Portland State University
PDXScholar

Spring 6-13-2014

System-wide Performance Analysis for Virtualization

Deron Eugene Jensen
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer and Systems Architecture Commons, and the Data Storage Systems Commons

Recommended Citation

System-wide Performance Analysis for Virtualization

by

Deron Eugene Jensen

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Karen Karavanic, Chair
James Hook
Charles Wright

Portland State University
2014

Abstract

With the current trend in cloud computing and virtualization, more organizations are moving their systems from a physical host to a virtual server. Although this can significantly reduce hardware, power, and administration costs, it can increase the cost of analyzing performance problems. With virtualization, there is an initial performance overhead, and as more virtual machines are added to a physical host the interference increases between various guest machines. When this interference occurs, a virtualized guest application may not perform as expected. There is little or no information to the virtual OS about the interference, and the current performance tools in the guest are unable to show this interference.

We examine the interference that has been shown in previous research, and relate that to existing tools and research in root cause analysis. We show that in virtualization there are additional layers which need to be analyzed, and design a framework to determine if degradation is occurring from an external virtualization layer. Additionally, we build a virtualization test suite with Xen and PostgreSQL and run multiple tests to create I/O interference. We show that our method can distinguish between a problem caused by interference from external systems and a problem from within the virtual guest.

i

Contents

List of Tables

List of Figures

Definitions

Hypervisor A thin kernel layer that abstracts the physical hardware and presents virtual hardware to the guests.

VMM Virtual Machine Manager. A kernel and OS that configures the hypervisor and virtual guests. In Xen this is Dom0.

Guest A complete OS (with applications) that is running under a hypervisor. In Xen this is DomU.

Virtual Resource A physical resource (such as Disk, CPU or memory) that is managed by a hypervisor and allocated to a guest.

Virtualized Changing a physical system to a virtual guest system.

Overcommit Assigning more resources than are physically available.

System-wide profiling Both the guest and VMM are profiled.

System noise Interrupts from daemons or other kernel processes that need to perform some task. [53]

Paravirtualization A virtualization technique where the native instruction set is not completely implemented. Usually the Guest OS needs to be modified to know it is virtualized. This is the technique used by Xen. [27, 42]

Working set The size of most of the data an application needs. For database servers if the working set can fit into RAM, it is much faster than going to disk and swapping data.

Overhead The additional cost (time) required for virtualization. For each operation, there may be additional resources required to virtualize the operation instead of interacting directly with the hardware.

Snapshot A complete state of the entire virtual machine saved to non-volatile disk for later use. A snapshot is used to create a template or for configuration management and roll back to a previous state.

PMU Performance Monitoring Unit. Physical hardware that counts various hardware events.

1 Introduction

Virtualization is a way to emulate hardware in order to allow multiple operating systems to run concurrently on a physical system. The hypervisor provides an abstraction and isolation of each guest to ensure protection. Virtualization allows data centers to reduce cost and power by overcommitting and sharing system resources across disparate operating systems with common hardware (Figure 1).
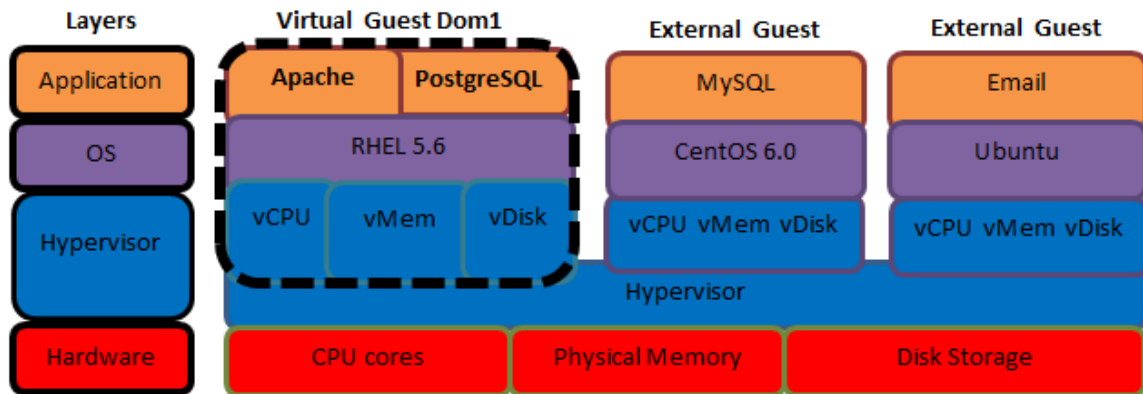
Figure 1: Virtualization Example Domain 1 is running with some external systems. The hypervisor divides, shares, and overcommits the physical resources between the 3 guest domains. Each guest has access to virtual resources and not physical hardware.

In a single server environment or HPC cluster there can be *interference* [10] or *system noise*[53] caused by complex software layers (Application, OS, and Hardware), that causes poor application performance. The hypervisor, as well as multiple external guest virtual machines, compete for system resources and add to this interference. Current performance tools in the virtual guests are unable to distinguish between performance problems in the guest machine and external interference. We have developed a method to aggregate resource counters from multiple layers of virtualization to accurately diagnose interference as the root cause of a performance bottleneck.

1.1 Motivating Example

A large corporation, with over 10,000 employees, used a suite of applications which were specifically written for this business. The software stack consisted of Redhat based Linux with 2.6 kernel with current patches, PostgreSQL 8.3.6, and Apache 2.2.3. The physical hardware had 40 CPU cores (4 sockets 10 cores each) and 256 GB of RAM with a high performance RAID SAN used for the data store. At most times during the business day the system would run with less than 60% total CPU and would cache nearly all data inside active RAM. The *working set* was less than the available memory. The system ran optimally for several months in this configuration.

At some point the customer and data center decided to virtualize the application to add disaster recovery [52] and save power [26]. After several weeks in this configuration, the end users started to complain and enter trouble tickets about slow response times. Application developers and support quickly found issues and developed fixes. Usually these fixes were to reduce the number or reads or writes to some file in the file system. However, after each fix, sometimes days later, a new problem would show up. Eventually the database engineers and system engineers began examining the recurring problems.

The symptom was that sometimes the system would go into a state where all 32 virtual CPUs [1] would run at 100% CPU time, multiple threads would go to a D state, [2] and multiple processes would fail to complete. The additional CPU time in the virtualized guest was shown to be in *System Time* and multiple tools confirmed this. Memory and I/O used on the guest and host server were consistent, and engineers monitoring the SAN did not see any latencies or problems measuring disk IO. Sometimes this would only last a few seconds and sometimes it would last for hours. Tracing system calls with *strace* showed

---

[1] The virtualization platform was limited to 32 virtual cores

[2] According to the man page for 'ps': A process in the D state is neither running nor sleeping. It is "Uninterruptible sleep (usually IO)"

some process would wait for several seconds on *open*, *lseek*, and *write* (Figure 2).

```
12:46:02 open("", O_WRONLY|O_CREAT|O_
12:46:16 fstat64(61, {st_mode=S_IFREG|
12:46:16 lseek(61, 0, SEEK_CUR) = 0
12:46:24 write(61, "O:6_"..., 8192) =
12:46:24 write(61, "R1339 "..., 8192)
12:46:24 write(61, "ct "..., 8192) = 8
12:46:31 write(61, "cription "..., 819
12:46:31 write(61, ";s:11 "..., 8192)
12:46:31 write(61, "7:for "..., 3288)
12:46:31 close(61) = 0
```

Figure 2: *strace* shows system calls to open, lseek, and write take up to 14 seconds to complete

Data center engineers examined the statistics in the host VMM, which showed exactly what was noticed on the guest: High CPU usage. However, from the point of view of the VMM, there was no information about the guest applications or kernel. Furthermore, the VMM could not distinguish between guest kernel time and guest user time. The problem could not be confirmed to be caused by virtualization, but no symptoms appeared until the application suite was virtualized. The guest was unable to perform profiling with hardware counters on this version of the hypervisor [44] to see if there was a kernel bug. Was there a bug or misconfiguration in the application, guest OS, hypervisor, or hardware? Or was there just too much overhead from virtualization during peak usage?

The temporary solution was to reboot the guest machine periodically, which would clear the working cache memory, and possibly have other unknown results. After several weeks of this, enough of the load was reduced in the application to prevent this strange interference. At the time of this writing, it is still unknown why this occurred when the system appeared to have plenty of physical and virtual resources available. There may have been interference from virtualization or a defect in the guest kernel or some combination that was not detectable.

Similar issues have been seen at large scale streaming applications hosted in massive

3

public clouds. It is argued that Netflix will completely shutdown and restart a virtual machine on a separate server if they can detect interference from external virtual machines [54].

1.2 Measurement Challenges

On a physical server, we can assume the hardware speed is known. For a CPU we can measure in hertz or floating point operations per second (FLOPS), and for a disk we can measure in I/O per second (IOPS) or latency. When that system is virtualized, the availability of the virtual resources are dynamic. The resource may (or may not) be available to the guest and the performance tools become nearly meaningless. Without knowledge of the availability, it is difficult to diagnose or find a root cause from a guest application. There is almost no way to distinguish between a poor running application, OS kernel issue, or external interference. From the Hypervisor view, there is no information about the guest applications or guest kernel functions, so there is no way to relate a problem to a guest application. Furthermore, the entity managing the hypervisor layer is likely a completely different organization than the entity managing the virtual guest.

Since each guest machine only uses a portion of the available resources at any given time, the total resources allocated to all guest VMs can exceed the total physical resources [2, 25, 45]. This idea of overcommitting resources is the same as preemptive multitasking, where multiple processes share a single CPU; and OS virtual memory, where the total memory available to applications exceeds the physical memory capacity. In order to maximize resources, IT data centers overcommit the resources, with the hope that multiple virtual guest machines do not need all resources concurrently. Even when the CPU and memory are not overcommitted there is still shared cache and memory bus. For network and disk I/O multiple guests usually share access to a single physical resource, which can cause interference when accessed concurrently.

4

Existing performance tools fall into three categories, but they are unable to accurately diagnose or find a root cause of performance problems of production guest systems. First, profiling tools are available on limited platforms, and require significant guest kernel and hypervisor support and coordination. Additionally, profiling is a great tool for identifying problems in test and development, but may not be available on production systems. Second, performance monitoring tools are readily available on most every operating system. They are very valuable for physical hardware but they assume that the resources are constant. The performance data used by the tools when external systems are causing interference is indistinguishable from a change in the application workload. Finally, hypervisor tools can show each guest and the physical resources, but they do not have access to the applications in the guest. There is no way to relate a problem seen in the hypervisor to specific processes in the guest or the guest kernel.

1.3 The growing importance of virtualization

Due to the cost savings and decreased physical administration overhead, data centers and businesses are changing from physical server to virtualized environments. In 2008 Gartners research found that virtualization is not just for server consolidation, but for efficient use of shared resources [34]. They attribute the growth of cloud computing to standardization of technologies, increase use of the Internet, and virtualization. Research from Ramya and Edwin show tremendous growth in Platform As A Service (PAAS) where an entire system platform is dynamically provisioned in a cloud computing service [36]. The NIST explicitly states that for PAAS the consumer does not manage or control the underlying infrastructure including networks, servers, and storage. They state that cloud systems should provide a *measured service* and resources utilization is monitored, controlled, and reported by providing transparency of the service [35]. Massive data centers are able to provide virtual systems, and manage large clusters of shared resources, for a fraction of the

5

price of building a physical server for each customer.

1.4 Contributions

This research presents a method for analyzing performance data at runtime in the guests and Virtual Machine Monitor (VMM). With this framework we can detect the I/O interference by sharing physical and virtual I/O performance statistics between multiple guests. We believe that this method can be extended to find interference from other resources. This could significantly reduce time spent troubleshooting and analyzing performance problems in virtualized environments.

Thesis statement: We can measure external interference from a guest in a virtual environment by relating virtual and physical resource usage across multiple layers.

The contributions of this thesis are:

1. **Overhead** We designed a new method to measure the overhead of virtualization.

2. **Test Suite** We developed an interference test suite that causes measurable I/O interference in a virtualized server.

3. **Interference** We designed a method to collect resource metrics at several layers and quantify the I/O interference from virtualization with minimal perturbation.

The remainder of this document is organized as follows: In section 2 we examine related work. In section 3 we define our methodology. Section 4 shows our experimental results for two different architectures. We conclude and discuss future work in section 5.

2 Related Work

A number of recent research efforts have been dedicated to analyzing the performance of applications running in a virtualized environment. This section reviews this research and the current approaches to identifying performance problems. First, we examine research that has shown interference and overhead of virtualization. Then we examine root cause analysis and the end-to-end approach of identifying problems. Finally, we review the current state of profiling and performance monitoring tools for virtualization.

2.1 Interference

Disk and network I/O is a major problem when running multiple virtual guests with I/O intensive applications such as file servers and database servers. Research at Georgia Tech [10] analyzes the impact of combinations of I/O, CPU, and memory intensive applications running in various combinations. They conclude that when a virtualized file server and database server are placed together they result in a 25% - 65% degradation. This is in contrast to CPU application research [1, 2] which can be allocated per VM (CPU core pinning) and will operate with about 5% interference. They identify the three categories that contribute to performance issues as: virtualization type, resource management configuration, and workload profile. Furthermore, they show that CPU intensive applications can scale at a rate near optimal 1/x. For example: if 4 CPU intensive applications are running at rate R on 4 separate cores, then they will all run at about $1/4R$ capacity if placed on the same core. Disk I/O and memory intensive applications do not scale in this manner, because of both visible and invisible interference [3].

Other research in virtualization performance shows that there are 'invisible' resources such as the shared LLC cache [3]. They show that scheduling two CPU intensive VMs on the same socket but different cores will result in significant degradation. Conversely, an

7

I/O intensive application with a CPU intensive application in the same configuration will run with little interference. However, if this I/O and CPU load runs on the same CPU core, then the I/O intensive application will perform optimally, but will significantly degrade the CPU intensive application.

Ultimately, the goal of this type of research is to be able to automatically know *a priori* how a virtual machine will run in a given environment. Recently, researchers at Florida International attempted to predict performance in a virtual environment through several performance models [5]. Through on-line and off-line modeling, and training data, they are able to predict with about a 20% error rate through various linear regression models. They also use an artificial neural network statistical model which was able to predict at under 6% median error rate. In order to predict this performance, the application load rate would need to remain constant. Research from Tikotekar (et. al) at Oak Ridge National Laboratory show that it is extremely difficult to generalize a work load [19]. They show that similar HPC application benchmarks, which are both CPU bound, can have significantly different results when paired with other applications. They stress the fact that performance isolation is extremely difficult.

Researchers at IBM [25] try to minimize I/O interference by introducing vIOMMU. The idea is to give the guest machine access to the IOMMU through virtualization of the IOMMU. They show that they can use a *sidecore* (a dedicated core for I/O) and can map and unmap memory for the guest machines, and still provide protection from device drivers in the guest.

2.2 Overhead

Cherkesova (et. al) show that they can measure and quantify the CPU overhead for moving from a physical to virtual machine. Additionally, they formulate a method for charging a machine for excessive I/O which ends up causing CPU contention on an unrelated VM

8

[6]. They measure overhead by monitoring memory page exchanges between Dom0 and DomU. Research at VMware also show this contention, and formulate that it is due to additional Inter Processor Interrupts (IPI) [24]. They say that for a high I/O it can cause a high CPU overhead due to handling of all the interrupts. They attempt to dynamically increase the rate of interrupt coalescing based on the number of I/O requests from the quests.

Other research attempts to quantify the overhead of virtualization by measuring the performance drop of benchmarks executing inside physical server and a virtualized environment. Huber (et. al) [1] attempt to measure the overhead when moving from a physical to virtual environment. They run multiple benchmarks with native hardware and when it is virtualized and calculate the performance drop between the two environments. At VMware [45] they show an increase in the TLB miss is the biggest issue between native hardware performance and virtualization. They state that I/O needs to be processed twice; once in the guest and once in the hypervisor. They show that for almost every resource and workload they can achieve *near native* performance except for very high I/O workloads.

Most of this research uses *offline* modeling (a benchmark without virtualization). Researchers at Intel [3] suggested an *online* modeling approach for CPU cores. They monitor both the CPU core utilization and the maximum CPU core utilization when running alone as a guest. They use this as a baseline to predict the performance of the guest VM when run with other systems.

2.3   Root Cause Analysis

There has been research to find a *root cause* of application degradation which could identify the layer of the problem in traditional [7] and high-performance [8] systems. This research shows the importance of analyzing external parts of the system and their impacts on performance. It has been shown that analysis needs to be competed from a full

9

end-to-end perspective of the system [15]. Gupta [17] also urges that we that there should be transparency of each layer and each layer should be charged performance points. The research also argues that upon changes in workload the application should inform systems about the current needs, in the hope that the system can accommodate those needs. An alternate argument could be made that the system should inform the layer above about the availability of the resources.

2.4   Profiling

Profiling allows a user or system administrator to see which functions are being called the most frequently or consume the most time. Oprofile [18] is a profiler that can profile both application and kernel functions, by registering with the Performance Monitoring Unit (PMU) on the CPU. The PMU will track hardware counters and will notify the registered profiler when a specific hardware counter reaches a threshold set by the profiler [11]. When Oprofile receives notification, through an NMI, it will record the PC and other information about the system, and over time will have the most frequently called function when the specified hardware event occurred. Some typical hardware event counters are: TLB miss, LLC cache miss, or CPU instructions retired. Other profilers like OSprof use latency distribution analysis [7]. As function calls are entered and exited to or from the stack, the profiler tracks the start and end time of the functions.

Research at VMware [44] shows the importance and challenges of implementing hardware counters on the PMU for analyzing interference from a virtual guest perspective. The difficulties with running a profiler like Oprofile in a virtualized guest is that the hardware counters in the guest may not be available for all hypervisors [45]. Additionally, the guest is not guaranteed to receive interrupts in a correct order. Tracking the wall-clock time and CPU time on virtual machines is extremely difficult and there are many additions in the hypervisor to help guest machines accurately track time. On many systems, kernel

10

level profiling is not enabled. A patch or configuration change is needed to enable profiling, which makes it difficult on production systems. In the Xen environment, Xenoprof allows the virtual guest to read hardware performance counters from the guest OS perspective [4, 43]. However, Xenoprof does not analyze the results or draw any conclusions about the counters.

In virtualization there are 2 types of profiling: Guest-wide profilers only profile a single guest (or domain), and System-wide which monitors all guests and the VMM [42]. In order to provide Guest-wide profiling for virtual systems, the profiler runs in the guest and the VMM needs to provide synchronous delivery of interrupts from the hardware to the guest machines. For System-wide profiling the profiler runs in the VMM and requires interaction between the guest OS and VMM. Specifically, the VMM needs to know which application and kernel functions are running on the guest machine at the time of the interrupt.

Xenoprof uses Oprofile to provide the profiling and uses Xen's hypercalls to share information between the guest and Xen Hypervisor. The Xenoprof toolkit supports system-wide coordinated profiling in a Xen environment. The underlying Oprofile uses statistical profiling across domains and can track functions running in the user space, kernel space, and in the hypervisor. Xenoprof provides a performance event interface which maps physical hardware events to virtual HW counters [20, 21]. The guest-level profiling registers with the virtual CPU, and the hypervisor registers with the physical hardware. Since it is not possible to call the domain level virtual interrupt handler synchronously, Xenoprof collects the samples for the guest domain and logs them to that specific domain buffer, and then Xenoprof sends a virtual interrupt to the guest domain.

In KVM, a Linux Kernel VM subsystem, a customized profiler can implement both guest-wide and system-wide profiling [43]. Linux perf is another profiler that runs in the Host Linux on KVM, and can measure software and hardware event counters as well as tracing. It is aware of the memory layout of the guest and can monitor the Kernel space of

11

the guest VM, but not its user-space application.

Profiling results also require manual inspection and may not show how interference from external layers impact the guest machine [7, 8]. Additionally, the user running the profiler needs to have some "guess" or prior knowledge about what event to sample, and the desired frequency of the sample. Should the user sample cache miss, TLB miss, or some other counter? As more events are profiled, the more perturbation is encountered. In a virtual environment a user may need to start several profilers on several virtual machines each monitoring various hardware counters, system state, and applications. Only after several runs in many different configurations and manual inspection could an advanced administrator make some determination as to the root cause of the problem.

2.5 Performance Monitoring

Modern operating systems have a method of monitoring the current performance of the system. User tools use statistics tracked in the kernel to monitor access to resources, and are not (necessarily) related to the PMU on the CPU. For example a specific CPU may track hardware events such as L2 cache miss, or TLB cache miss, but the kernel will track time spent reading or writing to a physical disk, or virtual memory cache information. User tools aggregate and relate these statistics to provide the user some inference about the system.

In a Linux OS there are administration tools such as top, iostat, ps, and vmstat. Windows uses *Performance Monitor* or *Resource Monitor* to show the current performance. Most kernels will write statistics when each resource is used, and these tools read and analyze the kernel data about resource usage, through the method provided by the kernel. By examining these counters and aggregating the data with tools, an administrator can see the current threads and how each of the resources are being used.

When a system is virtualized, the guest OS kernel only tracks the statistics from its own viewpoint. In other words, the resource statistics are for the virtualized resource, and

12

not the physical resource. The counters do not collect information from external guests or show the interference from external guests or the hypervisor. The host system kernel will track all the guests, but is unable to identify which process in the guest may be degraded. In order to accurately analyze performance issues on complex virtualized systems, we need to collect and analyze resource usage from all systems and the hypervisor. We need the ability to perform System-wide performance monitoring similar to the System-wide profiling available.

2.5.1 Linux Tools

On Linux the kernel resource counters are tracked in the /proc file system, which is a dynamic view into the kernel data structures and statistics [48]. For example */proc/stat* contains statistics about the time (since boot) that the the CPU was idle, running in user mode, or running in kernel mode. By running a test load and examining the /proc file system, one can reasonably determine the resources used and which process or thread is using them. Without virtualization, these tools can quickly narrow the scope of possible problems for a degraded system.

```
top - 15:57:05 up 109 days,  6 users, load average: 0.12, 0.17, 0.11
Tasks: 164 total,   1 running, 163 sleeping,   0 stopped,   0 zombie
Cpu(s):  6.0%us,  0.7%sy, 93.3%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  2062348k total, 1868648k used,  193700k free, 185776k buffers
Swap: 2095100k total,  389996k used, 1705104k free, 699236k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  999 root      20   0 94876  52m 4900 S  4.0  2.6 373:01.16 Xorg
 5362 deron     20   0  317m  92m  62m S  2.0  4.6 300:20.40 soffice
25333 deron     20   0  140m  15m 5684 S  1.0  0.8   3:33.53 gnome
 1859 deron     20   0  166m 5820 4028 S  0.3  0.3   3:15.13 metacity
```

Figure 3: *top* collects information from Linux kernel through *proc* and displays information about CPU, Memory, and processes

13

2.5.2 Windows Tools

On Windows OS the counters are similar and are tracked by the Windows kernel about processes and threads. The user space API [51] is used by the user application to display graph or report. The GUI administration tool can display any number of statistics about the system. Similar to the /proc file system in Linux there is little or no information provided about the overhead and interference from external layers.
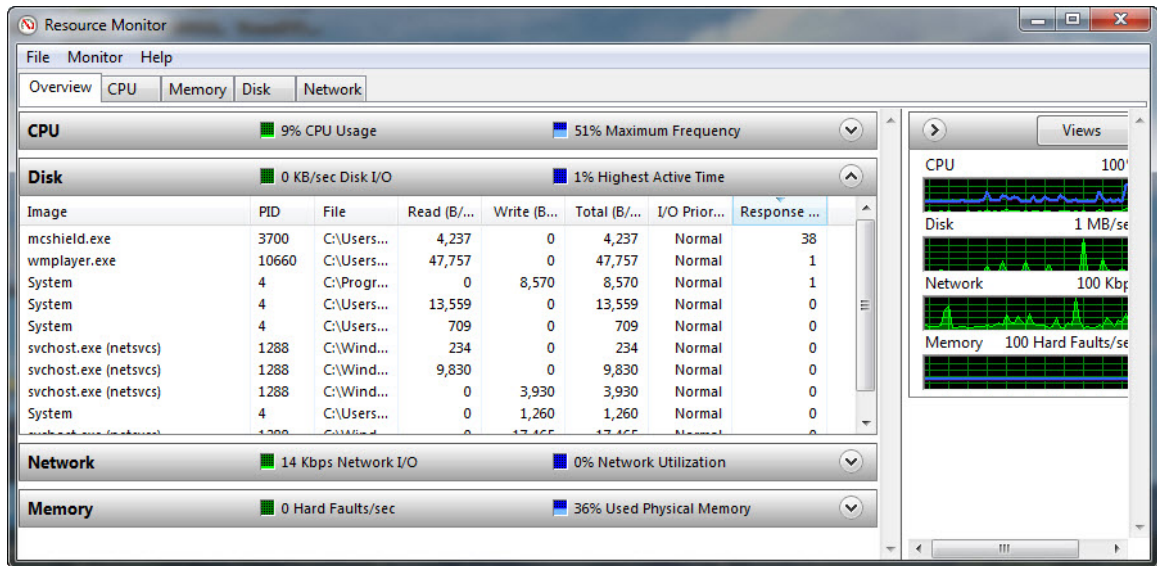
Figure 4: Microsoft Windows Resource Monitor. Reads data from Winperf API tracked in Windows kernel

2.5.3 Hypervisor Tools

Similar tools have been created in the hypervisor, *xentop* for Xen and *esxtop* for VMware, to show each virtual machine running and the resources that the entire machine is using. These tools use the same information from the *proc* file system as VMware and Xen use a Linux based kernel. From this view we can see the guest machines, and the hypervisor, but we can't easily determine how they impact each other. There is also no way to relate a virtual process running in the guest machine from the hypervisor tools. The hypervisor

14

has no access to the guest information. Even if the these could show everything running in the guest machines, the administrator for the hypervisor and guest machines are different people or even completely different organizations.

```
xentop - 17:07:00   Xen 4.2.2-23.el6
5 domains: 3 running, 2 blocked, 0 paused, 0 crashed, 0 dying,
Mem: 4182752k total, 4182236k used, 516k free    CPUs: 4 @ 2133MHz
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k)
  Domain-0 -----r      78969   16.9    2035712   48.7    2097152
  Test_VM1 -----r       6474   11.5     524288   12.5     524288
  Test_VM2 -----r       4642   12.7     524288   12.5     524288
  Test_VM3 --b---       4210    0.1     524288   12.5     524288
  Test_VM4 --b---       4137    0.1     524288   12.5     524288
```

Figure 5: *xentop* 4 Guest domains. Hypervisor does not get information about guest applications.

2.5.4   Steal Time - Complete View

Due to the fact that the hypervisor has one view of the system, and the virtualized guests have a different view of the system, the Xen Hypervisor has added a *steal time* statistic and made it available to the guest operating systems. New Linux distributions have implemented this new kernel option as *steal: involuntary wait* [48]. Additionally, administrative tools top, sar, and vmstat have all implemented this new counter. On most systems top will show as '0.0%st' (Figure 3), but if it is running on a Xen Hypervisor the paravirtualized guest will communicate with the hypervisor and collect information. Amazon cloud uses Xen and this has become a valuable tool for guests to determine if they are having performance problems from external interference [54]. We did not know about the CPU *steal time* when we started this work, but it has shown to be a valuable metric for applications running in the cloud. Our work is different from this in that *steal time* is a specific implementation for the CPU resource on a Xen hypervisor with a Linux Guest. Our method is a general approach that can be used on other platforms. More importantly,

15

I/O and not CPU time is the focus of our approach.

3 Methodology

In this section we present a framework to collect and analyze data needed for performance diagnosis from the point of view of a virtual guest system. First, we define the layers of abstraction and virtual resources unique to virtual environments. Then we identify the additional information needed from these layers, and a method to collect the data. Finally, we describe a method to analyze the additional data to determine if the guest machine is experiencing external I/O interference.

In order to accurately determine if interference is due to external interference, we need to first calculate the overhead and establish a known baseline for the resources in that configuration. Then, when run in production with other systems, we can compare the latency and throughput of the resources at the guest and hypervisor layers.

3.1 Abstraction Layers and Resources

On a single physical server, the application, OS, and hardware layers need to be considered as a potential problem for application performance problems. The resources are physical hardware such as a disk, memory (RAM), and the CPU cores (Figure 6). Applications access the physical resources through the kernel, and the kernel tracks statistics about the usage of the resource. The availability of the physical resource is known or meets some sort of guarantee from the hardware.
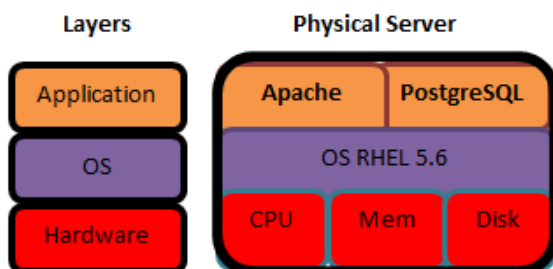
Figure 6: Layers in physical server. OS kernel tracks statistics about physical hardware.

17

In a virtual environment, the OS layer in the guest virtual machine does not have direct access to the physical resources. Instead the hypervisor divides the physical resources between the guest systems and presents a virtual resource to the guest OS (Figure 7). This is similar to an OS kernel dividing up the CPU time between multiple processes, where each process only runs for a portion of the total CPU time. The major difference between an OS kernel dividing up a physical resources, and the hypervisor dividing up a resources is that the OS layer provides APIs to the application layer in order to monitor the resources. By default (on Linux and Windows) any process can determine the percentage of resources that it is using. Unlike the OS kernel layer, the hypervisor layer provides little or no information to the layer above it about the true availability and use of the resource.
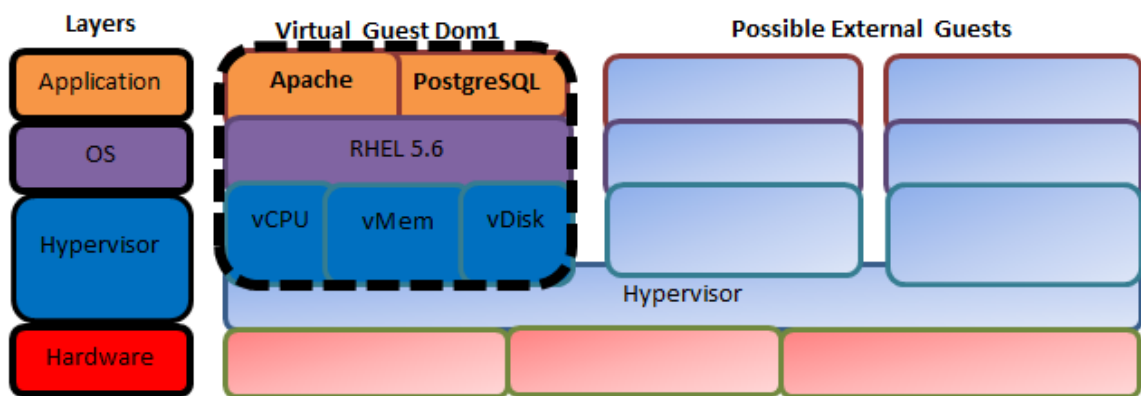
Figure 7: New layer **Hypervisor** should share information with guest OS about physical resources. The guest OS kernel now tracks statistics about the virtual resources.

The physical resource may be used by an external virtual machine or by the hypervisor, and there is little information that a guest OS or application can see about the availability of the physical resource. Furthermore, the hypervisor view does not know about specific processes running in the guest. If the VMM performance tools could identify the resource availability, there is not any way to relate that to a specific guest process. Both the virtual guest administrator and hypervisor administrator would need to monitor resources concurrently and share infomration through an external channel. Our method collects and

18

relates the usage of resources at both the guest and hypervisor layers to provide immediate information about the usage of the resource.

Resource	Definition
vCPU	The virtual core allocated to the guest
vMemory	The virtual RAM allocated to the guest
vDisk I/O	The virtual I/O block device allocated to the guest

Table 1: Virtual resources which may experience interference from hypervisor or external guest.

3.2 Performance statistics

To monitor applications and the kernel, administration tools will read kernel statistics over some period of time. Then the tool will aggregate and relate the data for that time period. In some cases, additional inference can be made from different parts of the data. For example the *sar* utility can show the average wait time, average service time, and bandwidth utilization for I/O. On a physical server, where the hardware availability is deterministic, this information is valuable for analyzing and tuning applications. However, when the resource is virtualized, these kernel resource statistics for virtual resources are almost meaningless, because the availability of the physical resource is not known. Our tests will show that for disk I/O statistics, there is little difference in these counters when comparing application load changes and interference from external virtual guests.

$interval \leftarrow 5$
$stat \leftarrow$ DiskRead
$pre \leftarrow$ READ $stat$
loop
  SLEEP $interval$
  $post \leftarrow$ READ $stat$
  $result \leftarrow (post - pre)/interval$
  PRINT $result$
  $pre \leftarrow post$
end loop

Figure 8: Example to display reads per second *reads/s* every 5 seconds.

19

Our method will collect, aggregate, and analyze the statistics kept by the guest OS and hypervisor layers in order to measure interference from external domains and the hypervisor. We find that there are statistics that will show the interference from external machines, when collected from all layers. For our examples we choose disk read and virtual memory statistics to analyze I/O read bottlenecks. However, this method could be used for identifying other types of performance problems if the data was available to the guest.

(a) Virtual memory paging performance counters [50]

pgpgin	count of memory pages in.
pgpgout	count of memory pages out.
pgfault	count minor (or soft) page faults.

(b) I/O read performance counters [47]

r_sectors	number of sectors read successfully.
r_ms	number of milliseconds spent by all reads.
r_total	number of reads completed successfully.

Table 2: Statistics collected from all guests and hypervisor

By collecting these statistics at two points in time as in Figure 8, we can infer other data such as the throughput in reads per second *reads/s*. We can also calculate the average wait time for reads *AvgRdWait*.

$$AvgRdWait = \frac{r\_ms}{r\_total} \tag{1}$$

$$reads/s = \frac{r\_total}{TotalTime} \tag{2}$$

3.3   I/O Virtualization Overhead

For each virtual resource (Table 1) there is a performance cost to making that resource virtualized. If the guest had direct access to the hardware, at all times, the virtualization

20

cost would be zero. Since most system calls from the guest kernel need to go through the hypervisor, we need to account for this additional time.

Several researchers [6, 1] have called this cost the *overhead*, and have quantified the overhead for a given configuration. This previous research used an off-line modeling technique by running a benchmark with and without virtualization. By calculating the percent difference of these two benchmarks they can calculate the overhead. The problem with this technique is that physical servers would need to be provisioned for this exercise. Any configuration changes in hardware or anywhere in the software stack, may require a new test.

Our method uses a similar approach, but does not require physical hardware for each guest. We calculate the percent difference between the counters for the physical resources in the hypervisor and the virtual resources in the guest. We claim that the time spent waiting for an operation to complete in one layer depends on the layer below. In a physical server the application depends on the kernel and the kernel depends on the hardware. Our overhead calculation finds this additional time for I/O virtualization, by using the *AvgRdWait* statistic in both the guest and hypervisor without interference to calculate the overhead.

We need to calculate the overhead before running a guest virtual machine in production. In data centers and cloud systems, templates are usually created before virtual machines are used in production. A template is a complete snapshot image of a virtual machine that has been built and tested to meet some need. For example a Redhat 6.2 system with an Apache web server may need to be used on several machines. A system could be built, tuned, and tested for that environment and then made into a template. Future users can deploy a new virtual machine from that template. We suggest to calculate the overhead from virtualization before it is made into a template.

To calculate the virtualization overhead, create a single virtual machine with dedicated

21

resources on isolated hardware. There are many performance benchmarks that can be used [16, 19, 39] to place a desired load on the virtual machine. Then run the benchmark or load on the virtual machine and begin monitoring the performance statistics for that resource using the algorithm in Figure 8 in both the guest OS and hypervisor. Save the results of the guest, hypervisor, and time in the guest as this is the baseline statistic without external interference. It is important that the guest and hypervisor collect *pre* and *post* statistics as close to the same time as possible.

Since we are measuring from the perspective of the guest OS, we want to discover how the guest degrades in comparison to the hypervisor. For some statistics such as the *AvgRdWait*, we can approximate the overhead of virtualization, by calculating the average wait time in the hypervisor $AvgRdWait_H$ and the guest $AvgRdWait_G$.

$$Overhead_{IO} = \frac{AvgRdWait_G - AvgRdWait_H}{AvgRdWait_H} \tag{3}$$

3.4 I/O Virtualization Interference

We should be able to distinguish between a guest that is degraded because the hypervisor layer (including external interference) is using the resource and a guest that is degraded because of an application or OS layer issue within the guest. In other words we should be able to distinguish between a problem inside the guest and a problem where the guest has no control. An secondary goal is to measure or analyze the interference. We should be able to answer the question, "How much of the resource is available to the guest?".

It may seem possible to that we can determine degradation by examining only the resource statistics inside of the guest machine. However, with some simple tests, we were able to change the I/O wait and throughput in a guest domain by only changing the guest workload and data size (Table 3).

When an application is running in the virtual guest with interference, the resource may

22

Test	AvgRdWait	reads/s
Baseline	5.96	376
Increase DB size	8.04	563
Increase DB Connections	20.3	875
Decrease DB Connections	4.94	144
Baseline with Interference	9.86	242

Table 3: Statistics from only the guest view of virtual resources. The first 4 tests are without interference from external domains.

not be available to the guest for some time. From this information (Table 3), it is difficult to determine whether the changes are due to external interference or application changes. We can increase or decrease the $reads/ms$ simply by changing the number of connections to the database. In our experimental results we show how or calculation for interference can distinguish these cases.

In order to immediately determine if the guest domain is experiencing interference we need to also examine the counters in the hypervisor. We can examine the throughput of the guest $read/s_G$ and the throughput of the hypervisor $read/s_H$ over the same time period calculate the interference as:

$$Interference_{RPS} = \frac{read/s_H - read/s_G}{read/s_H} \tag{4}$$

By analyzing the throughput from both the guest view and hypervisor view, we can help determine if a guest application is degraded due to external interference from external domains.

There is a case when this could erroneously tell us there is interference when there is not any interference. So far we have only examined systems where the application is running at 100% of possible throughput. One reason virtualization is successful is that not all guests run at full capacity at all times. In a theoretically perfect virtualization, a physical resource R would be divided between n guests, and each guest would use $1/n$ of resource

23
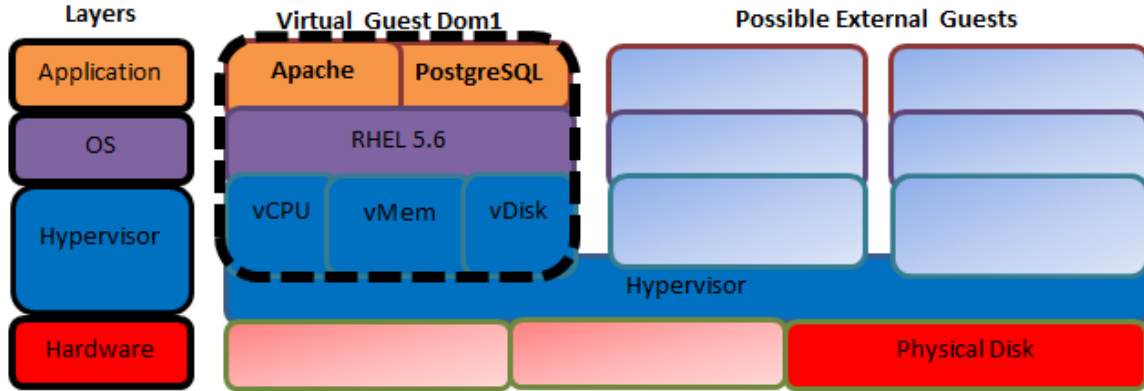
Figure 9: By passing the performance data from the physical resource to the guest Dom1 machine, the guest can determine if the interference is from an external layer.

R at all time. For example, if a physical disk could handle a throughput rate of 400 reads per second, and 4 virtual machines each requested data at a rate of 100 reads per second. In this case, our method may inaccurately report 75% interference.

Since the guest has previously calculated the read wait time through the hypervisor *AvgRdWait*, we can use that information to determine if the guest is degraded due to interference from external systems. When external systems are exceeding the I/O throughput, the wait time time will increase significantly. We can calculate the interference by using the average wait time (without interference) and the average wait time when run with additional external domains. Since we are trying to determine if the hypervisor layer is the root cause, we need to use the information from the view of the hypervisor.

$$Interference_{ARW} = \frac{AvgRdWait_{Current} - AvgRdWait_{Overhead}}{AvgRdWait_{Current}} \tag{5}$$

If both of the calculated interference results are positive then we can assume that the hypervisor is waiting for I/O, and there is something in the external layer that is generating some of the I/O throughput. Only when both of these values increase, can we accurately determine there is external interference. We found that the best calculation for interference from I/O reads is the lower of these two values.

24

3.5 Hypervisor Interference

Another option we explored was not consistent between workloads and system architectures. The idea is to calculate the $Overhead_{IO}$ again during production when calculating interference. Instead of only using one guest we can calculate the total time spent waiting in all guests, and compare that to the hypervisor just as we did when calculating the overhead without interference. We call this calculation the $Overhead_{VALL}$. Since for one guest, the time in that guest was always greater than the time in the hypervisor, we assume that the total time for all guests would increase. The problem was that for some tests the hypervisor would generate extra reads and take more time than the guests. We believe that the $Overhead_{VALL}$ may be a useful statistic for other resources.

3.6 I/O Read Example Method

First, the guest machine needs to calculate the $Overhead_{IO}$ and save the $AvgRdWait_H$ for later reference. When the guest machine is put into production, it can calculate the interference when requested from a user space tool. Both the guest virtual machine and hypervisor can calculate the resource statistics according to algorithm in Figure 8 when requested. It is important that they collect *pre* and *post* statistics concurrently. Then the hypervisor can return the *AvgRdWait* and the *reads/s* back to the guest. Then the guest VM can calculate $Interference_{RPS}$ and $Interference_{ARW}$ as described previously. The guest can report interference as the least of the two calculated interference measurements.

$Interference_{EXT} \leftarrow 0$
if $Interference_{RPS} > 0$ **and** $Interference_{ARW} > 0$ **then**
    $Interference_{EXT} \leftarrow FLOOR(Interference_{RPS}, Interference_{ARW})$
end if
Figure 10: User tool for guest domain to calculate external read I/O interference.

The user space tool *iostat* reads disk performance counters in /proc/diskstats and will report transfers, bytes read, and bytes written per second. If an application was

25

experiencing I/O performance problems this would be a tool an administrator or application developer may monitor. Without knowing the interference this could be misleading as to the root cause of the problem. The following example shows a possible output from the perspective of the running guest when experiencing I/O interference from external guest machines.

```
Device: tps    kB_read/s    kB_wrtn/s
sda   577.20     41388.00     148073.00
  I/O interference 22.4%
```

Figure 11: Example: Possible *iostat* output in a virtual guest experiencing external interference. This is similar to the new *steal time* for the CPU resource.

3.7   Other Considerations

There are three other issues that need to be considered with this type of method. First, we examine the methods of passing information between the guests and hypervisor. Second, we see if there are any performance issues with the method we describe. Finally, we look at security concerns since passing information about external machines may reduce the security guarantees when a system is virtualized.

Most virtualization platforms have a message passing API native to the system. Usually this is to automatically configure the guest domain when it is provisioned or when changes are needed. For Xen Server, this is done through the Xenbus and Xenstore. VMware guest tools has a guest API (vmware-tools-guestlib package), and even has a setting to pass host (hypervisor) resource counter data to the guest [56]. However, this is disabled by default and security tuning documents encourage explicitly setting it off for security reasons. If there were not any native mechanism to pass data between the guest and the host system, we could still use some message passing over TCP/IP such as SOAP or Rest service with HTTP.

There is some performance overhead with this additional data collection, as with any

26

method that attempts to measure performance. We do not want a tool to negatively impact the performance, or report degradation from the tool itself. For I/O performance issues we believe that our method will not have a measurable impact. For a system that is waiting on I/O from disk, it should have CPU cycles available for data processing. Both the guest and the hypervisor need to read in memory kernel data twice. Each layer needs to compute the difference of the two statistics after they are collected. The hypervisor needs to pass the data to the guest VM, and then the guest needs to calculate the interference. There are several tools in Windows and Linux that perform similar to this now (other than passing information), and are widely accepted.

Obviously, security is a major concern for a virtual machine that may be running concurrently with untrusted external systems. The more information an attacker can gain about the system, the better chance that a vulnerability can be found. There is similar precedence for sending this type of data in a standard kernel. Currently (by default) all users can see the sum of the I/O and CPU processes. The OS kernel has the responsibility to provide protection, yet it provides access to global data about the system. This is the same type of data where we want to know how our virtual machine is effected by other virtual machines. An attacker may be able to gain knowledge about the run state of external machines, and possibly execute some covert channel attacks. It may be beneficial to limit the guest to only requesting data from the hypervisor at known time intervals. More research would be needed to see what types of vulnerabilities would be possible, and what types of controls could be added to reduce the risk of an exploit.

3.8 Virtualization Test Suite

In order to test our design, two virtualization stacks, a test infrastructure, and several benchmarks were created. Our test suite runs a load on the guest machines and coordinates the stop and start of the collection of multiple performance counters across the guests and

27

hypervisor. Our experiments, when run in this infrastructure, demonstrates performance problems guest machines may experience, from external interference or changes to the application workload. The performance counters are collected from all layers of the virtualization stack and the results of the application performance are also recorded. For each experiment, we run the test at least three times and average the results.

Both virtualization platforms have the same software stack installed (Figure 4a). We use the Xen hypervisor and CentOS release for both the guest (DomU) and VMM (Dom0). This software stack can be extended to mulptiple servers to produce a larger cluster of servers. We chose PostgreSQL as the database server for our tests because of its robustness and standard use in hosting facilities and applications. It is a good general purpose application that can be used to stress I/O, memory, and CPU resources, although we focused our test suite to measure database read operations.

The test suite can run in all sizes of virtualization environments (Table 4b). Tests which run in a *Personal* or *Business* virtualization platform can also be run in a *Cluster* or *Cloud* based system by increasing the size of the database or number of guest machines until a similar load is reached. For our experiments we run on an IBM x3650 *Personal* and Dell T410 *Business* sized virtualization environments.

Software	Version
Hyperviser	Xen 4.2
Domain 0	CentOS 6.5 (Kernel 3.4)
Guest Domains	CentOS 6.4 (Kernel 2.6.39) PostgreSQL 8.4

(a) Software installed virtualization test stack

Size	Specifications
Personal	IBM x3650 Quad Core 2GB Ram
Business	Dell PowerEdge T410 dual quad-core Xeon processors, 12GB Ram.
Cluster	Multiple small or medium servers clustered together with shared SAN data store.
Cloud	Amazon Cloud or similar PAAS provider.

(b) Virtualization sizes for tests

Table 4: Software and Hardware stacks

28

The test infrastructure generates a consistent and reproducible system load by using a PostgreSQL database server with PGbench. PGbench creates a TPC-B similar style workload and calculates transactions per second (TPS), so we can track the performance of each guest system from the application layer [41]. We use PGbench to both generate a load to measure overhead and create interference. We can change the database size and the number of clients that connect to the database to change the I/O workload from the guest perspective. PGbench will put a stress load on the limits of the system, and it is a good tool for simulating guest machines that are all consuming as much resources as possible. We measure the TPS from the benchmark when run as a single virtual machine and when run with multiple machines concurrently. We compare the degradation from the application to our proposed calculation for interference.

When the test begins, each guest VM starts a process which waits for the hypervisor to signal the start of the test. The Dom0 (VMM) then starts a process which signals all the guests to start, and which benchmark to run. Each guest then collects the *pre* resource counters and begins running the specified benchmark system load. Dom0 also collects the same resource counters from its view, but does not generate a load, since in Xen all disk I/O goes through Dom0, we use this as the hypervisor layer. When each guest completes the benchmark, it reads the *post* counters and sends the difference between the two counters back to Dom0 (Figure 8). When Dom0 has collected the information from all guests, it can display the counters for itself and each guest. Executing this once with a single virtual machine, and once with multiple virtual machines we can calculate the interference.

One of the challenges of this test suite is the coordination of the starting and stopping of the test, collection of resource counters, and the information sharing between the guests and hypervisor. Since we use Xen as our virtualization platform, we chose XenBus and XenStore [55] to both coordinate the stop and start of the benchmark, and the collection of data results between domains. In order for paravirtualized guest domains to communicate

29

with the hypervisor the XenStore tools for DomU domains had to be built as this is not currently part of the standard package of guest tools. Other options were to use the TCP/IP stack to send and receive messages between the domains.

We can use the TPS to show how the database performs at the application layer when external environment changes are made to the system. Our tests will select Memory and I/O resources to monitor. For measuring disk I/O it is a good to also collect virtual memory (in this case we are talking about OS virtual memory, not virtualization) as well as I/O data. Our infrastructure collects paging performance counters and disk read counters at all layers to measure interference from I/O.

4 Experimental Results

In the following experiments we evaluate the benefits of passing additional information through hypervisor layer to the guests for detecting interference. We first validate that our test infrastructure can accurately create I/O interference and we can measure that application throughput. We find that the application throughput is dependent on the size of the test database. Then we use our method to calculate the overhead on our *Personal* size server. We run two different sets experiments in this configuration to detect interference from both external memory pressure as well as external I/O. Similarly we calculate the interference on our *Business* size server, by adding one domain at a time. These results show that our method can measure the interference from external systems and account for the performance drop in the application. Our final test shows that our method does not report interference when our application is degraded from an application workload change.

4.1 Validate Test Infrastructure

In this experiment we verify that our test framework can generate a load that degrades the application when run with external interference. We expect a significant drop in performance from running a single guest test to running multiple guests concurrently. Monitoring resource counters in only the guest domain provides little information about external systems causing performance problems.

4.1.1 Without Interference

For I/O intensive workloads, the application tends to perform very well when the entire *working set* can fit into main memory, and the OS will cache reads from disk. However, when the *working set* approaches (or exceeds) the size of main memory, the application tends to degrade quickly. Our initial experiments to find an I/O workload highlight this

31

fact by changing the database size under load in a virtual environment. Before running the entire test suite we need to create a test database that will exceed the *working set* and increase the probability that a database read will need to fetch the data from disk storage. With PGbench the "-i" flag *scaling factor* is used to initialize a database at a specified size (prior to running the benchmark) so that our results can show changes between a memory bound system and I/O bound system.

To begin this exercise, we divide the physical memory and CPUs into four equal parts and create four individual guest virtual machines (Dom1 - Dom4). Each virtual machine is given an equal share of the memory and CPU resources so that no guest virtual machine would interfere with another machine if they were on separate physical systems. We start only our Dom1 machine and run our experimental test with PGbench. On our *Business* size server, Dom1 is allocated 2GB vRAM and 2 vCPU, while on our *Personal* server Dom1 is given 512MB vRAM and 1 vCPU.

The test starts by initializing a very small database then runs PGBench and collects the TPS. Then we increase the DB size slightly and run the benchmark again. After repeating this process several times, we can determine when the DB size changes to an I/O bound system when its performance drops significantly (Figure 12). This is due to the fact that it must fetch database rows from the disk and it is much slower.
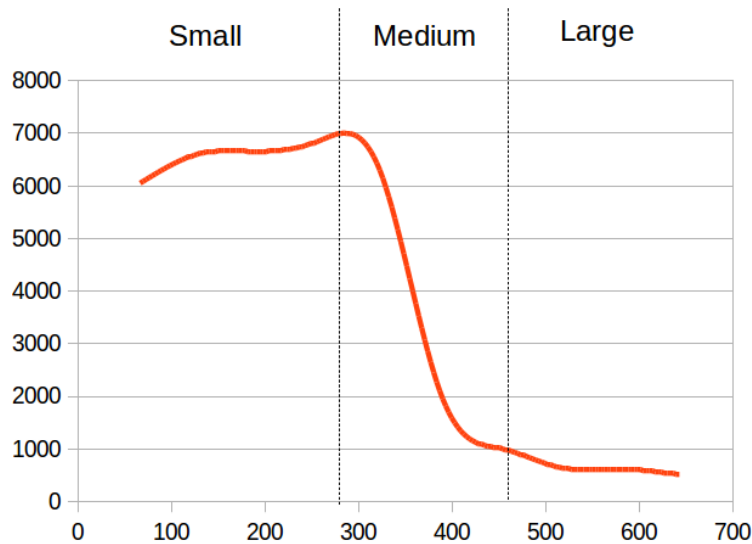
From these tests we can now see the size of the database needed on each system to generate an I/O workload.

Small DB The working set can fit into main memory. Performance is based on memory.

Medium DB The working set is swapped to disk occasionally. Performance is moving from memory to I/O.

Large DB Most reads need to go to disk. Performance is based on Disk I/O.

32

Figure 12: TPS on our *Personal* size server with 1 vCPU and 512KB vRAM. It changes from a memory bound application to an I/O bound application when the DB size approached the available RAM.

4.1.2 With Interference

We create external machine interference by running Dom1 concurrently with Dom2, Dom3, and Dom4. Each of the Dom2 - Dom4 systems are configured the same as Dom1. We create a Large 2GB database on each guest of the Dell, and a Large 1GB database on each guest of the IBM. We run PGBench in a loop to continuously create I/O interference on each external guest machine (Dom2 - Dom4). Concurrently we run our benchmark, and collect performance statistics only from Dom-1. When the system is not I/O bound (Small DB) there is about a 28% drop in performance (4434 TPS - 3208 TPS) on the *Personal* server and little change in the *Business* server (Table 5). On both servers the guest becomes an I/O bound system quicker with external interference.

Trying to analyze the performance drop in Dom1 without knowing about this external interference is difficult. There were no changes to Dom1 when run by itself and run with other guests. By using the system *vmstat* utility we can examine available memory, SwapIn and BytesIn/s to see if we can determine why the application benchmark is degraded

33

DB Size	Single	Interference	Drop
Small	4434	3208	28%
Medium	2149	216	90%
Large	260	197	24%

(a) Interference from *Personal* size server with 2GB RAM: Each Guest domain has 512MB Allocated.

DB Size	Single	Interference	Drop
Small	5772	5734	0.7%
Medium	1608	162	90%
Large	359	82	77%

(b) *Business* size server with 12GB RAM: Each Guest domain has 2GB Allocated.

Table 5: Dom1 TPS difference from interference for 3 database sizes.
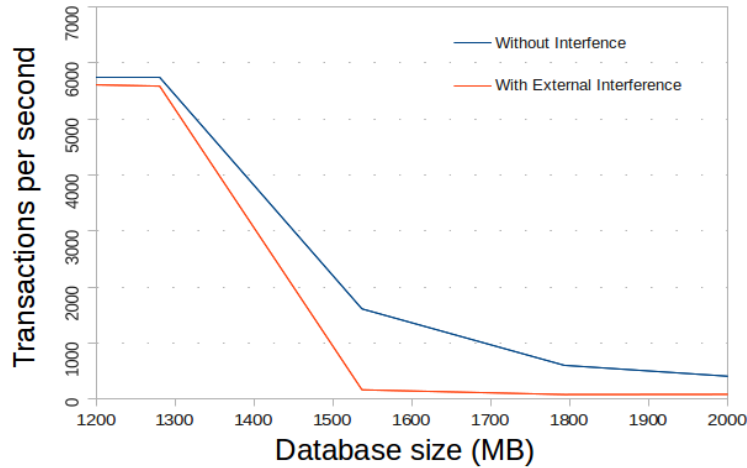
Figure 13: Application performance in transactions per second is degraded with interference.

without collecting external information (Figure 6). Memory is not used as efficiently, the system almost completely eliminates swapping data in, and also does not read as much data from disk. However, there is no indication that the problem was due to an external guest and hypervisor using those resources. A DBA looking at these numbers may conclude that kernel swap or DB tuning may fix the problem. The root cause of the performance drop is due to external interference, which is not known from examining the data available.

VMstat	Single	Multiple	Drop
SwapIn/s	1,480	85	94%
BytesIn/s	6,877	4,438	35%
CPU IOwait	92%	93%	1%

Table 6: Statistics using performance tool *vmstat* on guest Dom1 with a Large database while running alone and with external interference.

4.2 Personal Server

In the previous experiment we showed the interference that can occur when external I/O interference is applied to a guest domain, and only looking at the performance counters in the guest does not indicate the problem. In this experiment, we verify our design to measure interference from external systems, when the system is I/O bound. The results are from our IBM x3650, where each guest is configured to use 512MB of vRAM. First, we run the benchmark with a large 640MB database without interference and calculate the overhead. Then we start Dom2 - Dom4, while running the benchmark in Dom1, and calculate the interference. We compare the results of the calculated interference with the performance drop in the application. In all of these results we show the average of at least three test runs.

```
# xm list
Name                     ID    Mem VCPUs      State  Time(s)
Domain-0                  0   1988     4      r-----  88871.2
Test_VM1                  1    512     1      -b----  60187.0
Test_VM2                  2    512     1      -b----   5722.0
Test_VM3                  3    512     1      -b----   5273.0
Test_VM4                  4    512     1      -b----   5203.5
```

4.2.1 Overhead

We calculate the overhead at 4.7% using the resource counters defined in our design and Equation 3 (Table 7). The guest (Dom0) needs to wait about 5% longer than the physical writes take to complete. While calculating the overhead we also collected the application performance at 415 TPS without any external interference.

One thing to notice is that the *pgpgin* (count of pages in using */proc/vmstat*) from the Dom0 is twice as much as Dom1. We also see in Dom0 that pgpgin is exactly the same as

35

Counter	Dom0	Dom1	$Overhead_{IO}$
pgpgin	332,504	165,901	
pgfault	25,648	132,691	
r sectors	332,504	331,803	
r_ms	63,356	72,428	
r total	11,145	12,166	
reads/s	372	406	
AvgRdWait	5.68	5.95	4.7%

Table 7: Overhead on the *Personal* size server.

the sectors read, while there is exactly twice as many read sectors in Dom1. After much research reviewing the *sar* man page [3], we found that the pgpgin is reporting 1 Kbyte pages. The *fdisk -l* command showed the sector size as 512 bytes per sector. We verified this on physical hardware by examining the statistics while copying files. There were 2 sectors read for each page in. We also ran this test on only Dom0 and found the same results: There were 2 sectors read per page in. The expected results should be 2 read sectors for each page in, but when reading from Dom1, Dom0 always showed a 1:1 ratio.

After closer examination of the method used to pass reads with Xen, we found that there is an additional block device in Dom0 [57]. Although this does not map to a physical device it does explain the strange *pgpgin* between Dom0 and Dom1. In the virtual guest (Dom1), when it reads from a virtual disk it uses the blkfront driver. This communicates with the blkback driver in Dom0. There is an additional virtual disk in Dom0 used to pool requests. Although this is not a physical device it still uses the virtual memory of the kernel and pages in for Dom0. So each read sector (from physical disk) in Dom0 results in 2 pages in which is exactly what our data shows.

This additional work in Dom0 is an example of the overhead from virtualization, and partially explains the additional wait time in the guest. We measure the additional time time waiting in the guest domain to account for all of the overhead the hypervisor must do in order to virtualize the physical device.

---

[3]We also verified this by looking at the sar code and kernel code for virtual memory

36

4.2.2 Memory Interference

We repeat the previous experiment with 4 guest domains all running at the same time. Since the DB size in Dom1 is 640MB and there is 512MB of vRAM on Dom1, we can say that Dom1 is bound by disk I/O speed. We run the experiments with the other 3 guest domains running a memory bound database. We create a Small DB of 128MB in each of the external domains to generate memory interference on Dom1.

Counter	Dom0	Dom1	$Interference_{RPS}$
pgpgin	328,144	128,176	
pgfault	57,293	95,591	
r sectors	328,144	256,352	
r ms	103,483	70,751	
r total	13,365	9,547	
reads/s	446	318	28.6%
AvgRdWait	7.74	7.41	

Table 8: Calculated Interference with external interference from small database in Dom2 - Dom4. TPS dropped by 20%

We also need to calculate the increased wait time using the wait time *AvgRdWait* in the hypervisor without interference (5.68*ms*) and the new wait time in the hypervisor (7.74*ms*). We calculate $Interference_{ARW} = 26.6\%$ (Equation 5). Since this is slightly less than the interference from reads per second (28.6%) we calculate the $Interference_{EXT} = 26.6\%$.

From these tests, Dom1 had 331 TPS, and was degraded from 415 TPS, a performance drop in the application of 20%. It is not shown in these results, but Dom2 - Dom4 was able to cache the entire working set in memory and did not issue any read requests. Dom1 was the only domain to issue read request, but there was additional read time in the guests and hypervisor.

4.2.3 I/O Interference

Now we create I/O interference in the three guest domains by creating a 640MB Large DB in Dom2 - Dom4. Since each guest has 512MB vRAM, this should cause significant I/O

37

contention in Dom1.

Counter	Dom0	Dom1	$Interference_{RPS}$
pgpgin	549,419	97,092	
pgfault	58,201	86,765	
r sectors	549,419	194,163	
r ms	285,334	71,585	
r total	20,723	7,259	
reads/s	691	242	65%
AvgRdWait	13.8	9.86	

Table 9: Interference calculated Large 640MB DB in Dom2 - Dom4. TPS dropped by 39%

We also need to calculate the additional wait time in the hypervisor (Equation 5). Without interference the read wait time is 5.68ms and with interference it is 13.8ms. We calculate $Interference_{ARW} = 58.7\%$ Since this is less than the interference from throughput we calculate the $Interference_{EXT} = 58.7\%$.

During these tests, the application benchmark in Dom1 decreased to 254 TPS. This is much less than when run without interference (415 TPS) and with memory interference (331 TPS). All of the read counters showed significant interference. Additionally we are showing some interference from page faults.

4.2.4 Results Analysis

In both cases (Figure 14) we calculate a higher interference than the application degrades. At a minimum this implies that there is interference to the guest application. Although the database is I/O bound there is still a percentage of the application that relies on the CPU speed which is not part of the interference.

4.3 Business Server

For this experiment we use our Dell T410 server with 12GB of physical RAM. We configure all Dom1 to use 4GB of vRAM and configure Dom2 - Dom4 as noted below.

38
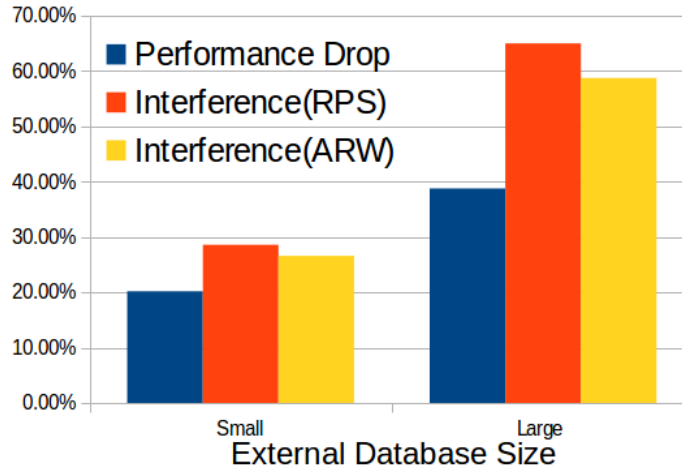
Figure 14: Application performance drop compared to calculated interference, for two types of external interference.

We assign some different CPUs to the guest machines. We create a medium database of 3.2GB in Dom1. And create Medium and Large databases in the external guest domains create different workloads in the guests.

Name	ID	Mem	VCPUs	State	Time(s)
Domain-0	0	872	16	r-----	27116.4
TestVM1	1	4096	1	-b----	1127.7
TestVM2	2	3072	2	-b----	1735.8
TestVM3	3	2048	1	-b----	1588.1
TestVM4	4	2048	2	-b----	1906.1

4.3.1   Overhead

We calculate the $Overhead_{IO}$ in Dom1 using equation 3. The overhead from virtualization on this platform (2.4%). When comparing the these results on this size platform to the overhead in the previous size platform 7 we can see that our reads/s decreased, but our

39

database throughput is faster at 1,348 TPS. In this server the database size is slightly smaller than the available memory (Table 10).

Counter	Dom0	Dom1	$Overhead_{IO}$
reads/s	275	281	
AvgRdWait	13.6	13.9	2.4%

Table 10: Overhead on the *Business* size server.

4.3.2 Interference

In the previous experiment for calculating interference we used 3 external guest domains, and changed the workload in the guest domains. In this experiment, we create interference adding 1 external domain at a time. The first result is a single guest domain Dom1 without external interference. Then we run Dom1 with Dom2 and calculate the interference with one external domain. We continue to add guest domains and calculate the interference and application performance drop in TPS (Table 11).

| | TPS | reads/s | reads/s | AvgRdWait | $Inter_{RPS}$ | $Inter_{AWR}$ |
Experiment	Dom 1	Dom0	Dom1	Dom0	$Inter_{EXT}$	
Dom1 (Only)	1,348	275	281	13.6	-2.1%	0.0%
Dom1 + 1 guest	703	214	137	33.2	35.8%	56.0%
Dom1 + 2 guests	543	246	108	44.0	56.1%	67.1%
Dom1 + 3 guests	378	259	76	59.5	70.6%	77.2%

**Table 11:** Interference generated from 0, 1, 2, and 3 external guest domains.

By examining these results as more external domains are added, the application TPS performance degrades as expected. We are also able to measure the interference which increases as more domains are added. The interference increases similar to the rate at which domains are added and the performance drop.

4.3.3 Results Analysis

First we examine the results of three tests in each configuration with external interference. In this test we can see how the calculated interference relates to the performance drop in

40

each experiment. As more domains are added, the application performance drop increases similar to the measured interference (Figure 15).
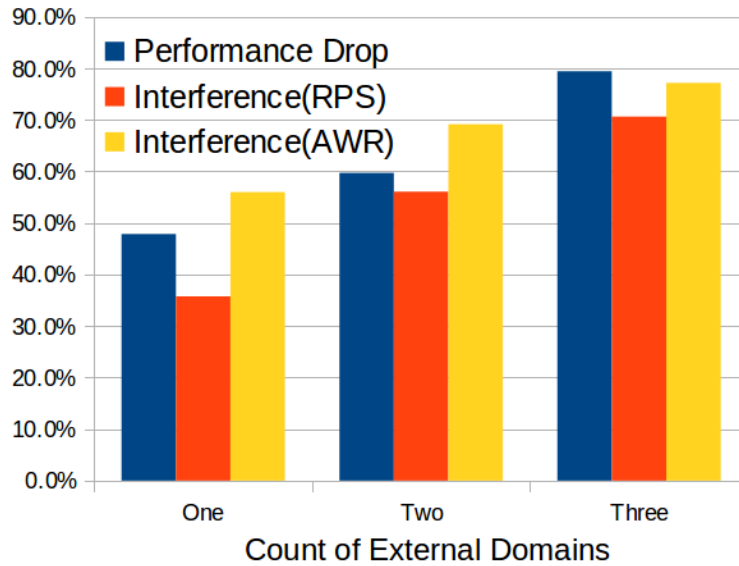
Figure 15: *Business* size application performance drop compared to calculated interference, for one, two and three external domains.

4.3.4 Statistical Analysis

In the previous experiments, we only had a sample size of 3 runs. Although each run was for 30 seconds, we need to see how each run changes. Now we run this experiment 30 times in each configuration to see how much change is occurring in each test. We receive similar results as when only with 3 tests, but we can see that the standard deviation increases in both our application measurement and measured interference as more external domains are added (Table 12) (Figure 16).

Finally, we need to show that the calculated interference relates to the application performance drop. We use a linear regression over the 90 runs. Although one and two external domains were similar, each test run would have different calculated interference

41

Experiment	TPS	reads/s	AvgRdWait	$Inter_{RPS}$
Dom1 (Only)	1,247 (55)	317 (15.9)	14.7 (0.5)	N/A
Dom1 + 1 guest	585 (28)	163 (8.3)	37.3 (2.1)	36.6% (3.3%)
Dom1 + 2 guests	537 (43)	148 (12.3)	43.1 (5.0)	40.9% (4.6%)
Dom1 + 3 guests	430 (61)	118 (16.5)	55.6 (9.7)	51.8% (5.8%)

Table 12: Mean (Standard Deviation) from 30 runs of 0, 1, 2, and 3 external guest domains.
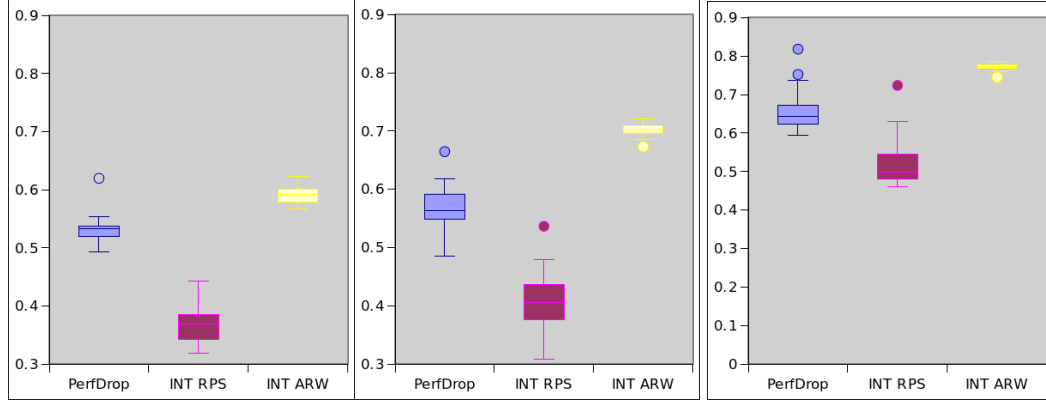
(a) 1 external domain   **(b)** 2 external domains   **(c)** 3 external domains

Figure 16: Box plot with quartiles and outliers for application performance and calculated interference for 90 test runs.

and application performance. We show that the calculated interference can explain the application performance drop (Figure).

4.4 Verification Without Interference

In the previous two experiments, we showed how our method can give a guest domain vital information when it is degraded from external interference. However, what if the guest domain was degraded due to an application bug, DB change, or misconfiguration? In that case we want our method to show that there is not any external interference.

We have previously demonstrated that the database performance can become degraded as the size of the database increases. Now we show that by simply increasing the size of the database, that our method does not report external interference. In this case the application is degraded because the working set exceeds the available memory. We also change the number of database connections to change the read wait time. In either case,

42
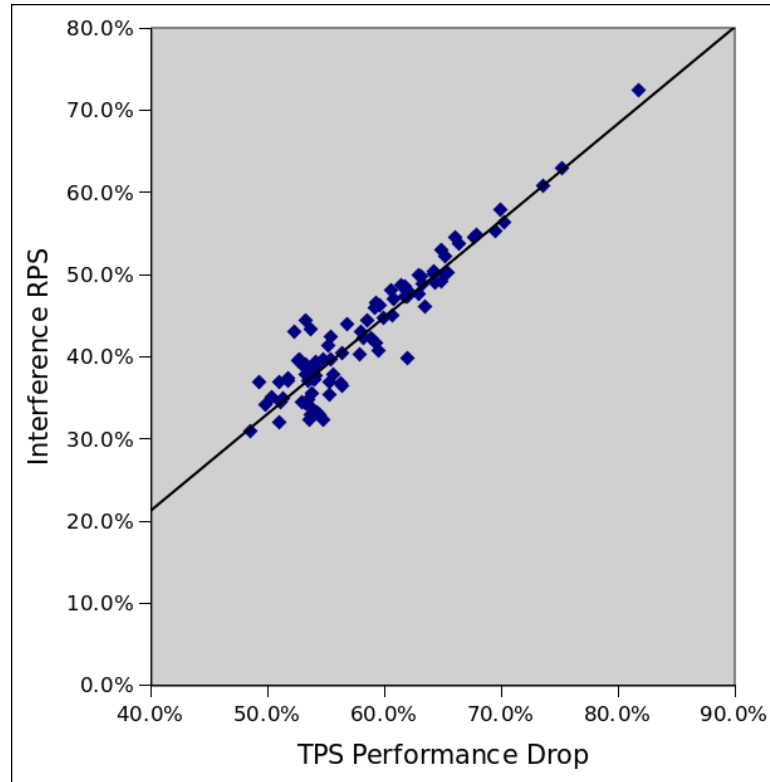
Figure 17: Linear regression of interference. $R^2 = 0.8882$

the application is not degraded due to external interference, and our method accurately

indicates there is not any external interference.

Experiment Test	reads/s Dom0 Guest	reads/s Dom1 Hypervisor	AvgRdWait Dom0 $Interference_{RPS}$	$Interfer_{RPS}$	$Interfer_{AWR}$
Baseline	372	406	5.7	-6%	0.0%
Increase DB size	538	563	7.8	-6%	26.9%
Increase DB Connections	836	875	15.4	-5%	63.0%
Decrease DB Connections	279	144	4.7	48%	-17%
With 3 ext domains	691	242	13.8	65%	59.0%

Table 13: Tests without interference, our method does not report interference.

One interesting result is that when we decreased the number of DB connections, we

experienced a significant increase in reads in the hypervisor. We ran this experiment

several times, and verified that this was accurate. For some reason the hypervisor increased

the reads (since there were not any external domains). Since our calculation determines

43

interference from external domains, it did not show interference because there was no increase in the wait time.

By analyzing the throughput from both the guest layer and hypervisor layer, we can see that our method reports interference only when external interference is applied to the guest domain. When we showed the guest view (Table 3) it was difficult to determine if the problem was from a change in the guest application or an external domain. With these tests we can see that we need to have both an increase in wait time in the hypervisor as well as additional throughput from external domains.

4.5 Results Analysis

We completed several different experiments on two different virtualization platforms. In section 4.1 we verified our test method and showed how performance can decrease without interference by increasing the size of the database. We also showed that for each database size when external guest domains run concurrently they cause interference and degrade the guest application. In section 4.2 we looked at our *Personal* size server which had 2 GB ram and 4 CPUs. We were able to calculate the interference from external domains that only used memory, and external domains that were also I/O bound. Our calculations for interference increased as the application performance decreased. In section 4.3 we used our *Business* server and calculated the overhead and interference from virtualization. We created external interference from different numbers of guest domains, and found as more external guests are added our calculated interference increased similar to the application performance drop. Finally, in section 4.4 we validated that when the application degrades due to internal changes, our method does not report it as external interference from virtualization.

44

5 Discussion and Conclusions

Our research has reviewed the current work in virtualization and the trends of data centers and cloud computing to move physical servers to virtual servers. Virtualization can reduce cost and power and improve resource utilization. With faster computers and more cores, low use applications do not need a complete physical server. Data centers can virtualize systems and easily deploy thousands of virtual servers across fewer physical servers. Both VMware and Xen provide tools to manage large clusters of physical servers each running multiple guest virtual machines.

Current research in managing these types of environments show the difficulties in identifying performance due to overhead and interference from virtualization. Most of the research is to try to find the optimal solution, given set of resources, a set of workloads, and a set of applications. However other research showed that different workloads can have severely different results, and it is difficult to generalize a workload. Assuming that a workload can (and will) change, we need to be able to identify at runtime why an application may be degraded. We must also consider the fact that the workload will change not only for the guest virtual machine in question, but also for external domain workloads.

Our research uses the end-to-end argument in research and methods of performance analysis. We identify the layers and the resources in each layer. We find that looking at both the hypervisor and the guest machine gives the user of the guest machine a significant advantage in determining the root cause of the problem. Without this it would cause a significant impact on the developer or administrator, since the user of the guest would not have access to the hypervisor. Additionally, the hypervisor does not have a view of the applications on the guest. Our experiments only use I/O performance and we only measure interference from read counters. A next step is to determine if this will work for write I/O and both read and write I/O concurrently. Similarly, we should try to analyze network

45

I/O. After that is completed we can try to determine how memory intensive workloads can measure interference, and how they effect the I/O interference. For Xen with Linux guest machines there is already a metric *steal time* that shows the CPU time used by another guest, and the current guest was waiting to use the CPU.

Another area to explore is testing this on other platforms. All of our tests were run on Xen with a Linux guest OS. We need to test on other hypervisors such as VMware ESX and Microsoft HyperV. We also need to run other types of guests since the counters in each guest could be different. We believe our method is generic and most hypervisors and guests already have the data but it may need to be collected and analyzed differently.

We also calculated the overhead of virtualization $Overhead_{IO}$ and stated that it was important, but did not go into details about this statistic. We believe this is useful for tuning the hypervisor and guest. It can also be used to compare virtualization techniques. We also found several interesting results across multiple counters above and beyond what was in this document. We found that by looking at the sum of the counters of all guests and comparing that to the hypervisor we can calculate $Overhead_{Vall}$. Often, the difference between these two values were meaningful depending on the platform and the workload. However, we were unable to draw any meaningful conclusions about this difference.

5.1 Conclusions

We need the ability to accurately provide performance counters and statistics to the guest applications based on the current availability of the physical resources. Our research provides a method to show the interference caused by external guest applications for I/O workloads by passing resource counters through the additional layers of virtualization.

References

[1] Huber, Nikolaus, et al. *Analysis of the performance-influencing factors of virtualization platforms.* On the Move to Meaningful Internet Systems, OTM 2010. Springer Berlin Heidelberg, 2010. 811-828.

[2] Huber, Nikolaus, et al. *A Method for Experimental Analysis and Modeling of Virtualization Performance Overhead.* Cloud Computing and Services Science. Springer New York, 2012. 353-370.

[3] Tickoo, Omesh, et al. *Modeling virtual machine performance: challenges and approaches.* ACM SIGMETRICS Performance Evaluation Review 37.3 (2010): 55-60.

[4] Menon, Aravind, et al. *Diagnosing performance overheads in the xen virtual machine environment.* Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. ACM, 2005.

[5] Kundu, Sajib, et al. *Application performance modeling in a virtualized environment.* High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on. IEEE, 2010.

[6] Cherkasova, Ludmila, and Rob Gardner. *Measuring CPU overhead for I/O processing in the Xen virtual machine monitor.* Proceedings of the USENIX annual technical conference. 2005.

[7] Traeger, Avishay, Ivan Deras, and Erez Zadok. *DARC: Dynamic analysis of root causes of latency distributions.* ACM SIGMETRICS Performance Evaluation Review 36.1 (2008): 277-288

[8] Knapp, Rashawn L., Karen L. Karavanic, and Douglas M. Pase. *Detecting runtime environment interference with parallel application behavior.* Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE, 2007.

[9] Kenistion, J., Panchamukhi, Prasanna, Hiramatsu, Masami. *Kprobes* `https://www.kernel.org/doc/Documentation/kprobes.txt`

[10] Paul, Indrani, Sudhakar Yalamanchili, and Lizy K. John. *Performance impact of virtual machine placement in a datacenter.* Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International. IEEE, 2012.

[11] Mucci, Philip J., et al. *PAPI: A portable interface to hardware performance counters.* Proc. Department of Defense HPCMP Users Group Conference. 1999.

[12] Mohror, Kathryn, and Karen L. Karavanic. *Towards scalable event tracing for high end systems.* High Performance Computing and Communications. Springer Berlin Heidelberg, 2007. 695-706.

[13] Kufrin, Rick. *Perfsuite: An accessible, open source performance analysis environment for linux.* 6th International Conference on Linux Clusters: The HPC Revolution. Vol. 151. 2005.

[14] Jafar, Anderson, and Abdullat (2008). *Comparison of Dynamic Web Content Processing Language Performance Under a LAMP Architecture* Journal of Information Systems Applied Research, 1 (1). `http://jisar.org/1/1/.ISSN:1946-1836.`

[15] Saltzer, Jerome H., David P. Reed, and David D. Clark. *End-to-end arguments in system design.* ACM Transactions on Computer Systems (TOCS) 2.4 (1984): 277-288.

[16] Katcher, Jeffrey. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

48

[17] Gupta, Vishakha, Rob Knauerhase, and Karsten Schwan. *Attaining system performance points: revisiting the end-to-end argument in system design for heterogeneous many-core systems.* ACM SIGOPS Operating Systems Review 45.1 (2011): 3-10.

[18] Levon, John, and Philippe Elie. *Oprofile: A system profiler for linux.* 2012-05-05. http://oprofile,sf.net(2004).

[19] Tikotekar, Anand, et al. *An analysis of hpc benchmarks in virtual machine environments.* Euro-Par 2008 Workshops-Parallel Processing. Springer Berlin Heidelberg, 2009.

[20] Santos, Renato J. Tutorial: Profiling in Xen http://xen.xensource.com/files/summit_3/xenoprof_tutorial.pdf HP Labs Xen Summit, 2006

[21] Menon, Santos, Yoshio, Janakiraman. XENOPROF Performance profiling in Xen. User Guide. http://xenoprof.sourceforge.net/xenoprof_2.0.txt 2005.

[22] Joukov, Nikolai, et al. *Operating system profiling via latency analysis.* Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006.

[23] Gupta, Vishakha, et al. *Pegasus: Coordinated scheduling for virtualized accelerator-based systems.* 2011 USENIX Annual Technical Conference (USENIX ATC11). 2011.

[24] Ahmad, Irfan, Ajay Gulati, and Ali Mashtizadeh. *vIC: Interrupt coalescing for virtual machine storage device IO.* USENIX Annual Technical Conference (ATC). 2011.

[25] Amit, Nadav, et al. *vIOMMU: efficient IOMMU emulation.* USENIX Annual Technical Conference (ATC). 2011.

49

[26] Lim, Harold, Aman Kansal, and Jie Liu. *Power budgeting for virtualized data centers.* 2011 USENIX Annual Technical Conference (USENIX ATC11). 2011.

[27] VMware, Inc. Performance Study. *Understanding Memory Resource Management in VMware ESX 4.1.* http://www.vmware.com/files/pdf/techpaper/vsp_41_perf_memory_mgmt.pdf

[28] Boutcher, David, and Chandra Abhishek. *Does Virtualization Make Disk Scheduling Pass?* Univerisity of Minnesota. http://www-users.cs.umn.edu/~chandra/papers/hotstorage09/paper.pdf

[29] Diskeeper, Inc. *Virtualization and Disk Performance* http://files.diskeeper.com/pdf/virtualization_performance.pdf

[30] IBM, Inc. *Assessing disk performance with the sar command* http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prftungd/doc/prftungd/assess_disk_perf_sar.htm

[31] Soundararajan, Gokul, and Cristiana Amza. *Towards end-to-end quality of service: controlling I/O interference in shared storage servers.* Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware. Springer-Verlag New York, Inc., 2008.

[32] Lumb, Christopher R., Arif Merchant, and Guillermo A. Alvarez. *Faade: Virtual storage devices with performance guarantees.* (2003).

[33] Knapp, Rashawn L., R. L. Pase, and Karen L. Karavanic. *ARUM: application resource usage monitor.* 9th Linux Clusters Institute International Conference on High-Performance Clustered Computing. 2008.

50

[34] Bittman, Thomas J. *Cloud Computing: Defining and Describing an Emerging Phenomenon.* Gartner Research, March (2008).

[35] Mell, Peter, and Timothy Grance. "The NIST Definition of Cloud Computing." NIST special publication 800-145 (2011).

[36] Catherine, M. Ramya, and E. Bijolin Edwin. *A Survey on Recent Trends in Cloud Computing and its Application for Multimedia.* International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)2.1 (2013): pp-304.

[37] VMware, Inc. *Storage system performance analysis with IOmeter.* http://communities.vmware.com/docs/DOC-3961

[38] Suse Chapter 13. *Tuning I/O Performance.* http://doc.opensuse.org/products/draft/SLES/SLES-tuningsdraft/cha.tuning.io.html

[39] *HPL Benchmark: the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.* http://www.netlib.org/benchmark/hpl/

[40] *HPC Challenge:* http://icl.cs.utk.edu/hpcc/

[41] Momjian, Bruce. *PostgresSQL Hardware Performance Tuning.* http://momjian.us/main/writings/pgsql/hw_performance/

[42] Du, Jiaqing, Nipun Sehrawat, and Willy Zwaenepoel. *Performance profiling in a virtualized environment.* Proc. HotCloud(2010).

[43] Du, Jiaqing, Nipun Sehrawat, and Willy Zwaenepoel. *Performance profiling of virtual machines.* ACM SIGPLAN Notices. Vol. 46. No. 7. ACM, 2011.

[44] Serebrin, Benjamin, and Daniel Hecht. *Virtualizing performance counters.* Euro-Par 2011: Parallel Processing Workshops. Springer Berlin Heidelberg, 2012.

[45] Buell, J., Hecht, D., Heo, J., Saladi, K. and Taheri, H. R. *Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications.* VMware Technical Journal Vol 2. No. 1. June 2013.

[46] Buell, J., Hecht, D., Heo, J., Saladi, K. and Taheri, H. R. *Overcommitment in the ESX Server.* VMware Technical Journal Vol 2. No. 1. June 2013.

[47] Lindsley, Rick. *I/O statistics fields.* https://www.kernel.org/doc/Documentation/iostats.txt

[48] Bowden, T., Bauer, B., Nerin, J., Feng, S. *THE /proc FILESYSTEM.* https://www.kernel.org/doc/Documentation/filesystems/proc.txt

[49] Merchand, Jerome. *I/O statistics of block devices.* https://www.kernel.org/doc/Documentation/ABI/testing/procfs-diskstats

[50] Memory Resource Controller. https://www.kernel.org/doc/Documentation/cgroups/memory.txt

[51] Windows Performance Toolkit. http://msdn.microsoft.com/en-us/library/windows/hardware/hh162945.aspx

[52] Wood, Timothy, et al. "Disaster recovery as a cloud service: Economic benefits and deployment challenges." 2nd USENIX Workshop on Hot Topics in Cloud Computing. 2010.

[53] Tsafrir, Dan, et al. *System noise, OS clock ticks, and fine-grained parallel applications.* Proceedings of the 19th annual international conference on Supercomputing. ACM, 2005.

52

[54] Link, David. *Netflix and Stolen Time* http://blog.sciencelogic.com/netflix-steals-time-in-the-cloud-and-from-users/03/2011

[55] *XenBus* http://wiki.xen.org/wiki/XenBus

[56] *Vshpere Guest Programming Guide* https://www.vmware.com/support/developer/guest-sdk/guest_sdk_40.pdf

[57] *Citrix XenServer 6.1.0 Storage Performance Guide*. Citrix, Inc. 2013.