2016 IEEE 9th International Conference on Cloud Computing

MIST: Mitigating Host-side Interference for Storage Traffic in Virtualized Data Centers

Hee Won Lee and Moo-Ryong Ra
AT&T Labs Research

***Abstract*—In today's data centers, storage hardware is getting virtualized and shared across multiple tenants. The storage virtualization is expected to significantly increase resource utilization, and consequently provide infrastructure providers with tremendous cost savings. In a shared environment, however, demanding tenants can severely impact the IO performance of co-located tenants. Therefore, in order to properly materialize expected benefits, it is important to provide virtualized IO traffic with performance isolation and a certain level of assured performance. Unfortunately, most prior work typically provided partial solutions by focusing on a single performance bottleneck, e.g., IO scheduler. In this paper, we present the design and implementation of MIST that can significantly mitigate the impact of IO performance interference. To achieve the goal, we devise a novel interference detection metric, I_{cpu}, and use a collection of control mechanisms that exploit the knobs provided by modern operating systems. Through the evaluation based on our MIST prototype, we demonstrate that MIST can provide effective protection for IO traffic against a wide range of interfering workloads. Moreover, we show that MIST has low performance overhead for its control actions.**

I. Introduction

The demands for cloud storage space is rapidly growing. A market forecast report, for instance, predicted that storage traffic related to personal content is expected to increase from 1.7 EB (ExaByte) in 2012 to 20 EB in 2017 [1]. To meet the future demand, both industry and academia have paid attention to virtualization technologies. With virtualization, compute resources and storage space can be shared across multiple tenants and the resource will be provisioned to tenants much faster than before, e.g., in the order of minutes/hours rather than weeks/months. Therefore, cloud service providers anticipate the higher resource utilization of underlying infrastructure.

When the infrastructure is virtualized, however, it is important for the providers to ensure strong performance isolation among tenants because they can interfere one another in a disastrous way. Moreover, today's cloud providers often oversubscribe system resources to maximize overall utilization [24], [19], which makes this problem even more difficult. At the same time, guaranteed bandwidth and/or latency is also important for many performance critical applications.

Storage traffic, the focus of this paper, has more complex dynamics compared to network traffic. Once IO requests are generated inside a VM, they will travel across many software layers including the guest operating system's IO stack, the hypervisor, the host operating system's IO stack, and then go through the networking stack if the storage medium is remotely placed. Each layer consumes system resources – CPU, IO scheduler, and network – and these resources are shared across multiple tenants. If not carefully managed, the co-located IO traffic will be interfering one another and cause significant performance degradation. Numerous evidences can be found in the literature [23], [11], [14], [12], [32], [2], [7], [22], [28].

Then how should we deal with the interference problem for storage traffic in today's data centers? It is a very challenging problem as identified by many researchers. One of the important reasons is that the design principles of modern operating system components, e.g., process scheduler, have evolved in a way that encourages the higher utilization of system resources rather than dedicating resources to a certain process. In a cloud setting, this makes the providers inherently hard to proactively reserve CPU/IO/network resources for a specific tenant. When a proactive reservation is not possible for a tenant, an alternative approach may take a reactive adaptation strategy based on runtime characteristics of the system. However, it requires two mechanisms: a) a mechanism to detect the resource contention and b) a mechanism to eliminate the interference without affecting the reservations of other VMs in the same host. What makes the task more difficult is that there are no explicit translation rules between application metrics (IOPS/latency) and system resources (CPU, IO scheduler, interrupts, etc.).

Unfortunately, the state-of-the-art techniques in this problem space either focus on a disk scheduler [11], [12] or assume that there already exists a mechanism to provide a guaranteed performance for storage traffic [32]. For the former, if interference is caused by a different system resource, e.g., other than disk scheduler, it is not clear how they can protect the traffic. The latter case is specific to a proprietary platform and its protection mechanisms are not exposed to the community.

In this paper, we devise a collection of techniques that can be used to mitigate the impact of interference for storage traffic and present the design and implementation of MIST. MIST can alleviate contentions on multiple system resources such as CPU, storage block, and network. For CPU contention, MIST accurately detects the interference using a carefully designed metric, called I_{cpu}, and prioritizes VMs using the realtime scheduler to eliminate the contention. In case of the block layer, MIST proportionally allocates resources using the Linux CFQ IO scheduler available in Linux. Moreover, MIST introduces a mechanism to reduce the impact of network interrupts against storage traffic by leveraging CPU affinity in order to segregate CPU cores for network interrupt processing from CPU cores for scheduling vCPUs of IO generating VMs.

The paper makes the following contributions. First, we explored the problem space and investigated the IO dynamics

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.42

IEEE computer society

with/without resource contention. Based on the observations, we devise a novel metric, I_{cpu}, that can accurately detect interference and compare it to an existing metric such as VM scheduling latency (§III). Second, we develop a set of control mechanisms in order to properly handle contention in each system resource. For instance, we use real-time (RT) scheduling to mitigate the impact of CPU interference and control CPU mapping for interrupt handling to alleviate the interference caused by network traffic (§IV). Third, we describe the design and implementation of MIST system and present evaluation results based on the prototype implementation (§IV and §VI).

II. Goal and Scope

Goal. Our goal is to achieve the highest possible utilization of resources in virtualized data centers, while still providing QoS guarantees to VMs of interest. If the infrastructure provides guaranteed performance, it is naturally desirable to oversubscribe resources, i.e., accommodate as many VMs as possible in a given host. In practice, a today's virtualization platform provides a set of knobs to oversubscribe resources. As an example, one can configure more number of vCPUs than the number of physical cores available to the machine. In practice, the ratio of physical cores to vCPUs is ranging from 1:6 to even 1:16 [24], [19]. While the oversubscription of system resources is maximizing resource utilization, it also makes performance isolation and QoS guarantees even more important. Under the oversubscribed host machine (the machine where VMs are running), we consider that two types of VMs could exist at a given time; VMs with IO throughput reservations and/or VMs that have no reservations. The former needs a protection from demanding VMs co-located on the same host. The latter runs as a best-effort manner and will tolerate some performance degradation when interference occurs.

Scope. For storage system configuration, we assume a virtualized data center environment where compute VMs are physically separated from the location of storage volumes. These volumes are typically connected through SAN (Storage Area Network) technologies such as iSCSI, Fibre Channel, etc. Under the configuration, the storage path will have complex dynamics because it passes through not only many hardware- and software-layers on the host but also network, middleboxes, remote storage system's front-end, and reaches to the storage backend media. Each layer could be shared by multiple tenants. In this paper, our focus lies in mitigating the impact of host-side interference and leave end-to-end QoS enforcement as future work. In terms of performance metric, this paper particularly focuses on IO throughput.

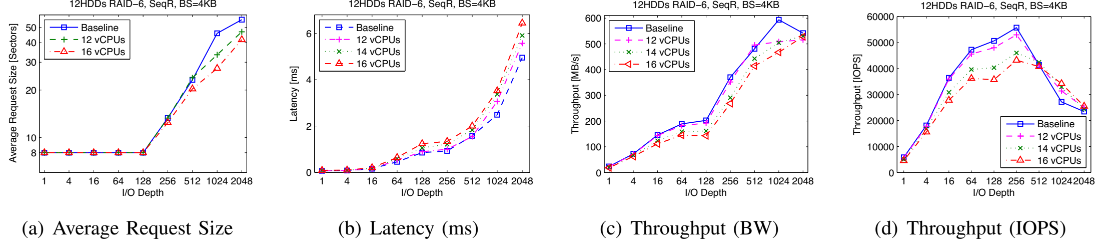
Our Position. When we try to maximize system utilization (and consequently to reduce cost) by oversubscribing resources, the importance of QoS enforcement becomes more salient. In other words, it is generally desirable to run as many VMs as possible at a given host as long as they do not violate QoS requirements. However, the design goal of modern operating systems, e.g., Linux, which are popular in data center environments, deviates somewhat from QoS provisioning. Rather than allocating dedicated resources to a few processes, it evolves its control mechanism to balance the loads across multiple processors and/or to get higher system utilization. For example, Linux process scheduler can preempt running processes in order to perform load balancing operation. MIST tries to carry out QoS enforcement in such an environment. Under this circumstance increases the importance of dynamic detection of interference and proper isolation mechanisms (§III and §IV). In this paper, we aim to deliver a practical solution to mitigate the impact of interferences in the context of oversubscribed cloud environment.

III. IO Dynamics and Resource Interference

In this section, we conduct a measurement study to better understand the problem space and discuss how IO performance metrics are affected by workload changes and resource contention. We devise a novel metric that can more accurately detect CPU interference than existing alternatives, e.g., task scheduling latency provided by perf tool in Linux.

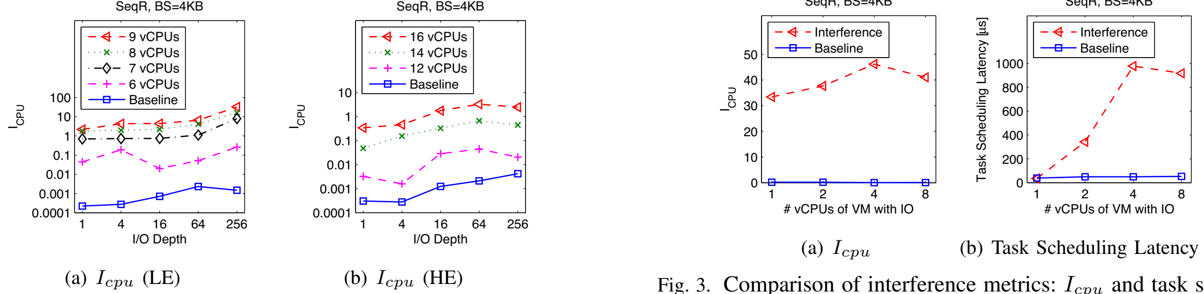
Measurement Setup. For the measurement study, we use two different machines that can capture low-end consumer class servers (LE) and high-end enterprise class servers (HE). LE server is a Dell PowerEdge R210 server with 8 physical cores – Intel Xeon CPU X3460 2.80 GHz, a quad-core CPU with hyperthreading – and 16 GB of main memory and 8 MB cache. The machine is backed by a Samsung 830 Series 128 GB SSD disk with 256 MB built-in cache. HE server is a Dell PowerEdge R820 with 64 physical cores – Intel Xeon CPU E5-4620 2.2GHz, 4 x 8-core CPUs with hyperthreading – and 128 GB of memory with NUMA (Non-Uniform Memory Access) architecture. Each cell (or socket) has 256K, 2M, and 16M of caches for L1, L2, and L3 respectively. A Dell PowerVault MD3600f disk array with 12 HDDs is attached to the host via 8Gb Fibre Channel. The 12 magnetic hard drives are configured as RAID-6. The HE machine additionally has three local hard disks configured as RAID-5. For host operating system, we use Ubuntu server 12.04 LTS. The KVM hypervisor with virtio IO driver is set for all experiments. In each experiment, there are one VM generating IO traffic using the FIO workload generator [10] and separate VMs interfering the IO traffic. FIO is a widely used IO load generator and provides a set of control knobs: IO type, block (or request) size, the number of jobs, the number of concurrent IO requests (called IO depth), control r/w ratio, IO engine, e.g., sync or async, etc. It also provides related statistics such as throughput (in IOPS and bandwidth) and latency. To collect data, we use several system monitoring tools available in Linux (iostat, top, htop, mpstat, pidstat, and sar.).

For CPU interference experiments, we measure IOPS (IO operations per second) and bandwidth (Megabytes per second). A VM generates a sequential read traffic with different intensity, i.e., varying number of IO depth. Note that the host operating system controls the IO request size based on the IO demand internally. As shown in Fig. 1(a), if the workload exceeds a certain threshold, the operating system aggregates incoming requests and batches them in order to maximize bandwidth usage. This control decision trades off latency



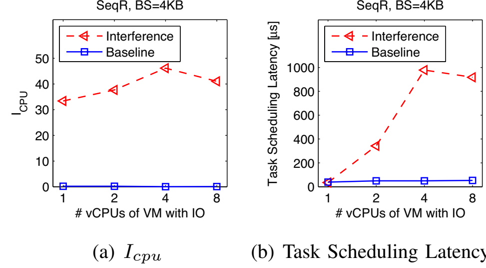
(a) Average Request Size (b) Latency (ms) (c) Throughput (BW) (d) Throughput (IOPS)

Fig. 1. IO dynamics with/without CPU Interference in HE (of 16 physical cores in a NUMA cell). "Baseline" means that there is no interfering workload when IO operations are performing. "n vCPUs" means that the interfering workload is using n vCPUs fully with $n \times$ 100% CPU utilization. Throughput in IOPS and BW could be degraded up to 40%.



(a) I_{cpu} (LE) (b) I_{cpu} (HE)

Fig. 2. CPU interference metrics (I_{cpu}). Performance degradation was not found until 6 vCPUs' interference in LE and 12 vCPUs' interference in HE.



(a) I_{cpu} (b) Task Scheduling Latency

Fig. 3. Comparison of interference metrics: I_{cpu} and task scheduling latency. "Baseline" means no interference case.

against increased bandwidth usage as shown in Fig. 1(b) and Fig. 1(c) respectively. In Fig. 1(d), IOPS increases until IO depth reaches 128∼256 and decreases beyond those points. Those transition points can be different along with the different degree of interference. Similar observation is made with an experiment with LE server [1].

CPU Interference.

1) Methodology & Baseline Results.: To understand the interfering behavior of VMs, we use our own workload generator. Our custom workload generator executes 'while' loop while generating one random number per cycle. It can take the number of vCPUs as input and saturate the capacity of the configured number of physical cores. To see the impact of interference, we increase CPU loads generated by the workload generator to the point that fully saturates the capacity of available physical cores, i.e., 800% for LE and 6400% for HE. We use all eight CPU cores available for the experiment on LE server. However, for HE server, we take one NUMA cell (16 physical cores) among four for our experiments[2].

Figure 1 shows that CPU contention can degrade IO performance up to 40% in HE. (The performance degradation in LE was similar.) This result demonstrates how CPU interference can degrade IO performance. When malicious tenants can act just like our load generator, another tenant could experience a significant decrease in IO performance.

[1]In LE, transition points spread out IO depths ranging from 16 to 64.

[2]In NUMA architecture, inter-cell interference due to the CPU resource is rarely occurred since each cell has a dedicated memory hierarchy, i.e., its own caches and memory space.

2) Detecting Interference with I_{cpu}: Accurately detecting interference takes an important role in a control system such as MIST. Inaccurate policy enforcement may result in inefficient resource allocation, so the overall system resources could be underutilized. To detect interference, we developed a novel metric that can detect CPU interference accurately. The metric, called I_{cpu}, can be described as follows.

$$I_{cpu} = \frac{N_{nvc}}{N_{vc}} \quad (1)$$

where N_{nvc} is the number of non-voluntary (or involuntary) context switches and N_{vc} is the number of voluntary context switches.

a) Why I_{cpu} Works: If a VM makes sufficient IO requests, the number of voluntary context switches are high due to the frequent invocation of IO-related system calls. When performance degradation happens, a running process is more likely to yield CPU involuntarily. During our empirical evaluation (Fig. 2), we realized that the ratio between two types context switches (i.e., I_{cpu}) reflects the existence of interference very well. We can experimentally find a threshold on I_{cpu} to detect interfering behavior (we use 0.05 as our threshold for our prototype implementation). When using the metric, we exclude the cases if the total number of context switches ($= N_{vc} + N_{nvc}$) is too small (i.e., the VM is idle) where I_{cpu} falsely indicates the interference.

b) Why not use existing metrics?: Some hypervisor implementations provide explicit information on compute resource contention such as VM scheduling latency, i.e., the amount of time a runnable VM's vCPU remains queued before being dispatched to execute on a physical processor (a.k.a. "ready time"). VMWare provides information on ready-time and the recent version of Hyper-V by Microsoft provides a

similar counter. Similarly, Linux provides task scheduling latency. Since our environment is based on Linux, we compared I_{cpu} with task scheduling latency and the result is reported in Fig. 3. We found that task scheduling latency was not able to detect a crucial case – VM with one vCPU – while I_{cpu} was robust irrespective of the number of vCPUs. VMs with a single vCPU are perhaps most common in cloud environments. In this experiment, we vary the number of vCPUs for an IO generating VM, while running an interfering VM with 8 vCPUs that utilizes almost all the CPU cycles ($\approx$ 800%).

Bottleneck at IO Scheduler. IO scheduler could be another source of bottleneck for block storage performance in a virtualized environment. Block storage volumes and VMs can form either one-to-one or one-to-many mapping. For the former case where a single VM is connected to a given volume, there are no contenders that share IO scheduler for the volume on the host. Thus, the interference will occur out of the IO scheduler's regime. For the latter case where multiple VMs are sharing a volume, multiple VMs may contend one another on a single IO scheduler. In this case, the interference needs to be explicitly handled on the scheduler (§IV).

Interrupt Processing. When the Linux kernel processes hardware interrupts generated by network or block IO operations, it is next processed by softirq mechanism [35]. However, network operations (send/recv) have a higher priority than block IO operations, and thereby IO performance will be greatly affected if a large number of network requests are co-located with the same CPU. Since the priorities of softirq entries are static (hard-coded in the Linux kernel), there is no easy way to change it out of box. We developed a practical mechanism (§IV); in this paper, however, this problem might be further mitigated using more involved techniques such as [21], [18].

Factors that are not included. In this study, we exclude network outside of the host since our focus is host-side IO traffic protection. Apparently, when enforcing end-to-end IO bandwidth reservation, the system requires a mechanism that can ensure protection on the network between the host and the storage backend. Regarding the issue, there exists an extensive body of work in SDN (Software-Defined Networking) literature and will be complementary to our work for future development. Additionally, IO performance could be restricted by other hardware components such as interconnects like PCIe, available memory bandwidth, etc. For these components, more sophisticated control might be necessary to achieve similar impact. For instance, Majo et al. [20] argues that, in order to optimize performance due to memory bandwidth, the system needs to handle data locality and cache contention simultaneously and implements a user-mode extension to Linux scheduler to achieve the goal. MIST system does not explicitly address potential issues on these components.

IV. System Design

Architecture Overview. Figure 4 shows the software architecture of MIST system. It is composed of four components: *REST-API*, *admission controller*, *enforcer*, and *performance monitor*. The *REST-API* component will receive a REST API request from a MIST user. The request contains a storage policy, e.g., bandwidth reservation request. Then the *admission controller* will decide whether to accept the request or not based on the current capacity and reserved resources. The admission controller will look up its database to see whether there is remaining capacity and make a decision. To make a correct decision, MIST system accounts available system resources. If the request is admitted, the *enforcer* will actually enforce the rule to a appropriate place, e.g., set an upper limit to a specific block device in cgroup pseudo file system, etc. Lastly, the *performance monitor* will periodically collect various performance statistics and, based on the information, it may trigger control actions to the *enforcer* if it observes both lower performance (less than the reserved amount) and resource contention (interference) simultaneously.

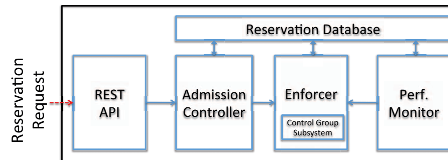


Fig. 4. MIST System Architecture

In many cases, interfering behavior could be transient. When there is no interference, MIST encourages using default control mechanisms built in Linux. Only when the system does not meet QoS requirements due to the interference, MIST needs to take an action to mitigate the interference. The *performance monitor* periodically samples I_{cpu} and the number of interrupts/softirq. If I_{cpu} deviates from a threshold set by the platform provider, the system will take a relevant control action appropriately. The details are discussed later.

Eliminating Interference on CPU Scheduling. The main goal of MIST CPU control is to minimize the impact of performance interference among multiple tenant VMs on the same host. In MIST, the *performance monitor* will trigger a CPU control action to the *enforcer*. Then the *enforcer* prioritizes the VM's task by switching its scheduler from CFS (default Linux scheduler) to RT (real-time) scheduler. Linux provides two types of RT scheduling, FIFO and RR (round-robin). We use RR algorithm (SCHED_RR) because there might be multiple VMs that the MIST system needs to protect at a given time. We demonstrate that the use of RT scheduler is actually very effective and can actually eliminate the interference on the scheduler side (§VI).

Why real-time (RT) scheduler: To isolate the impact of CPU interference, it is tempting to use cgroup's proportional sharing feature. However, in our measurement study, we found that proportional CPU time division does not work as expected especially under heavy loads. The reason was that, under the interfering condition, a demanding process (or task) consumes most of the CPU time and even if we set a very high value in CPU share, we were not able to eliminate the impact of interference. This is due to CFS scheduler's time slice management and interaction with load balancing, etc. [33]. The CPU pinning is seemingly an alternative way to resolve the

interference issue. Linux already exposes a set of API for users to pin their process(es) to particular CPU(s). vCPU is a regular Linux process on a host, so we may use this mechanism. However, it significantly limits the flexibility of CPU resource allocation, which is the main purpose of the oversubscription for better resource utilization. Similar observations are made in [15].

The use of RT scheduler offers an enough isolation to VMs while it still allows us to maximize the aforementioned flexibility. It enables MIST to provide robust isolation against CFS jobs (§VI). Unlike pinning, it will not cause lower utilization of CPU resource; when prioritized VMs belonging to RT scheduler are not running, CFS will schedule jobs without compromising its design goal.

Proactive Reservation on IO Scheduler. MIST needs to handle two types of volume configuration. As discussed earlier (§III), when there is a one-to-one relationship between a VM and a volume, interference will not occur on the IO scheduler since it is dedicated to the path. When the relationship is many-to-one, i.e., multiple VMs on a host share a remote volume (shared storage volume), many VMs share an IO scheduler and they could compete each other. Thus, MIST needs to explicitly control active IO traffic to ensure an assigned bandwidth. To achieve the goal, MIST relies upon CFQ [4] available in Linux kernel. The CFQ provides a way to implement proportional bandwidth(or IOPS) sharing. The mechanism is conceptually similar to classic weighted fair queueing and MIST ensures minimum throughput by setting weights (shares) for each VM and volume pair. One technical challenge is to accurately estimate the capacity of the IO path. Accurately accounting the capacity of a given storage path is a difficult problem since it depends on many factors such as workload, the behavior of other tenants who share the storage media, etc. One may use an advanced technique to estimate a storage IO capacity as in [2]. In this work, we measure the worst-case IO throughput as the capacity of a given storage path, i.e., the number of IOPS for random r/w workload.

Mitigating the Impact of Competing Interrupts. Similar to the use of RT scheduler, it is tempting to have a functionality to explicitly prioritize block IO requests against other types of operations, e.g., network. There exist related efforts in the research community [31], [21], [18]. However, since currently the priority of softirq jobs are static in the kernel implementation, changing block IO operation's priority requires kernel modification and more importantly the implication of changing softirq priority is not well-studied. MIST system does not have a mechanism to explicitly boost block IO interrupt (softirq) priority against network operations. Instead, MIST takes an alternative approach and use CPU affinity such that a set of CPU cores that process network requests are mutually exclusive with the CPUs that process block IO requests. We can achieve this functionality by allocating a block IO operation's IRQ to a specific CPU core and make interference VMs' network traffic use remaining CPU cores other than the core used by block IO. In most cases that we explored, this strategy was able to almost eliminate the interference. Apparently, this area has a room to be significantly improved as pointed out in the foregoing discussion.

V. IMPLEMENTATION

Control Groups. Control Groups [5] (cgroup), is a Linux kernel feature that enables group scheduling of system resources. It can aggregate processes into a group and associate control policies. MIST system utilizes two subsystems of cgroup infrastructure: *cpu and blkio*. Each cgroup subsystem has a different set of tunables, e.g., set an upper limit and/or weights, and provides useful statistics. As an example, MIST uses RT scheduling parameter *cpu.rt_runtime_us*[3] for its CPU control. By choosing a proper RT runtime *cpu.rt_runtime_us*, MIST system can explicitly allocate more CPU resources to a given storage path. Suppose that an RT period is 1000ms and the total RT runtime is 200ms. In this environment, if a VM requires 10ms of RT runtime to support 5000 IOPS, MIST system can run up to 20 VMs with a guaranteed performance of 5000 IOPS by setting *cpu.rt_runtime_us* as 10ms (see §VI-A for real data). Additionally MIST also uses block IO parameter *blkio.weight* for its block layer control.

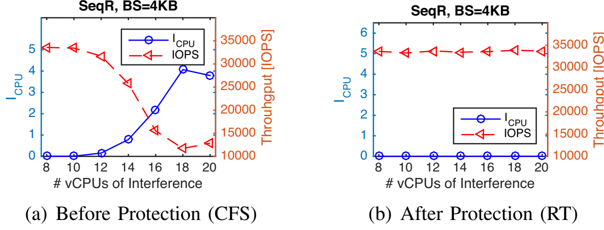
MIST Prototype. We implement a prototype of MIST system on the Ubuntu 12.04 LTS server edition. It is written in python. Each component of Fig. 4 is implemented as a POSIX thread and runs independently. For the *REST-API*, our API interface is similar to that of OpenStack [25], i.e., the requests are based on HTTP protocol and response messages are encoded in the JSON format. The *enforcer* exploits two cgroup subsystems – *cpu* and *blkio*. Each subsystem provides slightly different semantics for users to control the system resources. For instance, *blkio* subsystem provides mechanisms to ceil (or throttle) the traffic and control weights for proportional sharing of scheduling opportunities. The *enforcer* uses *cpu* subsystem to change the process scheduler for reducing CPU interference, and *blkio* to reserve bandwidth on IO scheduler.

VI. EVALUATION

In this section, we evaluate our prototype implementation. The primary goal of the evaluation is to answer the following questions: a) Can MIST protect IO workloads from a variety of interfering conditions? b) Will MIST perform well under real-world scenarios? c) How much overhead will be imposed?

For the experiments, we use the same machines, i.e., LE and HE, described in Section III. Additionally, we use mid-level enterprise class servers (ME). ME server is a Dell PowerEdge R710 server with 16 physical cores (Intel Xeon CPU E5520 2.27 GHz, an 8-core CPU with hyperthreading) and 32 GB of memory and 8 MB cache. We first perform micro-benchmarks on each control point. Then we conduct a macro-benchmark where we apply more realistic scenarios MIST. Lastly, we present results on overhead imposed by MIST system.

[3]In order to use cpu.rt_runtime_us, Linux kernel should be recompiled with the option CONFIG_RT_GROUP_SCHED=y.



(a) Before Protection (CFS) (b) After Protection (RT)

Fig. 5. CPU control: (a) I_{cpu} reflects the degree of interference well. (b) RT protection mechanism eliminates CPU interference.

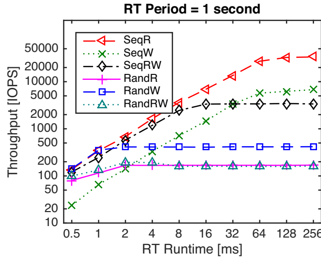


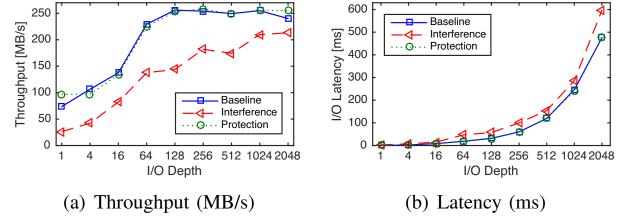
Fig. 6. A profile of IOPS vs. RT runtime

A. Micro-Benchmark

Handling CPU interference. Our goal of CPU control is to reduce the impact of scheduler interference by prioritizing a target VM. To show the effectiveness of our metric (I_{cpu}) and control mechanism, we attempt to answer three questions.

1) How effectively can I_{cpu} detect CPU interference?: To answer this question, we created a VM with 2 vCPUs in ME server; it generates sequential-read IO traffic. Next we created an interfering VM with 20 vCPUs. This VM interferes with the IO-generating VM by increasing the number of worker threads from 8 up to 20; each worker thread uses 100% of one CPU core. Fig. 5(a) shows that as CPU interference increases, IOPS decreases. Note that our CPU interference metric, I_{cpu}, captures the degree of CPU interference very well. When our RT protection mechanism takes an action (i.e., when $I_{cpu} > 0.05$ from §III), the CPU interference was almost eliminated (Fig. 5(b)).

2) How should we set RT parameters?: We set up two VMs in ME server: an IO-generating VM with 2 vCPUs and an interfering VM with 20 vCPUs. Then we executed an IO-generating VM with an RT scheduling policy (SCHED_RR). While changing the VM's RT runtime (see §V), we measured IOPS for diverse types of IO pattern. Fig. 6 shows that IOPS increases in a nearly linear relation with RT runtime for each IO type. The profile shown in Fig. 6 is used by MIST system's *enforcer*. Whenever MIST needs to take an action to eliminate CPU interference, the *enforcer* changes the VM tasks' scheduling policy to SCHED_RR, and then finds the RT runtime matching the requested IOPS from Fig. 6. For example, if a requested IOPS is 2000 for a mixed sequential read and write (i.e., SeqRW), the *enforcer* chooses 8ms for the RT runtime so that the IOPS of 2000 is achieved. In this fashion, our RT protection mechanism guarantees IOPS throughput. Since the combination of a host machine and a storage device has its own performance characteristics, it is required to profile IOPS vs. RT runtime for each combination.



(a) Throughput (MB/s) (b) Latency (ms)

Fig. 7. Reducing CPU interference via RT-sched (Hotmail workload)

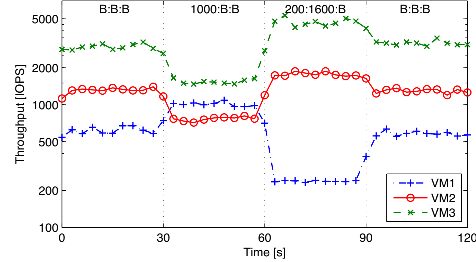


Fig. 8. Guaranteed IOPS

3) Can MIST protection mechanism handle real-world workloads?: At the beginning, an IO-generating VM with 4 vCPUs was running with default CFS scheduler in LE server (*Baseline*). We borrowed real-world IO workload characteristics from a Hotmail service [32]. For this experiment, the VM generates INDEX workloads where SeqR/RandW ratio is 75%/25% and IO block size is 4/64KB. Then we added one interfering VM with 16 vCPUs running our CPU load generator (*Interference*). Then, we enable our protection mechanism (*Protection*). The result (Fig. 7) shows that MIST's mechanism actually works for real-world workloads, i.e., throughput and latency are recovered to those of the *Baseline*.

Bandwidth reservation on IO scheduler. In this subsection, we evaluated our control mechanism on the block IO scheduler. For this experiment, we configured a shared storage volume. We created three VMs and connected them to a volume in a remote machine. The volume is shown as a block device to the host. As a result, all three VMs (i.e., VM1, VM2, and VM3) are attached to a single IO scheduler configured to the block device. For workloads, we tried both uniform and mixture of different workloads. Due to space limitations, only mixed case is presented. Specifically, VM1 has sequential rw, 10%/90% rw ratio, and block size 64KB. VM2 has sequential rw, 75%/25% rw ratio, and varying block size (50% 4KB, 50% 32KB). VM3 has random rw, 50%/50% rw ratio, and block size 8KB. We use an SSD in LE for this experiment. The result is presented in Fig. 8. At the beginning, all three VMs were running with no reservations (Best-effort). Every 30 seconds, we change reserved IOPS for each VM. At 30s, we reserved 1000 IOPS for VM1 and two other VMs still had no reservations (1000:B:B). At 60s, we changed the reservation of VM1 to 200 IOPS and make a new reservation for VM2 as 1600 IOPS (200:1600:B), etc. Overall, the reservations are met correctly whenever they are enforced to any VM. We also had tried a couple of different workload mix and conducted similar experiments in HE server. The results were

qualitatively similar. We omitted the results for brevity.

Mitigating the impact of network interrupts. MIST system also has a mechanism to reduce the impact of network traffic co-located on the same host. For brevity (to reduce the redundancy), we postpone the discussion on our mechanism to the next subsection (§VI-B).

B. Macro-Benchmark: Varmail with Transcoding Service.

Next we conducted a macro-benchmark using more realistic workload and interferer. For IO workloads, we use varmail workloads in Filebench software [9]. The varmail workload performs delete/create/append/fsync/read/open operations on the file. It spawns 16 threads, deals with 1000 files, and uses the 16k IO block size. When running inside a VM on LE server, the varmail generates approximately 4000∼6000 IOPS. We provision one VM with one vCPU for varmail workloads and allocate 500MB of memory. As for interfering VMs, we chose to use VLC media player [34]. The VLC software can execute as a streaming client or server. We set up six VMs and use one for a streaming server and the other five for streaming clients. The server streams 720p HD videos encoded with H.264 codec. This configuration nicely captures a mixture of compute, network, and IO operations occurring all together in the same hardware. All VM sizes were the same as the first VM. The experiment is performed in LE server.

Figure 9 shows the result. At the beginning, the varmail VM was running alone without any competing workloads and shows about 4000∼6000 IOPS. Next we made a reservation, i.e., 3000 IOPS. After 60s, all other VMs started to run and the varmail VM's performance is greatly affected (degraded to 1700∼2000 IOPS), which is apparently below the reserved bandwidth. MIST's *performance monitor* detected the interference since I_{cpu} value surges to more than 25, which is a way bigger than our threshold (0.05). Now *enforcer* triggered two control mechanisms of MIST system. The first mechanism that took an action was CPU control. Both VLC's video data streaming and media file decoding are CPU-intensive operations [17]. MIST changes the scheduler of VM1 (varmail) to RT scheduler and the interference is eliminated as shown in Fig. 9(b). However, it was not sufficient alone since a large volume of network requests were being generated by VLC server/clients. To mitigate the impact of network traffic, MIST partitioned CPU cores so that network and block IO interrupts are handled in a disjoint set of CPU cores. Specifically, in this experiment, MIST allocates CPU0 for block IO interrupts and other CPUs (i.e., CPU1-CPU7) for network interrupts. Fig. 9(c) illustrates the impact of this control. It first shows that the co-located interrupts generated by network and block IO operations interfered each other (see 60s∼120s). Then it demonstrates the effectiveness of our control; it is clear that the separation of different types of demanding interrupt requests actually helps both workloads. As a result of the control action at 120s, the number of softirqs for both block IO and network operations are boosted. With these control actions, the IOPS is recovered close to the original state and consequently meet the QoS requirement, i.e., 3000 IOPS that we reserved before.

C. MIST System Overhead

In this subsection, we measure and present performance overhead imposed by MIST system. First of all, MIST system's overhead comes from the use of cgroup and its associated scheduler. The *enforcer* uses the cgroup interface and scheduling infrastructure provided by the Linux OS. We compare the performance penalty of using CFQ against the default block IO scheduler of Ubuntu 12.04 LTS, i.e., deadline scheduler. (The graph is omitted for brevity.) When backed by high IOPS storage media such as SSDs[4], the deadline scheduler outperforms the CFQ scheduler. The throughput (IOPS) difference could be up to 30% and latency difference is about 23ms ($\simeq$ 45.8ms of CFQ $-$ 22.7ms of Deadline on average) for read operations and 14ms ($\simeq$ 51.2ms of CFQ $-$ 37.6ms of Deadline) for write operations, when the three VMs were competing each other. The result is also roughly consistent with other research efforts [16], [8]. Although these performance differences are non-negligible, we would argue that the problem strictly comes from CFQ implementation and can be improved separately as in [29], [26]. For *cpu* subsystem of cgroup, we do not observe any performance degradation due to the scheduler change. Finally, MIST system needs to maintain system statistics, N_{nvc} and N_{vc}, in order to calculate I_{cpu} value. The MIST *performance monitor* will collect the data every one second. It consumes negligible CPU cycles.

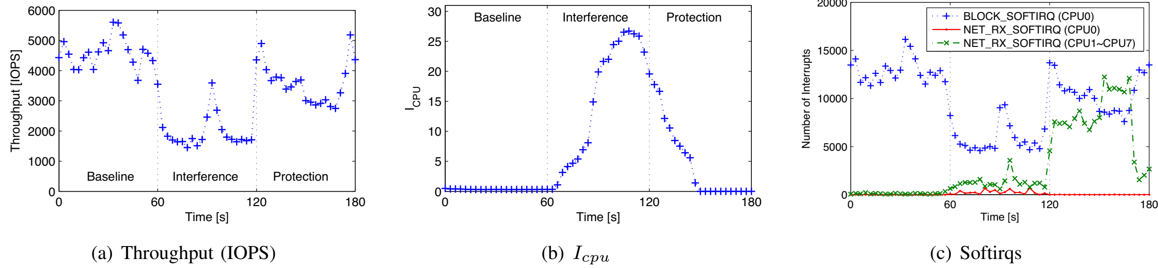
VII. Related Work

Many prior work on IO scheduler is based on proportional allocation or sharing of IO resources. Many of these are based on classic weighted fair queueing (WFQ) and many variants of WFQ have been developed in various context. Among them, mClock [11] proposes a tag-based queue control mechanism that can protect IO traffic from other tenants sharing the same hardware. Stonehenge [14] implements a custom disk scheduler that specifically provides QoS functions to a virtual disk. Unlike MIST, most of them modify the IO scheduler to achieve the goal and does not address the contention from other resources, e.g., CPU and interrupts. There are more work in this context [3], [27].

Another body of work aims to realize end-to-end bandwidth reservation mechanism. Gulati et al. [12] proposes tree-based data structure to enforce distributed IO bandwidth reservation. In addition, IOFlow [32] presents an SDN-like system where a centralized policy controller takes a high level storage policy and directs host-side clients and servers to enforce the rules. Pulsar [2] aims to provide end-to-end performance isolation by introducing VDC concept. Through the VDC, the system provides a tenant with aggregated bandwidth guarantee, introducing an intermediate throughput metric. These systems are complementary to our work since our system could provide host-level enforcement for such systems.

Next body of related work lies in the area of CPU/memory/network/IO sharing and prioritization. Silva et al. [30] proposes a cgroup-based VM performance isolation mechanism.

[4]We also performed similar experiments on HDDs, but it does not show meaningful difference between the two schedulers.



(a) Throughput (IOPS) (b) I_{cpu} (c) Softirqs

Fig. 9. Varmail: the interference has been successfully controlled.

However, they show that their mechanism fails to isolate IO performance (which is our paper's main focus). Das et al. [7] deals with CPU sharing problem in the context of relational database applications. Their LDF metric is qualitatively similar to task scheduling latency metric in Linux. Thus, we do not expect for LDF to solve our problem due to the reason discussed in §III. Cucinotta et al. [6] uses RT scheduler to provide performance isolation for a network application and inspired our use of RT scheduler. Gupta et al. [13] implemented the knobs on hypervisor to control aggregated resource consumption although the system is specific to the Xen platform. Moreover, there exists a body of work on reducing the impact of resource contention for NUMA (Non-Uniform Memory Access) architecture [22], [28]. In this paper, we implicitly mitigate the impact of NUMA-related contention problems by boosting a scheduling priority using RT scheduler.

Another set of work focuses on detecting performance interference. DeepDive [23] constructs a model using machine learning algorithm to identify resource contention. Infrastructure providers can trigger appropriate actions, e.g., migrating VMs to a different host. CPI^2 [36] uses the CPI and throttle the source of CPU interference to relieve the impact. The methodologies explored in these papers are complementary to our work and can potentially enhance the detection accuracy of MIST system.

VIII. Conclusion

In this paper, we present a collection of control mechanisms that can help us significantly mitigate the impact of performance interference from other demanding VMs sharing the same host machine. With the control mechanisms, MIST can handle the interference occurred in multiple system resources including CPU, IO scheduler, and network. Our novel metric, I_{cpu}, can effectively detect the contention on CPU resources, thereby triggering our control actions appropriately. Based on our prototype implementation, we demonstrate that MIST system can handle different types of interfering workloads and a wide range of IO workload types with low system overhead. Lastly we acknowledge helpful comments from Minsung Jang.

References

[1] Cisco Global Cloud Index: Forecast and Methodology, 2014-2019, http://goo.gl/t4qfZ4. Technical report, Cisco, 2013.
[2] Angel et al. End-to-end performance isolation through virtual datacenters. In *OSDI*, pages 233–248. USENIX Association, 2014.
[3] J. C. Bennett and H. Zhang. WF^2Q: worst-case fair weighted fair queueing. In *INFOCOM*, volume 1, pages 120–128. IEEE, 1996.
[4] CFQ, https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.
[5] Cgroups, https://kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.
[6] Cucinotta et al. Providing performance guarantees to virtual machines using real-time scheduling. In *Euro-Par 2010 Parallel Processing Workshops*, pages 657–664. Springer, 2011.
[7] Das et al. Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. In *VLDB*, volume 7, page 12. Very Large Data Bases Endowment Inc., 2013.
[8] A. Elnably, H. Wang, A. Gulati, and P. Varman. Efficient qos for multi-tiered storage systems. In *HotStorage*, 2012.
[9] Filebench, http://sourceforge.net/projects/filebench/.
[10] Flexible I/O tester, https://github.com/axboe/fio.
[11] Gulati et al. mclock: handling throughput variability for hypervisor io scheduling. In *OSDI*, pages 1–7. USENIX Association, 2010.
[12] Gulati et al. Demand based hierarchical qos using storage resource pools. In *USENIX Annual Technical Conference*, pages 1–13, 2012.
[13] Gupta et al. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006*, pages 342–362. Springer, 2006.
[14] Huang et al. Multi-dimensional storage virtualization. In *SIGMETRICS Performance Evaluation Review*, volume 32, pages 14–24. ACM, 2004.
[15] Iancu et al. Oversubscription on multicore processors. In *IPDPS*, pages 1–11. IEEE, 2010.
[16] IBM. Best practices for kvm, 2nd edition. Technical report, 2012.
[17] H. W. Lee, M. L. Sichitiu, and D. Thuente. High-performance emulation of heterogeneous systems using adaptive time dilation. *International Journal of High Performance Computing Applications*, 2014.
[18] M. Lee, J. Lee, A. Shyshkalov, J. Seo, I. Hong, and I. Shin. On interrupt scheduling based on process priority for predictable real-time behavior. *ACM SIGBED Review*, 7(1):6, 2010.
[19] S. D. Lowe. Best practices for oversubscription of cpu, memory and storage in vsphere virtual environments. Technical report, 2012.
[20] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *ACM SIGPLAN Notices*, volume 46, pages 11–20. ACM, 2011.
[21] Modena et al. Providing qos by scheduling interrupt threads. *Organized by the IEEE Technical Committee on Real-Time Systems*, page 53, 2008.
[22] Muralidhara et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *IEEE/ACM MICRO*, pages 374–385. ACM, 2011.
[23] Novaković et al. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX ATC*, pages 219–230, Berkeley, CA, USA, 2013. USENIX Association.
[24] OpenStack Overcommitting, http://docs.openstack.org/openstack-ops/content/compute_nodes.html#overcommit.
[25] OpenStack, https://www.openstack.org/.
[26] Park et al. Fios: a fair, efficient flash i/o scheduler. In *FAST*, 2012.
[27] Popa et al. Faircloud: sharing the network in cloud computing. In *SIGCOMM*, pages 187–198. ACM, 2012.
[28] Rao et al. Optimizing virtual machine scheduling in numa multicore systems. In *HPCA*, pages 306–317. IEEE, 2013.
[29] K. Shen and S. Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *USENIX ATC*, pages 67–78, 2013.
[30] Silva et al. Vm performance isolation to support qos in cloud. In *IPDPSW '12*, pages 1144–1151, Washington, DC, USA, 2012. IEEE Computer Society.
[31] Software interrupts and realtime, http://lwn.net/Articles/520076/.
[32] Thereska et al. Ioflow: A software-defined storage architecture. In *SOSP*, pages 182–196. ACM, 2013.
[33] P. Turner, B. B. Rao, and N. Rao. Cpu bandwidth control for cfs. In *Linux Symposium*, volume 10, pages 245–254. Citeseer, 2010.
[34] VLC media player - VideoLAN, http://www.videolan.org/vlc/index.html.
[35] M. Wilcox. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. In *Linux. conf. au*, 2003.
[36] Zhang et al. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, pages 379–391. ACM, 2013.