Tag Scheduler – An ARM I/O Scheduling Framework in KVM

Weihua Bai wb2331, Yukun Huang yh2977
Final Report
Computer Science Department
Columbia University

ABSTRACT

Virtualization enables multi-tenancy in data centers, however, it also introduces challenges to the resource management in traditional operating systems (OSes). In a traditional operation system, I/O requests of different applications/processes can be assigned with different I/O priority and the I/O scheduler is able to utilize such information to allocate time slices for each application to access the disk, e.g. by using the Completely Fair Queue (CFQ). On the other hand, in a virtual environment, I/O requests from applications running on the guest OSes are firstly handled by the I/O scheduler of the guest OSes and passed down to the hypervisor's scheduler afterwards. Passing down I/O requests from guests to host in this way, however, causes the hypervisor being unaware of the I/O priority of the guest applications. As a result, a low priority process running on a guest can usually block a high priority process from another guest.

In this project, we present Tag Scheduler to enforce I/O priority on guest application level, i.e. to ensure the I/O scheduler of hypervisor receive the assigned I/O priority of each application running on the guests for allocating resources more predictably and properly. The method enables priority tagging for processes across VMs. We implemented new system calls that would be able to inform the hypervisor's scheduler about the I/O priority of guest applications, and a modified I/O scheduling algorithm that could utilize the information.

KEYWORDS

Virtualization, I/O prioritization, Semantic Gaps

1 INTRODUCTION

Virtualization is becoming ubiquitous due to its support for multi-tenancy on cloud computing platforms and data centers. On the other hand, virtualization brings in a lot more layers of abstraction to the system that increases the semantic gap between physical and virtual environments. I/O prioritization is an important OS design to ease the performance bottleneck introduced by contention for I/O access of multiple processes. Without compromising throughput, I/O prioritization improves system responsiveness by serving applications with different priority with differentiated services in terms of e.g. the length of the time slice to access the disk and the number of requests allowed to submit etc. In reality, however, I/O prioritization in the virtualized environment is often less effective due to the semantic gap.

In a cloud computing platform or a virtualized multi-tenant data center, multiple guest operating systems often run on top of a hypervisor that manages the allocation of resources. According to Popek and Goldberg's virtualization requirements, a program running in a virtualized environment should have a behavior essentially identical to that in physical environments, i.e. virtualization is transparent to applications running in guest OSes. [1] Although each of the guest OSes has its own virtual disk, they actually share a single physical disk and the I/O requests are scheduled by the I/O scheduler of hypervisor to access the shared physical disk. With resource contention due to sharing, latency-sensitive applications running on a virtual machine are often blocked by latency non-sensitive applications on other VMs as the hypervisor is unaware of the priority assigned to the guest level applications and their corresponding I/O requests.

To verify the performance of I/O prioritization is to measure fairness. There are multiple definitions for fairness in the domain of scheduling problems. In this project, we measure fairness by comparing the average throughput of each process. We define a priority inverse case as if a low priority process in a VM is granted higher average throughput than a high priority process in another VM and we want to eliminate or at least alleviate the number of such cases.

In a modern virtualized environment, I/O requests of guest applications are scheduled by the I/O

schedulers on guest OSes before being passed down to the hypervisor's scheduler. The Completely Fair Queuing scheduler (CFQ), the Noop scheduler, the Deadline scheduler and the Anticipatory scheduler are the four I/O schedulers used by Linux kernel. The default Linux configuration is to run the CFQ scheduler both in the hypervisor and the guests. In section 5, we will demonstrate why this combination may not be satisfying by an experiment showing multiple priority inverse cases.

We argue that I/O prioritization in guest OSes is unnecessary since these I/O requests would eventually be scheduled by the hypervisor and the current guest OS schedule may be overridden and become invalid. In this project, we implement a method for the ARM architecture that tags guest level applications with priority and passes the priority with I/O requests all the way down to the hypervisor to be scheduled by its I/O scheduler. We use the Noop scheduler, a simple FIFO queue, at the guest level for the potential performance benefits compared with the much more complicated CFQ scheduler. In the hypervisor level, a new lightweight scheduler is implemented to take the additional priority messages into scheduling consideration.

As a graduate course project, our intention is to verify the idea more than demonstrating its power. We will show in the following how Tag Scheduler meets the requirement by alleviating the number of priority inverse cases introduced in other scheduling configuration, e.g. CFQ + CFQ, Noop + Noop, but may not eliminate it. The overall performance of our method compared to different combinations of schedulers used in the guest and host, e.g. CFQ + CFQ, Noop + Noop etc. is shown in section 5. In the benchmark test we used, Tag Scheduler is slightly faster than the default CFQ + CFQ configuration and both of them are slower than Noop + Noop. It should be noted that different schedulers in Linux kernel are optimized for different workloads. We are very interested in testing our method against more benchmarks with different characteristics. We will be discussing observations from other recent results in section 2.

Last but not least, traditionally, people regard the value of I/O scheduling mainly in the focus on minimizing disk arm movement on a spinning disk. In this project, Tag Scheduler is an ARM scheduling framework which runs on FLASH storage, without an access arm. We argue that the additional processing required to optimize the way I/O operations are delivered to underlying storage devices, i.e. I/O scheduling, is still useful and important.

2 RELATED WORKS

2.1 VMware Storage I/O Control [2]

In order to serve latency-sensitive applications, VMware introduces Storage I/O Control (SIOC) in vSphere. SIOC is a disk scheduler which monitors the datastores to determine if there is resource contention occurring. Whenever it detects resource contention, SIOC isolates the virtual machine disk (VMDK) that is causing the contention and takes action. Critical latency-sensitive applications are therefore protected from their noisy neighbor VMs as they are granted exclusive access to physical resources. However, this feature comes with side effects. The PCPUs exclusively given to the critical VM may not be used by other VMs even when these PCPUs are idle. This may lower the overall CPU utilization and the system throughput, compromising other VMs' performance.

2.2 Krishnamurthy et al.'s Paper [3]

In this paper, they present a similar method on the x86 architecture, however, we failed to find a workable version of their software. It's not open-sourced on GitHub and the code we found is not documented/readable. We see a gap to verify and improve the method.

2.3 Preserving I/O Prioritization in Virtualized OSes [4]

This is a recently published paper by Suo et al. on Xen and they have mainly discussed the problem of how to improve the performance by multiplexing I/O-bound tasks with compute-bound tasks. We don't think this is closely related to what we want to achieve in this project.

We want to emphasize on the following work, as it provides a model for comparing the combination of different schedulers used in guest and host.

2.4 Does Virtualization Make Disk Scheduling Passé? [5]

The paper shows some interesting experiment results by evaluating all combinations of the four I/O schedulers, i.e. the Noop scheduler, the CFQ, the Anticipatory scheduler, and the Deadline Scheduler in VMM and guest OSes. They used two benchmarks, one is the Flexible File System Benchmark (FFSB), a multi-threaded benchmark that provides a mix of read and write operations; the other is Network Appliance's PostMark, designed to emulate Internet applications such as e-mail, netnews, and e-commerce in a single thread, also known as small file and metadata-intensive workloads. Their experiments were performed on both Xen (Type 1) and VMWare Server (Type 2) hypervisor.

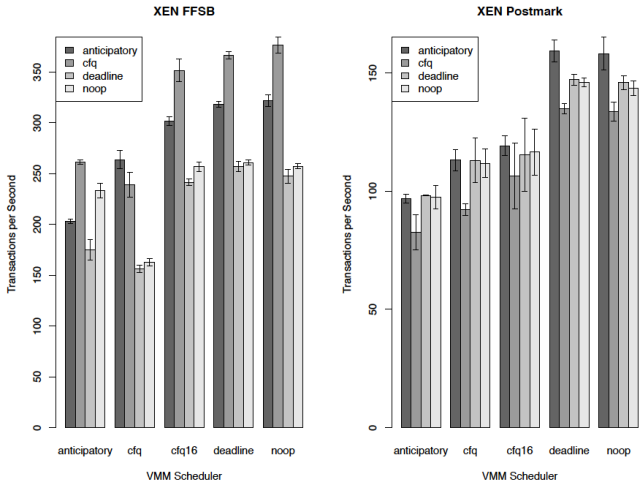


Figure 1: Throughput Comparisons on Xen

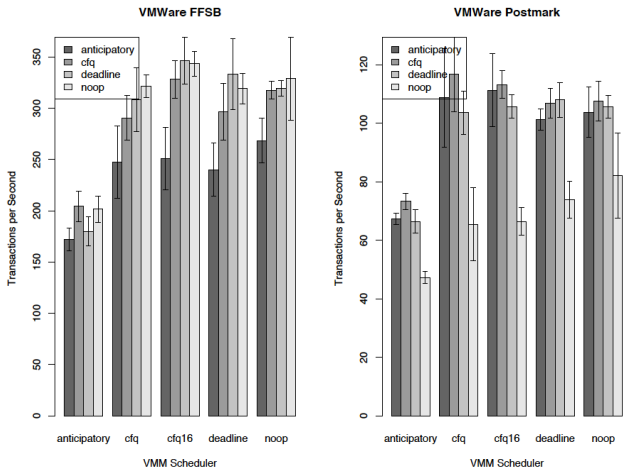


Figure 2: Throughput Comparisons on VMWare Server

In Figure 1&2, the I/O scheduling algorithms shown along the lower axis are schedulers running in hypervisor and the different shaded bars reflect I/O schedulers running in guest OSes. We can tell from Figure 1 that CFQ is the most preferred guest scheduler in FFSB no matter which scheduler is used in Dom0 of Xen. Similarly, for PostMark, the Anticipatory scheduler stands out. In this sense, the result verifies that different schedulers are optimized for different workloads, and the author claims that choosing the scheduler closest to the application, i.e. in the guest, has the greatest impact on performance. On the other hand, they suggest that using the Noop scheduler in the host can provide generally, if not necessarily, better performance compared to other more complicated schedulers. They conclude that there is no benefit in throughput from performing additional I/O reordering in the VMM, however, it may benefit fairness.

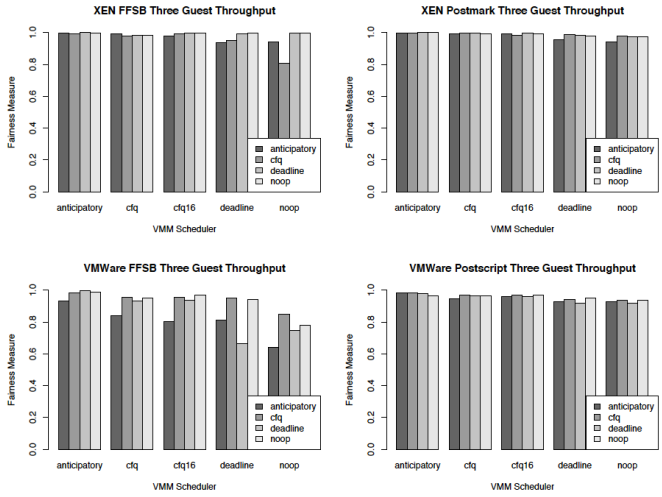


Figure 3: Fairness Comparisons

Figure 3 demonstrates the Jain's Fairness measure [5] for three VMs simultaneously running on top of the host, using the same guest scheduling algorithms. Jain's fairness measure ranges between 0 (completely unfair) and 1 (completely fair). We can tell from Figure 3 that using Noop at the host is the least fair among almost all combinations. More complex scheduling in the hypervisor can provide better fairness across multiple virtual machines.

The author finally concludes that there is a tradeoff between throughput and fairness in terms of which scheduling algorithm to use in the host, given that the Noop scheduler is top at throughput but bottom at fairness. On the other hand, choosing a scheduling algorithm in the guest should be judged by the specific workloads to work on.

2.5 Motivations from Related Works

The above paper is inspiring, but we didn't use their metrics for fairness measure. As mentioned in section 1, the average throughput for processes with different assigned priority is the metric we used in this project since we have knowledge about the priority of each process.

Surprisingly, we are not able to find many works related to disk I/O prioritization in virtualized environment, especially in ARM. We also find out that there are only a few implementations of application-level I/O prioritization across VMs in the x86 platform. This is the main motivation for us to work on Tag Scheduler.

3 METHODOLOGIES

3.1 Pass Modified System Call to hypervisor's I/O Scheduler

I/O system calls such as read, pread are handled by the following I/O stack (Figure 1). They are first called in the user space of the VM through the system call interface. This will lead to submitting an I/O request within the kernel space of the VM. Then it will pass through the virtual file system (VFS), the File System Buffer Cache and the block layer. In our implementation, the device driver used is virtio-scsi. The device driver will issue privileged instructions that cause virtio notifications/traps. These notifications/traps are then handled by the core KVM module within the hypervisor's kernel space. Privileged instructions are passed to QEMU and emulated by device-controller emulation module in QEMU. QEMU will inject an interrupt into the VM that issued the I/O system call and the guest OS will complete the system call.

In order to notify the I/O scheduler at the hypervisor of the guest-level application priority, a new system call with the priority arguments is created and the I/O stack is modified to be able to pass to process the new arguments. We will discuss the implementation in detail in section 4.

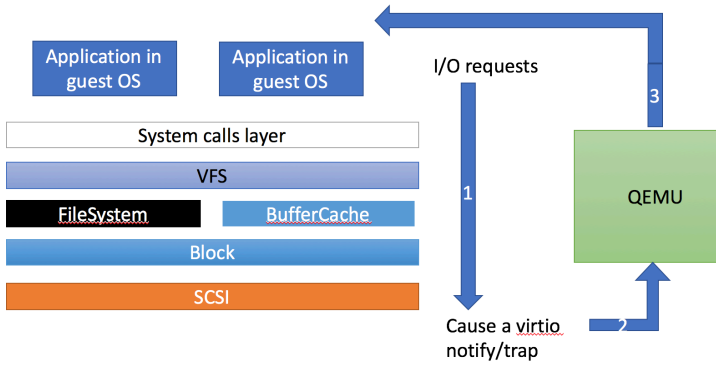


Figure 4: The I/O Stack of Handling a I/O request

3.2 Tag Scheduler Framework Configuration

As mentioned in section 1, the default configuration for Linux kernel is to use CFQ in both the host and guests, as illustrated in Figure 5.

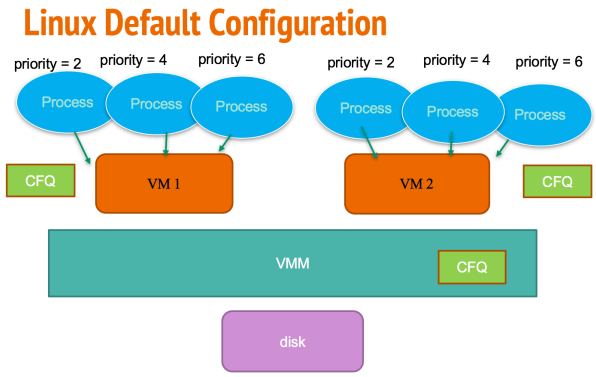


Figure 5: Linux Default Configuration

In Tag Scheduler, the Noop scheduler is used in guest VMs for high throughput, while in the hypervisor, we use a new scheduling algorithm. The new framework is illustrated in Figure 6.

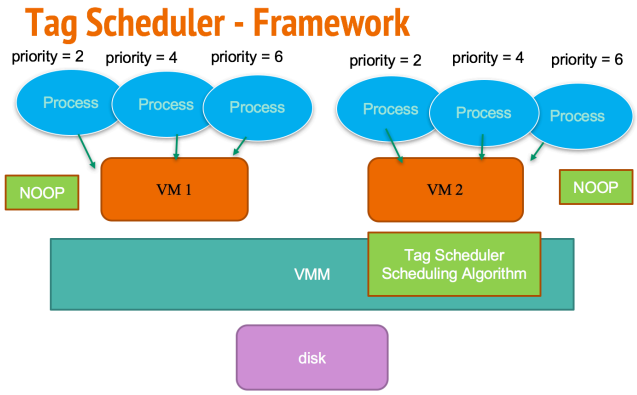


Figure 6: Tag Scheduler Configuration

3.3 The New I/O Scheduling Algorithm for the Hypervisor

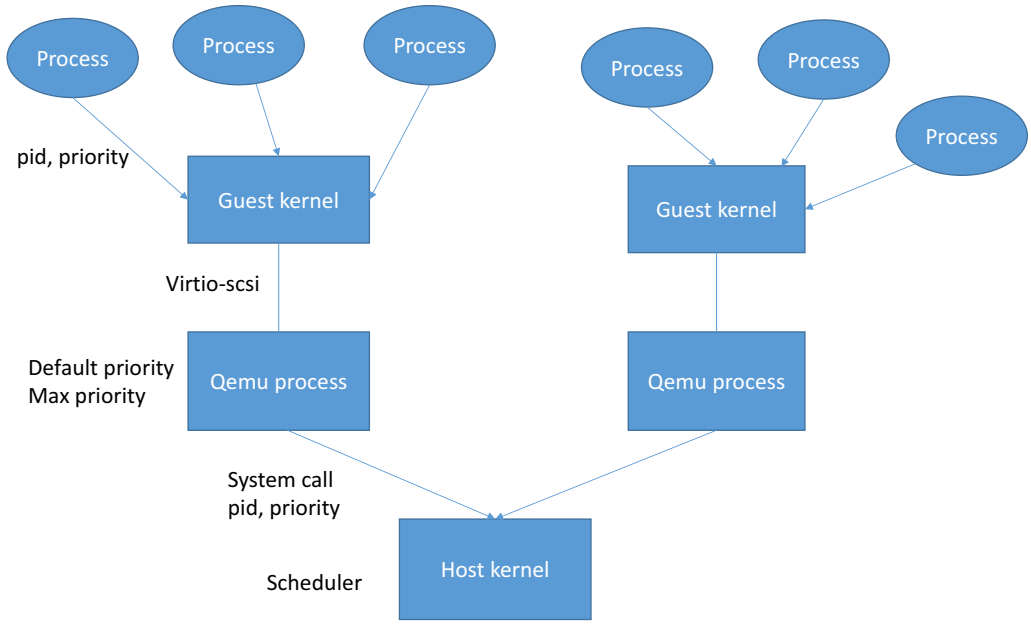
There will be a global counter G in the host.
The priority for each read request: Pi
The virtual I/O counter for each process that sends read request: Ck
Scheduling steps:

Select the process k which has the lowest virtual I/O counter and its latest request i
Increase Ck by G/Pi
For a new process or a process that has slept for a long time
Ck = max(min Ck across all the reading requests, Ck)
Dispatch the request i

Algorithm [3]

4 IMPLEMENTATIONS

Like shown in Graph 1, our framework is composed of five parts:



Graph 1

1. Transfer the given_priority of each process that is assigned by the guest user and the process id of each process from guest application to guest kernel. During the whole procedure that is composed of different layers, like shown in graph 2, the read request changes its format several times. We just added additional fields in the relative structures or functions to bind these two pieces of useful information. Finally the request will hit the device driver layer of the kernel for further dispatch. Because we will do the final scheduling in the host kernel, so there is no need to have any I/O scheduling algorithm in the guest kernel. So we just assign the I/O scheduler to noop.

2. Transfer the information we get in the last step into the qemu process. We set up the qemu process to use virtio-scsi as its I/O driver, which is a paravirtualization I/O emulation approach. The read request is transformed into a scsi command, and wrapped with a bunch of virtio structures. Then the request will be put into vring of virtio. The vring is actually some shared memory between guest kernel and qemu address space. So in the qemu side, the qemu can get the request from the vring, and after some processing, it can get the SCSI command the guest dispatches. So the request will be put into vring of virtio. And the format and data saved in the lowest layer of scsi command actually stays the same in both qemu and guest kernel. After testing, we found two unused bytes in the command and use those two bytes to transfer the given_priority and process id to the qemu process.

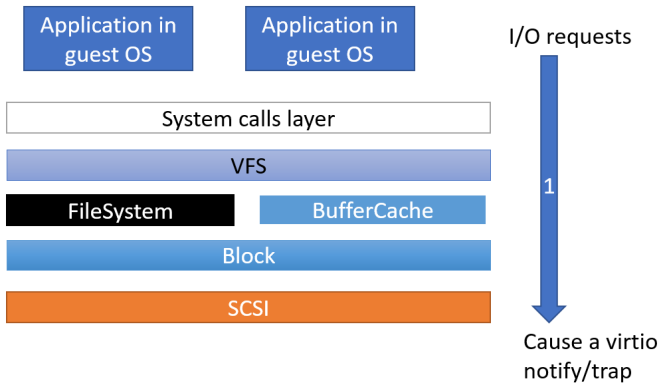


Figure 6

3. In the qemu process, we got the process id and the priority for each process. We also added two new parameters that can be assigned during the startup of guest virtual machines in the qemu: the default priority and maximum priority for this VM. So the final priority of each reading request will be decided by these three priorities: 1. The default priority of each virtual machine, called default_priority. 2. The maximum priority of each virtual machine, called max_priority. 3. The priority of each read request given by the guest VM. The reason why we break the assignment of priority into three parts are two, first, not every application wants to manually assign a priority for their requests. When they don't want to give an assigned priority, the request will use the default priority. Second, we want the host machine manager to have some control over all the virtual machines, in case that there is a malignant user to control the whole host system and throttle I/O process of other VMs by giving a very high priority to all his read requests. So there should be a maximum read priority that one virtual machine is able to give, which is why we also need a maximum priority for the virtual machine. So the final priority is:

If given_priority exists
    then final priority = min(maximum_priority, given_priority)
else
    then final priority = min(maximum_priority, default_priority)

4. Transfer the information from qemu to host kernel. We add two new system calls in the guest kernel called tag_pread64 and tag_preadv64, which are basically similar to the original system calls pread64 and preadv64, but they will also assigned a structure as the parameter for each read request. The structure is composed of three parts: priority of the request, process id of the request, virtual machine id of the request. And in the host kernel, the information still need to be bound with the read request all the way to the I/O scheduler. The procedure is basically same as that in the guest kernel, with some additional handling for direct I/O and device mapper.

5. Finally, the read request hits the I/O scheduler in the host kernel, with all the useful information we need.

Our algorithm is quite simple, with a thought of virtual io counter similar to the CFQ virtual time, shown below:

There will be a global counter G in the host.

The priority for each read request: Pi

The virtual I/O counter for each process k that sends read request: Ck

Scheduling steps:

Select the process k which has the lowest virtual I/O counter and its latest request i

Increase Ck by G/Pi

For a new process or a process that has slept for a long time

Ck = max(min Ci across all

the processes, Ck)
Dispatch the request i

We implemented this algorithm by mimicking and changing the code of Noop scheduler. Basically we created structures for each virtual machine and each process that currently has requests to dispatch. And order all the processes with two red black tree, one is ordered by (process id | (virtual machine pid << 16)), for quickly searching a process by its unique identifier in the host kernel. The other tree is ordered by the virtual I/O counter of processes, for quickly searching a process by virtual I/O counter.

5 EXPERIMENTS

5.1 Experiment Setup

1. We did all our experiment in CloudLab, using one host machine node m400:

2. M400:
   a. Eight 64-bit ARMv8 (Atlas/A57) cores at 2.4 GHz

   b. 64GB of ECC RAM (8x 8 GB DDR3-1600 SO-DIMMs)

   c. 120 GB of flash (SATA3 / M.2, Micron M500, hardware AES-256 encryption)

3. The host kernel and guest kernel are both using Linux-4.14.26

4. QEMU (version 2.11.1) and KVM are used to virtualize each VM. The I/O driver is virtio-scsi. No cache is used for each VM in the host kernel so that we can eliminate the influence of host cache.

5. The I/O request can be initiated by Filebench, but it seems that the results are influenced by the existence of guest cache. So, we also use our own benchmark.

6. Each VM is assign 1GB of RAM to shrink the influence of guest file cache.

7. The guest VMs are all created by the following parameters: double core, 16GB memory size, virtio-scsi I/O driver.

Our testbench is quite simple, there are three virtual machines overall, each virtual machine has two processes that can keep reading from three 1GB files. With a 256 KB reading stride each time. The result in the graph is their average throughputs. As mentioned in section 1, we observed an inversion in with the following experiement. There are three processes running in each VM, we assign priority 6, 4, 2 respectively to the processes. VM1 is assigned with priority equals 2 while VM2 being 5. As shown in Figure 7, processes running in VM1 with priority 4 and 6 are served with low throughput compared to processes with the same priority running in VM2. Two inversions can visible.

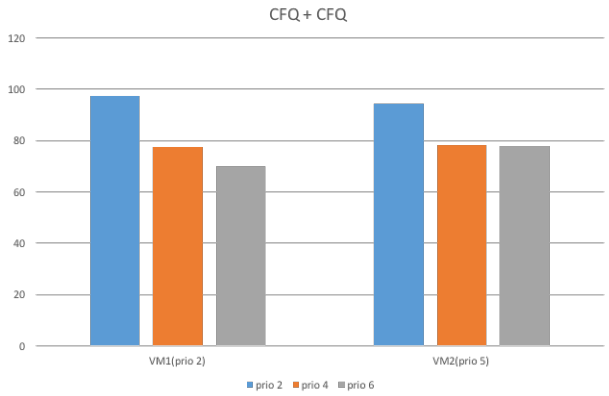


Figure 7

Then, we assign all the 6 processes with different priorities 12, 10, 8, 6, 4, 2. And for a comparison about their throughput, we also do the same experiments using CFQ and NOOP scheduler (since they cannot provide priorities across processes in different VMs, so we didn't assign any priorities for CFQ or NOOP)

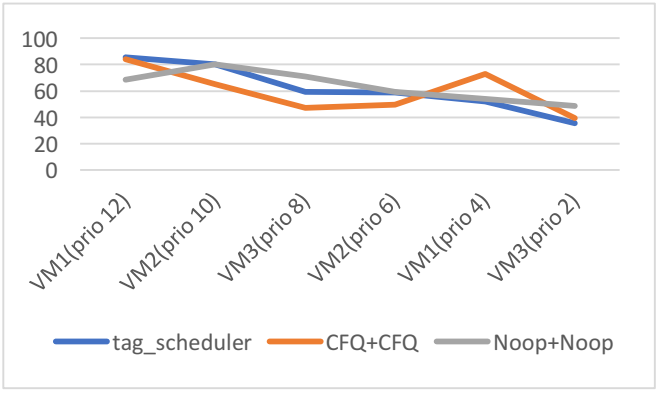


Figure 8

5.2 Analysis

So from the result, our framework can do prioritized scheduling across different virtual machines to some extent. And from the final result and experiments, our scheduler, with an average throughput 61.1 MB/S for six processes, can be slightly faster than CFQ, with an average throughput 59.5 MB/S, and slower than noop, with an average throughput 62.9MB/S. Because our scheduler just has one level of I/O scheduler in the host level instead of two, and our scheduling logic is much simpler than CFQ. About the priority, the processes basically have throughput that are consistent with their priorities. And also, compared with CFQ, our scheduler can support up to 254 priorities value (0 and 255 are kept for other functions), which is much more than CFQ.

However, this graph is one of the best results we can get in our experiments. Sometimes, some processes in our scheduler have opposite throughput relationship regarding their priorities. Doing the same experiments multiple times, sometimes we will get a graph like below:

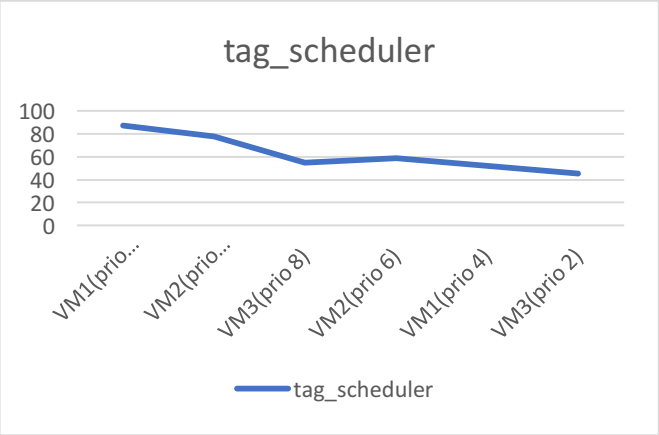


Figure 9

In the above result, the priority-8 process and priority-6 process have a wrong throughput relationship, though the rest processes have correct throughputs. And in some cases, the throughput of any process in one virtual machine is slower than the other two virtual machines, though this doesn't happen frequently. And this situation also happens in CFQ and NOOP. As we debug the VMs, we found that the slow virtual machine don't send any requests to the host machine during some time period while other VMs send their requests. So we think this problem actually comes from the qemu/kvm processes scheduling managed by the host kernel, which is not a problem that can be solved by the I/O scheduler.

After analyzing, we think a more reasonable plugger can help to solve this reverse priority problem. By printing debug information, we found that whenever our scheduler is going to dispatch, it will select the request of the process with lowest virtual counter, but sometimes it cannot find a request in the most suitable process, at that time, it will choose the process with next lowest virtual I/O counter until it actually finds one request. However, there are just a few requests in our queue when the scheduler is about to dispatch, since our scheduler is revised based on Noop scheduler which doesn't have any anticipatory logic, and there is only one simple plugger in the Noop framework for sorting and merging the request. And besides, arm server basically uses flash memory instead of spinning disk, so the speed to handle a request is very high, which makes the requests in the queue even less. Even if the guest VMs are doing intensive I/O, the number of requests in the scheduler is still not big when it is about to dispatch. So sometimes, our scheduler does act like a fifo, which is also the reason why sometimes the priority relationship can be reversed compared to throughput relationship in actual experiments. We think if we want to have a better priority pattern, we have to add a plugger to collect more requests before doing the dispatch in the correct place, though this may sacrifice some efficiency from the host's point of view.

6 FUTURE IMPROVEMENTS

Actually, although our framework can run normally for the most time so we can run the experiments and get the final results, the scheduler is still very buggy and has much space to improve further. There are some problems in the framework that we don't have time to correct or revise. If we have chance, we may do these further improvements in the future.

1. Time to delete the process structure. In our framework, all the information about guest machine processes exist in the host kernel, so when these processes die, we cannot easily know in the host kernel. So right now there is not any mechanism to delete the process structure when the process ends. The process structure is the basic unit that is ordered in the red-black tree, so as the time goes on, if more and more processes use our scheduler, the efficiency will decrease, which influences the throughput. So the scalability is a big problem in our framework. One possible way to solve this problem is that whenever a process ends, it sends a hypercall to inform the host kernel. But this will generate more trapping overhead. Another solution is to revise the code in the scheduler, set up a time for all the processes, if

only the request list of one process becomes empty, start this timer, if a new request comes in before the timer expires, then reset the timer, if the timer expires, then delete the process structure. If the process is not actually dead and sends another request in the future, just regards it as a new process that sends the request. We think the second solution makes more sense since it if easier to implement and doesn't introduce too much overhead.

2. Way to dispatch the request. In the I/O scheduling level, each I/O request structure has an embedded list_head structure called queuelist to link it into different queues. In the existent I/O scheduler like CFQ or deadline-mq, although the requests may be linked in different places, they all use this queuelist to link themselves. However, when I try to use this field to unlink it from its original list and, it will sometimes cause NULL pointer reference error in some irq handler, which we don't quite understand why. We guess that it is about some locks in the request structure or some checking on the existence of all the requests in the softirq handler. So we have to link all the requests into our structures by adding another list_head in the request. So the request is linked in both the original dispatch queue and our queue. We also add another flag member in the request, whenever the request is about to be dispatched, it will check whether this flag is set and decide whether it can be dispatched, if not set, the scheduler will try to dispatch the next request. We guess this may cause some synchronous error and decrease the efficiency to some extent, since the request exists in two different queues at the same time and may be undispatchable sometimes even it is in the dispatch queue. So in the future work maybe we can study the CFQ code further to see why this NULL pointer error exists and solve this problem fundamentally.

3. Set more reasonable plugger, like we have analyzed in the former part, the main reason why this scheduler doesn't work as well as we expect is the number of requests when it does the dispatching. So if we can add a plugger at the right place with suitable timeslice, the throughput pattern can be prettier, and the priority relationship will not be reversed anymore. The perfect pattern of our scheduler should be like this:

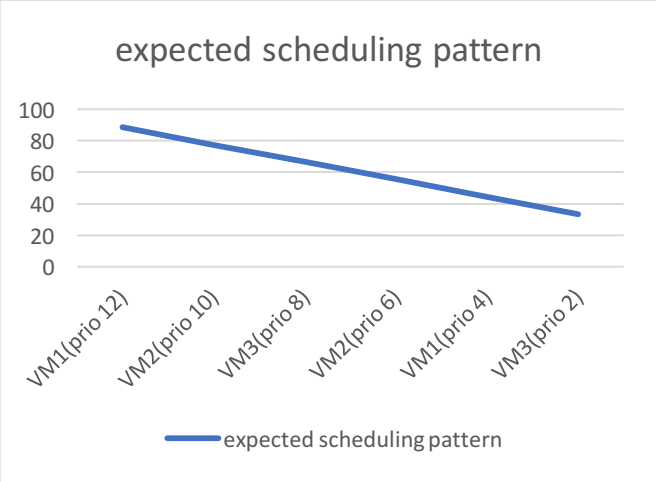


Figure 10

However, it's not very possible in our algorithm design, because we still cannot control the sequence and pattern of the coming I/O requests in the dispatch queue. Maybe a prioritized scheduler based on timeslice (like CFQ) is a better choice.

4. Support more sorts of I/O requests. Right now our framework just supports read request. We thought we can add support for write request, too with some minor revise, but actually there are many kinds of requests like flush and sync. Even for the write request, it is still more complex than the read, since during a write operation, the process just writes the data to a page buffer in the memory and the deamon process like pdflush will do the actual write back. Besides, the logging in the kernel is also a kind of write operation that needs to be considered.

REFERENCES

[1] Bugnion, E., Nieh, J., & Tsafrir, D. (2017). Hardware and software support for virtualization. Synthesis Lectures on Computer Architecture, 12(1), 1-206.

[2] Storage IO Control Technical Overview & Considerations for Deployment. (2010) VMware vCenter Server Performance and Best Practices, https://www.vmware.com/techpapers/2010/storage-io-control-technical-overview-considerat-10118.html

[3] Krishnamurthy, A., & Kowsalya, S. S. Differentiated I/O services in Virtualized environments.

[4] Suo, K., Zhao, Y., Rao, J., Cheng, L., Zhou, X., & Lau, F. (2017, September). Preserving I/O prioritization in virtualized OSes. In *Proceedings of the 2017 Symposium on Cloud Computing* (pp. 269-281). ACM.

[5] Boutcher, D., & Chandra, A. (2010). Does virtualization make disk scheduling passé?. *ACM SIGOPS Operating Systems Review*, *44*(1), 20-24.

[6] Jens Axboe. Completely fair queuing.

[7] Jens Axboe. Noop scheduler.

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

[9] Bellard, F. (2005, April). QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (Vol. 41, p. 46).