Differentiated I/O services in Virtualized environments

Anand Krishnamurthy, Salini S Kowsalya
Computer Sciences Department
University of Wisconsin, Madison
anand@cs.wisc.edu, salinisk@cs.wisc.edu

Abstract

There is a huge semantic gap between file system and storage in servicing an I/O request. In a virtual environment the gap increases even more, as the hypervisor's I/O scheduler is not aware of the semantics of the I/O request that the application issues. To address this limitation and to retain the semantics of the I/O requests in a virtualized environment, we tag each I/O request from the guest application all the way down to the hypervisor's I/O scheduler. By segregating I/O requests, the hypervisor can give differentiated I/O services to different applications both within and across the guest virtual machines. In this paper, we present the design and implementation of such a system on Kernel Based Virtual Machine. We evaluated our system on hard disks and solid state disks for different workloads. Our results show that preferential services are well achieved for applications running within and across guest virtual machines.

1 Introduction

A new era of cloud computing has begun with the advent of virtualization. Virtualization has improved hardware utilization allowing service providers to offer a wide range of applications, platform and infrastructure solutions through low-cost, commoditized hardware. However, virtualization introduces many layers of abstraction such as nesting of guest and host file system, making the system more complex. This leaves a huge semantic gap between host and guest file system in servicing an I/O request.

Multiple guest operating systems can run atop a hypervisor and often each guest runs a lot of applications. In most cases each of the guest operating systems has virtual disks which often share a single shared physical disk. So the I/O requests from the guests have to be scheduled in the hypervisor accordingly. In real world settings, latency sensitive applications need their I/O requests to be served immediately whereas I/O requests from applications like log pushers are primarily used for data mining and business intelligence and are not latency sensitive. Hence different applications often need different levels of I/O service.

VMwares Storage I/O Control [9] and Xen's credit scheduler [3] show that differentiated services can be provided across the guests running on a physical machine. Preferential services to applications running in a guest can be achieved by using appropriate scheduler in the guest. However this solution does not solve the problem of providing a differentiated service across applications running on different guests. In a real world environment, often a low priority process running on a guest can block a high priority process running on another guest. So it is necessary to give differentiated access to applications even across guests. This differentiated I/O service cannot be achieved with VM level prioritization.

In this paper, we address this issue and propose a solution for providing differentiated I/O services for applications running within and across guest virtual machines. This is enabled by passing the tags through the I/O stack of the KVM [6] virtual environment specifically modifying the I/O system call interfaces and disk drivers.

The rest of the paper is organized as follows. Section 2 motivates the need for providing differential services for different applications running across different virtual machines and gives a background and related work. We present our design in Section 3, our implementation details in Section 4 and evaluation in Section 5. We conclude with our learnings and future works in Section 6 and 7.

2 Motivation

Cloud environments host heterogeneous set of applications with different I/O requirements. Prior work related to I/O prioritization focusses on providing differentiated services between virtual machines or virtual disks. VMware's Storage I/O Control (SIOC)[9] provides storage I/O performance isolation for virtual machines to run important workloads in a highly consolidated virtualized storage environment. It protects all virtual machines from undue negative performance impact due to misbehaving I/O-heavy virtual machines, often known as the noisy neighbor problem. Furthermore, the service level of critical virtual machines can be protected by SIOC by giving them preferential I/O resource allocation during periods of contention. However it doesn't offer differentiated services across all applications accessing the datastore. Xen's credit scheduler is a proportional fair share scheduler that supports shares at domain level granularity.

Linux's Completely Fair Queuing(CFQ)[1] also supports I/O prioritization. Yet, there is no mechanism in the original virtualized architecture to get the priority of the I/O request in guest operating system down in the hypervisor's I/O scheduler. Hence in order to get differentiated services out of CFQ, prioritization must be enforced at the I/O schedulers of both the guest and the hypervisor. CFQ scheduler can then place synchronous requests submitted by virtual machines into a number of per-virtual machine queues and then allocate time slices for each of the queues to access the disk. The length of the time slice and the number of requests a queue is allowed to submit depends on the I/O priority of the given virtual machine. Within the guest operating system, differentiated services to processes could be provided in the same manner. Nevertheless using the above approaches leads to two issues

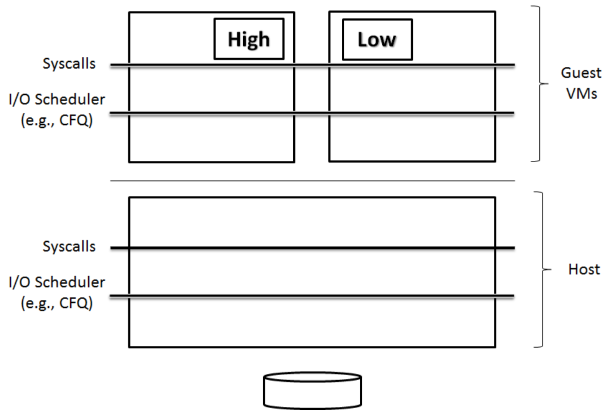

Figure 1: **High and low priority process in different VMs may be treated equally**

1. A low priority process running in a virtual machine and a high priority process running in another virtual machine may get treated equally. This is because the hypervisor's I/O scheduler is not cognizant of the priorities of I/O requests.

2. Another issue is that, this approach might introduce unintended delay in read latency. Since priorities are enforced at the guest I/O scheduler, it might block a low priority read request to serve high priority within the guest OS. Yet, if the data for the low priority read request is in the hypervisor's file system buffer cache, then it could have been served without hitting the disk. Thus blocking low prioritized I/O requests will introduce unwanted delays in servicing read requests.

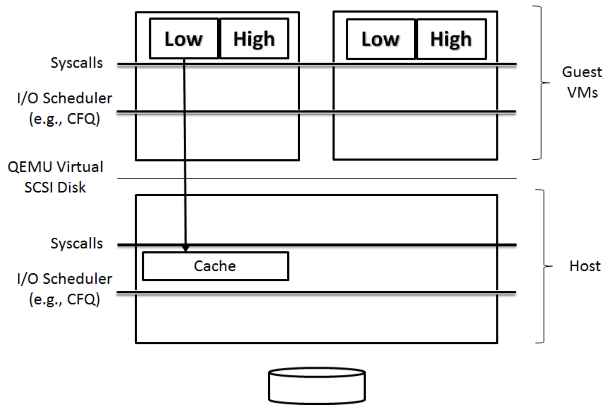

Figure 2: **Blocking low priority reads in guest OS may ignore hypervisor's buffer cache**

Hence with no intrinsic support from virtualization technologies to support guest process level priorities for I/O requests, we need to design a system that enlightens the hypervisor's I/O scheduler about the priorities of the I/O requests in the guest operating systems. As applications are well aware of their workload, they can tag their I/O requests with appropriate priorities. For tagging we use an approach similar to Differentiated storage services[7].

3 Design

To notify the priority of the application's I/O request, the system has to associate the tag which indicates the priority of the I/O request through the layers of I/O stack in the guest operating system and hypervisor. In this paper, we will focus only on reads and give pointers to our initial thoughts on designing for writes in our future work. We have implemented our design in KVM, while this approach could be easily incorporated in other virtualization technologies as well. KVM module which is part of the Linux kernel acts as the hypervisor and has three modes of execution.

- kernel-mode: switch into guest-mode and handle exits due to I/O operations

- user-mode: does I/O when guest needs to access devices
- guest-mode: execute guest code, which is the guest OS except I/O

It relies on virtualization support from the processor. KVM uses Qemu[4] for machine emulation. Qemu supports virtualization by co-operating with the KVM kernel module in Linux. Each virtual machine is a user space process. In the context of KVM, host and hypervisor can be used interchangeably.

3.1 Life cycle of a read request

We will briefly explain the lifecycle of a read request of an application in the guest operating system. The application issues an I/O system call like *read*, *pread* through the system call interface in the user space context of the virtual machine. This will lead to submitting an I/O request within the kernel space context of the virtual machine. Then it will pass through the VFS, file system buffer cache, block layer and eventually to the device dependent driver like SCSI, ATA compliant IDE. The device driver will issue privileged instructions to read the memory regions exported over PCI for the corresponding device. These instructions will trigger VM-exits that will be handled by the core KVM module within the hypervisor's kernel-space context. A VM-exit will take place for each of the privileged instructions resulting from the original I/O request in the VM. The privileged I/O related instructions are passed by the hypervisor to the Qemu process. These instructions will then be emulated by device-controller emulation modules within Qemu (either as ATA or as SCSI commands). Qemu will generate block-access I/O requests which will be handled by the traditional I/O stack of the hypervisor (Linux kernel). Upon completion of the system call, Qemu will 'inject' an interrupt into the VM that originally issued the I/O request. The guest operating system will complete the read system call originally issued by the application.

3.2 Modifications

We will now introduce the modifications that we made to support differentiated I/O services in Qemu-KVM.

3.2.1 New system calls to associate tags with I/O requests

The conventional I/O system call interface doesn't have an inherent way to associate a tag with the I/O

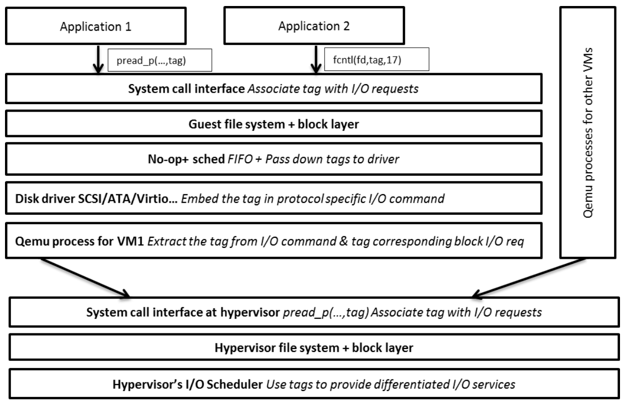
Application 1
Application 2
pread_p(…,tag)
fcntl(fd,tag,17)
System call interface Associate tag with I/O requests
Guest file system + block layer
No-op+ sched FIFO + Pass down tags to driver
Disk driver SCSI/ATA/Virtio… Embed the tag in protocol specific I/O command
Qemu process for VM1 Extract the tag from I/O command & tag corresponding block I/O req
Qemu processes for other VMs
System call interface at hypervisor pread_p(…,tag) Associate tag with I/O requests
Hypervisor file system + block layer
Hypervisor's I/O Scheduler Use tags to provide differentiated I/O services

Figure 3: **Differentiated I/O Services System Design**

request. Hence we introduced two new system calls to complement I/O tagging. Applications in guest operating systems can either tag I/O request while opening a file descriptors or during individual reads. The process after obtaining a file descriptor upon opening a file, calls fcntl to set a tag for the file descriptor. Subsequent read system calls on the file descriptor will carry the tags to the I/O scheduler layer. Similarly we expose a new system call *pread_p* analogous to pread except that it carries tags to the I/O scheduler layer. Applications can use this new *pread_p* system call to associate tags to individual read requests. Thus the system supports prioritization per read request within and across virtual machines.

3.2.2 Noop+ scheduler at the guest operating systems

Does Virtualization Make Disk Scheduling Pass [5] shows that choosing the right schedulers at the appropriate level of the virtualization stack can have a significant impact on the performance of the system. Because of the lack of coordination between the guest and host in a virtual environment, scheduling happens at two layers. First, by the I/O scheduler in the guest and next by the I/O scheduler in the host. This will hurt performance and wastes resource as 50% of the CPU cycles spent in scheduling a request happens at the guest I/O scheduler. This is not necessary as the request has to be scheduled at the hypervisor anyway.

In addition to that, the guest I/O scheduler does not have an overall picture about the I/O requests issued by the other guest virtual machines. So it cannot schedule the I/O requests efficiently. Scheduling at the lowermost layer always leads to better scheduling decisions because of the global picture. Hence we use a Noop+ scheduler in the guest akin to Noop[2] which inserts all incoming I/O requests into a simple FIFO queue. We call it Noop+ since it has extra logic to pass down the tags in addition to the I/O requests.

3.2.3 Editing disk driver to pass down the priorities to the hypervisor

The next modification is in the disk driver layer of the guest operating system to pass down the priorities to the hypervisor. This necessitates changes to the driver code for I/O protocols like SCSI, ATA, virtio etc. Each such I/O protocol requires different types of modifications to pass down the associated metadata.

3.2.4 Qemu modification to support tagged I/O requests

After all the interactions that we mentioned earlier in the Qemu-KVM I/O stack, the I/O commands are available in the Qemu emulation module. The tag is thus extracted from the I/O command. The underlying storage for the virtual disk can be either virtual disk image formats such as vmdk, qcow2 or physical disks. In both cases, Qemu I/O emulation layer issues block I/O requests for the corresponding I/O commands. We modified this part of Qemu to use our custom system call *pread_p* to pass down the extracted tag to the hypervisor's I/O stack.

3.2.5 Semantic aware I/O scheduler at the hypervisor

Since the tags associated by the application to the I/O requests have been passed down to the hypervisor, it knows about the semantics of the I/O requests. So based on the tags, the hypervisor's I/O scheduler can make intelligent scheduling decisions with any policies.

We envision that the design should be flexible to plug-in any scheduling algorithm and provide one such implementation, a stride[8] based scheduler. The details of the algorithm are presented in Section 4.

4 Implementation

In this section we will mention the specific changes that we made for implementing differentiated I/O services. We added a new fcntl function in fcntl.c to associate the tag with the file descriptor. A new system call was introduced in unistd.h. Changes were also made in file system layers read_write.c and blk-core.c to passdown the tags. Two elevator algorithms for I/O scheduling guest.c and host.c were used in guest and host kernels respectively. We edited the sd.c , the driver layer for disk. From the SCSI specification we identified an unused byte in the command descriptor block of READ, WRITE commands and we leveraged that to inject the tags. In Qemu, we edited block device emulation module to use the custom system calls pread_p to associate tags to the I/O requests. These tags are used then in the hypervisor's I/O scheduler for scheduling. ID_i is the tag which gets passed along the I/O stack. Whenever I/O scheduler has to dispatch a request to the disk, it invokes $Dispatchrequest()$. The following is the algorithm.

4.1 Stride Scheduler Algorithm

Id_i - Identity(ID) of the application A_i
$Shares_i$ - Shares assigned to Id_i
VIO_i - Virtual I/O counter for Id_i
$Stride_i$ - Global-shares / $Shares_i$

```
Dispatch request()
{
  • Select the request k which has the lowest virtual I/O counter
  • Increase $VIO_k$ by $Stride_k$
  • If process k has slept for long time then
      – $VIO_k$ = max(min($VIO_i$ which are non zero), $VIO_k$)
  • if ($VIO_k$ reaches threshold)
      – Reinitialize all $VIO_i$ to 0
  • Dispatch the request
}
```

5 Evaluation

We implemented our system, as described in the previous sections, in the KVM hypervisor. This section evaluates the functionality and performance of our implementation on hard disks and solid state drives(SSD).

5.1 Experimental setup

We consider two metrics - throughput and latency, for checking whether differentiated service is achieved across different applications running in the guests. A careful reader may think that checking for preferential service using throughput and latency will be a similar kind of measurement as latency of a process having higher throughput will be lesser than that of the process that has the lower throughput. But in some situations, two or more processes may have the same throughput, but their latencies may be different. Preferential service is considered to be achieved even in such situations.

We tested our system on a machine running Kubuntu 12.04 with Linux Kernel 3.2. The system includes 8GB of memory. The hard disk rotates at a speed of 7200 rpm and SSD delivers 35000 and 85000 IOPS for reads and writes respectively. Each guest virtual machine launched has a memory of 1GB and runs Ubuntu Server 12.04 with Linux Kernel 3.2. Our experiments involve launching same or different set of applications simultaneously in different guests and checking for differentiated services by measuring their throughput and latency.

We benchmarked our system with various workloads which tested the file I/O performance. The workloads which we used are as follows. **Sysbench** sequential read and random read workload checked for differentiated service across the applications during sequential reads and random reads respectively. These workload tested on 2000 files, each having a size of 16 MB(since the guest memory size is 1GB).

In order to check whether preferential service is obtained in real world workloads, we tested our system on **filebench**'s web-server and mail-server workloads. The former emulates a simple web-server I/O activity while the latter emulates a simple file-server I/O activity. We also tested our system on DHT(Distributed Hash Table) reads which is emulated by **voldemort** performance analysis tool. All the above workloads were run for three minutes. We also compare our results with linux's CFQ(Complete Fair Queuing) scheduler which has the ability to schedule different process with different I/O priorities.

5.1.1 Hard disk

This section shows the throughput and performance of I/O activity generated by different workloads on hard disks.

Figure 4 and Figure 5 shows the throughput and latency of our system while running the same kind of workload on five different applications(launched across different virtual machines) with different number of shares allocated to each of them. The shares are represented in X axis. Figure 4 shows that, for sequential read, random read and web-server workloads, there is an increase in throughput as the number of shares increases for different applications . But for mail-server workload, the throughput remains constant across applications with different shares.

Figure 5 shows that as the number of shares allocated to the applications increases, the latency decreases for random read, web-server and DHT workloads, whereas latency is not affected by shares for sequential read and mail server workloads. From these graphs we infer that preferential service is obtained in sequential read, random read, mail-server and web-server workloads. Mail-server workload does not obtain preferential service because it is dominated by both reads and writes. Since we have not implemented the I/O tagging for write system calls, preferential service is not observed.

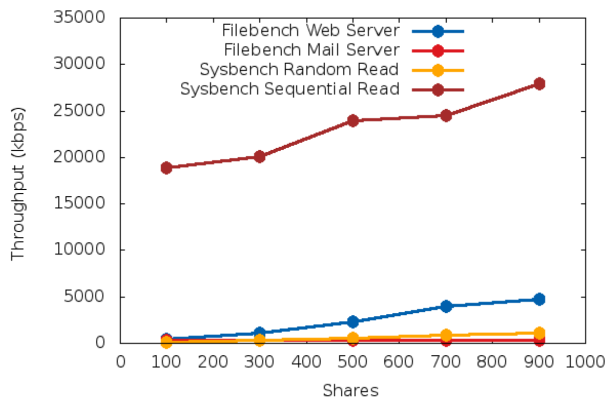

Figure 4: **Shares vs Throughput for different workloads**

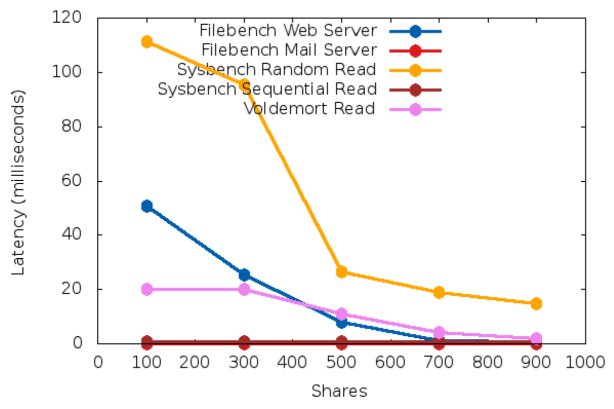

Figure 5: **Shares vs Latency for different workloads**

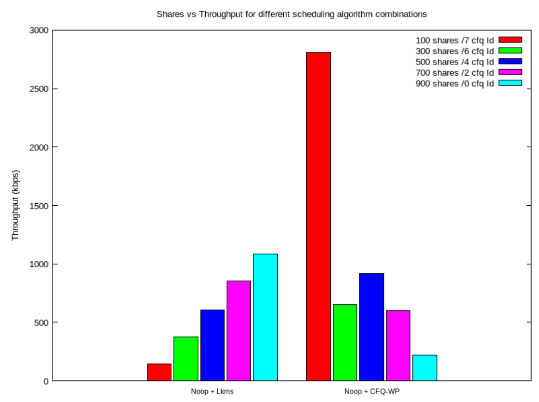

Figure 6: **Priority/Shares vs Throughput for different schedulers**

Figure 6 is an interesting graph which shows the comparison between running our system and running CFQ scheduler with priority. CFQ can run different processes with different I/O priorities levels. So one can use CFQ I/O scheduler in the hypervisor and launch virtual machines with different priority levels. Then we can check if we can achieve preferential service by running different applications with different virtual machines.

We compared the throughput obtained for applications from our system with that of the one obtained by launching different virtual machines having different I/O priorities. In CFQ, lower number corresponds to higher priority. Figure 6 shows that preferential service is obtained as expected in our system since the throughput increases as the number of shares increases[Noop+lkms]. But in CFQ with priority[Noop+CFQ-WP], clearly preferential service is not obtained as the throughput for a lower priority process is higher than that of the higher priority process.

We also checked for preferential service within the guest by launching different applications with different priorities with CFQ scheduler running in the guest. Figure 6 shows the graph obtained in such a setup. Even in this case, preferential service is not obtained.

The real world setting often is a mixed workload setting where different services run on different applications across virtual machines. So we tested our system by running different kind of workloads on three applications with different number of shares allocated to each of them. For comparison, we tested the same experiment with CFQ having different priorities for different guests.

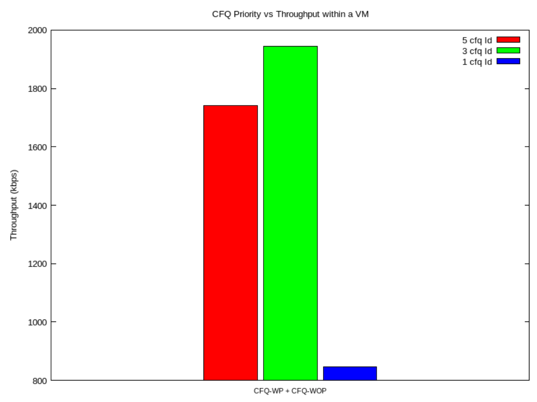

Figure 7: **Priority/Shares vs Throughput for different schedulers**

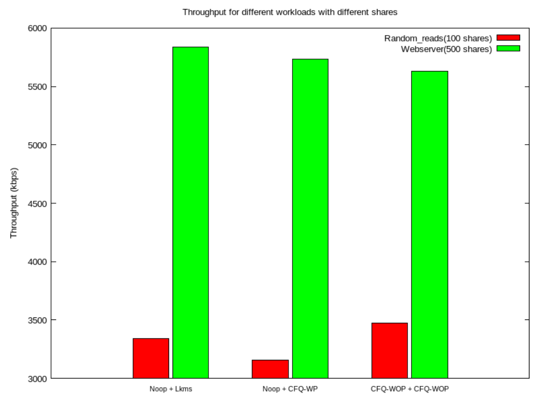

Figure 8: **Schedulers vs Throughput for different workloads**

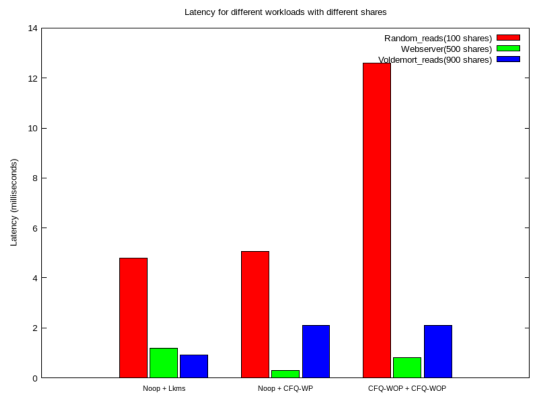

Figure 9: **Schedulers vs Latency for different workloads**

We compared the throughput and latency obtained for mixed workload applications obtained from our system with that of the ones obtained by launching different virtual machines having different

I/O priorities. We also compared our system with the default system which will not provide any differentiated services. Figure 7 and 8 show that throughput and latency of the mixed applications with different number of shares/priorities. Even in this setting, our system(Noop+lkms) gives preferential service as expected. In CFQ with priority (Noop+CFQ-WP), a lower priority process gets higher throughput and less latency than a higher priority process. Figures 6, 7 and 8 imply that CFQ with priority cannot operate in virtualized environments.

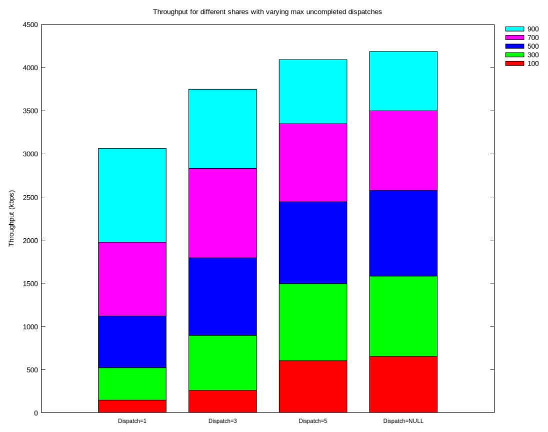

Figure 10: **Throughput for different dispatches**

Figure 10 is an interesting graph which shows the effective throughput for different dispatch numbers while running sysbench random read workload on applications having different number of shares. Dispatch number represents the maximum number of uncompleted requests a disk can have before I/O scheduler issues further more requests to it. Different colors in the graph represents different applications and applications are arranged in descending order of shares from top to bottom. For dispatch is 1 and 3, the system gives preferential service across different applications whereas for dispatch 5 and dispatch infinity preferential service is not obtained. But the effective throughput of the disk is high for dispatch 5 and dispatch infinity in comparison with dispatch 1 and 3. As our goal was to provide differentiated services, in our system reduction in effective throughput was compromised. However this situation can be eliminated by having smart disks. In this paper, we did not talk about smart disks.

5.1.2 SSD

We ran the same set of experiments on SSDs. This section shows the throughput and performance of I/O activity generated by different workloads on hard disks.

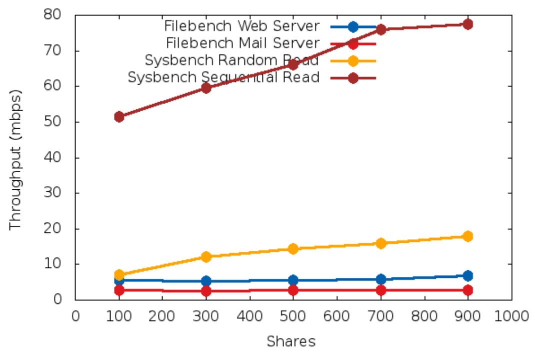

Figure 11: **Schedulers vs Throughput for different workloads**

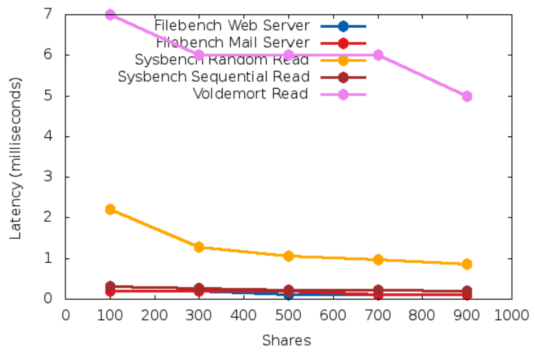

Figure 12: **Schedulers vs Latency for different workloads**

Figure 11 and Figure 12 show the throughput and latency of our system while running the same kind of workload on five applications(across different virtual machines) with different number of shares, which is represented in X axis. Figure 11 shows that the throughput of applications increases as the number of shares allocated to them increases for sequential read and random read workloads. However, for mail-server and web-server workloads, the throughput remains almost constant. Figure 12 shows that as the number of shares allocated to the applications increases, the latency decreases for random read, web-server and DHT workloads, whereas latency is not affected by shares for sequential read and mail server workloads. These graphs show that preferential service is obtained in sequential read, random read, DHT reads, but not for mail server and web-server workloads. This is because SSDs are faster than hard disks, and hence all the read requests are satisfied way faster than that of the hard disk. In order to observe the differentiated services in SSDs, we have to generate a larger number of read requests from the applications.

6 Future Work

So far we have provided differentiated access to only read requests. For providing differentiated access to writes, we have to find a mechanism to pass information from the guest to the hypervisor and vice versa. In the current implementation, we have assumed that the guest virtual machines are cooperative. But to deploy in a real world environment such as cloud, we need to incorporate mechanisms for security. We can also use our system for accounting purposes: one can account for the number of requests an application issues to the disk storage. If a smart disk is available, we can pushback the tags to the disk controller layer in the smart disk and thereby improving the effective throughput of the storage as well. We are also planning to compare our results with VMwares ESX server with SIOC.

7 Conclusions

In summary, we have presented a system to provide differentiated access to I/O storage on KVM hypervisor by reducing the semantic gap between the guest and host filesystem in a virtual environment. Our evaluation results show that by tagging each I/O request, one can improve the performance of latency sensitive applications in a virtual environment by giving higher priority to them. I/O requests of applications which collect statistics for business intelligence can be given lower priority. Since virtualization has marked a new era in the computer world, reducing the semantic gap in processing I/O requests may open a new area of I/O optimizations and many new conferences such as CADAVER!

References

[1] Jens Axboe. Completely fair queuing.

[2] Jens Axboe. Noop scheduler.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] David Boutcher and Abhishek Chandra. Does virtualization make disk scheduling passé? *Operating Systems Review.*

[6] KVM http://www.linux kvm.org. Kernel based virtual machine.

[7] Michael P. Mesnier and Jason B. Akers. Differentiated storage services. *Operating Systems Review*, 45(1):45–53, 2011.

[8] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, pages 1–11, 1994.

[9] VMWare Whitepaper. Vmware storage i/o control. *Storage I/O Control Technical Overview and Considerations for Deployment*, 2011.