Empirical Evaluation of Latency-sensitive Application Performance in the Cloud

Sean K. Barker, Prashant Shenoy
Department of Computer Science
University of Massachusetts Amherst
[sbarker, shenoy]@cs.umass.edu

ABSTRACT

Cloud computing platforms enable users to rent computing and storage resources on-demand to run their networked applications and employ virtualization to multiplex virtual servers belonging to different customers on a shared set of servers. In this paper, we empirically evaluate the efficacy of cloud platforms for running latency-sensitive multimedia applications. Since multiple virtual machines running disparate applications from independent users may share a physical server, our study focuses on whether dynamically varying background load from such applications can interfere with the performance seen by latency-sensitive tasks. We first conduct a series of experiments on Amazon's EC2 system to quantify the CPU, disk, and network jitter and throughput fluctuations seen over a period of several days. We then turn to a laboratory-based cloud and systematically introduce different levels of background load and study the ability to isolate applications under different settings of the underlying resource control mechanisms. We use a combination of micro-benchmarks and two real-world applications—the Doom 3 game server and Apple's Darwin Streaming Server—for our experimental evaluation. Our results reveal that the jitter and the throughput seen by a latency-sensitive application can indeed degrade due to background load from other virtual machines. The degree of interference varies from resource to resource and is the most pronounced for disk-bound latency-sensitive tasks, which can degrade by nearly 75% under sustained background load. We also find that careful configuration of the resource control mechanisms within the virtualization layer can mitigate, but not eliminate, this interference.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes;
D.4.8 [**Operating Systems**]: Performance—*measurements*

General Terms

Measurement, Performance

MMSys'10, February 22–23, 2010, Phoenix, Arizona, USA.

Keywords

Cloud computing, multimedia, resource isolation, virtualization

1. INTRODUCTION

1.1 Motivation

Cloud computing has emerged as a new paradigm where an organization or user may dynamically rent remote compute and storage resources, using a credit card, to host networked applications "in the cloud." Fundamentally, cloud computing enables application providers to allocate resources purely on-demand – on an as-needed basis – and to vary the amount of resources to match workload demand. The promise of cloud computing lies in its "pay-as-you-use" model – organizations only pay for the resources that have been actually used and can flexibly increase or decrease the resource capacity allocated to them at any time. Doing so can yield cost savings since organizations no longer have to maintain an expensive IT infrastructure that is provisioned for peak usage – they can instead simply rent capacity when needed and release it when the peak ebbs. Since cloud providers bill for usage at a very fine-grained level; e.g., based on hourly usage of servers or based on amount of disk or network I/O, the model is attractive to any organization or user that has variable computing or storage needs.

While the initial wave of cloud computing applications has focused on providing web-based services or running "batch" jobs based on MapReduce [10], the paradigm is well-suited for running multimedia applications as well, as depicted in the following hypothetical examples:

Consider a group of friends that wish to play an online game such as Doom, but they do not have a machine available to use as a private server. They decide to rent a cloud server for an evening to host their game server and run their game clients on their laptops. Since a cloud server may cost as little as 10 cents an hour to rent, an evening of entertainment costs less than a dollar. Furthermore, the cloud platform provisions each server with ample resources and a fast network connection while requiring no manual upkeep or configuration – users simply rent cloud servers for a few hours and terminate them when done.

Next, consider a user who wishes to convert her high-definition personal video library to the H.264 format for sharing on her smartphone or for posting on the web. Since transcoding hundreds of hours of home movies on a desktop may take hours or days, she decides to leverage the computational power of the cloud. She rents multiple fast cloud servers, partitions her library into groups of videos and assigns each group to a different server. The multi-core servers complete their tasks in a few hours, and at a total rental cost of a less than $10. Although uploading a large video collection from a home to the cloud may take inordinately long, it is conceiv-

able that the user has kept a backup of the videos on cloud-based storage (and thus need not upload any data) or can physically ship a portable disk containing the data to the cloud provider for faster uploading (some cloud providers already support such a service for uploading very large datasets).

On the commercial side, we envision a local newspaper or a local TV news channel that uses cloud servers to web-cast a local but popular event; since its normal web server infrastructure may not be provisioned for high-volume streaming, it is simpler to rent capacity from the cloud for the duration of the event for streaming. Similarly, small gaming companies may leverage the cloud to host new online games – since the popularity of a new game cannot be predicted a priori, a company may choose to initially host the game on a small number of cloud servers, and scale up the capacity if the game becomes popular.

While cloud servers and storage are well-suited for web-based applications that are more tolerant to delay and jitter, a key question is how well cloud platforms can service the needs of latency-sensitive multimedia applications. For instance, game users are very sensitive to delays from the online servers, and streaming applications are sensitive to both bandwidth fluctuations and jitter. In this paper, we empirically evaluate the efficacy of cloud platforms when servicing multimedia applications. In particular, we examine the resource control mechanisms available in cloud platforms and evaluate which mechanisms are best suited for multimedia applications and scenarios under which these mechanisms expose interference from other competing applications running on the same hardware.

1.2 Research Contributions

Our paper makes three sets of contributions.

EC2 evaluation. We run a series of microbenchmarks on the Amazon EC2 public cloud over a period of several days to evaluate the jitter and delay seen by latency-sensitive CPU, disk and network tasks. Our results show that, despite the resource control mechanisms that are designed to provide performance isolation across co-located virtual machines, there may be significant jitter in the CPU availability, and disk latency and throughput seen by a cloud-based application. We also find a certain degree of variation in the first-hop network latency as well. Overall these results show that a cloud platform may expose interference from competing applications that are running on other VMs on the same physical server, causing the performance of latency-sensitive tasks to fluctuate over time.

Evaluation of hypervisor resource control mechanisms: To better understand the cause of such interference and to determine configuration settings to minimize their impact, we conduct a detailed evaluation of the resource control mechanisms employed by the Xen hypervisor—the same hypervisor employed in EC2. We introduce a controlled background CPU, disk and network load and study the impact of varying this load on the performance of latency-sensitive tasks; we also evaluate the impact of using different "knobs" exposed by the resource control mechanisms in Xen and Linux on the degree of performance isolation seen by competing applications.

We find that Xen's CPU scheduler generally provides fair shares even under stress but may expose jitter when multiple VMs share a CPU. The worst interference is seen for a shared disk, which lacks a proper isolation mechanism in Xen; we also find that network interference can be mitigated by properly configuring bandwidth sharing mechanisms to isolate VMs.

Multimedia application case studies: We evaluate the performance of two latency-sensitive applications — the Doom 3 game server and the Darwin Streaming Server on a laboratory cloud server. Our goal is to evaluate the real-world impact of various types of background interference, particularly network, which is harder to predict. We find that background network I/O has a significant impact on latency and jitter in addition to decreasing available throughput. Using the Linux utility `tc` allows us to manage this latency impact and provides a choice between sharing and dedicated bandwidth that results in a tradeoff between lower latency and jitter.

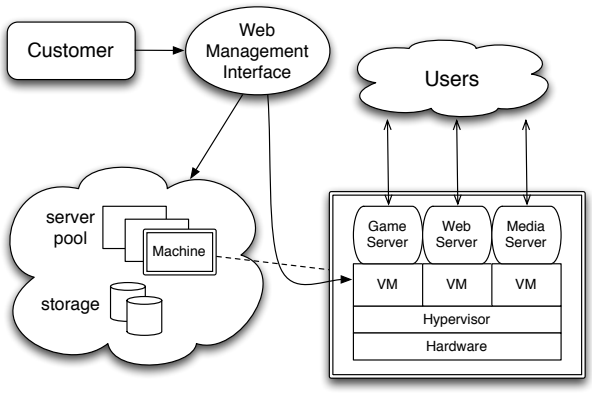


Figure 1: Cloud platform architecture.

Overall, our evaluation finds that while latency-sensitive applications on cloud servers are subject to harmful interference, proper server configuration can mitigate, but not totally eliminate, the impacts of such interference. Such configuration involves tuning Xen's hypervisor CPU settings as well as using the `tc` utility to isolate network performance; however, the lack of disk isolation mechanisms in Xen remains a problem for disk-bound latency-sensitive tasks.

The rest of this paper is structured as follows. We provide a brief background on cloud computing platforms in Section 2. We present our Amazon EC2 results in Section 3. We discuss Xen's resource control mechanisms and present the results of our Xen-based evaluation in Section 4. Our multimedia application case studies are presented in Section 5. We discuss related work in Sec 6 and conclude in Section 7.

2. CLOUD COMPUTING PLATFORMS: AN INTRODUCTION

To better understand the efficacy of running latency-sensitive multimedia applications on cloud platforms, we must first understand the internals of how cloud platforms are designed. A cloud platform is essentially a virtualized data center with a self-service web portal.

The data center consists of a cluster of servers, each running a hypervisor or a virtual machine monitor (see Figure 1). Customer applications are run inside virtual machines that are mapped onto these physical servers; each server may run multiple virtual machines, each belonging to a different customer. In the cloud platform, a customer uses the web portal to request a new server (and possibly storage). Upon receiving a new user request, the cloud management software locates a server with sufficient unused capacity and uses the hypervisor to create a new virtual machine. The user-specified VM image is copied over to the server and attached to the virtual machine; the new virtual server is then started up using the OS kernel in the user-specified image. Upon setup, a cus-

Instance Type	Memory	ECUs	Disk	Cost
Small (default)	1.7 GB	1	160 GB	$0.10
Large	7.5 GB	4	850 GB	$0.40
Extra Large	15 GB	8	1690 GB	$0.80
High-CPU Med.	1.7 GB	5	350 GB	$0.20
High-CPU XL	7 GB	20	1690 GB	$0.80

Table 1: Amazon EC2 instance types. One ECU (EC2 Compute Unit) is roughly equivalent to a 1.0-1.2 GHz 2007 Opteron or Xeon processor. Costs are per hour running a Linux instance as of Oct. 2009.

tomer gets full (i.e., privileged) access to her virtual server and can then install and run applications of her choice.

To simplify initial setup, virtual machines running particular software are packaged into portable images (typically single files) called *virtual machine images*. These images are then made available to users, removing most of the installation and configuration necessary to set up a new VM. This makes individual VMs much more disposable than traditional machines, and may be quickly set up or destroyed on particular physical machines. For example, a new VM in Amazon EC2 is set up by selecting the desired VM image, which is then copied to an automatically selected physical machine in an EC2 data center and booted, resulting in a fully installed and configured system in minutes.

In a virtualized environment, physical servers are controlled by a privileged *hypervisor*, which acts as the intermediary between hardware itself and the set of virtual machines running on it. Individual virtual machines are typically indistinguishable from native machines from an end-user perspective, exposing their own disks, memory, and operating system software. Multiple VMs on the same machine need not even be running the same operating system and are oblivious to the particular configurations of co-located VMs.

Typical cloud providers support servers of different "sizes"—for instance, Amazon's EC2 supports small, large and extra large server instances; these instances differ in the number of cores and the memory allocated to them and are priced accordingly (see Table 1). From a cloud platform perspective, multiple virtual servers may be mapped onto a single physical server. Thus, if the cluster comprises eight-core servers, each physical server can house eight uni-core small VMs, four dual-core VMs and two quad-core VMs. The underlying hypervisor partitions resources on the physical server across VMs so as to allocate the amounts specified in Table 1 and to isolate the VMs from one another. While the cloud management software can specify a resource partitioning to the underlying hypervisor, it is up to the resource control mechanisms in the underlying hypervisor to enforce these allocations at run-time and to provide the desired isolation. The goal of our work is to empirically evaluate how well these resource control mechanisms work in terms of performance isolation. In particular, we empirically evaluate whether a VM that sees a sudden CPU, disk or network load spike can impact the performance seen by a different VM running a latency-sensitive application.

Experimental methodology: We begin our experimental study by running a sequence of latency-sensitive micro-benchmarks on the Amazon EC2 public cloud over a period of several days. Our micro-benchmarks measure the variations in the CPU, disk, and network performance seen by a VM. If our results show that the allocations of these resources is stable and does not vary over time, then this will imply that existing cloud platforms are well-suited for running latency-sensitive applications. On the other hand, if our experiments uncover jitter in the resource allocations, then this will imply that the underlying hypervisor is unable to fully isolate the VMs from one another. We must then examine the hypervisor itself and determine what resource control mechanisms (and what settings) are best suited for latency-sensitive applications.

As we will show later, our EC2 experiments do reveal jitter in the performance seen by a latency-sensitive application. However, since EC2 does not expose the hypervisor control mechanisms to a customer and nor does the EC2 platform provide any control over what other VMs are placed on a given physical server, we conduct our hypervisor-based experiments on a laboratory cloud platform. Such a laboratory-based cloud allows complete control over background load in other VMs (allowing repeatability of experiments) and also allows full control how the resource allocation mechanisms in the underlying hypervisor are configured. We use the Xen virtual machine platform for our laboratory-based cloud experiments. We choose Xen for two reasons. First, Xen is an open-source platform allowing us to examine (and, if necessary, modify) the underlying resource control mechanisms. Second, the Amazon system is based on a variant of Xen [1], allowing our laboratory cloud to better approximate the environment in which latency-sensitive applications run on EC2.

3. MICROBENCHMARKING EC2

In this section, we empirically evaluate the efficacy of a public cloud platform, Amazon's EC2, for hosting latency-sensitive applications. At the outset, we note that Amazon's EC2 SLA *does not* provide any explicit performance guarantees for latency-sensitive applications (other than specifying the approximate hardware resources allocated to a customer's virtual machine as shown in Table 1). Thus, if an application experiences variable performance due to background load from other VMs, this does not imply any SLA violations, since no explicit performance guarantees are provided. Our goal is to quantitatively determine whether such background load can impact application performance on a cloud server and by how much.

Time-shared systems such as Linux are designed to support interactive applications, and their underlying schedulers do not provide performance guarantees or isolation between applications. Thus, background processes can easily impact latency-sensitive applications in such systems. To address this limitation, there has been more than a decade of research addressing the design of fair-share schedulers that provide a guaranteed share of the resource to an application and also provide performance isolation [19, 16, 2]. Unlike OS kernels that typically support time-sharing, modern hypervisors such as Xen and VMWare actually implement variants of these fair-share schedulers to provide performance isolation to resident VMs (in Section 4 we discuss the specific resource control mechanisms implemented in hypervisors in more detail). Thus, one may expect the degree of performance isolation in hypervisors and, by extension, in cloud servers, to be significantly better than in time-shared systems. The key question is whether this performance isolation is "good enough" for running latency-sensitive applications, even though cloud system do not provide any explicit SLA performance guarantees to this class of applications.

Microbenchmarks and Methodology: We rented an Amazon EC2 server for several weeks and ran several latency-sensitive microbenchmarks on this server; we chose the standard "small" EC2 configuration for our experiments, as it represents the most common setup. It is important to note that we have no control whatsoever over the background load (i.e., the load imposed by other resident VMs) on this physical server, nor do we have any control over how many VMs are collocated on the machine. Consequently, we repeatedly

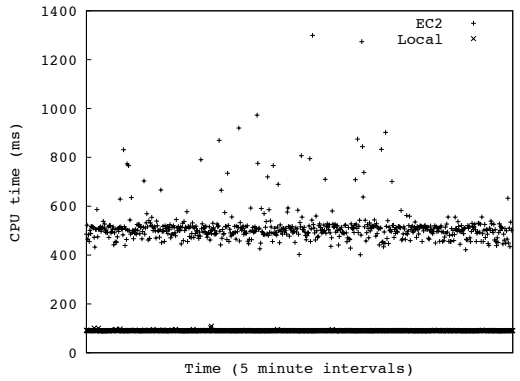


Figure 2: CPU jitter on an EC2 server.

ran each benchmark over a period of several days to maximize the chances of experiencing varying background loads and to evaluate its impact on the performance of our benchmarks.

We loaded our EC2 instance with an Amazon Machine Image (AMI) running CentOS 5.3 with a Linux 2.6.18 Xen kernel. Our microbenchmarks tested the three primary shared resources–CPU, network, and disk—to understand the impact of background load on the performance of each resource. Our VM software consisted of a server that ran our tests on-demand whenever a request was received. We ran the tests and collected results every 5 minutes – the test battery generally took less than 10 seconds to complete, leaving the VM idle the rest of the time. We describe each benchmark task here.

Our CPU micro-benchmark is a single threaded task that performs a small, fixed number of floating-point computations. Since the task is strictly CPU-bound with no I/O, its completion time provides a reasonable measure of the CPU available to the virtual machine—if the measured completion times fluctuate, this implies that the underlying hypervisor is allocating different amounts of CPU to the VM, possibly due to a varying background load.

Our disk micro-benchmark consists of four independent tests. We wish to measure both the sustained disk throughput available to the VM (both reading and writing) as well as the latency of small read and write operations. Whereas jitter-free disk throughput is important for applications such as streaming servers, jitter-free small I/Os are important to applications such as game servers. Benchmarking disk I/O performance is complicated by the presence of caching both at the disk level and at the OS level. Where possible, we attempt to isolate the effects of caching from our results by clearing the disk cache between successive benchmark runs.

For our read evaluation, we created a set of several hundred 5 MB files of random data to use for the disk tests. Each small read test consisted of picking a random file and measuring the latency to read a randomly chosen 1 KB block from it. The sustained read test was similar and involved reading several entire files in succession. Write tests were the same as for reads, except they wrote data into the files rather than reading them.

Our network benchmark also had several subcomponents. In order to isolate interference from other VMs from regular Internet traffic variations, we used `traceroute` to measure ping times *within* Amazon's EC2 infrastructure. We measured both the ping from our server to the next immediate hop as well as the sum of the first three hops – the former captures jitter seen at the network interface of the server, while the latter captures a wider range of jitter occurring within Amazon's internal routers. As a more comprehensive but less specific test, we also measured the time to transfer a 32 KB block back and forth between our local server and the EC2 VM. In addition to any delay occurring within EC2, this measurement includes delay and jitter seen by standard wide-area Internet traffic.

CPU microbenchmarks.

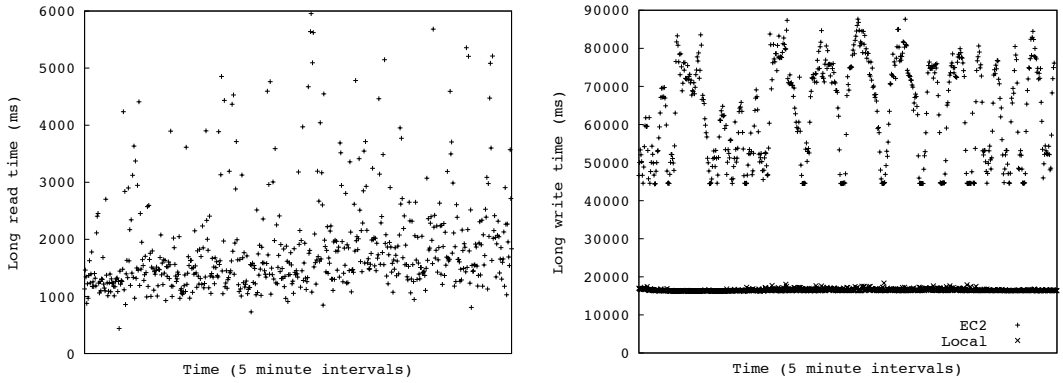
Figure 2 depicts the times taken to complete each invocation of our CPU microbenchmark over a period of several hours. We observed significant variations in the completion time of the task. While most of the tests completed in roughly 500 ms, there was frequent variations between 400 and 600 and many outliers taking significantly longer; a few even took more than an entire second. To verify that this behavior was due to other processes executing on the machine and not to our experimental setup or scheduling artifacts, we also ran our benchmark on a Xen virtual machine in a local server that was guaranteed to be otherwise idle (i.e., no co-located VMs performing any work). The results of this local test are also depicted in Figure 2. As the hardware in our local machine was different than the hardware provided in our EC2 VM, the absolute comparison of the completion times is not relevant to our discussion. However, our local benchmark observed almost no variation from the mean completion time, and certainly far less than we observed in EC2. Given that our EC2 VM was provisioned with an ostensibly fixed CPU capacity, the degree of variation we observed was surprising and strongly suggests that other VMs running on our EC2 server were impacting the CPU performance of our VM. Also notable is that the highest outliers in our EC2 test appeared to cease during the latter part of our test – this behavior could be explained by the shutdown of a collocated VM, resulting in less background interference.

Overall, these results suggest that the fair-share CPU scheduler in the hypervisor, as configured in EC2 systems, is not able to fully isolate VMs from one another. Our experiments demonstrated that amount of CPU available at any given time can fluctuate subject to interference out of the VM's control.

Disk microbenchmarks.

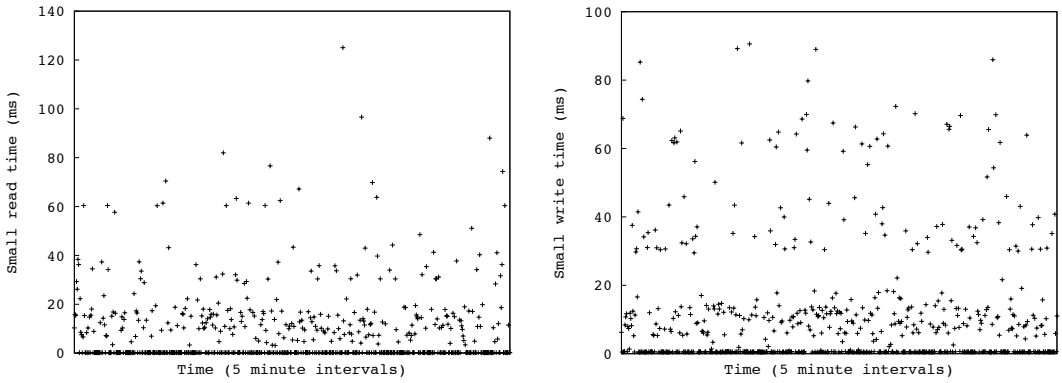
The results of our disk microbenchmarks are shown in Figures 3 and 4. Figure 3 (a) and (b) shows the observed jitter in the disk throughput for "streaming" reads and writes—depicted as the completion time for large reads and writes. Both display a significant amount of variation, particularly the write task, which appears to display an ebb and flow of available write bandwidth. This curious pattern (which, we note, is not correlated with the time of day) again prompted us to run the task on a local VM – as in the CPU task, our local VM was extremely consistent and displayed none of the wide swings we observed in the EC2 VM. This is a result of some concern, as it demonstrates that available write bandwidth can easily vary by as much as 50% from the mean. The read bandwidth results also display significant variation, but we refrain from drawing stronger conclusions regarding them – the low completion times overall strongly suggest that caching or read-ahead, likely within the hypervisor, is influencing our results.

The results for our small disk latency tests are similar. Figures 4 (a) and (b) plot the amount of time used to read or write a small 1 KB block of a random file. In both graphs, the presence of many data points close to zero indicate that in many cases, a copy of the selected file was cached somewhere and no disk I/O actually occurred. However, in those which did incur a disk operation (probably most data points above 20 ms), the completion time varied widely within a factor of about 5 (up to 100 ms).



(a) Streaming Reads (b) Streaming Writes

Figure 3: Jitter in disk throughput for large/streaming reads and writes.



(a) Small reads (b) Small writes

Figure 4: Jitter for small disk reads and writes.

These results, despite caching artifacts, still clearly indicate that there is significant room for improvement in isolating disk I/O between VMs. The wide variation in bandwidth available to a VM is likely to be problematic for a latency-sensitive application that relies on a certain level of service from the hardware.

Network microbenchmarks.

Figure 5 displays the results of our network microbenchmarks for three-hop internal latency (a) and the small wide-area data transfer (b). The results of the one-hop latency test are omitted, as we saw effectively no jitter at that granularity. In the case of the first three hops, while the majority of the pings displayed times typical of a LAN (less than 5 ms), we also observed a significant number of pings taking an order of magnitude (or more) longer. Based on the our one-hop results, there is no reason to attribute these spikes to virtualization or lack of isolation, but rather to queuing at the internal switches and routers, given the undoubtedly significant amount of traffic that traverses Amazon's infrastructure. However, these variations may still cause difficulties for latency-sensitive applications running in high-bandwidth environments such as data centers.

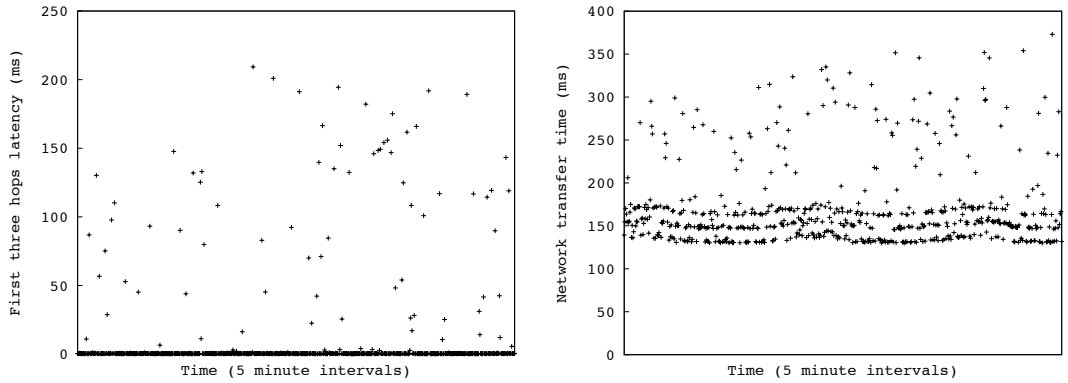
The results of the network transfer operation display a reasonably wide variation, as well as the regular pattern of increasing transfer times during peak work hours (mornings and early afternoons) and decreased times during evenings and nights. These results are fairly typical of any wide-area Internet application and do not suggest any performance isolation issues specific to EC2.

Overall, our network-based tests suggest that a certain amount of latency variation is to be expected when running in a cloud-based environment, but do not implicate resource sharing as the cause of this variation. However, it is entirely possible that our VM was simply not collocated with anything performing notable amounts of network I/O. We evaluate the effect of network interference in a more controlled setting later in Section 5.

4. RESOURCE SHARING IN XEN

Given the performance results from our EC2 study, we next conduct several experiments to understand the resource control mechanisms available to the hypervisor. We run these experiments on a Xen-based laboratory cloud, which enables us to control the background load as well as exercise the settings of the various resource control mechanisms to determine their impact on the jitter seen by applications.

We first discuss Xen's mechanisms for sharing each resource—CPU, disk, and network—and then present the results of experimental evaluation for that resource. Since RAM is statically partitioned between VMs in Xen, no significant interference is expected for memory, and thus we do not consider memory sharing in this paper. Unless noted otherwise, we run the same microbenchmarks as our EC2 experiments but in a more controlled setting. Section 5 then presents application case studies where we run full-fledged applications to evaluate their performance with and without background interference.



(a) Internal traceroute latency (b) Network transfer time

Figure 5: EC2 Network microbenchmarks.

4.1 CPU Sharing

Consider a multi-core physical server that runs multiple virtual machines. In general, the number of virtual machines can be greater than the number of cores in the system. To enable flexible sharing of the cores, Xen defines the notion of a virtual CPU , or *VCPU*. Each virtual machine is assigned one or more VCPUs at VM creation time; the hypervisor allows for a flexible mapping from VCPUs to physical cores in the system. It is possible to bind a VCPU exclusively to a physical core, or each core can be shared between multiple VCPUs.

In the event where multiple VCPUs are assigned to a core, or where all cores are shared across all VCPUs, the hypervisor must employ a CPU scheduler that divides time on the cores between the VCPUs. Xen implements several CPU scheduling algorithms. For instance, it implements Borrowed Virtual Time [14]—a variant of the start time fair queuing (SFQ) proportion fair-share scheduler [16, 15], which functions by assigning each VM a *virtual time* and attempting to keep them in sync by scheduling the VCPU with the smallest virtual time (thus fairly sharing between VCPUs in proportion to their weight). Xen also supports a variant of the Earliest Deadline First scheduler, which uses a model of real-time guarantees for scheduling. Both of these, however, have been deprecated in favor of the newly implemented default scheduler, the Credit Scheduler. This scheduler operates by having each VM expend periodically reissued credits for CPU time. When a VM exceeds its credit allowance, it is scheduled after other VMs waiting for CPU time until credits are reissued.

Per-VM credit allocation is managed by two parameters: a **weight** and an optional **cap**. The weight is simply a relative measure of how many credits should be allocated to the VM. For example, a VM with a weight of 2 should receive twice as much CPU time as a VM with a weight of 1. Be default, the credit scheduler is *work conserving*, which means that a VM that has expended all of its credit will be allocated additional CPU time if no other VM is using its allocated share (i.e., unused CPU cycles are redistributed to needy VMs rather than being wasted). The scheduler uses the notion of a cap—when defined, it specifies the maximum amount of CPU the VM is allowed to consume, even if the rest of the CPU is left idle. For example, a VM with a cap of 70 is only allowed to consume 70% of a single core. If a VM has been assigned multiple VCPUs, then it may use more than a single physical CPU concurrently; due to the decoupling of CPUs and VCPUs, the number of VCPUs simply defines an upper limit on the CPU consumption of the VM. VCPU allocations are automatically load balanced across all physical CPU cores, but may be pinned to specific physical CPUs using administrator tools in the hypervisor (domain-0 in Xen).

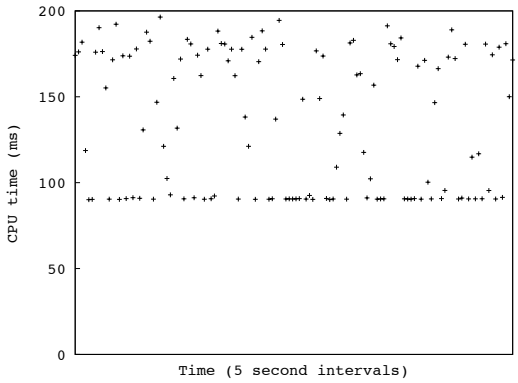
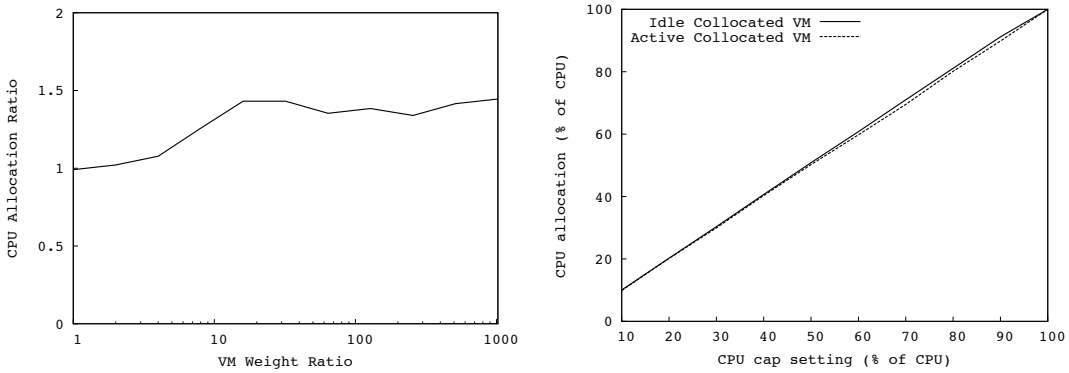


Figure 6: CPU benchmark performance with randomized background load.

Our experiments evaluate the impact of different weight and cap assignments to understand their efficacy in isolating the performance of latency-sensitive applications.

CPU scheduler benchmarks: The credit scheduler allows a hypervisor to specify a CPU share (via the VM's weight) and hard limits (via the cap). We conducted a series of experiments to evaluate the impact of both parameters on the performance and jitter seen by a latency-sensitive application. We created two virtual machine on a Xen server; the first virtual machine ran the same CPU microbenchmark as our EC2 experiment and the second VM was used to introduce different amounts of background CPU load on the server. Both VMs were bound to the same single CPU core in order to evaluate the impact of per-core sharing. First, we gave the VMs equal weights and evaluated the jitter seen by our microbenchmark under variable background load in the second VM. This evaluates a scenario possibly like the one we observed on EC2. Next, we varied both the weight and the cap assigned to the foreground VM while keeping the background VM constantly busy and measured the CPU share of the first VM.

Figure 6 plots the jitter seen by the first benchmark, and as in the EC2 case, we see that the completions times are unpredictable over time. Figures 7 (a) and (b) depict the results obtained by varying the weight and the cap, respectively. For the weight test, we fixed one VM's weight and gradually increased the weight of the other,



(a) Varying VM Weights

(b) Varying VM Caps

Figure 7: Xen credit scheduler tests.

then plotted the ratio of their weights versus the ratio of their CPU shares. According to the credit scheduler specification, these values should always be nearly equal. At equal weight, this was true; both VMs received exactly half of the CPU. However, this was not the case with different weights; increasing the weight of one VM only increased its CPU allocation by a small amount. After a ratio of 10 or above, the higher-weighted VM was only receiving about 33% more CPU than the other.

For evaluating the effect of caps, we varied the cap of one VM from 10% of a CPU to 90% of a CPU and varied the other in reverse (so the total was always 100%). This represents a situation in which each VM has a dedicated piece of the CPU that may not be used by other VMs. We also ran this test a second time in which the second VM was not executing (in this case, the total CPU usage would be less than 100%). In both cases, the CPU allocation of the first VM varied linearly with the cap, exactly as the scheduler specifies it should. Furthermore, the presence of a competing VM using the remainder of the CPU had no significant effect on the CPU allocated to the original VM.

These results suggest that sharing a CPU using weight but no caps will cause latency-sensitive tasks to experience jitter. Furthermore, the notion of a weight in the credit scheduler seems different from that in fair-share schedulers—a higher weight yields a higher share but the weight does not precisely denote the proportion of CPU allocated to a VM For a latency-sensitive application that requires a guaranteed certain CPU share, such a guarantee can be made by assigning appropriate caps to each resident VM or dedicating CPU cores. Finally, although not evaluated here, in multi-core systems, strong isolation can be achieved by dedicating an entire core to a VM's VCPU, in which case it will see minimal interference from activity on other cores. This may not be true every case, since cores may share resources such as cache, but should prevent the possibility of starvation.

4.2 Disk Sharing

A typical cloud server will consist of a fast hard disk with hundreds of gigabytes of disk storage; the disk capacity and bandwidth is typically shared between the resident VMs. This may be achieved by placing multiple VM machine images— each of which act like a virtual disk to its VM —on a single hard drive. In such a setup, although disk space is statically allocated to each VM, actual disk bandwidth is shared between them.

The Xen I/O model places the hypervisor (running on the Domain-0 VM) between the disk and the virtual machines. All I/O requests from the virtual machines are routed through the hypervisor, which is the only entity with direct access to the physical disk. A simple round-robin servicing mechanism is employed to handle batches of competing requests from each VM; there is presently no mechanism to explicitly partition disk bandwidth between VMs. Thus, an overloaded or a misbehaving VM may impact the disk performance in other VMs.

Disk sharing benchmarks: We conducted a set of experiments to determine the extent of the performance isolation between the VMs. We used the same disk microbenchmarks as our EC2 experiments and measured the same four statistics as described in Section 3: small I/O operations and overall throughput for both reads and writes. We ran our EC2 disk benchmarks in one VM while the other performed heavy I/O by performing continuous `dd` operations to read and write random files in multiple parallel threads (one reader thread per writer thread). The benchmark VM repeated the tests while we varied the number of background thread pairs in the second VM. We used the default disk schedulers set by Xen: the CFQ scheduler in Domain-0 and the simpler NOOP scheduler in the guest VMs [5]. While [5] suggests that throughput could be substantially improved by modifying the schedulers used, the impact on fairness is demonstrated to be fairly minimal, so we did not try other scheduler combinations.

Figure 8 shows the performance degradation (measured by the time required to complete the benchmark tasks) experienced by the first VM with varying background load in the second VM (depicted by the number of thread performing I/O operations in the latter VM). In the ideal fair case, we expect each VM to get 50% of the disk bandwidth. Ideally, there should be no impact on performance regardless of the background load (i.e., no performance degradation). The actual result, however, is that the second VM is able to capture more than its fair share of disk bandwidth (or deprives competing VMs of their fair share). With a single thread pair, throughput in the first VM dropped by roughly 65% and read and write latencies increased by nearly 70% and 80%, respectively. Increasing the number of thread pairs had a noticeable additional impact on read latency and write throughput, resulting in degradation of about 75% for each.

This figure also shows that despite this unfairness, there is a limit on the amount of degradation seen by the foreground VM. In other word, while we were able to dampen disk performance in the first VM by performing heavy I/O in the second, we were never able to actually cripple disk performance – after a certain point, increasing the amount of I/O occurring in the second VM had no additional impact on the first, which remained responsive in spite of the heavy

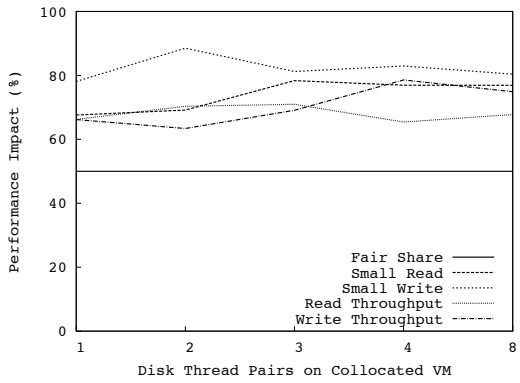


Figure 8: Disk performance with varying background load.

interference. We attribute this behavior to the round-robin servicing of disk requests from each VM – this round-robin policy appears to provide a degree of performance isolation by continuing to service requests from each VM even when one of the VM has a large number of outstanding requests.

For applications not performing intensive I/O, even a 75% drop in available throughput or disk latency may not have a major impact on application performance. However, for applications relying on high disk throughput (such as media servers), such an unpredictable impact on performance is quite problematic. Similarly, the variable disk latency of small reads and writes can impact latency-sensitive applications that rely on small I/O requests.

4.3 Network Sharing

As with disk bandwidth, Xen has no built-in control mechanism for distributing network bandwidth between VMs. All network traffic is passed through a bridge in Domain-0 and onto a shared external network interface. However, bandwidth distribution and shaping may be accomplished by employing the Linux traffic control (`tc`) tool. This tool performs bandwidth shaping and can be used to both to both limit and share bandwidth between VMs. Such a setup works well since all traffic from VM passes through Dom-0, and by implementing traffic shaping in Dom-0 for each VM, Xen eliminates the need to modify individual VMs.

Traffic shaping in `tc` is accomplished by creating *flows*, which may be rate limited or shared among child flows in a hierarchical fashion. Packets are then assigned to particular flows as desired. A simple `tc` setup to rate-limit VMs is to create a flow for each IP address of a DomU VM and assign packets to particular flows based on source or destination IP addresses. For example, if the machine has 3 megabits of upload bandwidth and wishes to provide 1 megabit to `VM1` and 2 megabits to `VM2`, the administrator can use `tc` to create two flows (rate-limited to 1 and 2 megabits, respectively) and then assign all packets with a source address matching `VM1`'s IP address to one network flow and similarly for `VM2`.

The `tc` utility allows both for strict caps on network bandwidth and shared bandwidth with performance guarantees. The former is a simple limit as described above. The latter means that bandwidth may be used by more than one VM, but each VM is guaranteed at least its rated speed. For example, the previous example could be modified to provide bandwidth sharing as follows: a parent flow could be created that is allowed to use the entire 3 megabits of bandwidth, then the two other flows could be assigned as children. This means that if either `VM1` or `VM2` is not using their bandwidth, the other may send at the full link speed. However, if both are sending as quickly as possible, then `VM1` will only receive one megabit, while `VM2` will receive two. Finally, both caps and sharing may be employed; for example, `VM1` might be allowed to use `VM2`'s excess bandwidth, but only to a maximum speed of 2.5 megabits.

We conducted several experiments using `tc` rate-limiting to verify that it performed as desired using the HTB (hierarchical token bucket) queuing discipline. We chose HTB because of its straightforward configuration and good documentation relative to other queuing disciplines available in `tc`. Our tests confirmed the expected behavior of rate-limiting using `iperf`, which exhibited sustained throughput within 10% of the limits specified. In our experiments where two VMs competed for bandwidth, `tc` reliably gave each VM its guaranteed rate while fairly dividing slack bandwidth. We conclude that `tc`'s rate-limiting mechanism works well for controlling throughput. However, for our applications, *we are particularly interested in the impact of* `tc` *on latency*. We explore this impact using several real applications, as described in detail in Section 5.

5. REAL-WORLD CASE STUDIES

While our previous results used micro-benchmarks to quantify the behavior of latency-sensitive tasks on shared cloud servers, we now explore the effects of such interference on real latency-sensitive applications. We pick two applications for our case studies: an online game server and a streaming media server and discuss each in turn.

5.1 Game Server Case Study

Many types of computer games have multiplayer or networked components that operate across the Internet. Some, such as online card games, can tolerate significant latency with little impact on gameplay. Real-time games such as a first-person shooter (FPS), however, are extremely sensitive to even small delays – such games rely on quick reaction times from players and fast response times from the game. High latency can result in game stutter and frustration for players when their actions do not take effect quickly (popularly known as 'lag'). For a representative FPS, we chose to use id Software's 2004 title Doom 3. Running the game's dedicated server in a VM in our lab, we conducted tests of two metrics: map loading times and server latency.

Map Loading Tests: Multiplayer games such as Doom 3 take place in a game map, of which several are generally available. Loading another map is a fairly common operation which may occur several times during the course of a single gaming session. Load times should be minimized, of course, to prevent forcing players to wait before continuing. Doom 3 includes a variety of multiplayer maps that may be used. To evaluate map loading times, we had the game server cycle the active map several times and took the average of the load times for each individual map. We repeated this test several times (restarting the server in between tests to reset the cache) with a variety of background interference occurring in another VM.

The results of this test are shown in Figure 9. With the second VM idle, each map load took approximately 2 seconds. With background disk activity, the load time increased by approximately 25%, as the server was unable to access the map resource files on disk as quickly. We ensured that this increase was due to disk interference alone and not a byproduct of the CPU usage of disk I/O by fixing the competing VM to a different CPU than the server. When we moved the VM to the same CPU and reran the test with a background CPU load, the average load time doubled – this makes sense for a CPU bound task, as 50% of the CPU is diverted away from the server to the computing VM. With both disk and CPU in-

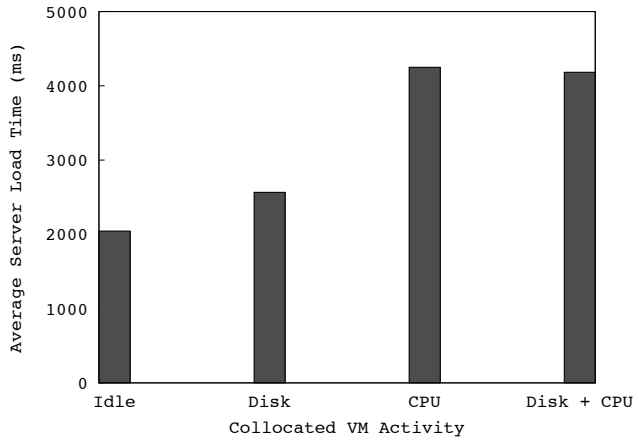


Figure 9: Game server map loading times with a collocated resource-hog VM.

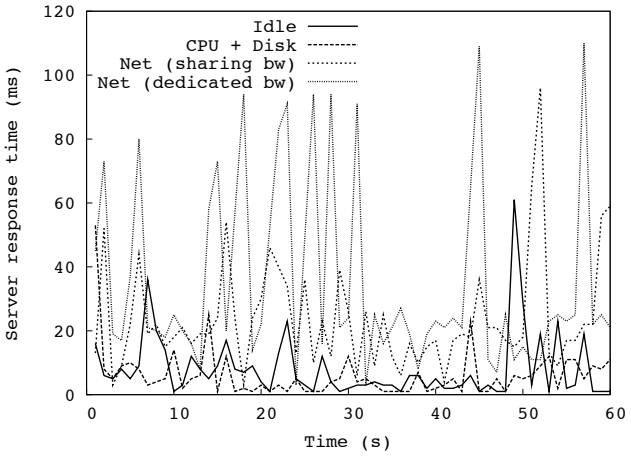


Figure 10: Game server latency with a collocated resource-hog VM and `tc` bandwidth shaping.

terference, the load time remained at roughly double the original time, reinforcing that the map load task was CPU-bound. Overall, this test confirmed our hypothesis that background work by other VMs can have a very real effect on application performance.

Latency Tests: To measure server latency in our test VM, we used the `qstat` utility, which is able to gather basic information (including latency) from a variety of game servers. Using `qstat` gives us an application-level measurement, which is more useful and reliable than the low-level machine response time that we would get from an ordinary ping. Given that the primary factor in latency is network conditions, the primary sharing mechanism we examined was `tc`. We evaluated two ways to use `tc` to share upstream bandwidth (the primary scarce resource) between VMs. One method is to take all available upstream bandwidth, divide it between competing VMs, and prohibit VMs from using any bandwidth that is not in their share. We refer to this method as *dedicated* bandwidth. The other method is to guarantee each VM its fair share, but allow them to draw on the complete pool of upstream bandwidth as well. We refer to this method as *shared* bandwidth. Finally, we can opt to not use `tc` at all, which simply shares the link between VMs as it would between processes in a single VM.

For our latency experiment, we started the dedicated server and connected a single local client to ensure that the server was actually using the network. Tests of more than one client did not show appreciable differences in overall latency, so we did not have more players join the server during the test. Since our experiment was conducted on our LAN, we used `tc` to create a bandwidth-constrained environment by limiting the machine's total upload to 1 megabit per second – while this may seem excessively low, the actual throughput used by the server is minimal (less than 100 kbps), especially with few connected players. We conducted five sets of latency tests, the results of which are shown in Figure 10 and Table 2. In each test, we measured latency once a second for 60 seconds while running a particular background load in a second VM and sharing bandwidth between the two VMs with a particular `tc` configuration. In all cases, `tc` was limiting the machine's total outgoing bandwidth. We used the following five setups: idle (in which the second VM did nothing, and `tc` was not performing shaping), CPU and disk interference, network interference (many concurrent `iperf` connections) without using `tc` to distribute bandwidth, and then network interference using `tc` to provide either dedicated or shared bandwidth.

Figure 10 shows the response times over the course of a minute for each setup, while Table 2 gives the average latency, standard deviation, and percentage of the 60 pings that failed to return a response. With no background interference, the server latency was steady at around 10 ms with some fluctuations but nothing that constituted a problem. The same was true with CPU and disk interference, indicating that (at least in our small server setup), nothing was bottlenecked by the CPU or disk. Heavy background network I/O, however, had a serious impact. When `tc` was not used, the background VM crippled the game server, causing 100% of our pings to time out and resulting in the game client effectively freezing as it waited for data from the server. The results show the importance of using bandwidth control to ensure that one VM does not starve others of bandwidth.

The application performance differed depending on whether bandwidth was shared or dedicated to each VM. In both cases, the game server remained responsive and gameplay on the client was generally smooth. However, the dedicated case had higher jitter when compared to the shared case. While fair sharing of the link with the background VM caused the average latency to moderately increase versus dedicating bandwidth, it also had the effect of significantly smoothing the jitter (due to the ability to utilize unused bandwidth from the bursty background VM). Dedicating bandwidth lowered the overall latency but also resulted in many latency spikes and higher jitter. As shown in Table 2, this issue manifested as a higher standard deviation in the dedicated case as well as a higher timeout rate. We ruled out the possibility that the server was simply starved of available bandwidth at the dedicated level by rerunning the test with a slower total link speed and no interference, confirming that this was not an issue by giving us similar times to the original idle setup (<10 ms). Thus, we conclude that fair sharing of bandwidth using `tc` yields somewhat higher latencies but can lower jitter and

Interference	Avg. Time	σ	Timeouts
Idle (none)	8.1	10.2	0%
CPU + Disk	6.2	7.9	1.7%
Net (no `tc`)	N/A	N/A	100%
Net (`tc`, dedicated)	23.6	29.6	6.7%
Net (`tc`, sharing)	33.9	16.9	1.7%

Table 2: Game server latency statistics.

timeouts in the presence of bursty background loads; dedicating bandwidth to VMs lowers the mean latency but can increase jitter and timeouts when the guaranteed allocation is temporarily exceeded.

Our game server experiments highlight the importance of using a tool like `tc` to fairly distribute bandwidth in a potentially competitive network environment, as well as illustrate the tradeoffs that result from the configuration of such a tool. In particular, the configuration of `tc` presents a tradeoff between the lower average latency and lower jitter.

5.2 Streaming Media Server Case Study

Our second case study focused on the performance of a streaming media server. Streaming servers continuously transmit data across a network to clients, and as such require a fast, reliable connection to avoid stuttering or loss of quality in the video or audio feed being served. Techniques such as server-side and client-side buffering can help mask network fluctuations, but a variable throughput and/or excessively high latency connection can still create significant problems for a streaming server.

We chose Apple's open-source Darwin Streaming Server (DSS) to use for our streaming server experiments. DSS also comes with a tool to generate artificial client loads on the server, which greatly simplifies testing. We were interested in two primary metrics: the total throughput served by the server to a set of streaming clients, and the average jitter of those streams. Here, jitter refers to the variability in the arrival intervals of packets; a connection on which packets arrive at highly irregular intervals will have undesirable high jitter, while one on which packets arrive regularly and on-schedule will have desirable low jitter. The previous section has already shown that imperfect disk isolation in Xen can negatively impact the disk throughput of an application under background load, given the lack of disk QoS mechanisms. Here we were primarily interested in server latency characteristics under background network interference.

As in our game server experiments, we looked at four different `tc` configurations to evaluate our two metrics: idle (no background interference), off (interference without traffic shaping), shared `tc`, and dedicated `tc`. The full machine upload speed was limited to 10 Mbps in all cases. To evaluate the total bit rate, we started up several 1 Mbps streams of a movie file and ran them for one minute, after which the individual overall bit rates were summed. To evaluate jitter, we performed the same test and ran a single `iperf` connection transferring 1 Mbps alongside the regular streams, which provided jitter statistics at the completion of the test.

Since the full upload speed was 10 Mbps (capable of serving around 8 streams), the server's "fair share" of bandwidth was only half of that when the collocated network hog was running. To account for this, we ran all tests with both 4 and 8 concurrent streams. Our results are shown in Figures 11 (bit rates) and 12 (jitter).

With no background interference, the streaming server was able to service 4 clients at full quality or 8 clients at roughly 60% quality. Once background load was introduced, however, the overall throughput dropped by about 30%. Running `tc` either with shared or dedicated bandwidth caused the average bit rate to recover significantly, though not fully to the rates observed before any background network activity was introduced. The behavior of the jitter, however, was more varied. Jitter increased considerably when introducing the background load across all configurations. However, the amount of jitter was even worse when dedicating bandwidth with `tc` than in fair sharing case.

These results support the conclusions from our game server study: fair sharing of bandwidth yields lower jitter than in the case where

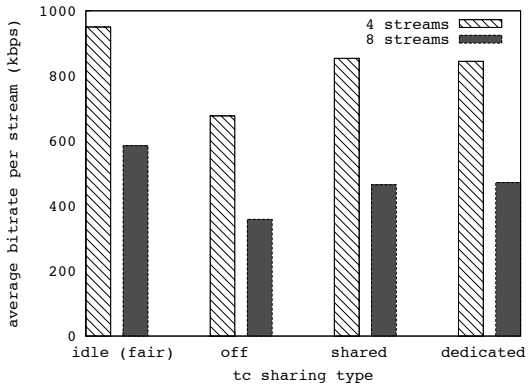


Figure 11: Average bit rate over 4 or 8 concurrent streams with a variety of network sharing setups.

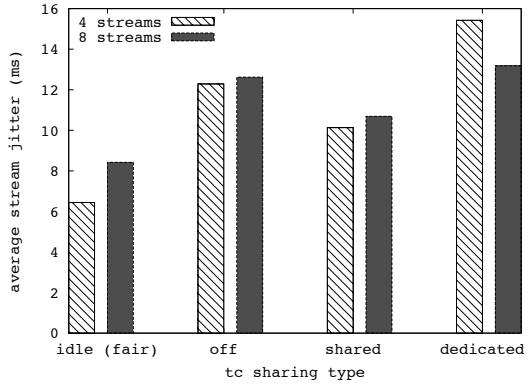


Figure 12: Stream jitter over 4 or 8 concurrent streams with a variety of network sharing setups.

the link bandwidth is dedicated to VMs. Given that the shared and dedicated setups performed equally well in our throughout experiments, fair sharing appears to be the better choice for the streaming server application. For latency sensitive applications in general, the choice of police type presents a tradeoff between mean latency, latency variance (jitter), and timeouts (which are, in this case, a side effect of high jitter).

6. RELATED WORK

The importance of predictable resource allocation in certain applications has resulted in a significant amount of work on fair-share schedulers. Several techniques have been proposed for predictable allocation of the processor [14, 15, 19, 25], network bandwidth [7, 11, 16] and disk throughput [2, 23, 18] Broadly speaking, these efforts have been targeted at adding quality-of-service (QoS) to operating systems. Several systems have been developed to integrate many resource allocation schemes into a single cohesive QoS-aware system. These systems include QLinux [31] (based on the Linux kernel), Nemesis [2, 27] (built from the ground-up), and Eclipse [4] (based on FreeBSD). Versions of Solaris have also included a resource manager that enables fine-grain allocation of various resources to processes and process groups [30]. These mechanisms vary from automatic resource allocations by the OS to application-specified resource requirements. We explore hypervisor-level fair-share mechanisms to divide resources on a VM rather than process level.

Work in hypervisor techniques have resulted in several established virtualization systems. Two of the most well-known hypervisors are Xen [3, 8] and VMware ESX Server [32]. Xen in particular has been the focus of a significant amount of work regarding performance isolation between VMs [17]; this has included evaluating Xen's different CPU schedulers and its I/O driver model. In addition to managing the interactions between VMs, there have also been efforts to reduce the overhead of Xen's driver stack relative to bare-metal Linux [22]. We leverage Xen's capabilities to explore the performance guarantees that can be made between competing VMs. One of ESX Server's key features not yet existing in Xen is a shared memory model that allows administrators to provision memory to multiple VMs, trading off higher memory availability with greater performance consistency [32].

Cloud computing has become an increasingly active area of research, particularly with the advent of commercial wide-scale deployments such as EC2 [1]. One of the primary questions in such deployments is how to perform dynamic resource allocation, particularly with regard to VMs. Both Xen [9] and VMware [24] have implemented "live" VM migration techniques that allow the movement of VMs between machines to be used as a resource allocation technique, such as in [28]. The requirements of SLAs specifying certain levels of performance are another area of work in cloud computing; a variety of shared resource models have been developed to deal with the issues of allocating resources on a multi-machine and multi-service level [6, 13]. Estimating SLA resource requirements by inspecting application behavior has been explored in work such as [20]. SLA requirements are relevant to our work in that such requirements are likely to be significantly more stringent for a latency-sensitive multimedia application than for a typical web application.

Performance isolation in virtualized systems has been studied from several perspectives, including the role of virtual machine monitors in I/O performance [29], design of fair virtual I/O schedulers [26], and benchmarking suites for evaluating performance isolation [12, 21]. We build on this existing work by focusing on multimedia and latency-sensitive application performance at a user level. We also consider the potential role of system administrators to configure and manage the risks of performance interference.

7. CONCLUSIONS

Motivated by the increasing popularity of cloud platforms for running hosted applications, in this paper we conducted an empirical study to evaluate the efficacy of these platforms for running latency-sensitive multimedia applications. Since multiple virtual machines running disparate applications from independent users may share a physical cloud server, our study focused on whether dynamically varying background load from such applications can interfere with the performance seen by latency-sensitive tasks. We first conducted a series of experiments on Amazon's EC2 system to quantify the CPU, disk, and network jitter and throughput fluctuations seen over a period of several days. We then turn to a laboratory-based cloud and systematically introduced different levels of background load and studied its ability to provide application isolation under different settings of the underlying resource control mechanisms. In addition to several, micro-benchmarks, we also evaluated the performance of real-world applications—the Doom 3 game server and Apple's Darwin Streaming Server—under background load.

Our EC2 experiments revealed that the CPU and disk jitter and the throughput seen by a latency-sensitive application can indeed degrade due to background load from other virtual machines. Our hypervisor experiments indicated similar jitter when sharing the CPU and we observed fair throughput when CPU allocations are capped by the hypervisor. Our experiments revealed significant disk interference, resulting in up to 75% degradation under sustained background load. We also found that network interference can be mitigated using traffic shaping tools in the hypervisor. Our application-level experiments revealed two main insights: the lack of proper disk isolation mechanisms can hurt performance, and that network isolation mechanisms in the hypervisor present a tradeoff between mean latency and metrics such as jitter and timeouts—dedicated caps yield lower average latency, while fair sharing yields lower timeouts and somewhat lower jitter due to the ability to use unused capacity from bursty background loads. Overall, our results point to the need to carefully configure the hypervisor resource control mechanisms, which can mitigate, but not totally eliminate, this interference.

Acknowledgements

We thank David Irwin, Tim Wood, and the anonymous reviewers for their comments. This research was supported in part by the National Science Foundation under Grants CNS-0720616, CNS-0720271 and EEC-0313747, and a gift from IBM. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

8. REFERENCES

[1] Amazon elastic computing cloud. `http://aws.amazon.com/ec2`, 2006.

[2] P. Barham. A fresh approach to file system quality of service. In *Proceedings of NOSSDAV'97, St. Louis, Missouri*, pages 119–128, May 1997.

[3] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), Bolton Landing, NY*, pages 164–177, October 2003.

[4] J. Blanquer, J. Bruno, M. McShea, B. Ozden, A. Silberschatz, and A. Singh. Resource management for qos in eclipse/bsd. In *Proceedings of the FreeBSD'99 Conference, Berkeley, CA*, October 1999.

[5] David Boutcher and Abhishek Chandra. Does virtualization make disk scheduling passe? In *Proceedings of 2009 Workshop on Hot Topics in Storage and File Systems (HotStorage '09)*, October 2009.

[6] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2001.

[7] S. Chen and K. Nahrstedt. Hierarchical scheduling for multiple classes of applications in connection-oriented integrated-services networks. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems, Florence, Italy*, June 1999.

[8] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Open Source Software Development Series. Prentice Hall, 2007.

[9] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfiel. Live migration of virtual machines. In *Proceedings of Usenix Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of the Symposium on Operating System Design and Implementation (OSDI'04), San Francisco, CA*, December 2004.

[11] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.

[12] Todd Deshane, Demetrios Dimatos, Gary Hamilton, Madhujith Hpuarachchi, Wenjin Hu, Michael Mccabe, and Jeanna N. Matthews. Performance isolation of a misbehaving virtual machine with xen, vmware and solaris containers.

[13] R Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of Fourth USENIX Symposium on Internet Technologies and Systems (USITS ' 03), Seattle, WA*, March 2003.

[14] K. Duda and D. Cheriton. Borrowed virtual time (bvt) scheduling: Supporting lantency-sensitive threads in a general-purpose scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC*, pages 261–276, December 1999.

[15] P. Goyal, X. Guo, and H.M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle*, pages 107–122, October 1996.

[16] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of ACM SIGCOMM'96*, pages 157–168, August 1996.

[17] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006), Melbourne, Australia*, November 2006.

[18] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP'01)*, August 2001.

[19] M B. Jones, D Rosu, and M Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France*, pages 198–211, December 1997.

[20] Abhinav Kamra, Vishal Misra, and Erich Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *International Workshop on Quality of Service (IWQoS)*, June 2004.

[21] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, and Michael McCabe. Quantifying the performance isolation properties of virtualization systems. In *Workshop on Experimental Computer Science (ExpCS '07)*, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.

[22] A Menon, A. Cox, and W Zwaenepoel. Optimizing network virtualization in xen. In *Proceedings of USENIX Annual Technical Conference 2006*, 2006.

[23] A. Molano, K. Juvva, and R. Rajkumar. Real-time file systems: Guaranteeing timing constraints for disk accesses in rt-mach. In *Proceedings of IEEE Real-time Systems Symposium*, December 1997.

[24] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference*, 2005.

[25] J. Nieh and M S. Lam. The design, implementation and evaluation of smart: A scheduler for multimedia applications. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France*, pages 184–197, December 1997.

[26] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '08)*, pages 1–10, New York, NY, USA, 2008. ACM.

[27] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report No. 376.

[28] Paul Ruth, Junghwan Rhee, Dongyan Xu, Rick Kennell, and Sebastien Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *IEEE International Conference on Autonomic Computing (ICAC)*, June 2006.

[29] Seetharami R. Seelam and Patricia J. Teller. Virtual i/o scheduler: a scheduler of schedulers for performance virtualization. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE '07)*, pages 105–115, New York, NY, USA, 2007. ACM.

[30] Solaris resource manager 1.0: Controlling system resources effectively. Sun Microsystems, Inc., http://www.sun.com/software/white-papers/wp-srm/, 1998.

[31] V Sundaram, A. Chandra, P. Goyal, P. Shenoy, J Sahni, and H Vin. Application performance in the qlinux multimedia operating system. In *Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA*, November 2000.

[32] C. Waldspurger. Memory Resource Management in VMWare ESX Server. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, December 2002.