Server virtualization in autonomic management of heterogeneous workloads

Malgorzata Steinder*, Ian Whalley*, David Carrera†, Ilona Gaweda‡ and David Chess*
*IBM Thomas J. Watson Research Center
Hawthorne, NY 10532
Email: {steinder,inw,chess}@us.ibm.com
†Barcelona Supercomputing Center (BSC) - Technical University of Catalonia (UPC)
Barcelona, Spain
Email: david.carrera@bsc.es
‡AGH University of Science and Technology
Krakow, Poland
Email: ilona.gaweda@gmail.com

***Abstract*—Server virtualization opens up a range of new possibilities for autonomic datacenter management, through the availability of new automation mechanisms that can be exploited to control and monitor tasks running within virtual machines. This offers not only new and more flexible control to the operator using a management console, but also more powerful and flexible autonomic control, through management software that maintains the system in a desired state in the face of changing workload and demand. This paper explores in particular the use of server virtualization technology in the autonomic management of data centers running a heterogeneous mix of workloads. We present a system that manages heterogeneous workloads to their performance goals and demonstrate its effectiveness via real-system experiments and simulation. We also present some of the significant challenges to wider usage of virtual servers in autonomic datacenter management.**

I. Introduction

Many organizations rely on a heterogeneous set of applications to deliver critical services to their customers and partners. This set of applications includes web workloads typically hosted on a collection of clustered application servers and a back-end tier database. The application mix also includes non-interactive workloads such as portfolio analysis, document indexing, and various types of scientific computations. To efficiently utilize the computing power of their datacenters, organizations allow these heterogeneous workloads to execute on the same set of hardware resources and need a resource management technology to determine the most effective allocation of resources to particular workloads.

A traditional approach to resource management for heterogeneous workloads is to configure resource allocation policies that govern the division of computing power among web and non-interactive workloads based on temporal or resource utilization conditions. With a temporal policy, the resource reservation for web workloads varies between peak and off-peak hours. Resource utilization policies allow non-interactive workload to be executed when resource consumption by web workload falls below a certain threshold. Typically, resource allocation is performed with a granularity of a full server machine, as it is difficult to configure and enforce policies that allow server machines to be shared among workloads. Coarse-grained resource management based on temporal or resource utilization policies has previously been automated [1], [2].

Once server machines are assigned to either the web or the non-interactive workload, existing resource management policies can be used to manage individual web and non-interactive applications. In the case of web workloads, these management techniques involve flow control [3], [4] and dynamic application placement [5]. In the case of non-interactive workloads, the techniques involve job scheduling, which may be performed based on various existing scheduling disciplines [6]. To effectively manage heterogeneous workloads, we need a solution that combines flow control and dynamic placement techniques with job scheduling. To the best of our knowledge, no such solution has been proposed.

We present a system that considerably improves the way heterogeneous workloads are managed on a set of heterogeneous server machines using automation mechanisms provided by server virtualization technologies. The system introduces several novel features. First, it allows heterogeneous workloads to be collocated on any server machine, thus reducing the granularity of resource allocation. This is an important aspect for many organizations that rely on a small set of powerful machines to deliver their services, as it allows for a more effective resource allocation when any workload requires a fractional machine allocation to meet its goals. Second, our approach uses high-level performance goals (as opposed to lower-level resource requirements) to drive resource allocation. Hence, unlike previous techniques [7] and [8], which manage virtual machines according to their defined resource requirements, our technique provides an application-centric view of the system in which a virtual machine is only a tool used to achieve performance objectives. Third, our technique exploits a range of new automation mechanisms that will also benefit a system with a homogeneous, particularly non-interactive, workload by allowing more effective scheduling of jobs.

Integrated automated management of heterogeneous workloads is a challenging problem for several reasons. First, performance goals for different workloads tend to be of different types. For interactive workloads, goals are typically defined in terms of average or percentile response time or throughput over a certain time interval, while performance goals for non-interactive workloads concern the performance of individual jobs. Second, the time scale of management is different. Due to the nature of their performance goals and short duration of individual requests, interactive workloads lend themselves to automation at short control cycles. Non-interactive workloads typically require calculation of a schedule for an extended period of time. Extending the time scale of management requires long-term forecasting of workload intensity and job arrivals, which is a difficult if not impossible problem to solve. Server virtualization helps us avoid this issue by providing automation mechanisms by which resource allocation may be continuously adjusted to the changing environment. Third, to collocate applications on a physical resource, one must know the applications' behavior with respect to resource usage and be able to enforce a particular resource allocation decision. For web applications, with the help of an L7 gateway, one can rather easily observe workload characteristics and, taking advantage of similarity of web requests and their large number, derive reasonably accurate short-time predictions regarding the behavior of future requests. Non-interactive jobs do not exhibit the same self-similarity and abundance properties, hence predicting their behavior is much harder. Enforcing a resource allocation decision for web workloads can also be achieved relatively easily by using flow control mechanisms [3], [4]. Server virtualization gives us similar enforcement mechanisms for non-interactive applications.

While server virtualization allows us to better manage workloads to their respective SLA goals, it also introduces considerable challenges in order to use it effectively. They concern the configuration and maintenance of virtual images, infrastructure requirements to make an effective use of the available automation mechanisms, and the development of algorithmic techniques capable of utilizing the larger number of degrees of freedom introduced by virtualization technologies. This paper addresses some of these challenges.

This paper is structured as follows. In Section II we present the architecture of our resource management system. In Section III we describe some fundamental aspects of virtualization, focussing on those aspects that our system controls. Section IV presents components of our system that are specific to managing virtual resources. In Section V we evaluate our system through experimentation. Related work is discussed in Section VI. We conclude the paper in Section VII.

II. System architecture

Figure 1 shows a simple example of a system we consider in this paper. The managed system includes a set of heterogeneous server machines, referred to henceforth as *nodes*. Web applications, which are served by application servers, are replicated across nodes to form application server clusters. Requests to these applications arrive at an entry router which may be either an L4 or L7 gateway that distributes requests to clustered applications according to a load balancing mechanism. Long-running jobs are submitted to the job scheduler, placed in its queue, and dispatched from the queue based on the resource allocation decisions of the management system.

Our management architecture takes advantage of an overload protection mechanism that can prevent a web application from utilizing more than the allocated amount of resources. Such overload protection may be achieved using various mechanisms including admission control [9], flow control [3], [4], or OS scheduling techniques [10]. Server virtualization mechanisms could also be applied to enforce resource allocation decisions on interactive applications.

In the system considered in this paper, overload protection for interactive workloads is provided by an L7 request router which implements a flow control technique. The router classifies incoming requests into flows depending on their target application and service class, and places them in per-flow queues. Requests are dispatched from the queues based on weighted-fair scheduling discipline, which observes a system-wide concurrency limit. The concurrency limit ensures that all the flows combined do not use more than their allocated resource share. The weights further divide the allocated resource share among applications and flows.

Both the concurrency limit and scheduling weights are dynamically adjusted by the *flow controller* in response to changing workload intensity and system configuration. The flow controller builds a model of the system that allows it to predict the performance of the flow for any choice of concurrency limit and weights. This model may also be used to predict workload performance for a particular allocation of CPU power. In this paper, we use this functionality of the flow controller to come up with a utility function for each web application, which gives a measure of application happiness with a particular allocation of CPU power given its current workload intensity and performance goal. The flow control technique implemented by the flow controller and request router has been introduced in [4] and further enhanced in [11], and will not be further discussed in this paper.

Long-running jobs are submitted to the system via the *job scheduler*, which, unlike traditional schedulers, does not make job execution and placement decisions. In our system, the job scheduler only manages dependencies among jobs and performs resource matchmaking. Once dependencies are resolved and a set of eligible nodes is determined, jobs are submitted to the *application placement controller* (APC).

Each job has an associated performance goal. Currently we only support completion time goals, but we plan to extend the system to handle other performance objectives. From this completion time goal we derive an objective function which is a function of actual job completion time. When job completes exactly on schedule, the value of the objective function is zero. Otherwise, the value increases or decreases linearly depending on the distance of completion time from the goal.

The job scheduler uses APC as an adviser to where and when a job should be executed. When APC makes a placement

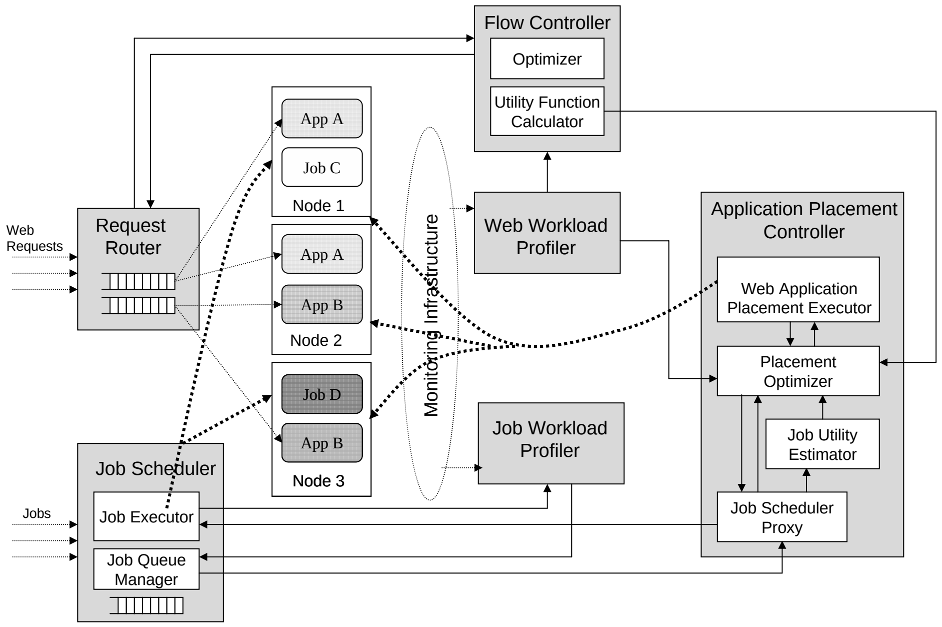

Fig. 1. Management system architecture for heterogeneous workloads. Thin dotted lines indicate request or job paths. Thin solid lines indicate interactions among management system components. Thick dotted lines show management actions applied to the managed system.

decision, actions pertaining to long-running jobs are returned to the scheduler and put into effect via its *job executor* component. The job executor monitors job status and makes it available to APC for use in subsequent control cycles.

From the point of view of this work, APC is the most important component of the system. It provides the decision-making logic that affects placement of both web and non-interactive workloads. To learn about jobs in the system and their current status, APC interacts with the job scheduler via its *job scheduler proxy*. The *placement optimizer* calculates the placement that maximizes the minimum utility across all applications. In [5], we have introduced a technique that provides such dynamic placement for web applications. It is able to allocate CPU and memory to applications based on their CPU and memory requirements, where memory requirement of an application instance is assumed not to depend on the intensity of workload that reaches the instance. APC used in this system is a version of the controller presented in [5] that has been enhanced in several ways. We modified the algorithm inputs from application CPU demand to a per-application utility function of allocated CPU speed. Permitting resource requirements to be represented by non-linear utility functions allows us to better deal with heterogeneous workloads which may differ in their sensitivity to a particular resource allocation. The attention to workload sensitivity to resource allocation is important when system is overloaded and resource requirements of some applications cannot be satisfied in full. We also changed the optimization objective from maximizing the total satisfied CPU demand to maximizing the minimum utility across all applications, which focuses the algorithm on ensuring fairness and, in particular, prevents it from starving some applications. In addition, we have improved the heuristics used be the algorithm, which resulted in a significant reduction of its computational complexity.

Since APC is driven by utility functions of allocated CPU demand and (for non-interactive workloads) we are only given objective functions of achieved completion times, we need a way to map completion time into CPU demand, and vice versa. Recall that for web traffic we already have a similar mechanism, provided by the flow controller. The required mapping is very hard to obtain for non-interactive workloads, because the performance of a given job is not independent of CPU allocation to other jobs. After all, when not all jobs can simultaneously run in the system, the completion time of a job that is waiting in the queue for other jobs to complete before it may be started depends on how quickly the jobs that were started ahead of it complete, hence it depends on the CPU allocation to other jobs. In our system, we have implemented heuristics that allow us to estimate CPU requirements for long-running jobs for a given value of utility function. We use this estimation to obtain a set of data-points from which we extrapolate the utility function. The utility function allows us to evaluate a placement of long-running jobs with respect to how well it is likely to satisfy their SLAs. The process of calculating the utility function is rather involved, and due to space limitations it will not be described in this paper.

To manage web and non-interactive workloads, APC relies on the knowledge of resource consumption by individual requests and jobs. Our system includes profilers for both kinds of workloads. The *web workload profiler*, which was introduced in [12], obtains profiles for web requests in the form of the average number of CPU cycles consumed by requests of a given flow. The *job workload profiler*, which is a subject of our ongoing research, obtains profiles for jobs in the form of the number of CPU cycles required to complete the job, the number of threads used by the job, and the maximum CPU speed at which the job may progress.

III. Value of Server Virtualization

Space does not permit a full discussion of the various types of virtualization and their relative merits here; the reader is referred instead to [13], [14]. Instead, we will briefly enumerate the features of virtualization of which our system is capable of taking advantage. The terminology here aligns with that of Xen [15].

- PAUSE When a virtual machine is paused, it does not receive any processor time, but remains in memory.
- RESUME Resumption is the opposite of pausing—the virtual machine is once again allocated processor time.
- SUSPEND When a virtual machine is suspended, its memory image is saved to disk, and it is unloaded.
- RESTORE Restoration is the opposition of suspension—an image of the virtual machine's memory is loaded from disk, and the virtual machine is permitted to run again.
- MIGRATE The virtual machine is first paused, then the memory image is transferred across the network to a target node, and the virtual machine is resumed.
- LIVE_MIGRATE A variant of migration in which the virtual machine is not paused. Instead, the memory image is transferred over the network whilst running.
- MOVE_AND_RESTORE When a virtual machine has been suspended, and needs to be restored on a different node, the saved memory image is moved to the target node, and the virtu machine is then restored.
- RESOURCE_CONTROL Resource control modifies the amounts of various resources that a virtual machine can consume. We consider CPU and memory.

While virtualization can be provided using various technologies, our system uses Xen as it is capable of providing the wide variety of controls discussed above—all of these controls are most directly accessible from a special domain on each node, labeled domain 0.

IV. Managing server virtualization

In this section, we discuss how our system makes use of virtualization technologies to manage heterogeneous workloads. Recall from Section II that, to manage web workloads, our system relies on an entry gateway that provides flow control for web requests. The entry gateway provides a type of high-level virtualization for web requests by dividing CPU capacity of managed nodes among competing flows. Together with an overload protection mechanism, the entry gateway facilitates performance isolation for web applications. Such virtualization technology exists in some application server middleware systems, of which WebSphere Extended Deployment [16] is the example most familiar to us.

Server virtualization could also be used to provide performance isolation for web applications. This would come with a memory overhead caused by additional copies of the OS that would have to be present on the node. Hence, we believe that middeware virtualization technology is a better choice for managing the performance of web workloads.

Since middeware virtualization technology can only work for applications whose request-flow it can control, a lower level mechanism must be used to provide performance isolation for other types of applications. As outlined in the previous section, server virtualization provides us with powerful mechanisms to control resource allocation of non-web applications.

Several server virtualization technologies have been developed thus far [13] and they all could be used by our system, although possibly with a limited set automation mechanisms. Our implementation makes use of Xen [15].

A. VM management

To manage VMs inside a physical Xen-enabled node, we have implemented a component, called the *machine agent* (Figure 2), which resides in domain 0 so as to have access to the Xen domain controls. The machine agent provides a Java-based interface to create and configure a VM image for a new domain, copy files from domain 0 to another domain, start a process in another domain, and to control the mapping of physical resources to virtual resources. During its life-cycle, a domain can transition between various states in accordance with the transitions shown in Figure 3.

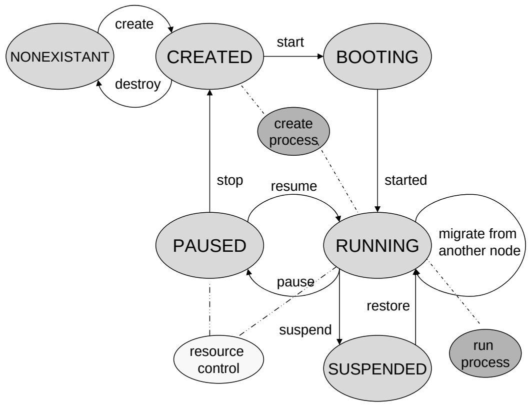

Fig. 3. Life-cycle of a Xen domain.

We use Xen to provide on-line automation for resource management, hence we want to make management actions light-weight and efficient. This consideration concerns the process of creating virtual images, which may be quite time consuming. We avoid substantial delays, which would otherwise be incurred each time we intend to start a job, by pre-creating a set of images for use during runtime. The dispensing of these pre-created images is performed by the *image management* subsystem. Images once used to run a process are scrubbed of that process data and may be reused by future processes. In our small-scale testing thus far, we

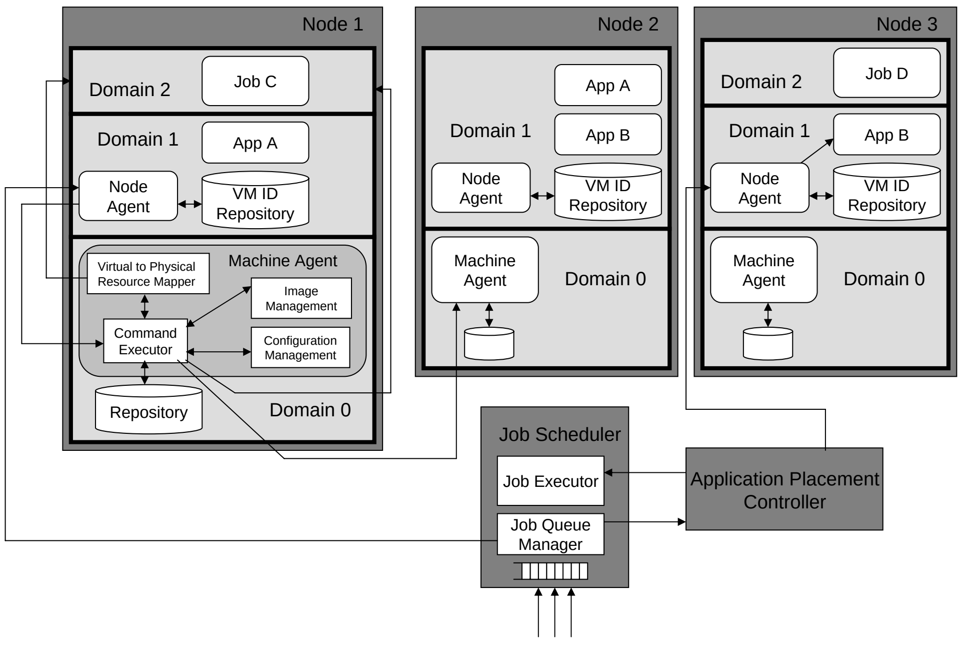

Fig. 2. Management architecture for Xen machines.

found it sufficient to pre-create a small number of images, however, we plan to extend the *image management* subsystem to dynamically extend the pool of available images, if needed.

Inside a created image, we can create new processes. This is done by populating the image with the files necessary to run that new process. In our system, we assume that the files required for of all processes that may run on the node are placed in its domain 0 in advance. Hence, we only need to copy them from domain 0 to the created image. Clearly, there are mechanisms that would allow us to transfer files from an external repository to a node where the process is intended to run, but we have not used them in our prototype.

Before it may be booted, an image must be provided with configuration files to set up its devices and networking. This functionality is encapsulated by the *configuration management* subsystem. To assign an IP address and DNS name, a DHCP server can be used, although in our system we have implemented a simpler, more restrictive, module that selects configuration settings from a pool of available values.

An image, once configured, may then be booted. Once in the running state, it may be suspended or paused. New processes may be created and run inside it. An image that is either running or paused may also be resource controlled. Migration may be used to transfer the image to another node. Since at the time of writing this paper we do not have the shared storage infrastructure required to use migration, we have implemented a suspend-move-and-restore mechanism by which the domain is suspended on one machine, the checkpoint and image files are copied to another node, and the domain is restored on the new host node. This is obviously quite inefficient mechanism, which nevertheless allows us to study the benefits of migration.

Xen provides resource control mechanisms to manage memory and CPU usage by its domains. We set memory for a domain based on configured or profiled memory requirements. We set CPU allocation for a domain based on the decisions of APC, which result from its optimization technique. The CPU allocation to a domain may be lower that the amount of CPU power actually required by a process running inside a domain. Both memory and CPU allocations to a domain may change while the domain is running based on changing process requirements and decisions of APC.

CPU allocation to domains may be controlled in Xen using three mechanisms. First, the number of virtual CPUs (vCPUs) can be selected for any VM. Second, vCPUs may be mapped to physical CPUs. By ‘pinning’ vCPUs of a domain to different physical CPUs we can improve the performance of the domain. Finally, CPU time slices may be configured for each domain. When all vCPUs of a domain are mapped to different physical CPUs, allocation of 50 out of 100 time slices to the domain implies that each vCPU of the domain wil receive 50% of the physical CPU to which it is mapped. Xen also permits borrowing, by which CPU slices allocated to a domain that does not need them can instead be used by other domains.

In a default configuration provided by Xen, each domain receives the same number of vCPUs as there are physical CPUs on a machine. Each of those vCPUs will be mapped to a different physical CPU and receives 0 time slices with CPU borrowing turned on. In the process of managing the system, we modify this allocation inside the *virtual-to-physical resource mapper*. When a domain is first started, we allow Xen to create the default number of vCPUs and map them to different physical CPUs. We only set the number of time slices to obtain the CPU allocation requested by placement controller. While domain is running, we observe its actual

CPU usage. If it turns out that the domain is not able to utilize all vCPUs it has been given, we can conclude that the job is not multi-threaded. Hence, to receive its allocated CPU share, its vCPUs must be appropriately reduced and remapped. The *virtual-to-physical resource mapper* must attempt to find a mapping that provides the domain with the required amount of CPU power spread across the number of vCPUs that the job in the domain can use—clearly, this is not always possible.

All the VM actions provided by the machine agent are asynchronous JMX calls followed by JMX notifications.

B. Job management

To hide the usage of VMs from a user, we have implemented a higher-layer of abstraction, embedded inside the *node agent*, which provides the job management functionality. It provides operations to start, pause, resume, suspend, restore, and resource control a job. To implement these operations, the node agent interacts with the machine agent in domain 0 using its VM management interfaces. When a job is first started, the node agent creates (or obtains a pre-created) image in which to run the job. It records the mapping between the job ID and VM ID. Then it asks the machine agent to copy corresponding process binaries to the new image and to boot the image. Once domain is running, the job is started inside it.

Observe that we always place a job in its own domain. This gives us performance isolation among jobs such that we can control their individual resource usage, but it comes at the expense of added memory overhead. We plan to extend our system such that it allows collocation of multiple jobs inside a single domain based on some policies.

The node agent process is placed in domain 1, which is the domain we use for all web applications. There are two reasons for placing the node agent in a separate domain than domain 0. First, our application server middleware already provides a node agent process with all required management support, thus adding new functionality is a matter of a simple plugin. Second, domain 0 is intended to remain small and light-weight. Hence, we avoid using it to run functionality that does not directly invoke VM management tools. Like the machine agent, the node agent exposes its API using JMX.

C. Xen machine organization

In Figure 2 we show the organization of a Xen-enabled server machine we use in our system. We always run at least two domains, domain 0 with the machine agent, and domain 1 with the node agent and all web applications. Since resource control for web applications is provided by request router and flow controller, such collocation of web applications does not affect our ability to provide performance isolation for them. Domains for jobs are created and started on-demand.

V. Experiments and results

In this section, we experimentally evaluate our approach using both real system measurements and a simulation. We have implemented our system and integrated it with WebSphere Extended Deployment [16] application server middleware. We use WebSphere Extended Deployment to provide flow control for web applications and use Xen virtual machines to provide performance isolation for non-interactive workloads.

In the experiments, we use a single micro-benchmark web application that performs some CPU intensive calculation interleaved with sleep times, which simulate backend database access or I/O operations. We also use a set of non-interactive applications, which consists of well known CPU-intensive benchmarks. In particular, we use BLAST [17], Lucene [18], ImageMagick [19] and POV-Ray [20] as representative applications for bioinformatics, document indexing, image processing and 3D rendering scenarios respectively. BLAST (Basic Local Alignment Search Tool) is a set of similarity search programs designed to explore all of the available sequence databases for protein or DNA queries. Apache Lucene is a high-performance, full-featured, open-source text search engine library written entirely in Java. In our experiments, we have run the example indexing application provided with the Lucene library to index a large set of files previously deployed in the filesystem. POV-ray (Persistence of Vision Raytracer) is a high-quality free tool for creating three-dimensional graphics. ImageMagick is a software suite to create, edit, and compose bitmap images.

In the experiments, we submit six different jobs, whose properties are shown in Table I. We achieve differentiation of execution time by choosing different parameters, or by batching multiple invocations of the same application. All used applications except BLAST are single-threaded, hence they can only use one CPU. In addition, Lucene is I/O intensive, hence it cannot utilize a full speed of a CPU. We assign jobs to three service classes. Completion time goal for each job is defines relative to its profiled execution time and is equal to 1.5, 3, and 10 for *platinum*, *gold*, and *silver* class, respectively.

We experiment with our system on a cluster of two physical machines, `xd018` and `xd020`, each with two 3GHz CPUs and 2GB memory. We used the XenSource-provided Xen 3.0.2 packages for RedHat Enterprise Linux 4.

While testing our system, we determined that the resource control actions of our version of Xen are rather brittle and cause various internal failures across the entire Xen machine. Therefore, in our experiments, we have supressed resource control actions in the machine agent code.

A. Effectiveness of automation mechanisms

In this section, we study the effectiveness of automation mechanisms used by our system. We take three different jobs from our set, JOB1, JOB2, and JOB5, and perform various automation actions on them while measuring their duration. We do not measure migration, as we have not set it up in our system. Instead, we use move-and-restore (as discussed in Section IV). Clearly, this is quite an inefficient process, mostly due to the overhead of copying the image. We expect a dramatically different result once we put live-migration in place. Clark et al [21] report sub-second application downtime when live-migration is used.

TABLE I
JOBS USED IN EXPERIMENTS

	JOB1	JOB2	JOB3	JOB4	JOB5	JOB6
Job Type	BLAST	ImageMagick	POV-Ray	BLAST	Lucene	BLAST
Exec. time[min]	15	127	40	8	69	30
Class	Bronze	Platinum	Platinum	Platinum	Gold	Platinum
Max. speed [CPUs]	2	1	1	2	0.6	2
Memory [MB]	550	750	250	550	350	550

In Table II, the domain creation time includes the time taken to create the domain metadata, such as configuration files. Process creation involves copying process files into process target domain while domain is in running state. Suspend and restore operations involve creating a checkpoint of domain memory and saving it to disk, and restoring domain memory from checkpoint on disk, respectively. The checkpoint copy operation involves transfering checkpoint file between machines in the same LAN. The checkpoint file is practically equal in size to the amount of RAM memory allocated to a domain. Similarly, time to copy an image is measured between two machines in LAN. There is a clear relationship between domain RAM size and its checkpoint copy time, and between domain image size and image copy time. Both copy image and copy checkpoint can be avoided when shared storage is available. Migration time includes suspend, resume, copy image and copy checkpoint, and could be greatly reduced with the use of shared storage.

TABLE II
RUNTIME OF VM OPERATIONS FOR VARIOUS CONTAINED JOBS

	JOB1	JOB2	JOB5
Process files:			
File number	286	16	2693
Disk space [MB]	614	4	115
File size [MB]:			
Image	6900	6900	6900
Checkpoint	550	750	350
Execution times [s]:			
Create	3	3	3
Boot	33	33	32
Create process	61	2	79
Suspend	19	21	13
Restore	19	21	14
Copy checkpoint	51	70	35
Copy image	906	904	1000
Migrate	994	1016	1060

B. Managing heterogeneous workloads

In this section we describe an experiment we carry out to demonstrate the benefits of using server virtualization technology in the management of heterogeneous workloads. We deploy *StockTrade* (a web-based transactional test application) in domain 1 on two machines `xd018` and `xd020`. We vary load to *StockTrade* using a workload generator that allows us to control the number of client sessions that reach an application. Initially, we start 55 sesstions and observe that with this load, response time of *StockTrade* requests is about 380ms and approaches response time goal of 500ms, as shown in Figure 4. At this load intensity, *StockTrade* consumes about 5/6 of CPU power available on both machines. Then we submit JOB5 (A). Recall from Table I that JOB5 is associated with *platinum* service class and therefore has completion time goal equal to 1.5 to its expected execution time. After a delay caused by the duration placement control cycle (B) and domain starting time, JOB5 is started (C) in domain 2 on `xd020` and, in the absence of any resource control mechanism, allocates it the entire requested CPU speed, which is equivalent to 0.6 CPU. As a result of decreased CPU power allocaton to domain 1, on `xd020`, the response time for *StockTrade* increases to 480ms, but it stays below the goal. A few minutes after submitting JOB 5, we submit JOB1 (D), whose service class is *bronze*. JOB1 has a very relaxed completion time goal but it is very CPU demanding. Starting it now would take 2CPUs from the current *StockTrade* allocation.

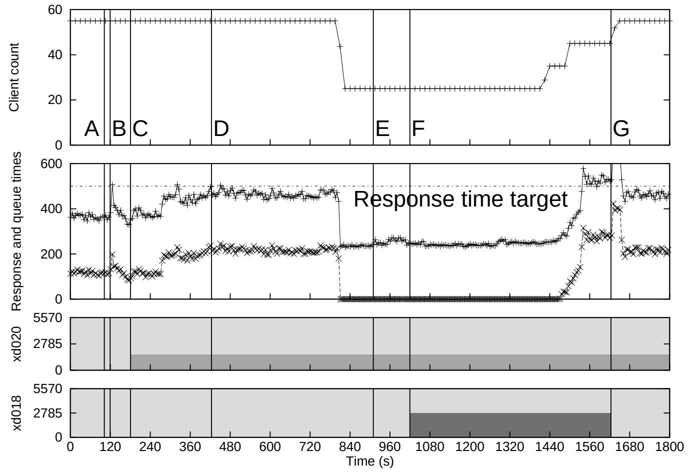

Fig. 4. Response time for *StockTrace* and job placement on nodes.

At 800s since the begining of the experiment, we reduce load to *StockTrade* to 25 concurrent client sessions. When CPU usage of *StockTrade* reduces to about 50% of each machine, the placement controller decides (E) to start JOB1 (F) on `xd018`. After 1000s, we increase the number of client sessions back to 55, placement controller suspends JOB1 (G). Typically, JOB1 will later be resumed when any of the following conditions occur: (1) JOB5 completes, (2) load to *StockTrade* is reduced, or (3) JOB1 gets close enough to its target completion time so as to necessitate its resumption, even at the expense of worsened performance for *StockTrade*.

However, the occurrence of the third condition indicates that the system is under-provisioned, hence SLA violation may not be avoided. This simple experiment demonstrates that with the use of server virtualization, our system is able to balance resource usage between web and non-interactive workloads.

C. Non-interactive workload management

In this paper, we also show the usefulness of server virtualization technology in the management of homogeneous, in this case non-interactive workloads. Using the same experimental set-up as in Section V-B, we run a testcase that involves only long-running jobs shown in Table I. The placement of jobs on nodes over time is shown in Figure 5.

We start the testcase by submitting JOB1 (A), which is started on `xd020` and takes its entire CPU power. Soon after JOB1 is submitted, we submit JOB2 and JOB3 (B), which both get started on `xd018` and each of them is allocated one CPU on the machine. Ten minutes later, we submit JOB4 (C), which has a very strict completion time requirement. In order to meet this requirement, APC decides to suspend JOB1 and start JOB4 in its place. Note that if JOB1 was allowed to complete before JOB4 is allowed to start, JOB4 would wait 5 min in the queue, hence it would complete no earlier than 13 min after its submission time, which would exceed its goal. Instead, JOB4 is started as soon as it arrives and completes within 10 min, which is within its goal. While JOB4 is running, we submit JOB5 (D). However, JOB5 belongs to a lower class than any job currently running, and therefore is placed in the queue. When JOB4 completes, JOB5 is started on `xd020`. Since JOB5 consumes only 1 CPU, APC also resumes JOB1 and allocates it the remaining CPU. However, to avoid Xen stability problems in the presence of resource control mechanisms, we supress the resource control action, and resolving CPU contention is delegated to Xen hypervisor.

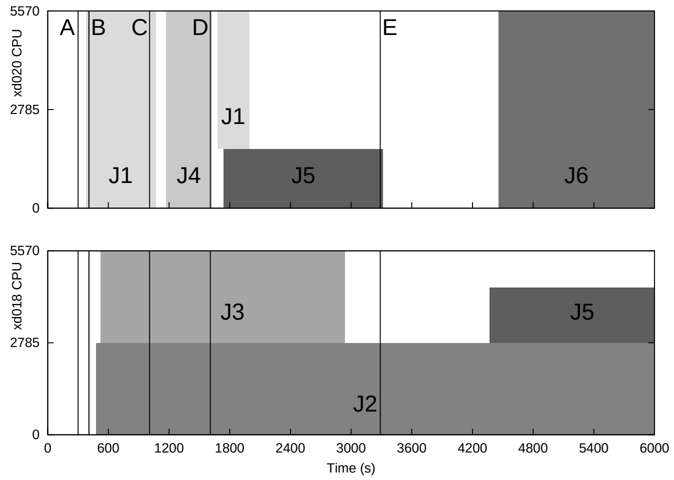

Fig. 5. Node utilization by long running jobs.

In the next phase of the experiment, we demonstrate the use of migration. We wait until the completion of JOB1 and JOB3, and then we submit JOB6 (E). When JOB6 arrives, JOB2 and JOB5 each consume 1 CPU on `xd018` and `xd020` respectively. Since JOB6 requires 2 CPUs, APC may either (1) make it wait in the queue, (2) suspend JOB2 or JOB5, (3) collocate and resource control JOB6 with either JOB2 or JOB5, or (4) migrate either JOB2 or JOB5. Options (1)-(3) would result in wasted capacity on one or both machines. Moreover, options (1) and (3) would result in having *platinum* class job receive proportionately less CPU power than JOB5, whose service class is *gold*. This would clearly not be the optimal decision. Hence, APC decides (E) to move JOB4 to `xd018` (which it will now share with JOB5) and start JOB6 on the now-empty `xd020`.

Even though this experiment shows that APC correctly uses migration when machine fragmentation makes it difficult to place new jobs, it also demonstrates a limitation of our optimization technique, which is currently oblivious to the cost of performing automation actions. Although in this experiment, 15 min is an acceptable price to pay for migrating a job, it is easy to imagine a scenario where this would not be the case.

D. Simulation study

In this section, we study potential benefits of using virtualization technology in the management of non-interactive workloads. We simulate a system in which jobs with characteristics similar to the ones in Table I are submitted randomly with exponentially distributed interarrival times. The workload mix includes 25% multithreaded jobs with execution time of 32 min, 25% multithreaded jobs with execution time of 23 min, 25% single-threaded jobs with execution time of 66 min, 15% single-threaded jobs with execution time of 45 min, and 10% single-threaded jobs with execution time of 127 min. The service class distribution for all jobs is 50%, 40%, and 10% for *platinum*, *gold*, and *silver* service class, respectively. We vary mean interarrival time between 8 and 30 min.

Our simulation does not model the cost of performing virtualization actions. Hence, the results concern the theoretical bound on the particular algorithmic technique we use.

We evaluate our placement algorithm (APC) with well known scheduling techniques: first-come-first-serve (FCFS) and earliest-deadline-first (EDF), in which we interpret completion time goal as deadline. We also execute our placement technique after disabling automation mechanisms provided by virtualization technology (APC_NO_KNOBS).

Figure 6 shows the percentage of jobs that missed their completion time goal as a function of interarrival time. When APC uses virtualization mechanims, it performs much better than FCFS and EDF. Throughout the experiment, it does not violate any SLAs, with the exception of a high-overload case corresponding to job interarrival time of 8min. In the overloaded system, our technique has 20-30% lower number of missed targets that FCFC and EDF, which is not shown Figure 6. When virtualiztion mechanisms are not used, our algorithm is no better or worse than EDF. This shows that the improvement observed in the case of APC is truly due to the use of virtualization technology and not due to some new clever scheduling technique.

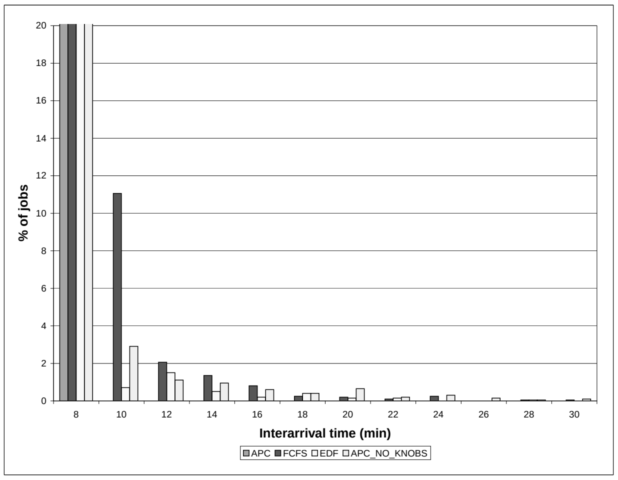

Fig. 6. Percentage of jobs that have not met their completion time goal.

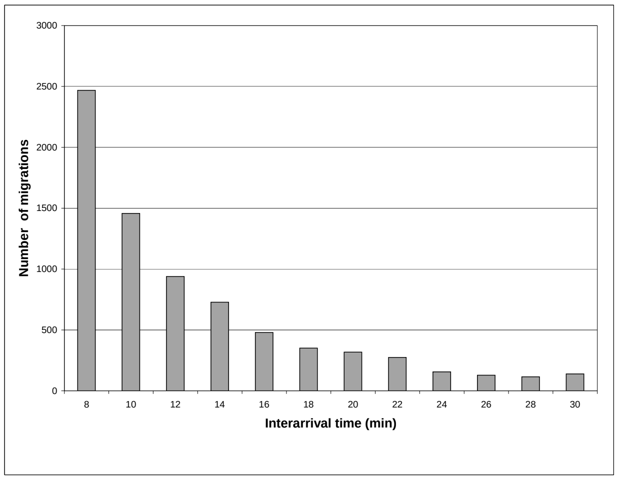

Fig. 8. Sum of migrations and move-and-restore actions.

Figures 7 and 8 show the number of suspensions and movements during the experiment. Not surprisingly, as load increases, the number of actions also increases. With very high load each job is suspended and moved more than once, which in practice will increase its execution time. In order to benefit from the usage of the automation mechanism in practice, it is therefore important to consider the cost of automation mechanisms in the optimization problem solved by APC. We plan to consider such costs in our future work.

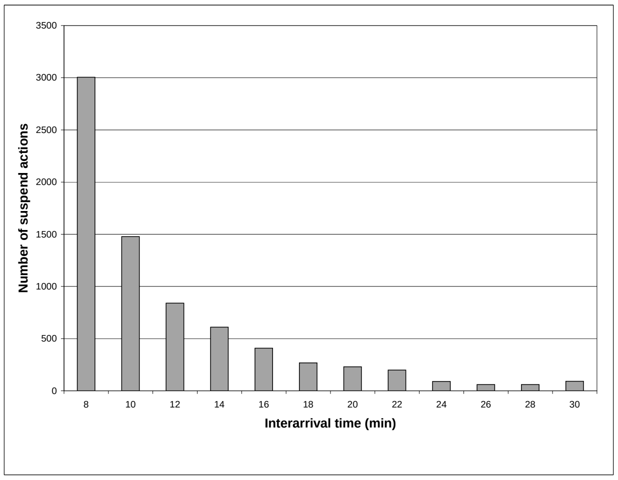

Fig. 7. Number of suspend operations.

VI. Related work

Our approach differs from prior virtual machine management techniques [22], [10], [23], [24] in a key aspect. While previous techniques concentrate on managing virtual machines as primary abstractions, our technique manages applications using automation mechanisms provided by VMs. We take an application-centric approach and strive to keep the usage of VM technology invisible to the end user.

Virtuoso [22], [10] provides a OS scheduling technique, VSched [10], for heterogeneous workload VMs. VSched enforces compute rate and interactivity goals for interactive workloads, including web workloads, and non-interactive ones. It provides soft real-time guarantees for VMs hosted on a single server machine. VSched could be used as a component of our system for providing resource-control automation mechanism within a machine, but clearly our approach is broader as it addresses resource allocation for heterogeneous workloads across a cluster of server boxes.

The management of clusters of virtual machines is addressed in [23], [25]. Fisher et al. [23] deal with the problem of deploying a cluster of virtual machines with given resource configurations across a set of machines. Czajkowski et al. [25] define an API for Java VM that permits a developer to define resource allocation policies. Neither of these techniques provide a technology to dynamically adjust allocation based on SLA objectives in the presence of resource contention.

VMware DRS [7] provides technology to automatically adjust the amount of physical resources available to VMs based on defined policies. This is a achieved using live-migration automation mechanism provided by VMotion. VMware DRS adopts a VM-centric view of the system: policies and priorities are configured on a VM-level. A approach similar to VMware DRS is proposed in [8], which proposes a dynamic adaptation technique based on rearranging VMs so as to minimize the number of physical machines used. The application awareness is limited to configuring physical machine utilization thresholds based on off-line analysis of application performance as a function of machine utilization. Runtime requirements of VMs are taken as a given and there is no explicit mechanism to tune resource consumption by any given VM.

Unlike [7] and [8], our system takes an application-centric approach—the virtual machine is considered only as a container in which an application is deployed. Using knowledge of application workload and performance goals, we can utilize a more versatile set of automation mechanisms than [7] and [8].

We can vary the number of VMs over which a clustered application is provided, suspend a VM for a long-running job, and decide how much resource a VM should be allowed to consume. In addition, our system is able to utilize various kinds of virtualization for various applications. For example, for web workloads, we chose to use virtualization technology provided by application server middleware technology.

The adaptation problem for virtual environments has also been studied in [24]. The problem there is to place virtual machines interconnected using virtual networks on physical servers interconnected using a wide area network. Given the nature of the network, the primary concern in this problem is to allocate network bandwidth for virtual networks. VMs may be migrated, but their resource allocation is taken as a given. Our problem deals with datacenter environments, in which network bandwidth is of lesser concern, and our solution considers VM placement as well as resource allocation.

Our work could also be compared to numerous prior publications on job scheduling [6]. This paper does not claim to advance the field of job scheduling. Nevertheless, we show that virtualization technology offers important control mechanisms that can be used to facilitate more effective scheduling of jobs and should be considered in future job scheduling techniques.

VII. Conclusions and future work

We present a system that allows us to manage heterogeneous workloads on a set of heterogeneous server machines using automation mechanisms provided by server virtualization technologies. The system introduces several novel features. First, it allows heterogeneous workloads to be collocated on any server machine, thus reducing the granularity of resource allocation. Second, our approach uses high-level performance goals (as opposed to lower-level resource requirements) to drive resource allocation. Third, our technique exploits a range of new automation mechanisms that will also benefit a system with a homogeneous, particularly non-interactive, workload by allowing more effective scheduling of jobs.

More work remains to be done. In the immediate future we plan to investigate problems with Xen VM monitor, which prevented us from utilizing resource control mechanism. We also plan to implement live-migration control knob and use it in our system. Other short-term goals include the implementation of workload profiler for long-running jobs and support multiple jobs in the same VM.

We believe that using server virtualization for automatic performance management is an exciting research problem. It requires novel resource allocation algorithms capable of reasoning of many new automation mechanisms and many different levels at which virtualization may be provided. It also requires techniques to deploy and manage virtual images and to model multi-level relationships among resources. We must also consider multiple resources that may be virtualized.

References

[1] K. Appleby, S. Fakhouri, L. Fong, M. Kalantar, G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano - SLA-based management of a computing utility," in *IFIP/IEEE Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[2] Y. Hamadi, "Continuous resources allocation in internet data centers," in *IEEE/ACM International Symposium on Cluster Computing and the Grid*, Cardiff, UK, May 2005, pp. 566–573.

[3] C. Li, G. Peng, K. Gopalan, and T. Chiueh, "Performance guarantees for cluster-based internet services," in *IEEE/ACM International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

[4] G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef, "Performance management for cluster-based web services," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 12, Dec. 2005.

[5] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic placement for clustered web applications," in *World Wide Web Conference*, Edinburgh, Scotland, May 2006.

[6] D. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling – a status report," in *10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004, pp. 1–16.

[7] "VMware DRS," http://www.vmware.com/products/vi/vc/drs.html.

[8] G. Khanna, K. Beatty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," in *IEEE/IFIP Network Operations and Management Symposium*, Vancouver, Canada, Apr. 2006.

[9] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites," in *International WWW Conference*, New York, NY, 2004.

[10] B. Lin and P. Dinda, "Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling," in *Proceedings of ACM/IEEE Supercomputing*, Seattle, WA, Nov. 2005.

[11] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef, "Managing the response time for multi-tiered web applications," IBM, Tech. Rep. Tech. Rep. RC 23651, 2005.

[12] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Dynamic estimation of cpu demand of web traffic," in *VALUETOOLS*, Pisa, Italy, Oct. 2006.

[13] S. Nanda and T. Chiueh, "A survey of virtualization technologies," Stony Brook University, Tech. Rep. TR-179, Feb. 2005.

[14] R. Figueiredo, P. Dinda, and J. Fortes, "A case for grid computing on virtual machines," in *International Conference on Distributed Computing*, Providence, RI, May 2003.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, 2003.

[16] "WebSphere eXtended Deployment," http://www-306.ibm.com/software/webservers/appserv/extend/.

[17] B. (BLAST), *The National Center for Biotechnology Information (NCBI)*. [Online]. Available: http://www.ncbi.nlm.nih.gov/blast/

[18] Apache Software Foundation, *Apache Lucene*. [Online]. Available: http://lucene.apache.org/

[19] ImageMagick (TM), *ImageMagick*. [Online]. Available: http://www.imagemagick.org/

[20] Persistence of Vision Pty. Ltd., *Persistence of Vision (TM) Raytracer*. [Online]. Available: http://www.povray.org/

[21] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. July, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005.

[22] B. Lin, P. Dinda, and D. Lu, "User-driven scheduling of interactive virtual machines," in *IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, PA, Nov. 2004.

[23] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, , and X. Zhang, "Virtual clusters for grid communities," in *IEEE/ACM International Symposium on Cluster Computing and the Grid*, Singapore, May 2006.

[24] A. Sundararaj, M. Sanghi, J. R. Lange, and P. A. Dinda, "Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments," Northwestern University, Tech. Rep. NWU-EECS-06-06, Jul. 2006.

[25] G. Czajkowski, M. Wegiel, L. Daynes, K. Palacz, M. Jordan, G. Skinner, and C. Bryce, "Resource management for clusters of virtual machines," in *IEEE/ACM International Symposium on Cluster Computing and the Grid*, Cardiff, UK, May 2005, pp. 382–389.