IOrchestra: Supporting High-Performance Data-Intensive Applications in the Cloud via Collaborative Virtualization

Ron C. Chiang
University of St. Thomas
cchiang@stthomas.edu

H. Howie Huang, Timothy Wood
George Washington University
{howie,timwood}@gwu.edu

Changbin Liu, Oliver Spatscheck
AT&T Labs, Inc.
{changbl,spatsch}@research.att.com

Abstract

Multi-tier data-intensive applications are widely deployed in virtualized data centers for high scalability and reliability. As the response time is vital for user satisfaction, this requires achieving good performance at each tier of the applications in order to minimize the overall latency. However, in such virtualized environments, each tier (e.g., application, database, web) is likely to be hosted by different virtual machines (VMs) on multiple physical servers, where a guest VM is unaware of changes outside its domain, and the hypervisor also does not know the configuration and runtime status of a guest VM. As a result, isolated virtualization domains lend themselves to performance unpredictability and variance. In this paper, we propose IOrchestra, a holistic collaborative virtualization framework, which bridges the semantic gaps of I/O stacks and system information across multiple VMs, improves virtual I/O performance through collaboration from guest domains, and increases resource utilization in data centers. We present several case studies to demonstrate that IOrchestra is able to address numerous drawbacks of the current practice and improve the I/O latency of various distributed cloud applications by up to 31%.

1. INTRODUCTION

Servicing requests quickly is critical for enhancing user experience of cloud applications. For example, the instant predictions of the Google search system require very short response times (within a few tens of milliseconds) to seamlessly update query results as a user types [14]. The latency of cloud service could incur nontrivial loss in business, e.g., Amazon.com would lose sales by 1% for every 100 ms delay in page load time and a similar test at Google also revealed that a 500 ms increase in displaying the search results could reduce revenue by 20% [28]. More importantly, for such cloud applications, improving the average latency is insufficient because it is the maximum latency that dictates the response time. In other words, the tail of the latency distribution, instead of the mean or median, determines the quality of service for cloud applications [25, 42].

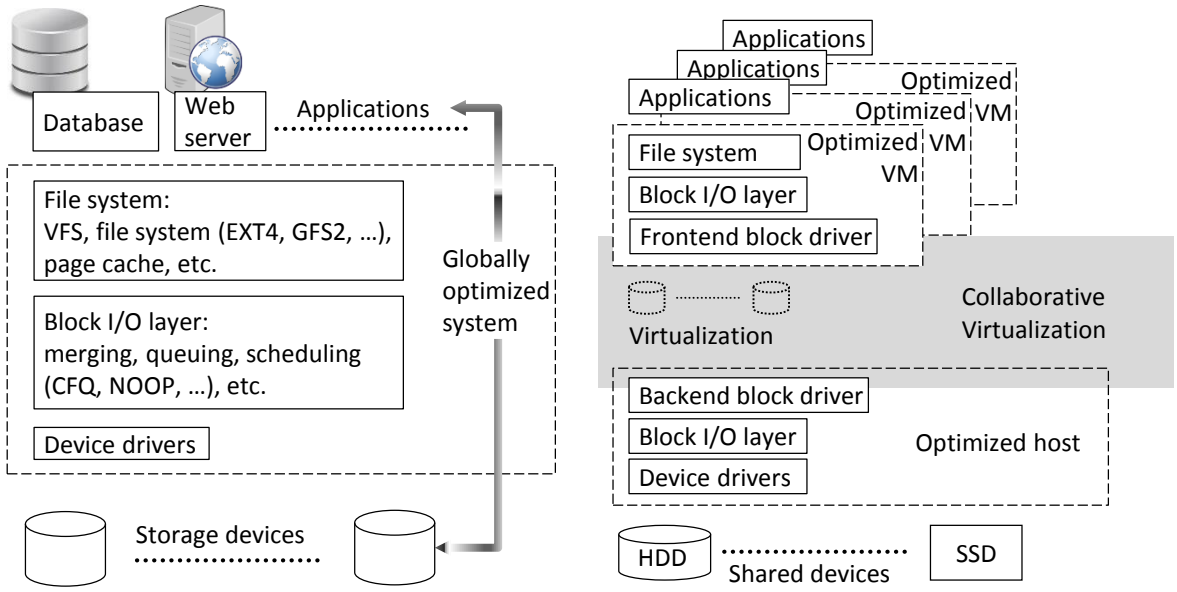
SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807633

The problem of long-tail latency is not new and has been discussed extensively in networked systems [2,35,40]. However, the widespread use of virtualization in cloud services poses a new set of challenges. Cloud applications often rely on a large number of different systems, e.g., a page request on Amazon typically involves hundreds of services with many inter-dependencies [15]. The problem is further exacerbated as the scale and complexity of cloud systems continue to increase. The I/O performance in virtualized systems has been identified as one of the culprits [12, 33], which is also the focus of this work. Because a cloud application that resides in a virtual machine (VM) does not have control over the host operating systems and physical devices, it can experience wide fluctuations in I/O performance even when the resource allocations are unchanged. In other words, a guest VM is normally unaware of changes outside its domain, and the VM monitor (VMM) or hypervisor also does not know the configuration and runtime status inside a guest VM, which in this paper we refer to as the semantic gap between VMs and the hypervisor. The causes of the gap are from both sides - the hypervisor treats VMs as black boxes and vice versa for the VMs.

Traditional computer systems are usually tuned for specific applications when working with a set of devices, as shown in Fig. 1(a). A harmonious status is achieved when the system is configured to suit the current context including application, hardware, background activities, etc. On the other hand, as the virtualization technique abstracts hardware devices, many assumptions no longer hold. Fig. 1(b) demonstrates the block I/O flow from applications in a guest VM to physical devices on a paravirtualized host machine. An I/O request starts its journey from the user space of a guest VM. Then, it travels through the complete I/O stack in the kernel space of the VM, and arrives at the frontend driver of the hypervisor. Paravirtualization utilizes shared memory space and event channels to connect frontend drivers to backend drivers, which will deliver I/O requests to the generic block layer in the host OS. Finally, the I/O request arrives at physical devices through the block device driver. Clearly, system tuning becomes more complex in such virtualized environments.

Fortunately, we have observed that virtualized systems can



(a) Physical machines

(b) Virtualized systems

Figure 1: (a) Physical machines where the systems are tuned for specific applications and I/O devices (b) Virtual machines where I/O performance tuning is challenging

deliver high performance if they are well-informed and properly configured. In this work, we design a new virtualization architecture, *IOrchestra*, to bridge the semantic gaps of system information across the domains, and coordinate them to improve the virtual I/O performance. While many prior works focus on modifying specific virtualization modules or reducing the overheads of virtualization [1, 17, 21, 26, 29, 31, 37], IOrchestra creates a communication channel for system management and keeps the original OS and virtualization design intact. That is, IOrchestra aims to facilitate *automatic collaboration* between VMs and the hypervisor. We have implemented a prototype in Linux and Xen [6]. The major features of IOrchestra are summarized as follows:

- *An automatic collaboration framework*, which makes key performance statistics readily available and makes use of cross-VM knowledge to determine the best system setting. A self-tuning system is desired because manually tuning all VMs is not feasible in the production environment.
- *Flexibility.* IOrchestra is designed for inter-domain communication and control. That is, it can be easily applied to other issues that require cross-domain collaboration. In this paper, we use several design cases to validate the usability and performance of IOrchestra.
- *Scalability.* IOrchestra is scalable because it has no centralized control. The VMs will work together to achieve high I/O performance.

It is easy to adopt IOrchestra - the cost is comparable to paravirtualization as it only modifies the hypervisor and drivers, that is, the changes to the guest OS is minimal. In virtualized data centers, IOrchestra can provide valuable benefit of coordinating many critical OS functions across the host OS and the guest OSes.

The rest of the paper is organized as follows: The next section gives the background and motivation. Sec. 3 presents the architecture of IOrchestra as well as its versatile functions. Sec. 4 describes the implementation of IOrchestra in Linux and Xen. The experiment setups and results are presented in Sec. 5. We discuss related works in Sec. 6 and conclude in Sec. 7.

2. BACKGROUND AND MOTIVATION

A number of studies have shown that various settings of I/O stack can lead to significant improvement in performance and resource utilization [7, 30, 43]. Table 1 lists a number of configurable I/O components and tuning methods[1]. One can see that some components are only adjustable in one domain, e.g., hardware-related settings, and others may exist in both the guest and host domains. In the latter case, it is likely that they work independently and can not be automatically tuned in event of new changes in the other domain. In this work, instead of controlling hardware and application specific settings, we focus on collaborative optimization between the guest and host domains.

Here we will use a simple test to illustrate the impact on I/O latency from the congestion control mechanism in Linux. In this test we run two VMs and each VM has four VCPUs and four GBs memory. Both VMs are using eight threads concurrently to read eight 1 GB files. Linux has a congestion avoidance scheme to reduce the performance impact caused

[1]Actual settings and practice might be different depending virtualization type (full, para, or hardware-assisted virtualization), deployment policy, etc.

Table 1: Virtual I/O System Parameters

Layer	Configuration or Function	Controlled by	Preferred Tuning Method
Application	Database schema	Guest	By app users
	File format		
File System	Flush dirty pages	Guest & Host	Via collaboration
	Read ahead		
Block I/O	Congestion control	Guest & Host	Via collaboration
	Scheduling		
Hardware	RAID settings	Host	By host administrators
	Parity check		

by a congested request queue. When using the default settings in the driver domain and VMs, the congestion avoidance scheme is activated in both guest VMs, which leads to the average measured latency of 220 ms. It turns out that the workloads from both VMs did not saturate the I/O bandwidth of the storage device, but the VMs were unaware that more I/O requests could be supported. If the default Linux congestion avoidance were disabled, the average measured latency would become 160 ms, a 37% improvement. In other words, falsely triggered congestion avoidance could introduce negative impacts on virtual I/O performance. It is important to note that this semantic gap problem also exists in many other system components, e.g., network, CPU, and memory.

In all, the semantic gap is a by-product of isolation provided by the virtualization architecture. Though isolation is a property that can not be compromised, it has prevented potentially useful communication across different domains. IOrchestra aims to achieve collaborative virtualization without compromising the isolation. The evaluation results of IOrchestra also show performance improvement when the guest OSes are integrated with IOrchestra's driver code.

3. ARCHITECTURE

IOrchestra allows different domains on the host to collaborate with each other through four major components as shown in Fig. 2:

- A shared system store for information exchange;
- An interface between the system store and all the VMs;
- A monitoring module in the hypervisor for collecting system status;
- A management module also in the hypervisor for updating the parameters.

In IOrchestra, each physical machine runs an IOrchestra-enabled hypervisor independently. We utilize a shared key-value store (KVS) in the hypervisor for exchanging information across the domains. Each guest domain stores their configuration data in KVS. All VMs have access to the store, but not all data fields. For security and privacy, each VM can only access its own data via certain system functions. Only the hypervisor has the access to the data of all VMs. IOrchestra also includes the monitoring and the management modules to facilitate the collective system optimization. The monitoring module collects and processes system statistics, such as latency, throughput, performance counters and access patterns. Through analyzing collected data, the

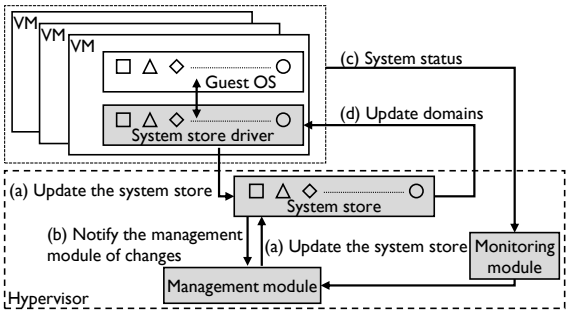


Figure 2: IOrchestra system architecture. Common actions are (a) The domains or the management module makes changes to the system store; (b) The system store notifies the management module about the updates; (c) The monitoring module periodically reports the system status for decision making; (d) The system store notifies the domains about newly updated configurations. The gray parts represent the IOrchestra modules. Various shapes inside the VM and the store are used to illustrate different system modules or parameters.

hypervisor can obtain the information of all VMs. The management module then utilizes this information and KVS values to calculate new configurations for guest domains. The hypervisor stores these configurations in the KVS, which in turn notifies the guest domains to retrieve these updated values and adjust the corresponding settings.

The design of IOrchestra uses the publish-subscribe model. That is, additional modules and functions can be easily included. Also, IOrchestra has low overheads in runtime because it only reacts to certain system events. By utilizing the observed behaviors and insights from our studies, IOrchestra makes decisions promptly with less than a hundred lines of code in each function.

While IOrchestra enables more control within domains, it is still as secure as paravirtualization in the sense that both of them use customized interfaces for improving manageability and performance. The access control to the system store is enforced by the hypervisor. That is, all attempts to access the system store require appropriate permissions. IOrchestra can be configured to identify malicious VMs by enabling anomaly detection in the management module. Although the features of current prototype do not result in any conflict, the management module can also be used to resolve potential configuration conflicts when extending IOrchestra to more functions.

In the following, we present three I/O functions and show how IOrchestra helps critical I/O components to close the semantic gap and achieve high-performance I/O.

3.1 Flushing Dirty Pages

In order to reduce the latency of write requests, operating systems tend to keep modified data in memory. These buffered data, known as dirty pages, will be written to disks when the system runs out of memory, the dirty pages have existed for too long, or there are too many dirty pages. There could be multiple flushing threads responsible for different device queues to avoid the write-back process from being blocked at a single device. The idea of multi-threaded

flushing is to improve the overall throughput by keeping high utilization across multiple disks.

In virtualized environments, the virtual disks in a VM could still be files or partitions on the same disk. As a result, the original idea of avoiding bursty and congested traffic on one device becomes meaningless. Moreover, multiple VMs could simultaneously flush many dirty pages onto the same disk because they are not aware of the real loading on the physical I/O subsystem. Consequently, bursty write requests may saturate the I/O subsystems and cause traffic congestions on all VMs, which represents a control gap in the flushing mechanism in virtualized environments.

To address this problem, IOrchestra designs a cross-domain flush control technique. The idea is to spread out the flushing activities in the VMs to reduce the possibility of overloading the system. The management module actively flushes VMs' dirty pages when the underlying storage device is idle. To reduce the system overhead, only VMs with dirty pages will be scheduled to write back. In the prototype, a boolean variable *has_dirty_pages* for indicating the existence of dirty pages is maintained in the system store for each virtual disk. Linux uses a *bdi_writeback* structure for tracking dirty pages. On an IOrchestra-enabled platform, a guest OS sets a variable *has_dirty_pages*=1 in the system store if there exists dirty pages, that is, the parameter *nr* is greater than zero in the *bdi_writeback* of its virtual disk. The monitoring module collects physical disk status using *blktrace* and reports it to the management module. When the bandwidth usage of a block device is lower than one tenth of its capacity, the management module sets a variable *flush_now*=1 of the guest domain with the most dirty pages in the system store. The affected guest VM will be notified about this change. Then, the guest OS will trigger the *sync* system call, which will invoke *wakeup_flusher_threads* to flush dirty pages, and set *flush_now*=0. Algorithm 1 shows the pseudocode of the control algorithm.

Algorithm 1: Policy for flushing dirty pages

```
Data:
nr_i: number of dirty pages of VM_i;
flush_now_i: to notify VM_i to flush dirty pages;

//VM_i set dirty pages
if the number of dirty pages is changed to zero then
    has_dirty_pages_i = 0;
else
    has_dirty_pages_i = 1;
end

//Management module notifies a guest to flush
if has_dirty_pages_i = 1, ∀i and the device has low utilization then
    i = arg max_i nr_i
    //notify VM_i to flush dirty pages
    flush_now_i = 1;
end

//VM_i is notified to check flush status
if flush_now_i = 1 then
    sync();
end
```

3.2 Congestion Control

If the number of pending read/write requests in the OS reaches a pre-defined limit, the processes that try to submit read/write requests will be placed in a waiting queue and forced to sleep until the job scheduler wakes them up. Putting processes to sleep subverts the performance because of the excessive sleep time and extra operations for context switching [9]. In order to reduce the performance impact caused by a congested request queue, a congestion avoidance scheme is implemented in Linux. The basic idea is to slow down the rate of generating new requests before the queue is full and processes are forced to sleep. Linux uses fixed ratios to turn on/off the congestion avoidance scheme. When the number of pending requests is larger than 7/8 of the queue limit, Linux kernel slows down the generating rate of new requests. When the number of pending requests is smaller than 13/16 of the queue limit, Linux kernel turns off the congestion avoidance process. Similarly, the semantic gap in virtualized systems prevents VMs to achieve a good congestion control on the whole system level.

IOrchestra designs a collaborating congestion control technique shown in Algorithm 2.

Algorithm 2: Policy for congestion control

```
Data:
congested_i^j: 1 when device dev_j of VM_i is congested, and 0 otherwise;
release_request_i^j: to notify VM_i to flush requests to dev_j;

//VM_i requests congestion status on dev_j
if congestion avoidance function is called on dev_j of VM_i then
    if Host device is overcrowded then
        congested_i^j = 1;
    else
        congested_i^j = 0;
        release_request_i^j = 1;
    end
end

//when release_request_i^j is set
if congested_i^j = 1 then
    notify VM_i to flush dev_j's request queue;
    congested_i^j = 0;
    release_request_i^j = 0;
end

//When a host device is relived
if ∃ congested_i^j = 1 on this host device then
    notify VM_i to flush dev_j's request queue;
    congested_i^j = 0;
    release_request_i^j = 0;
end
```

We first create two new boolean entries: *congested* for each device request queue, and *release_request* for each VM in the system store. When a guest VM wants to enable the congestion avoidance scheme, it will set *congested* as one in the corresponding device. The management module is then notified with this change and checks if the host I/O subsystem is congested. If the I/O subsystem is really congested, the entry *congested* will be set. Otherwise, the hypervisor will instead set *release_request* as one. In this case, the guest VM is notified and a corresponding call back function is triggered to unplug and flush the request queue. When the host's I/O subsystem is free from congestion, it will check if any VM still has *congested* as one and reset the status if needed. When a VM's *congested* is unset, it will wake up processes in the waiting queue. To avoid starvation and bursty traffic, VMs are woken up in a FIFO order and interleaved with a random time interval between 0 to 99 ms.

3.3 Inter-domain I/O Co-scheduling

Using dedicated I/O cores to process I/O requests have been proposed to reduce the virtualization overheads [4, 22, 23]. However, this also introduces additional challenges when a VM's VCPUs may be distributed over several sockets on a NUMA (non-uniform memory access) machine [8, 32]. In NUMA, accessing remote resources usually has a higher cost. Related works on NUMA-aware scheduler in virtualized environments mostly focus on placing CPU and memory on the same socket, e.g., VMware vSphere. A common practice for tuning performance is to fit VMs within physical cores' capacity [41]. The dedicated I/O core framework in [22] also assumes all VCPUs are on the same socket. As the computing technology is scaling toward exscale, such assumption greatly limits the scalability of VM size. To maximize resource utilization, large VMs may reside on different sockets and use multiple I/O cores. As a result, the load on I/O cores may not be balanced because VMs are not aware of it.

IOrchestra uses two design principles for the inter-domain I/O co-scheduling and dedicated I/O cores: 1) Interference-aware that aims to minimize performance degradation caused by resource contention. 2) Weighted resource sharing that preserves the priority across all the VMs.

First, not all VCPUs of one VM can always stay on the same socket. For some large VMs and also to maximize the resource utilization, some VMs may be assigned to different sockets. When such case happens, a NUMA-aware process scheduler inside the guest VM can be used to improve the performance and reduce potential interference with other VMs. This goal can be achieved with the help of IOrchestra. To reduce the remote I/O core access overhead and balance loadings, the cross-socket VM has a request buffer per resided socket and the corresponding I/O processes will work on its local buffer.

Assume a VM is spread over the sockets $SKT_i, i = 1 \cdots n$, and the average latency on the I/O core of each socket is L_i. To avoid overloading on an I/O core, the weight of I/O processes inside a cross-socket VM will be distributed to different sockets in the inverse proportion to L_i. That is, the cross-socket VM's I/O weight on SKT_i will be

$$\frac{\frac{\sum_{i=1}^{n} L_i}{L_i}}{\sum_{i=1}^{n} \frac{\sum_{i=1}^{n} L_i}{L_i}}.$$

The monitoring module collects latencies on all I/O cores. Then, the managing module updates the weights to the system store. To avoid the overhead from frequent updates, the management module updates the weights every second or when the weight ratio among cores is changed by over 50%. Once updated, the registered callback function inside a guest VM is invoked to distribute I/O processes to different VCPU's according to the updated information.

Because the I/O cores keep polling guest's request buffers, the time sharing on the I/O cores directly affects guest domain's I/O performance. The scheduling in [22] assumes static equal share of all VMs on the same socket. Such assumption leads to the inaccurate resource sharing in virtualized systems. Each I/O core should also have different priorities because they usually are not serving the same group of VMs. Therefore, we consider the weights of each co-located VM when scheduling the polling interval.

Algorithm 3: Polling request buffers by one I/O Core

```
Data:
m active VMs on this I/O core;
i ∈ 1, ..., m;
B_i is the request buffer of VM_i;
C_i is the credit of B_i;
Q_i = BW_max · S_SKT^VM_i, the quantum of VM_i;
BW_max is the maximum bandwidth of the device;
S_SKT^VM_i is updated periodically by IOrchestra;
while TRUE do
    if m ≥ 1 then
        for i ← 1 to m do
            C_i = Q_i + C_i;
            while C_i > 0 and B_i ∉ ∅ do
                ReqSize = The size of the first request in B_i;
                if ReqSize ≤ C_i then
                    Process the first request in B_i;
                    C_i = C_i − ReqSize;
                else
                    //credit is not enough now
                    break;
                end
            end
            if B_i ∈ ∅ then
                C_i = 0;
            end
        end
    end
end
```

Assume VM_i has I/O share $S_i^{(VM)}$, where $\sum_{i=1}^{m} S_i^{(VM)} = 1$ and m is the number of VMs in the system. Let VM_i have $VCPU_k^{(VM_i)}, k = 1 \cdots x$. Because of dynamic process scheduling, VCPUs have various I/O priorities and intensities at any given time. Let the weight of a process l be $P_l, l = 1 \cdots y^{(VM_i)}$, where $y^{(VM_i)}$ is the total number of processes in VM_i. Therefore, the process weight on $VCPU_k^{(VM_i)}$ is

$$W(VCPU_k^{(VM_i)}) = \sum P_l, \forall P_l \in VCPU_k^{(VM_i)},$$

and the process weight of VM_i on a socket SKT is

$$W_{SKT}(VCPU_k^{(VM_i)}) = \sum W(VCPU_k^{(VM_i)}),$$
$$\forall VCPU_k^{(VM_i)} \in \text{the socket } SKT.$$

Therefore, the I/O share of VM_i on a socket SKT is

$$S_{SKT}^{VM_i} = \frac{W_{SKT}(VCPU_k^{(VM_i)})}{\sum_{l=1}^{y^{(VM_i)}} P_l} S_i^{(VM)}.$$

Calculating the fine-grained I/O sharing requires the support from IOrchestra. More specifically, $W(VCPU_k^{(VM_i)})$ is periodically updated to the system store. Then, the managing module calculates the I/O shares $S_{SKT}^{VM_i}$ for the VMs. The sum of all I/O shares on a socket is used as its I/O core's weight when accessing a storage device shared with other I/O cores. A storage device uses cgroups with these I/O cores' weights for controlling shares. Algorithm 3, which is adapted from the deficit round-robin method, utilizes $S_{SKT}^{VM_i}$

as the quantum in controlling the time-sharing of all VMs on the same I/O core.

4. IMPLEMENTATION

We implement a prototype of IOrchestra with Xen in about 3,000 lines of code. More specifically, the XenBus and XenStore modules are modified to provide new functionalities for IOrchestra. XenStore is a shared key-value store (KVS) for system configuration and device information. XenStore maintains all domain's settings in a hierarchical key-value structure. A directory or key is accessible to a domain if it has the permission. Interactions between a domain and XenStore are accomplished through XenBus, an abstract channel for inter-domain communication. During the system initialization, domains are required to register configurations and callback functions to XenStore. If certain values are updated, XenStore will notify corresponding domains. Then, notified domains will probe XenStore to retrieve updated information or call registered functions to handle events.

Fig. 3 illustrates the interaction between the system store and the management module. The left-hand side of Fig. 3 shows a sample XenStore hierarchical content. The process starts with the event of *value changes* on monitored items. When triggered, the management module will determine a new configuration and update it to the store. For making appropriate configurations, the management module may also consult the information collected by the monitoring module. After a new configuration is decided, the management module updates the corresponding fields in XenStore. Then, all affected domains will be notified of the changes, and the callback functions will be invoked to carry out specific operations.

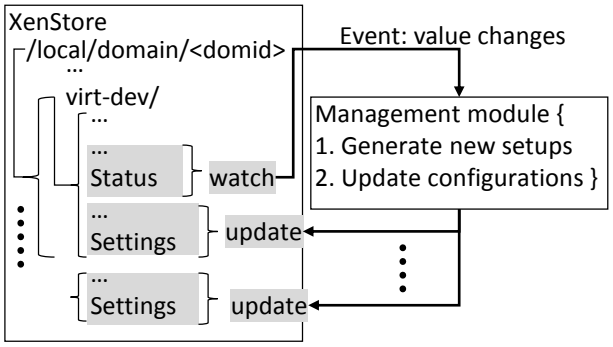


Figure 3: The management module is called when there is a change on watched items. Then, it will make necessary changes on the configuration store

5. EXPERIMENTS

All experiments are conducted on a cluster of machines. Each machine has two six-core 2 GHz Intel Xeon E5-2620 processors, 32 GB memory, and a 960 GB RAID0 drive (eight 120 GB Intel520 SSD drives). The OS in both host and guest machines are Linux 3.5 with Xen 4.0 paravirtualization. The baseline systems in all tests are running the original Linux 3.5 kernel with official Xen 4.0. The IOrchestra prototype includes the proposed framework with support for three use cases in Section 3. We also emulate two related projects for comparison: static dedicated cores (SDC) [22,29] and disk idleness based flush (DIF) [17]. The number of VMs and their sizes are varied in each testing scenario.

We evluate IOrchestra with a number of cloud applications:

- Olio is a Web 2.0 social events calendar;
- YCSB (Yahoo! Cloud Serving Benchmark) provides a performance measurement framework for cloud data serving [13]. The experiments use two of its core workloads on Apache Cassandra: YCSB1 generates an update heavy workload with the read:write ratio of 50:50; and YCSB2 is a read mostly workload with the read:write ratio of 95:5;
- Basic Local Alignment Search Tool (BLAST) [3] is one of the most widely used algorithm for identifying local similarity between different protein sequences. To run BLAST on multiple machines, we use the mpiBLAST implementation [19]. As the inputs, the nucleotide and protein databases used in this work are NCBI's (National Center for Biotechnology Information) NT and NR databases that contain the full-set of non-redundant DNA and protein sequences;
- Cloud9 uses distributed resources to provide a high-quality on-demand software testing service;
- File Server (FS) emulates typical workloads on a file system, e.g., create, read, write, delete on a directory tree; Web Server (WS) issues mostly read operations on web pages, and writes log to a file; Video Server (VS) emulates a video server where a number of threads are reading videos and a thread is adding new videos into the repository; And multi-stream read sequentially reads multiple files. The FS, WS, VS, and multi-stream read are from the FileBench suite [18].

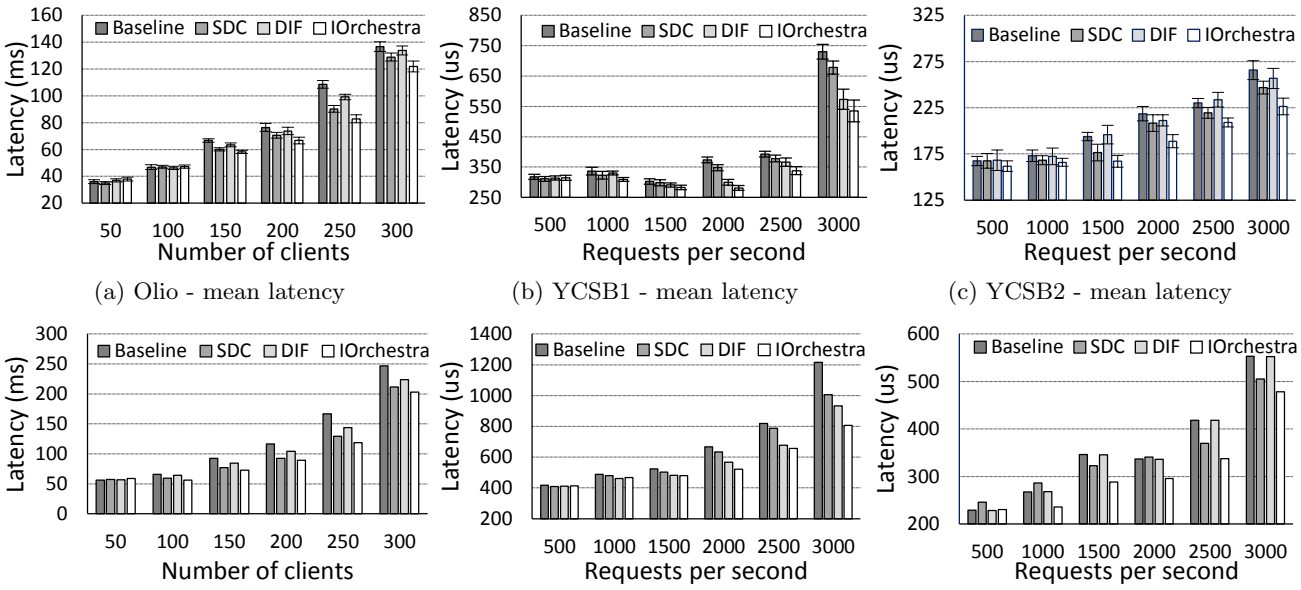
5.1 IOrchestra in Action

As we stated earlier, a cloud application may use multiple VMs as different service components and have various cloud applications running concurrently. There are two real-world cloud applications with various workloads in this test:

1) Olio: It uses three VMs. We install an Apache HTTP server and the PHP version of Olio in one VM. We use another VM as the database (MySQL server) and populate it with a data set of 500 users (about 40 GB). The last VM is a file server for the web frontend. Each VM has two VCPUs and 4 GB memory. We use CloudStone [38] as user behaviors emulator in Faban, and run Faban on another physical server as the workload generator.

2) The multi-node Appache Cassandra key-value store: One data store uses two VMs as its data nodes. Each VM also has two VCPUs and 4 GB memory. There are two Cassandra data stores in this test, which are running YCSB1 and YCSB2 workloads respectively.

We run all applications concurrently for ten minutes in one test and repeat the test with different numbers of clients or requests. Fig. 4 shows the 99.9th percentile and mean latency of all testing applications when running with the baseline, SDC, DIF, and IOrchestra. IOrchestra effectively reduces both the mean and 99.9th percentile latency, especially on the tail latency. The overall average improve-

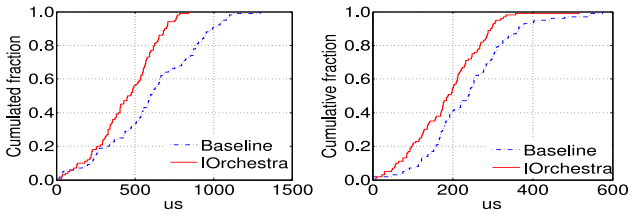


(a) Olio - mean latency (b) YCSB1 - mean latency (c) YCSB2 - mean latency

(d) Olio - 99.9th percentile latency (e) YCSB1 - 99.9th percentile latency (f) YCSB2 - 99.9th percentile latency

Figure 4: Latency at different workload intensities and applications. The whisker on the mean latency is the standard deviation

ment on the mean and 99.9th percentile latency are 9 and 12% respectively. Because of intensive write in YCSB1, passing disk idleness information allows DIF to outperform SDC and baseline. IOrchestra provides even more benefits from advanced flushing and congestion control. The average improvement on the mean and 99.9th percentile latency of YCSB1 are 13 and 16% respectively. Fig 5 shows the cumulative percentages of request latency of YCSB1 and YCSB2 when the workload intensity is 3000 requests per second. The latency distribution clearly shows the advantage of IOrchestra over the baseline case.



(a) YCSB1 (b) YCSB2

Figure 5: Latency distribution of YCSB1 and YCSB2 at 3000 requests per second

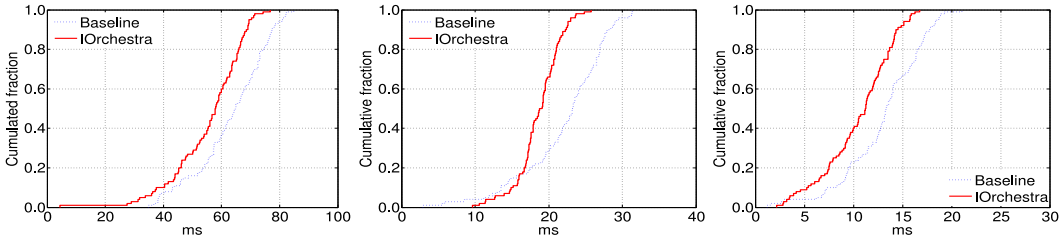
To demonstrate IOrchestra's effect on the distributed multi-tier cloud applications, we analyze the changes of Olio's latency at each tier. Fig. 6 shows the cumulative percentages of request/query latency at Olio's web server, database, and file server VMs. In each sub-figure, the dotted blue line represents a baseline system and the solid red line represents IOrchestra. Fig. 6(a) shows the overall performance of the Olio application, which has an average improvement of 11.2%. Fig. 6(b) and (c) are the latency distributions on the database and file server VMs, which have average improvements of 21.6% and 19.8% respectively. The overall improvement is less than those on the database and file server VMs, because other factors, e.g., CPU times, also affect the final performance.

To summarize, IOrchestra effectively improves the I/O virtualization and its latency, especially in shortening the tail of latency distribution. In the following subsections, we will explore IOrchestra in detail at different scenarios and functions.

5.2 Scaled-Out Cloud Applications

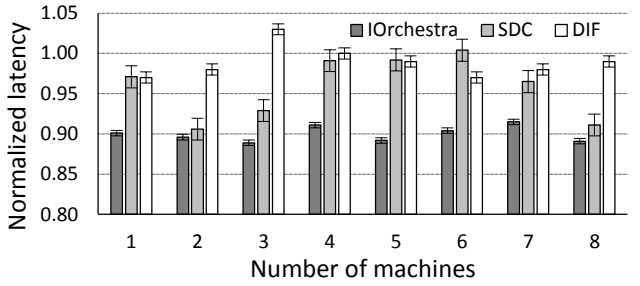
This test is to verify IOrchestra's effectiveness on applications performance in multiple VMs and hosts. We create three VMs on a physical machine. In a dynamic cloud environment, different types of workload may be hosted on the same physical machines. Thus, these three VMs are running different cloud applications, Cloud9, mpiBLAST, and YCSB1. We keep these applications running for one hour each with the baseline and IOrchestra. Because these applications can use multiple interconnected machines to speed up their processes, these tests are also scaled to eight machines.

Fig. 7 shows the normalized mean I/O latency of mpiBLAST and YCSB1 with different numbers of VMs. The X-axis represents the number of VMs used by an application and the Y-axis represents the latency normalized to the baseline. Fig. 7(a) is comparing SDC, DIF, IOrchestra and Baseline using mpiBlast, while Fig. 7(b) using YCSB1. As Cloud9 is a CPU-intensive workload, we do not observe significant changes in its performance, thus exclude it from Fig. 7. With IOrchestra the average latency of mpiBLAST and YCSB1 in all cases are improved by 10.1% and 12.9% respectively. YCSB1's latency is getting longer as the number of VMs increases, because additional inter-node traffic introduces more performance overhead. On the other hand, mpiBLAST's improvement on latency is stable across all number of machines. In order to understand the main

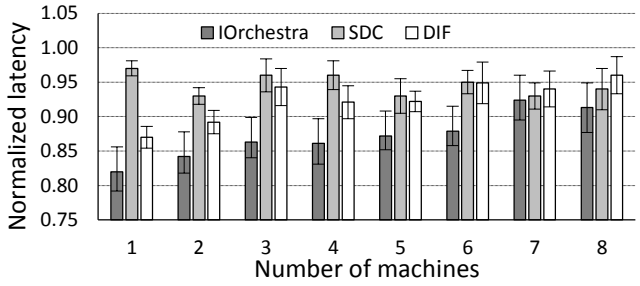


(a) Web server (b) Database (c) File server

Figure 6: Latency distribution at different layers of a distributed cloud application



(a) mpiBLAST



(b) YCSB1

Figure 7: Normalized mean I/O latency of (a) mpiBLAST and (b) YCSB1 at different cluster sizes (numbers of working VMs)

reasons for this latency improvement, we repeat the same test for three times. In each test, only one of the functions in Section 3 is enabled. We find out that mpiBLAST's improvement is mainly from the new function of congestion control because the BLAST algorithm sequentially checks the patterns. YCSB1's improvement is mostly contributed by the functions of congestion control and flushing because of YCSB1's write intensive behavior.

5.3 Flushing Dirty Pages

Write operations in file systems tend to be buffered first and flushed later. Therefore, we use FS's write throughput to evaluate the dirty page flushing schemes. The goal of this subsection is to evaluate IOrchestra's performance on flushing dirty pages. Therefore, only the flushing control is enabled in IOrchestra in this test. The default dirty page ratios defined as the number of dirty pages to the total number of memory pages may range from 10 to 40%, and the number of VMs on the same host can also be different. Fig. 8 demonstrates the write throughput improvement with regards to different dirty page ratios and the number of concurrent VMs. For exercising the flushing scheme, the working set size is larger than twice of a VM's memory size. The X-axis shows the number of VMs, from 2 to 20 VMs. Each VM runs FS, and has one VCPU and 1 GB memory. The Y-axis shows the average improvement from ten runs.

As we have described in Section 3, the baseline systems lose their ability to balance loadings and avoid process starvation in virtualized systems. When all VMs have large dirty page ratio and large amount of requests, VMs may experience a long response time because many VMs are writing huge amount of dirty pages concurrently. IOrchestra, on the other hand, helps VMs to flush when spare bandwidth is available. The largest improvement of 21% is achieved at 20 concurrent VMs and the 40% dirty page ratio. When there are 20 VMs, the average improvement across four ratios is 12.7%.

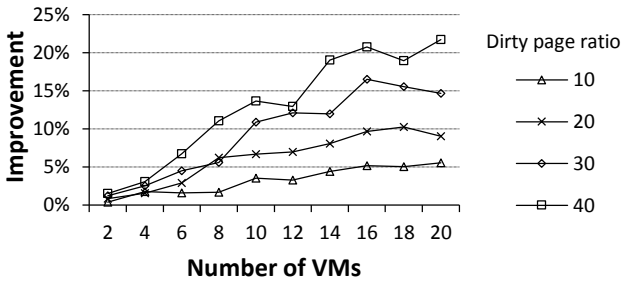


Figure 8: Write throughput improvement of FS workload at various VM numbers

We also test the new flushing scheme with various VM sizes, numbers and arrival rates. The VM arrival rate follows a Poisson process with an average rate of λ VMs per minute. When a VM arrives, its number of VCPUs and memory sizes will be randomly selected from 2, 4, 6, 8, and 10 VCPUs (and GB memory). All VMs are served in FIFO order. The application to run in a VM is randomly selected from FS, YCSB1, and Cloud9. YCSB1 is included because it intensively updates data and buffers data in memory, and Cloud9 is used to represent CPU-intensive workloads. The number of application threads is the same as its VCPUs. Each application is configured with a fixed problem size, e.g., a YCSB VM is removed after finishing 50,000 operations and a FS VM is removed after completing 2 GB data transmission. We test $\lambda = 4, 8, \cdots, 20$ VMs per minute, and the total testing time for each λ is one hour.

Table 2 shows the improvement on aggregate write throughput with various arrival rates. When the VM arrival rate

is small, flushing operations may not be blocked because the bandwidth is not saturated yet. The improvement becomes bigger as the arrival rate rises because our method actively uses underutilized bandwidth to flush dirty pages. The dynamic workload types and intensities create more opportunities, which leads to a higher performance, that is, the average improvement across all cases in Table 2 is about 22.1%.

Table 2: Write throughput improvement with various arrival rates

λ	4	8	12	16	20
Improvement (%)	6.6	19.1	24.5	29.8	30.6

5.4 Congestion Control

We use latency to evaluate the new congestion control scheme. In this test, each VM has 1 VCPU and 1 GB memory. Fig. 9 shows the normalized latency when running FS, WS, and VS with various numbers of VMs. The normalized latencies are getting closer to one as the number of VM increases. Interestingly, with the help from IOrchestra, the normalized latency can be as small as 0.9 when running FS. In this case, FS issues more small mix I/O requests than WS and VS, where guest VMs would falsely trigger more congestion avoidance while the I/O subsystem was not really congested.

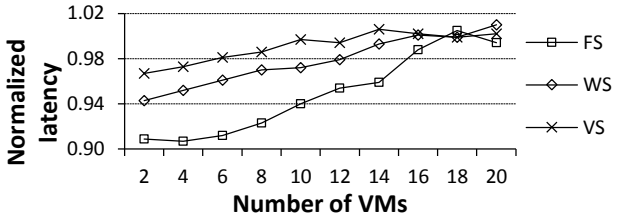


Figure 9: Latency of various workloads normalized to corresponding baselines

5.5 Inter-domain I/O Co-scheduling

The goal of this experiment is to verify the ability of coordinating in-guest I/O processes. We integrate IOrchestra with *cgroup* in guest to run and control testing applications. We create a big VM which has 10 VCPUs and 10 GB memory. Because each VCPU is pinned to one physical core, all physical cores are in use. We concurrently run one CPU-intensive application (Cloud9) and one I/O-intensive benchmark (multi-stream read) in this big VM. In order to test different I/O intensities, the number of threads and working set sizes are varied, specifically from high (80%) to low (20%) I/O intensities. Fig. 10(a) shows the I/O throughput improvement at different I/O intensities. The X-axis shows the ratio of data-intensive threads. The numbers are the average values and standard deviations of ten runs. The throughput improvement ranges from about 2% to 14%. The improvement is bigger at moderate I/O intensity (40% and 60% I/O threads) because the I/O cores may be extremely unbalanced in the baseline. These two cases also have high variance. On the other hand, the improvement for low I/O intensity (20%) is small due to the light workload. For high I/O intensity (80%), the improvement is small too because in this case I/O cores in both the baseline and IOrchestra have the same heavy workload.

Similar to the test setting in Table 2, we also test the inter-domain I/O co-scheduling with various VM sizes, numbers, and arrival rates $\lambda = 4, 8, \cdots, 20$ VMs per minute. Again, the total testing time for each λ is one hour. Fig. 10(b) shows the improvement on the number of VMs completed. When the arrival rate is small, the system is not fully loaded. A higher arrival rate makes the system fully loaded and highly utilized, which is the case that mostly needs co-scheduling. Therefore, the improvement is increased as the λ increases. Our method outperforms the baseline by up to 6.6% when λ is equal to 20. This is because our framework helps to balance the I/O core loading. And since not all VMs can be hosted on the same socket immediately, our cross-domain framework can help to improve the performance of those VMs that reside on different sockets. Moreover, our method also outperforms the original dedicated I/O core method.

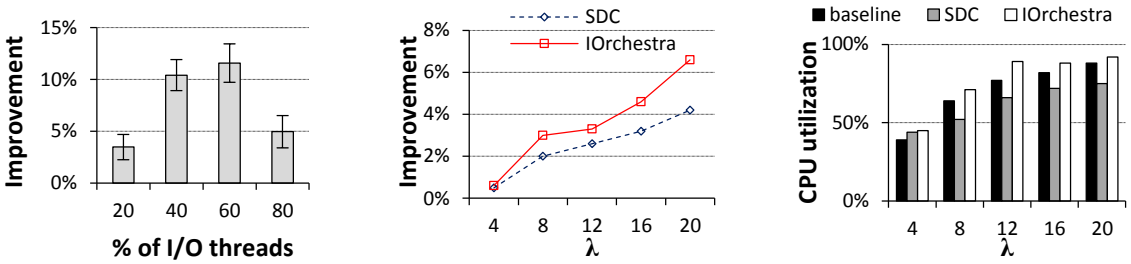
Because some cores are dedicated for I/O, Fig. 10(c) shows the overall CPU utilization at different workload intensities. When λ is low, the baseline has a lower CPU utilization because it does not keep spinning one core for I/O. As the VM arrival rate is increasing, the CPU utilization of the baseline and IOrchestra become higher than the dedicated core scheme because of its restriction on cross-socket assignment.

Using dedicated core improves the throughput for the baseline. Fig. 11 shows the throughput improvement at various λ. One can see that the original dedicated core method does not scale at the high VM arrival rate. On the other hand, IOrchestra helps to balance the workload on I/O cores. The enhanced ratio can be double when λ reaches 20.

5.6 Bursty Writes

In this subsection, we show that the flushing and congestion control techniques in IOrchestra deliver low latency even with bursty writes. We use skewed request inter arrival times with the write-intensive YCSB1 to create the bursty workloads. That is, the inter-arrival times do not follow the same pattern all the time. Specifically, there are short periods of intensive workloads, although the average rate might be reasonable. We use the methodology described in [5, 25]. There are multiple clients generating requests at a fixed rate with synchronized burst periods. During a burst period, the maximum request rate is limited at 10 times of the overall average rate. To fairly compare IOrchestra with other methods, we control the number of requests in a burst to ensure they are the same during different tests.

Fig. 12 shows the 99.9th percentile of latency for the baseline and IOrchestra at burst periods of 50 and 100 ms. The latency of baseline exceeds 1 ms quickly at 800 and 500 requests per second when the burst length are 50 and 100 ms respectively. DIF performs better than SDC because the transparent disk information helps to schedule bursty requests. On the other hand, IOrchestra outperforms the others because of the advanced congestion control and comprehensive co-scheduling. IOrchestra can deliver more requests within a millisecond than the baseline. The average latency improvement across all cases is 31.8%. Note that without the bursty writes, the baseline can manage up to 2,500 requests per second with a latency of less than a millisecond shown in Fig. 4(b).



(a) The I/O throughput improvement at various intensities

(b) Improvement on the number of VMs completed at dynamic VM sizes and arrival rates λ

(c) Average CPU utilization at different arrival rates

Figure 10: Experiment results of inter-domain I/O co-scheduling

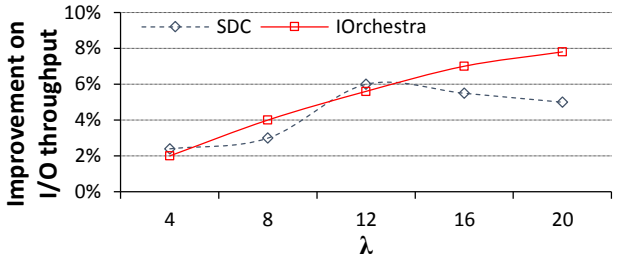
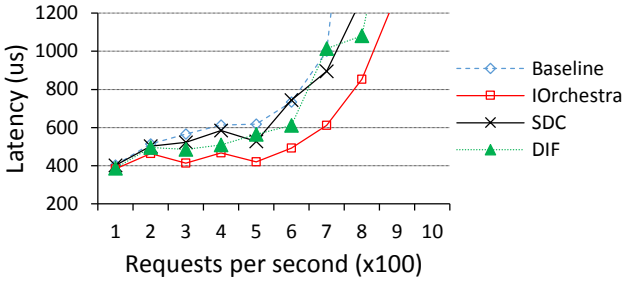
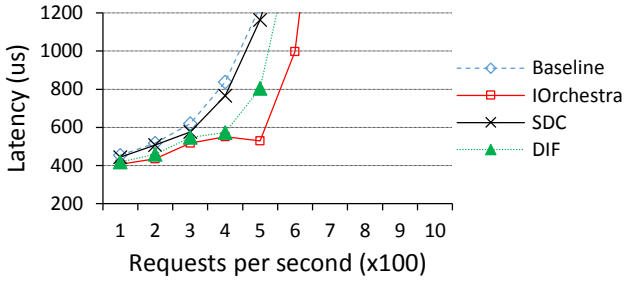


Figure 11: Normalized I/O throughput at different arrival rates



(a) 50 ms burst length



(b) 100 ms burst length

Figure 12: YCSB1 latency at 50 and 100 ms burst length respectively

6. RELATED WORK

Virtualization architecture design: Liu et al. propose the VMM-bypass I/O to improve I/O performance for VMs, which allows time-critical I/O operations to be processed without a hypervisor or a driver domain in the middle [31]. On the other hand, Santos et al. suggest to keep the driver domain because the space isolation provided by driver domain effectively prevents unexpected system crashes caused by drivers or malicious guest VMs [37] . ELI [21] and vIC [1] demonstrate the expensive overheads for interrupt handling. While these works focus on modifying specific virtualization architectures and reducing the overheads of virtual I/O, IOrchestra utilizes a communication channel for system management and keeps original virtualization design intact. In this manner, IOrchestra is open and applicable to various OS and virtualization designs.

XHive globally manages caches of VMs in order to accommodate a shared working set in host machine memory [26]. While XHive supports an holistic control over memory, IOrchestra provides a comprehensive framework by building bridges for eliminating the semantic gaps across domains. SplitX utilizes a set of dedicated cores for the guest and the hypervisor to avoid the expensive VM exit operation [29]. As SplitX uses extra cores and hardware to improve I/O performance, IOrchestra can be used to complement SplitX by providing administration channels across cores and thus optimizing systems globally. Elango et al. propose a new design to pass disk idleness information to VMs such that I/O schedulers can find a good time for flushing dirty pages [17]. IOrchestra aims at a bigger picture beyond the disk idle information, to bridge all semantic gaps.

MIKELANGELO [33], one of the European Commission's Horizon 2020 projects, was initiated to improve virtual I/O by integrating the thin OS, the remote direct memory access, and several enhancements in the KVM. While MIKELANGELO reduces the I/O overheads with a special designed OS and I/O stack, IOrchestra utilizes a communication channel for system management and keeps the original OS design intact. In this manner, IOrchestra is applicable to different OSes and virtualization systems.

Inter-domain communication: XenSocket [44] provides a socket-based interface for inter-domain communication and utilizes the shared circular buffer to transmit control packets and data between domains. XenLoop [39] adds an additional layer under the network layer to monitor outgoing packets. If the destination of packets is a co-located VM, XenLoop will redirect packets through the shared memory channel. Xway [27] provides similar features as XenLoop by bypassing

TCP/IP stacks. Fido [11] is an extra software component in guest domains for inter-domain communication. Fido modifies Xen event channel, Xen store, and memory mapping module to accomplish fast inter-virtual-machine communication. Although IOrchestra focuses differently on bridging the semantic gaps for optimizing system performance, it benefits from these inter-domain communication techniques in the framework design and event channel usages.

Gray-box techniques and VM introspection (VMI): Burnett et al. [10] use gray-box techniques to identify cache replacement policies in OS, and demonstrate the performance improvement of a web server which is aware of the system cache status. Geiger [24] extends similar gray-box techniques to monitor the buffer cache in a virtualized environment. VMI is an active research topic in digital forensics [20, 34], intrusion detection [36], and malicious VM detection [16]. In comparison, IOrchestra is an efficient and direct method in optimizing the virtual I/O performance.

7. CONCLUSION

We design IOrchestra to bridge the semantic gaps of virtual I/O stacks across different domains. IOrchestra provides a foundation for building an advanced virtualization with fine-grained control. IOrchestra can be integrated with various functions to effectively improve virtual I/O latency. A prototype in Linux and Xen is implemented to automatically tune the system configurations of all guest VMs. The evaluation results show that IOrchestra is able to coordinate all domains with low overhead and improve the latency by up to 31% with various real-world distributed cloud applications running on multiple virtual and physical machines. As future work, IOrchestra will be extended to additional system components that may suffer performance degradation from the semantic gap, e.g., network buffer sizes, window sizes, packet queues, etc.

Acknowledgment

We thank our shepherds, Craig Lee and David Abramson, and anonymous reviewers for their valuable suggestions. This work is supported in part by National Science Foundation grants CNS-1350766, CNS-1320226, CNS-1253575, and IOS-1124813. Ron Chiang did part of this research at the GWU and AT&T Labs.

8. REFERENCES

[1] I. Ahmad, A. Gulati, and A. Mashtizadeh. vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *USENIX Annual Technical Conference*, 2011.

[2] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of molecular biology*, 215(3):403–410, Oct. 1990.

[4] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. vIOMMU: Efficient IOMMU Emulation. In *USENIX Annual Technical Conference*, 2011.

[5] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, USITS'97. USENIX Association, 1997.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP'03, pages 164–177. ACM, 2003.

[7] P. Berenbrink, A. Brinkmann, T. Friedetzky, D. Meister, and L. Nagel. Distributing Storage in Cloud Environments. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), IEEE 27th International*, pages 963–973, 2013.

[8] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 557–558. ACM, 2010.

[9] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[10] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *USENIX Annual Technical Conference*, pages 29–44, 2002.

[11] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. In *USENIX Annual Technical Conference*, 2009.

[12] R. Chiang and H. Huang. TRACON: Interference-Aware Scheduling for Data-Intensive Applications in Virtualized Environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, pages 47:1–47:12, 2011.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[14] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Eextensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62. ACM, 2008.

[17] P. Elango, S. Krishnakumaran, and R. H. Arpaci-dusseau. Design Choices for Utilizing the Disk

Idleness in a Virtual Machine Environment. In *In Workshop on the Interaction between Operating Systems and Computer Architecture*, WIOSCA, 2006.

[18] FileBench. http://filebench.sourceforge.net/wiki/index.php/main_page.

[19] M. K. Gardner, W.-c. Feng, J. Archuleta, H. Lin, and X. Mal. Parallel Genomic Sequence-searching on an Ad-hoc Grid: Experiences, Lessons Learned, and Implications. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06. ACM, 2006.

[20] T. Garfinkel and M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine based Computing Environments. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, HOTOS'05, 2005.

[21] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. *SIGARCH Comput. Archit. News*, 40(1):411–422, Mar. 2012.

[22] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky. Efficient and Scalable Paravirtual I/O System. In *USENIX Annual Technical Conference*, pages 231–242, 2013.

[23] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/O Scheduling Model of Virtual Machine based on Multi-Core Dynamic Partitioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC'10, pages 142–154. ACM, 2010.

[24] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, 2006.

[25] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14. ACM, 2012.

[26] H. Kim, H. Jo, and J. Lee. XHive: Efficient Cooperative Caching for Virtual Machines. *Computers, IEEE Transactions on*, 60(1):106 –119, jan. 2011.

[27] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim. Inter-Domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08. ACM, 2008.

[28] R. Kohavi and R. Longbotham. Online Experiments: Lessons Learned. *Computer*, 40(9):103–105, 2007.

[29] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In *Proceedings of the 3rd conference on I/O virtualization*, WIOV. USENIX, 2011.

[30] D. Le, H. Huang, and H. Wang. Understanding Performance Implications of Nested File Systems in a Virtualized Environment. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.

[31] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference*, 2006.

[32] C. McCurdy and J. Vetter. Memphis: Finding and Fixing NUMA-Related Performance Problems on Multi-Core Platforms. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96. IEEE, 2010.

[33] MIcro KErneL virtualizAtioN for hiGh pErfOrmance cLOud and hpc systems. `http://cordis.europa.eu/project/rcn/194319_en.html`, 2015.

[34] K. Nance, M. Bishop, and B. Hay. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6(5):32–37, Sept. 2008.

[35] K. Nichols and V. Jacobson. Controlling Queue Delay. *Commun. ACM*, 55(7):42–50, July 2012.

[36] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, 2008.

[37] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *USENIX Annual Technical Conference*, 2008.

[38] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0. In *In Proceedings of Cloud Computing and its Applications*, CCA, 2008.

[39] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: A Transparent High Performance Inter-VM Network Loopback. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08. ACM, 2008.

[40] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61. ACM, 2011.

[41] Xen. Tuning. `http://wiki.xen.org/wiki/Tuning`.

[42] Y. Xu, M. Bailey, B. Noble, and F. Jahanian. Small is Better: Avoiding Latency Traps in Virtualized Data Centers. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 7:1–7:16. ACM, 2013.

[43] Y. Yu, Y. Wang, H. Guo, and X. He. Optimisation Schemes to Improve Hybrid Co-Scheduling for Concurrent Virtual Machines. *Int. J. Parallel Emerg. Distrib. Syst.*, 28(1):46–66, 2013.

[44] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, 2007.