPU value can be set for each VM defining hard limits on the range of CPU usage by the VM. In our initial testing we have chosen to set each of these values to the CPU capacity assigned by the Resource Reallocator, i.e., Min_CPU = Max_CPU = $c(V)$. This ensures that the VM receives a hard slice for its capacity. Alternately, the soft slice behavior could be mimicked by setting the Max_CPU value greater than the assigned VM capacity if additional capacity exists on the PM. Ghost VMs are assigned a small fixed slice. Other virtualization products, such as Xen and Linux Vserver, support capacity caps or shares. An implementation of our approach could easily use these features.

Before moving on to our study of the approach we note that it could easily be deployed on a different, yet important, model for a data center. Some data centers employ a one-application-per-machine model meaning that only one active application server is allowed to execute on a physical machine at a time. This approach, often used to satisfy customers' desire for absolute performance and security isolation from other applications, results in poor agility as increased load requires starting a new machine or having a "hot spare" available. An alternate approach available with our algorithm is to deploy only one active VM per physical server, but use additional memory (rather than an entirely new machine) to run one or more ghost VMs of other applications. Thus if the load of one application declines, one of its active VMs can be demoted while a ghost VM of another application can be promoted.

5. STUDY

Our approach has been successfully deployed in two data centers—one using VMWare Server v1.0.1 for virtualization and the other using VMWare ESX Server v3.5.0. Each data center uses WebSphere V6 as the application server software with a back-end database server running Oracle. Each application server runs within its own VM.

The VMWare Server data center consists of three physical machines each hosting up to three virtual machines. Each physical machine has two CPU cores and 2GB of RAM. Each virtual machine is configured with one virtual core and 512MB of memory. A Cisco Content Switch 11501 is used on the front end.

The ESX Server data center consists of three physical machines each hosting up to four virtual machines. Each physical machine has four cores and 4GB of RAM. Each virtual machine is configured with one virtual core and 1GB of memory. A Nortel Alteon 2208 application switch is used on the front end.

Within each data center, each VM is assigned one virtual CPU core. According to previous work [6], this option has good performance and utilization of the system. Each application within each data center is deployed with at least one active and one ghost VM. Further details about experiments are presented along with results in the following section.

The primary testing tool we use is TPC-W [12], a well-known "bookstore" workload for multi-tiered applications. We use the "browsing" request mix of TPC-W for testing. Requests are generated by Emulated Browsers (EBs). To increase the generated workload, two changes were made to the standard TPC-W configuration: the mean think time was reduced tenfold from the default of 7s to 700ms; and the TPC-W client software was modified to confine a client to a specific range of users for a test. This change allows testing from multiple client machines under the control of a testing harness that we created.

Initial tests were performed to measure the agility of our approach. For these tests we measured the time for the algorithm to detect that an increased workload requires additional active VMs and the time for a promoted VM to begin handling requests after switch reconfiguration. Similarly, we measured the time to detect and demote an active VM when the application workload is decreased.

We next examined how the faster resource reallocation of our approach translates into an end-to-end platform agility. To this end multiple applications with different load scenarios were run causing all aspects of the algorithm to be employed in not only promoting ghost VMs to active, but also for the ghost manager to recognize and resume suspended VMs to the ghost state. These tests confirmed the correct performance of our approach, and showed significant performance benefits over a legacy system that can respond to load changes only by resuming or suspending VMs.

We compared performance of our platform using ghost VMs with the legacy approach using four performance metrics available from the TPC-W clients. We determine the *request error rate* and the *number of successful requests* during the 10-minute interval in which each TPC-W client executes. Overloaded application servers result in increased request errors and fewer client requests that are handled. We also used TPC-W to determine the *slow response rate*, which is when the response time for successful client requests is greater than a threshold. Finally, we used TPC-W to measure the median response time for all successful requests.

Finally, we examined the ability of our approach to dynamically reassign resources among competing applications within a data center in response to changing demand. Specifically, we contrasted resource allocation and the resulting performance of three approaches:

1. Variants of the algorithm—we use our algorithm to manage resources with different parameter settings for the HW and LW thresholds.

2. Fixed—a fixed number of VMs were allocated to each application at the beginning of a test and this number was not changed during the test. Two fixed cases were used, one in which the number of VMs was intentionally under-provisioned for at least some portion of the test and one in which the number of VMs was intentionally over-provisioned.

3. Manual—predetermined promotion and demotion changes are manually scripted for the test based on a priori knowledge of workload changes. This test is introduced to understand expected best case behavior, although there is some tradeoff as to whether maximize performance or minimize the number of VMs in use.

We compared these test approaches using the TPC-W client summary performance metrics described above as well as measuring the *overload rate* at each VM. This rate is the percentage of time that the CPU utilization of the VMs for an application are greater than 80%, which is used for consistency across all approaches.

6. RESULTS

We ran a number of experiments to test the correctness and performance of our approach under different load conditions. Tests were performed for both platforms, although results are only shown for both platforms where appropriate. The first set of tests measure the agility of our platform in responding to rapid increases or decreases in load by promoting and demoting ghost VMs. The next set of tests demonstrates all aspects of our algorithm and compares performance of our platform with that provided by legacy systems. We then compare performance of our approach with other approaches for different load growth rates. We go on to show results for a multi-application workload that we constructed. We conclude our presentation of results with a time-varying workload of three applications for each platform.

6.1 Ghost Promotion and Demotion

Two critical measures in resource reallocation are the time needed to detect when a load increase has occurred and then time to activate additional resources. Figure 4 shows a scenario where a TPC-W application is running against our VMWare Server platform causing a 40% CPU utilization for the VM serving this application. At time $t = 180$ seconds in the scenario, the client load is sharply increased causing the CPU utilization to similarly rise. Using a HW threshold of 70%, the algorithm soon detects that additional capacity is needed for this application and at time $t = 193$ seconds makes a call causing the switch to promote a ghost VM for the application. At time $t = 198$ seconds the newly activated VM begins handling requests as evidenced by the drop in CPU utilization for the first VM shown in Figure 4.

We performed four tests and observed a median detection time of 13 (range 11-15) seconds and median switch reconfiguration time of 5 (3-5) seconds for a total of 18 (14-20) seconds to detect and act upon a sharp increase in load. This result demonstrates good agility by our approach in rapidly responding to significantly increased resource demands of an application. Detection time could be improved by reducing the interval between load monitoring, which is currently done every 10 seconds with the average computed over the last three measurements. In contrast, creating additional capacity by migrating a VM, which was smaller than our VM, to an available machine took almost 20 seconds after detection [14], thus our approach reduced this delay by a factor of four.

While the response time is not as critical, we ran similar tests for a sharp reduction in the load for an application. We started with two active VMs for an application and dropped
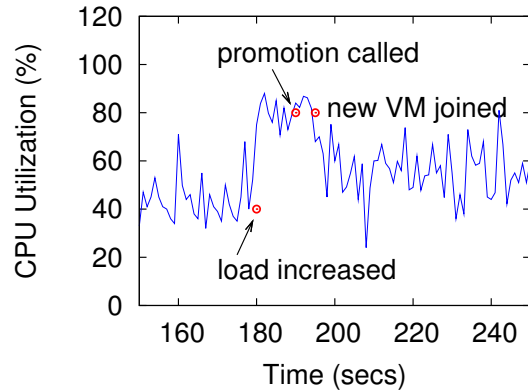

Figure 4: Detection and Promotion Delay (VMWare Server Platform)

the load so only one active VM is needed. In these tests we found the average detection time for demotion of a VM is 31 seconds with a total time of 35 seconds. These numbers make sense as we observe that the algorithm first tries to reduce the capacity of a VM before it eventually decides to demote the VM to a ghost.

6.2 Platform Agility

In our next set of tests we deployed three applications (all separate copies of TPC-W) on each of our platforms. Two of the applications, App1 and App2, are initially configured to have one active, one ghost and one suspended VM. The other application, App3, initially has one active and two ghost VMs. The initial load for each application is 10 emulated browsers on our ESX platform where each EB corresponds to the activity of one user in TPC-W.

During all tests, the load of two applications App1 and App3 is kept constant. However two separate tests are run where the load pattern of App2 is varied as shown in Figure 5. In the first test, a fast load growth to 85 EBs occurs for App2. In a separate test, a slow-growing load pattern is used that gradually increases the client load to 60 EBs.
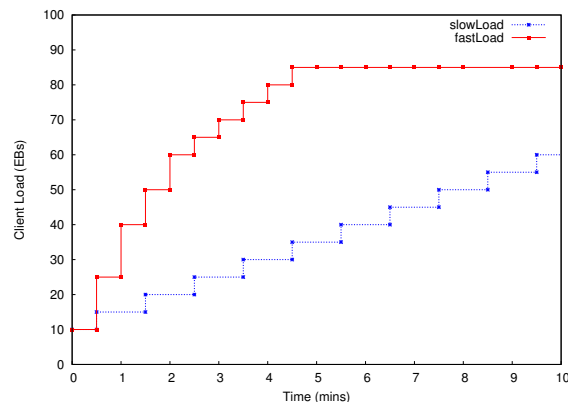

Figure 5: Load Growth Patterns used for Platform Agility Tests

Figure 6 shows a time-series graph of the CPU load as well as the capacity for each of the three VMs for App2 during the lifetime of the test for the fast-growing load pattern. Note that VM1 is initially active and has a 100% utilization that is allowed while VM2 is a ghost VM and as a nominal

136

Table 1: Performance Comparison of Ghost VM vs. Legacy Approach on ESX Platform

| Approach/ | # Errors | % Slow Requests | | | Median Req. | # Successful |
Growth Rate	(%)	>100ms	>500ms	>1000ms	Time (ms)	Requests
Ghost/Fast	23 (0.0)	18.9	2.6	1.6	46	84089
Legacy/Fast	128 (0.2)	36.6	5.7	2.1	69	61556
Ghost/Slow	14 (0.0)	15.8	3.4	2.1	38	43819
Legacy/Slow	24 (0.1)	14.2	3.4	2.2	37	35608

Table 2: Performance Comparison of Ghost VM vs. Legacy Approach on VMWare Server Platform

| Approach/ | # Errors | % Slow Requests | | | Median Req. | # Successful |
Growth Rate	(%)	>100ms	>500ms	>1000ms	Time (ms)	Requests
Ghost/Fast	386 (0.6)	67.9	12.9	2.7	165	65692
Legacy/Fast	2235 (3.9)	68.0	18.2	5.0	182	56747
Ghost/Slow	0 (0.0)	44.8	4.1	1.4	88	40281
Legacy/Slow	0 (0.0)	42.0	5.2	1.4	78	39694

slice of 12%[2] As shown at time 70s, the resource reallocator detects and promotes ghost VM2 for App2 to be active. In addition, a message is sent to the ghost manager indicating that no more ghost VMs exist for this application. Normally the ghost manager executes every 120s, but in this case the ghost manager causes the suspended VM3 for App2 to be resumed to the ghost VM state at time 116s. Finally, the resource reallocator determines that additional capacity is needed and promotes VM3 to active at time 132s. Due to the sharp increase in load for App2, the newly activated VM3 will not be fully resumed when it is transitioned to the active VM state, but at least it has already begun resumption. Alternately, we could wait until it is fully resumed before activation, but our approach allows the new VM to be used as soon as it is ready. Similar functioning for the fast load growth is shown in Figure 7 for the VMWare Server cluster, although the timing of actions is a bit different from Figure 6 as the physical servers have different capabilities in the two clusters.

We performed separate tests using the slow-growing load pattern, although a time-series graph of actions is not shown. In that test the ghost manager ran after a 120s interval and resumed another ghost VM for App2 because it had much more demand than the other applications. Thus App2 had two ghost VMs and when the load kept increasing, both of them were promoted to active.

To compare with a legacy system that does not utilize ghosts, we disabled the ghost manager portion of our platform so that the promotion and demotion of ghost VMs was not possible. Instead the platform could only resume or suspend VMs in and out of the active VM state. This approach mimics the behavior of a legacy system where resource reallocation can only be done with resumption and suspension of active VMs. We re-ran our fast- and slow-growing load test where initial ghost VMs in our platform are replaced with suspended VMs. Thus the initial state of VMs for App2 consists of one active VM and two suspended VMs. We re-tested each of the load growth patterns for App2 with this legacy platform and resource reallocation causing resumption of suspended VMs in the face of increased load. Table 1 shows a summary comparison of the TPC-W performance

metrics for each of the four testing scenarios, for the ESX cluster, and Table 2 shows the same results for the VMWare Server cluster.
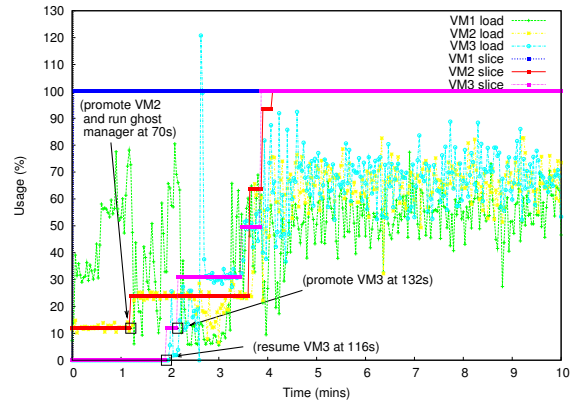

Figure 6: Platform Performance and Response for Fast Growing Load (ESX Cluster)
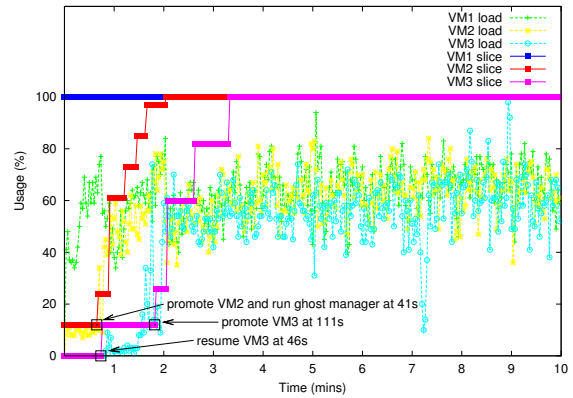

Figure 7: Platform Performance and Response for Fast Growing Load (VMWare Server Cluster)

Both tables show similar performance results across the virtualization technologies. In the fast growth case, our approach significantly outperforms the legacy system in all metrics we considered: the error rate, response time, and the number of completed requests. The higher agility of our approach allows the system to reassign resources among ap-

---

[2]All VMs are configured to run on one CPU core, and our utilization numbers all refer to the utilization of a single core. For example, the ghost's 12% slice means it overall consumes at most 3% of our 4-core physical server.

137

plications quickly to keep pace with the demand patterns. The legacy system experienced periods of degradation due to slow resource reassignment.

In the slow growth scenario, both ghosts and legacy platform were agile enough to cope with the growing demand. In fact, the legacy systems had somewhat better median response time, which we explain by the ghost overhead. Still, in the ESX case, the legacy system processed fewer total requests than our system. Because TPC-W clients wait for a current response before submitting the next request, this indicates that the legacy system experienced occasional slowdown periods.

6.3 Dynamic Resource Allocation

This section compares the resource allocation and resulting performance of our approach with that of the fixed and manual resource allocation as described earlier. We first examine the scenario where the load for an application exhibits linear growth followed by linear decline back to the original level. This workload was first proposed in [3], to study the behavior of a system under a flash crowd of varying severity. We then consider a more dynamic scenario where multiple applications experience varying levels of demand.

6.3.1 Steady Load Growth/Decline

Similar to [3], we performed an experiment to show how our algorithm performs with different load growth rates. The four workload patterns that we tested are shown in Figure 8 where each pattern exhibits a four-fold increase in load with different rates of increase.
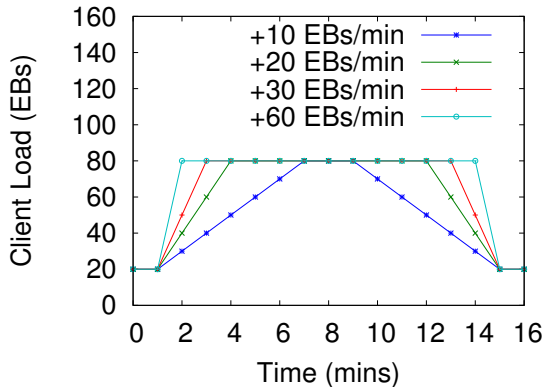
Figure 8: Load growth rate patterns.

We tested these workloads with different HW and LW thresholds of our algorithm as well as for fixed over- and under-provisioning. We also tested with a "manual" approach where reallocation decisions were scripted based on knowledge of the workload. During the tests we measured the performance metrics described in Section 5.

Figure 9 shows representative results that we found for our VMWare Server platform. The graph focuses on the slow response rate metric measured by the TPC-W client with the most shallow growth rate for five test approaches. The two algorithm parameter sets are based on experience for what parameters are a good fit for the data center.

The figure shows results for the percentage of responses to the TPC-W that take more than one second. For each approach this result is plotted against the mean number of
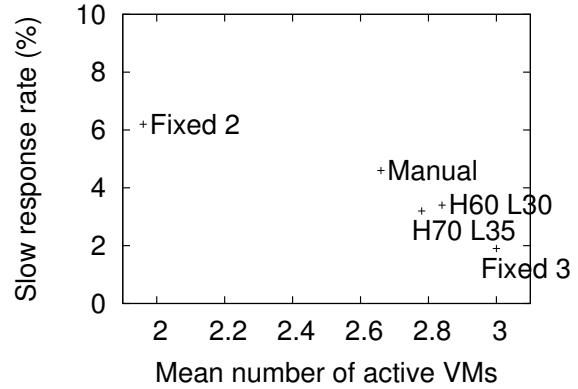
Figure 9: Slow Response Rate Performance for +10 EBs/min Growth (VMWare Server Cluster)

VMs that were active over the course of the test. Thus the "Fixed 2" approach is under-provisioned at two active VMs for the application and yields the worst performance. At the other extreme, the "Fixed 3" approach is over-provisioned and provides the best performance. In between these extremes are the two instances of our algorithm along with the manual approach. The results for these three approaches are comparable, although the manual approach is not as aggressive in activating a new VM as the algorithm. The result is more slow responses compared with using fewer VM resources.

This comparison shows our algorithm demonstrates competitive performance with the result if manual intervention was used for the known workload. The other performance measures show similar orderings and tradeoffs between the five approaches tested over the set of workloads.

We also performed tests on the ESX Server platform using the same load growth patterns shown in Figure 8. Again focusing on the slow response rate performance, results are shown in Figure 10 for the +10EBs/min growth rate. These ESX Server results contain two important distinctions from the previous VMWare Server results to account for the platform and its more powerful servers. First, the slow response threshold is 100ms rather than 1sec so it still represents on the order of 10% of responses. Second, because a hard CPU slice is used to enforce the desired capacity in the algorithm, the mean number of active VMs is weighted by their capacity. In looking at the results for this platform their tone is the same as what we previously observed with the two versions of our algorithm providing competitive performance with a manual intervention.

6.3.2 Multiple Applications Workload

We finally tested the algorithm with a workload of multiple applications. For this test we used two separate TPC-W applications (with separate databases) and a third application running as a simple WebSphere test with no database access. The initial configuration of this system is one active VM for each application as well as additional ghost VMs for the first two applications. The regular and periodic workload for each application over the course of our test is shown in Figure 11.

The summary performance metric results for the first application are shown in Figure 12 for our VMWare Server
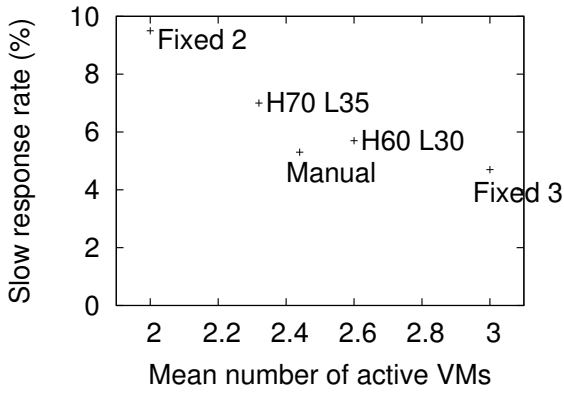
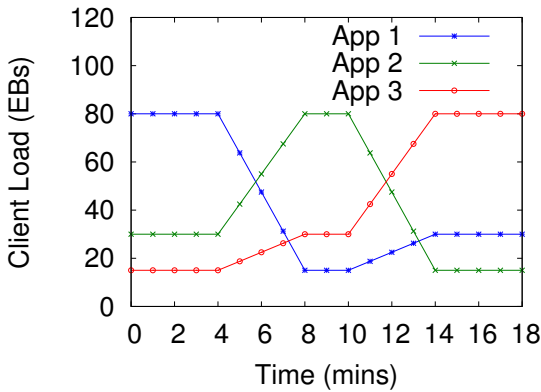**Figure 10: Slow Response Rate Performance for +10 EBs/min (ESX Cluster)**

Figure 11: Load Pattern in Multiple Applications Workload

platform. Results similar in nature were found for the ESX platform. The overall results are consistent with expectations with the under- and over-provisioned fixed cases defining the range of performance. The Fixed 2 approach is clearly under-provisioned with a relatively high overload and error rate. The better slow response rate for successful responses is deceptive because of the relatively high error rate. The Manual approach provides the best performance across the set of metrics with the HW=70,LW=35 algorithm providing competitive performance albeit with more resource usage.

7. RELATED WORK

Dynamic placement and allocation of resources to applications has been examined in other work such as [7, 11]. Most of these studies use simulation to evaluate their approaches. We explore our algorithm using a real prototype in two distinct virtual environments. Andrzejak et al. [1] found up to 50% savings could be found by dynamically redistributing resources among applications in utility computing. Virtual machines were investigated to adjust resource shares across tiers in a multi-tier platform [9].

Xen [4] and VMWare [8] have implemented "live" migration of VMs. They use a "pre-copy" mechanism that shows short downtime during migration. Although the downtime
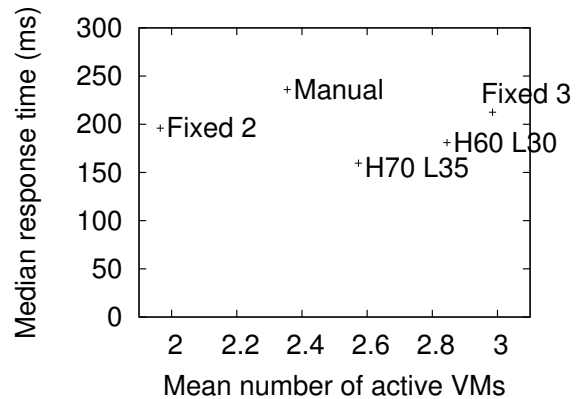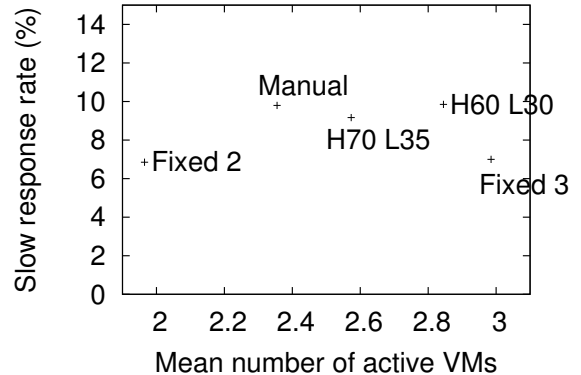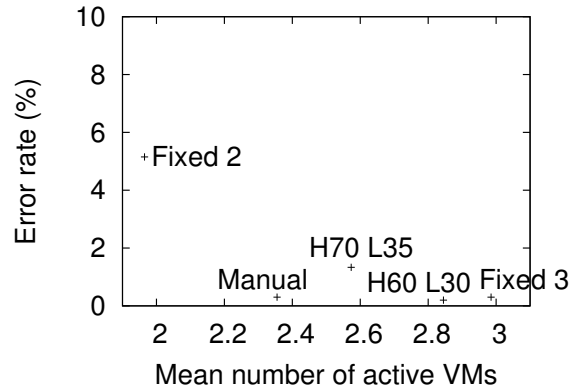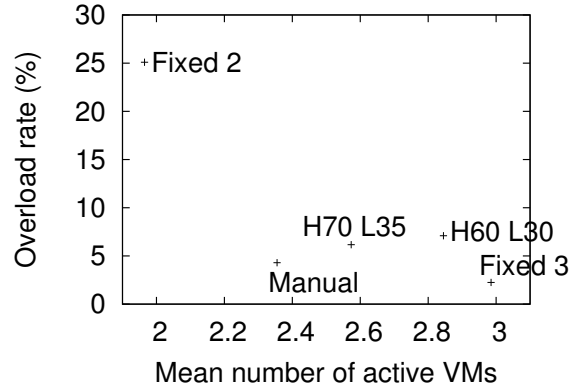

Figure 12: Performance for Multiple Applications Workload (VMWare Server Cluster)

of migration is short (less than one second), the service degradation time is quite long and memory-size dependent. According to [4] and [8], the "pre-copy" period could last from 10 seconds to over one minute. Migration of a 256MB VM may consume 25-30% of the CPU and 20% of network bandwidth on a gigabit link. It could be more and take longer if we migrate a larger VM.

VM migration has been used to solve the problem of hot spots in virtual machine environments [13]. This work reports resolving single server hot spots in 20 seconds with migration, although there may be additional time for complete migration of the VM. Also the source machine will not be relieved of the overload due to the operation of the source VM until the migration is complete. In our work we use ghost VMs that are "hot spares" already participating members of an application cluster that can quickly provide additional capacity for an application.

8. SUMMARY AND FUTURE WORK

In this work we have developed, implemented and tested an algorithm the management of resources in a virtualized data center. This algorithm makes use of ghost VMs, which participate in application clusters, but are not activated until needed by an application experiencing a significant load increase. Using this approach, our algorithm exhibits good agility—being able to detect the need for and promote a ghost VM to be handling requests in 18 seconds. In comparison with legacy systems needing to resume VMs in the face of sharply increased demand, our approach exhibits much better performance across a set of metrics. The algorithm has been deployed on multiple virtualization platforms providing different features. We found that the algorithm demonstrates competitive performance when compared with scripted resource changes based on a known workload. It also works well when tested with multiple applications exhibiting periodic workload changes.

Moving forward, we plan to continue work on deploying the algorithm across different platforms and more fully testing all aspects of the algorithm. We plan to investigate additional considerations such as network utilization in our capacity determination as well as better understanding the relationship between the watermark thresholds we use and user-perceived measures.

An important aspect of our larger project is to understand how to manage geographically distributed data centers. An interesting problem is how the local resource manager that we have implemented should interact with a global resource manager to provide agility both within and across data centers.

Acknowledgments

9. REFERENCES

[1] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models. Technical Report HPL-2002-339, HP Labs, Dec. 2002.

[2] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. E. Long. Managing flash crowds on the internet. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:246, 2003.

[3] A. Chandra and P. Shenoy. Effectiveness of Dynamic Resource Allocation for Handling Internet Flash Crowds. Technical report, Technical Report TR03-37, Department of Computer Science, University of Massachusetts, USA, November 2003.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA USA, May 2005.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpack, I. Pratt, and A. Warfield. Live migration of virtual machines. In *USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[6] Using VMware ESX Server with IBM WebSphere Application Server. http://ibm.com/websphere/developer/zones/hvws, April 2007.

[7] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 595–604, New York, NY, USA, 2006. ACM.

[8] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, Marriott Anaheim, CA USA, April 2005.

[9] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the EuroSys 2007 Conf.*, Lisbon, Portugal, March 2007.

[10] H. Qian, E. Miller, W. Zhang, M. Rabinovich, and C. E. Wills. Agility in virtualized utility computing. In *Proceedings of the Second International Workshop on Virtualization Technology in Distributed Computing*, Reno, NV USA, November 2007. ACM Digital Library.

[11] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 331–340, New York, NY, USA, 2007. ACM.

[12] Transaction processing council. TPC-W (web commerce) specification. http://www.tpc.org/tpcw/.

[13] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA USA, 2007.

[14] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 2009. Accepted for publication.