Xenio

Controlling Virtual Machine Disk Access in Xen

Austin Brooks, Charles Dallachie, Andrew Forsman, William Gomez, Mike Lapping, CJ Norris, Alan Scherger

December 12, 2011

Abstract

Quality of service (QoS) through a hypervisor is currently an unsolved and unexplored problem. It involves keeping track of all disk IO through the hypervisor while keeping the smallest footprint so as not to hinder the performance of the virtual machines (VMs) and, when necessary, throttling disk access to a designated limit. In this paper, we describe our method of QoS inside the Xen hypervisor. We achieved this goal by modifying the *Dom0*'s backend block driver and providing a new module in the kernel of the *Dom0*. We then proceeded to use accounting data to throttle the VMs' disk IO to provide a system where basic QoS could occur.

1 Introduction

Currently in computing, companies are shifting into clouds and performing processing for clients on dedicated machines. A complication that arises here is that multiple users will compete for resources on a given machine. Though each client has its own VM, multiple VMs will battle over CPU resources, memory resources, and disk resources. As of now, companies have managed to implement ways to restrict VMs from accessing too much of the CPU resources or memory resources to enable fair usage. This way companies can charge different rates for resource consumption on a per VM basis. Unfortunately, as of now there is no way to restrict disk IO for each virtual machine. The development of Xenio, as described throughout this paper, attempts to solve this problem and allow for fair distribution of disk resources.

2 Background

Xen is an open source virtual machine monitor software package. It abstracts the interactions between guest operating systems (*DomU*s) and underlying hardware. Xen relies on a host OS, known as *Dom0* in Xen configurations, whose kernel must be modified to perform hardware operations upon request from Xen. Due to the modified kernel requirement, *Dom0*s are generally limited to open source, unix-like kernels such as Linux, Solaris, and BSD.

*DomU*s run in two modes: paravirtualized kernel (PV) and unmodified kernel (HVM). As it pertains to this project, PV *DomU*s perform disk IO in a different manner than HVM *DomU*s. PVs make use of Xen-specific system calls known as hypercalls[1] that map an efficient path for disk requests to flow through Xen to the *Dom0*. For HVM's, Xen presents an emulated disk interface with the help of QEMU. In both cases, disk requests are routed to *Dom0*, whose job is to perform those requests on behalf of the *DomU*s. Consequently, hypercalls are more efficient than the emulated disk interface. The goal of this project is to control disk IO for PV *DomU*s only.

PVs use a two-piece block device driver (split driver) to handle disk IO. The first piece of the driver runs within the PV *DomU* (frontend) and the second runs within the *Dom0* (backend). The frontend and backend block drivers share data using ring buffers. Ring buffers are consumer-producer data structures used for queueing IO requests; one is created for every block device (device pair) used by a PV *DomU*. The backend block driver running in *Dom0* is a threaded, kernel mode process that receives IO request events from the Xen events system and routes IO requests to and from the *Dom0*'s disk scheduler. Each thread, known as a block interface, handles a specific block device pair, communicating request information over the pair's dedicated ring buffer.

3 Xenio

Based upon the design of the Xen IO path, we implemented our Xenio system to allow significant flexibility while also remaining lightweight. Xenio specifically works within the *Dom0*. The main work of Xenio is done within a kernel module. To ensure performance, it was important to keep most computational work within the Xenio module. The Xenio module keeps track of any statistics and data structures and provides an interface for a user to view statistics or regulate enforcement. It was decided that a kernel module would be the best method because it allows the user to insert or remove the module as they desire. If the module is removed, the device driver for the *Dom0* functions as if Xenio does not exist at all on the machine, allowing for slightly better performance.

In order to grab these statistics, Xenio must also do some work within the backend block driver. Fortunately, there was a single function within the driver that has all of the information Xenio needs, and also was appropriate for enforcing IO QoS as well. This function is accessed individually by each *DomU*, which makes this an excellent space to restrict disk IO or perform accounting per *DomU*.

Xenio's work begins in the backend block driver, and starts to collect statistical data for each *DomU*, shown in Figure 1. It records this data in the module's memory space. The module's memory space has two major areas within the data structure used by the driver. The first major area of the memory segment deals with the disk IO of each guest VM and the total disk IO between all VMs for a certain time period. The other major area of the memory segment holds data about each guest VM's total disk IO over the life of the VM and the total disk IO between all VMs since the Xenio module was inserted. In order to view the statistical data that has been gathered at any point, the user simply needs to *cat* one of two proc files: */proc/xenio_timed* or */proc/xenio_totals*. The *xenio_timed* proc file keeps track of the timed data, and the *xenio_totals* proc file keeps track of the totals data. This method was chosen to keep the data from displaying on the screen in a cluttered manner and to make both categories of data much more accessible to the user. In addition

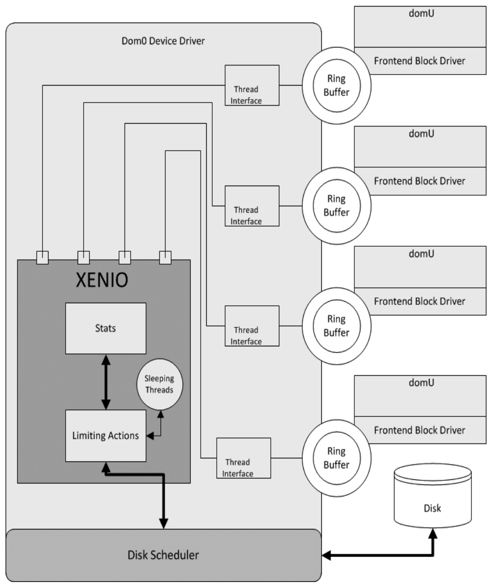

Figure 1: Xenio Design

to providing an output for guest disk IO statistics, both *proc* files provide the user with an input vector to the Xenio system. The *xenio_totals* file provides the user the ability to clear recorded data via writing into the file while the *xenio_timed* file takes a given *DomID* followed by a number to be used as a cap for the enforcement system.

3.1 Accounting

One of Xenio's primary goals was to account for disk IO. Accounting is essential in demonstrating the effect of enforcement. First, it is important to mention the limitations of accounting as they currently exist.

There are only two minor drawbacks to accounting, which can easily be fixed in the future. First, accounting keeps track of data based on the *DomID*. Because the *DomID* changes and no *DomID* is reused unless the hypervisor is rebooted, if a guest VM is shut down and restarted, its old data will stay tied to its old *DomID*, and the new data will begin from scratch. Due to this issue, the Xenio module cannot handle the *DomID* for a guest VM exceeding 40. If this happens, the Xenio module will crash. Second, as of now there is no place where any historical statistics are stored. It would be expected that future iterations will keep track of historical statistics using a database.

Despite these two drawbacks, useful data is available from the system that was implemented. Because a relatively low number of *DomU*s will run over Xen's runtime, *DomID*s will not usually grow close to 40. Accounting focuses on the total disk IO over the life of a VM. The data is never reset and keeps track of all of the stored data per *DomID*. Xenio is only privvy to the information accompanying incoming requests within the backend block driver, so the four meaningful statistics that are accounted for are the number of reads, the number of writes, the number of sectors read and the number of sectors written. While the accounting aspect is important and exists as the foundation of Xenio, enforcing is another major aspect and will be discussed next.

3.2 Enforcement

Xenio's second goal was to enforce basic QoS. Xenio enforcement is designed to control the flow of disk IO requests from a *DomU* to the disk. To accomplish the enforcement, Xenio creates a set of timed accounting data which allows us to check how much disk IO has happened in a single time period. Once a set limit is reached for a time period, the VM should not be allowed to complete more disk IO until the time period has ended.

A new set of data structures that holds only timed data was created along with a timing thread to control it. The timing thread is started upon insertion of the Xenio module and sleeps for a given time period. Once the thread wakes up, it clears the timed data, performs a check of *future* values, which we will describe later, stores a time value for the enforcement code in the backend block driver and sleeps again. The time value lets a thread which needs to sleep calculate the time left until the end of the current time period. For all our testing, the time period was set to five seconds.

In the backend block driver, the timed data is updated at the same time as the totals data for standard accounting. Before accounting, however, checks are made which determine if the current request should be allowed to complete. Theoretically, enforcement can occur based on any of the data stored in the timed data, but for our purposes we implemented enforcement based on the total number of sectors accessed by the VM in both read and write requests. If the next request causes the total number of sectors accessed to be greater than the set limit, the block interface thread is told to sleep for the remainder of the current time period before allowing the request to be completed. Otherwise, the request is allowed to complete as usual.

During our initial testing we realized there was a special case we had not yet handled. What if a low limit is set and a single request accesses more sectors than the limit allows? Without handling this properly, it is possible either to allow too much disk access to a VM over time, or to indefinitely hang a VM. Our system for handling the situation is the *future*s. When a single request accesses more sectors than the limit, the request is allowed to complete on the next cleared time period. The amount of sectors over the limit that it requests is stored as the VM's *future* value. The *future* value acts as a timed limit modifier, meaning that the next time period will act as if the limit is now $limit - future$. If the *future* value is greater than the limit, then the VM will not be allowed to complete any disk requests until the *future* value drops below the limit. *Future*s are decremented in the timing thread.

4 Results

Our final design allowed us the ability to not only monitor IO operations for guests but also to enforce IO limits on each guest individually. To ensure the capability of our design we performed several tests on our system, with various limits enforced across several VMs. To test our enforcement we used an IO benchmarking tool named IOZone which would make large amounts of disk requests. These requests are made in various sizes and patterns to allow anaylsis of performance. To ensure consistent data, we set up each VM to run IOZone at the same start time.

The first test we performed was a simple benchmark for a single VM with a limit, the purpose of this test was to illustrate the effectiveness of our enforcement, as can be seen in Figure 2. As can be

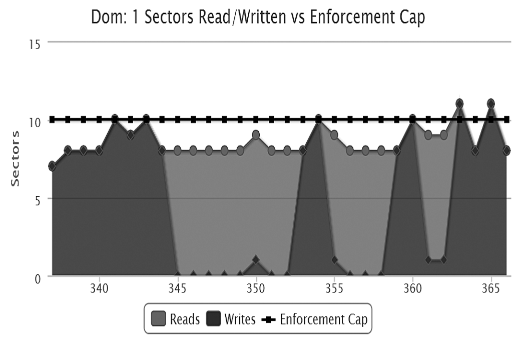

Figure 2: Limiting with Futures

seen from the graph, the VM is incapable of writing or reading over the imposed limit of 10 sectors. However, there is one notable exception to this enforcement policy. As previously mentioned, a single operation sector size can exceed the imposed limit, if the limit is low enough. For this instance, we allow the operation to occur, imposing a modified limit on the next time period based on the *future*s policy, as can be seen in Figure 2.

The next test we performed in the evaluation of our system was to examine the performance while running several guests at one time. For this test, we first recorded the total sectors per guest when running the same IOZone benchmark on 4 VMs at once given staggered limits at 1000, 2000, 4000 and 8000 sectors, respectively, as seen in Figure 3. This produced the expected behavior as each guest VM consumed up to, but not above, its limiting factor. The next part of this test was to then dynamically change the limits to a constant value. This change was reflected in our graph, as we can see the sectors consumed per VM converge on the same value of 2000 sectors.

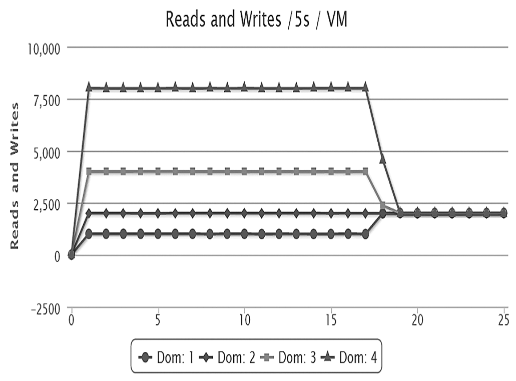

Figure 3: Limit Enforcement across VMs

Beyond proving that our enforcement worked, it was also imperative that we analyze the performance of our Xenio system. Since a large portion of the implementation involved modifying the backend block driver and executed on each IO operation, it is vital that our system not consume large amounts of resources. To determine the impact of our system, we compared the results of IO benchmarks run on a clean Xen installation with both our accounting implementation and our enforcement implementation with enforcing limits set exceptionally high as to prevent actually limiting during the benchmark.

From this comparision in Figure 4 we see that our

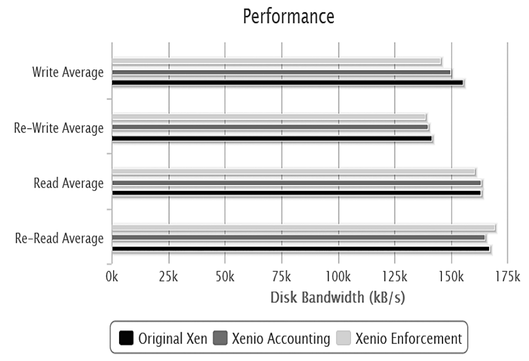

Figure 4: Performance Anaylsis of Xenio

accounting implementation only decreased disk performance 1.343% while our enforcement implementation incurs a 2.249% overhead. This data confirms that our solution is an efficient and effective method of control disk operations through the hypervisor.

5 Conclusion

We have presented an initial method of providing both disk IO accounting and QoS for VMs on the Xen hypervisor. In terms of accounting, our system manages to provide a modular solution to recording and analyzing disk operations for guest operating systems. When coupled with this accounting system, our enforcement methodology has shown that sleeping block interface threads assigned to overconsuming $DomU$s is an effective method for delaying the execution of disk operations and providing reliable QoS for multiple guests. In addition, we have shown that our solution provided these services in an extremely efficient manner, with low overhead and resource consumption. Finally, our system provides this functionality in a simple kernel module that supports dynamic changes to enforcement policy as well as giving the user access to detailed accounting information.

6 Future Work

So far, Xenio has made excellent progress with QoS. However, there is still much to be done. Currently, there are five improvements that are on the list to be implemented. First, a database to keep track of past statistics per VM, to enable persistent accounting and keep track of data over long periods of time. Second, the method for keeping track of statistics should be changed from being based off of the *DomID* to being based off of a unique identifier for each VM. This should enable more robust hypervisor use and keep the system from crashing. Third, implementing separate limits for reads and writes. Separating the limits allows for more user flexibility and also allows machines to access more of what they have paid for. Fourth, improve Xenio to allow for customizable enforcement policies, to allow users to change how they would like Xenio to limit disk resources for specific *DomU*s. Lastly, the limits could be changed remotely via a website or other web console. There are clearly many different ways to improve upon the Xenio module; however, even without these improvements, Xenio is still incredibly powerful and flexible.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. *University of Cambridge Computer Laboratory*, 2003.