Quantifying and Improving I/O Predictability in Virtualized Systems

Cheng Li, Íñigo Goiri, Abhishek Bhattacharjee, Ricardo Bianchini, Thu D. Nguyen
{*chenglii, goiri, abhib, ricardob, tdnguyen*}*@cs.rutgers.edu*
Department of Computer Science
Rutgers University, Piscataway, NJ 08854

Technical Report DCS–TR–697, September 2012, Revised February 2013

Abstract

Virtualization enables the consolidation of virtual machines (VMs) to increase the utilization of physical servers in Infrastructure-as-a-Service (IaaS) cloud providers. Unfortunately, our quantification of storage I/O performance across a range of workloads, virtual machine monitor (VMM) architectures, approaches to storage virtualization, and storage devices shows widespread performance *unpredictability* in the face of consolidation. Surprisingly, the use of solid-state drives (SSDs) can exacerbate the problem. Since many users may desire consistent performance, we argue that IaaS cloud providers should provide a class of predictable-performance service in addition to their existing (predictability-oblivious) services. Thus, we propose and evaluate VirtualFence, a storage system that provides predictable performance for this new class of service. VirtualFence uses three main techniques: (1) non-work-conserving time-division I/O scheduling, (2) a small SSD cache in front of a much larger hard disk drive (HDD), and (3) space-partitioning of both the SSD cache and the HDD. Our evaluation of a prototype VirtualFence implemented in the Xen VMM shows that VirtualFence improves predictability significantly. More fundamentally, our evaluation illustrates the tradeoff between predictability and performance. We conclude that current VMMs are far from providing predictability. Systems like VirtualFence can remedy this problem, while allowing the cloud provider to select an appropriate compromise between performance and predictability.

1 Introduction

With the advent of cloud computing, virtualization has become the primary strategy to consolidate diverse workloads (encapsulated in virtual machines or VMs) to ensure high utilization of physical machines (PMs). Among its various benefits, virtualization increases fault isolation and simplifies workload migration [1, 2]. Many IaaS cloud providers, such as Amazon EC2 and Rackspace, use virtualization and consolidation in offering their services. However, as we demonstrate in this paper, VM performance may vary significantly in the face of consolidation. In fact, VM performance is essentially unpredictable, since the number of co-located VMs and their workloads may change each time the VM runs, or even during a single run. For example, researchers have shown a single run of a fixed-load VM on Amazon EC2 to exhibit wild performance swings due to consolidation [3]. Since many users may desire consistent performance, we argue that IaaS cloud providers should offer a new class of predictable-performance service *in addition to* (and using different resources from) their existing (predictability-oblivious) services.

Along these lines, our research seeks to create virtualized systems that exhibit *performance predictability* for this new class of service. This property implies that the average throughput and response time experienced by each VM should be unaffected by any other VM executing on the same PM or the overall utilization of the PM. In fact, performance should be the same whether the VM runs in isolation or is co-located with any number (up to a pre-defined limit) of other VMs.

Importantly, note that our notion of performance predictability differs from *performance isolation* [2, 4, 5, 6, 7, 8, 9]. The goal of isolation is to ensure that each co-located VM achieves at least a minimum desired level of performance. This is one of the goals of predictability. However, in performance isolation it is typically acceptable to dedicate more resources than this minimum, if those resources are available. In contrast, dedicating any available resources beyond a fixed amount will likely ruin predictability. One may see performance isolation as a less strict form of performance predictability. As far as we know, only mClock [4] provides the ability to set both a minimum and a maximum (throughput) performance for a VM. However, the paper did not consider setting these bounds to the same value to achieve predictability.

In this paper, we address the specific case of *storage I/O* performance predictability (or simply I/O predictability). Compared to processing, memory, or networking, I/O predictability is the most challenging to achieve primarily due to the mechanical limitations of disk drives. In particular, high disk seek and rotational times make predictability difficult to achieve when multiple VMs share the same disk.

We divide the paper into two parts. The first part quantifies the impact of workload characteristics, virtual machine monitor (VMM) architecture, the approach to virtualizing storage, and storage device characteristics on I/O predictability. Overall, we measure the I/O predictability of three workloads running on sixteen configurations. Using a new "performance deviation" (or simply "deviation") metric, we find widespread unpredictability. Further, perhaps surprisingly, we find that using an SSD as the storage device can exacerbate the problem when the workload is write-intensive.

Based on this first study, the second part of the paper proposes and evaluates VirtualFence, a performance-predictable storage system. VirtualFence seeks to produce *consistent performance within the range defined by the best and worst performance levels that each VM may experience in a predictability-oblivious service* (i.e., in the absence of VirtualFence). Specifically, a VM would experience its best performance *BestPerf* when the provider co-locates no other VMs with it, and allows it to use all resources of the PM. The worst performance *WorstPerf* would occur when the provider co-locates other active VMs with it (up to the maximum capacity of the PM). VirtualFence seeks to produce performance between *BestPerf* and *WorstPerf*. In addition, it seeks to retain this performance regardless of changes in the number of co-located VMs, changes in the working set size of the other VMs, or any other external condition. The goal is for each VM to always run as if it were alone on a fixed and tightly controlled partition of the resources.

To achieve this goal, VirtualFence couples a small persistent SSD cache with a much larger HDD. It also implements a non-work-conserving I/O scheduling algorithm, partitioning time into a fixed number of relatively coarse-grained slots. Each I/O slot can only be given to one active VM, but a single active VM may receive multiple slots (the number of slots depends on how much the VM's owner is willing to pay the cloud provider). I/O accesses from a VM are only serviced during the VM's assigned slot(s). A static allocation of time slots while a VM is active ensures consistent resource allocation for predictable performance. The SSD cache and I/O time partitioning together minimize the impact of HDD head movement due to consolidation, whereas I/O time partitioning minimizes the impact of SSD block erasures. Finally, VirtualFence partitions both the SSD cache and the HDD, which is a non-work-conserving space allocation scheme that again ensures consistent resource allocation for predictability.

For simplicity, we implement VirtualFence for a virtualized system with direct-attached disks. However, the same ideas (one SSD per HDD, SSD space partitioning, and I/O time partitioning) can be easily implemented in file/storage servers, network-attached storage appliances, or distributed file/storage systems. Although these other systems may present additional sources of potential unpredictability, we expect disk-driven unpredictability to still dominate. The only requirement in these implementations is that the closest driver to the disks must be able to identify the VMs from which the accesses are coming.

Our evaluation demonstrates that VirtualFence improves I/O predictability (or, equivalently, that it reduces deviation) significantly, as long as we utilize all of its component techniques at the same time. In fact, we show that simply using an SSD as a cache of HDD data is *not* enough. More fundamentally, our evaluation illustrates the tradeoff between predictability and performance: the more we improve predictability, the worse average response time becomes. The challenge is finding the smallest response time that will produce enough predictability.

In summary, our contributions are:

- We study the impact of VMM architecture, approach to storage virtualization, and storage device on I/O predictability.
- We propose VirtualFence, a system that combines SSDs working as HDD caches, space and non-work-conserving time partitioning.
- We quantify the impact of each feature of VirtualFence on I/O predictability.
- Using VirtualFence, we investigate the fundamental tradeoff between predictability and performance.

The remainder of the paper proceeds as follows. The next section motivates our work. Section 3 discusses background and related work material. Section 4 describes our experimental methodology and workloads. Section 5 details the results from our VMM characterization. Section 6 describes VirtualFence, whereas Section 7 presents its evaluation. Section 8 discusses different aspects of VirtualFence and our results. Finally, Section 9 presents our conclusions.

2 Motivation

Many users desire performance predictability. Although most cloud users may not require predictable VM performance, many actually do. For example, streaming (video/audio) and gaming applications typically seek

to achieve a consistent rate (e.g., displayed frame rate) rather than the highest performance, if that performance might introduce unpredictability (jitter). There are also many cases where repeatable behavior is important, such as performance tuning, debugging, and diagnosis. In fact, it is impossible to evaluate the impact of changes to an application in the cloud, if its performance may constantly be affected by consolidation. Finally, many applications implement workflows/pipelines (e.g., Nutch [10], genome analysis [11]), where the performance that can be achieved in each stage depends on the expected performance of a previous stage. Properly designing such applications for the cloud is impossible if the performance of each stage can vary widely.

Predictability would benefit cloud providers and users. As predictability is important to many users, we argue that IaaS cloud providers should offer a new class of predictable-performance service. Importantly, this class of service should *not* replace their existing (predictability-oblivious) services. *Users who do not require predictability can still use the existing services.* Rather, the new class should be an additional service that uses a separate set of tightly managed hardware resources. (Section 8 discusses combining the two classes of service onto the same hardware infrastructure.) The amount of resources to be purchased for the new class of service can be small at first, and be increased as demand for predictable behavior increases. Current IaaS providers already offer a range of other classes of service, such as the Cluster Compute and Cluster GPU service classes of Amazon EC2.

The tight management of resources in this class of service would: (1) enable the provider to charge for exactly the pre-defined levels of performance and predictability that its users require; (2) enable the provider to conserve energy when resources are not used to guarantee the performance paid for by its users. For example, unneeded CPUs or memory modules can simply be turned off. In fact, the tight management of resources allows the provider to provision just enough servers, for additional (operating and capital) cost savings. Obviously, current services can use fewer servers by overbooking resources, but they cannot provide predictability; and (3) create an obvious relationship between the resources that customers pay for and the performance that can be achieved with those resources, i.e. users never complain that the performance of their VMs suddenly got worse (when the provider stopped dedicating more than the minimum set of contracted resources). Gulati *et al.* mention some of these same benefits to limiting maximum allocations [4].

Cloud users can also benefit from predictability for three reasons: (1) they can rely on it to implement applications for which predictability is more important than receiving as many resources as are available; (2) predictability can lower their cloud costs when the provider can save money by conserving energy or provisioning their data centers more tightly; and (3) they can predict their cloud costs into the future with the certainty that their VMs' performance will never be affected by changes in provider-side resource allocation.

Importantly, our approach enables a wide variety of performance and predictability levels. For example, a user may purchase $1/n$ (n is defined by the provider) of a PM and receive the performance that this fraction of resources produces. If this performance is not good enough, the user can purchase any multiple s ($s \leq n$) of this fraction. The larger the fraction, the more consistent the performance will be. Ideally, the price of a VM instance with s slices of a PM in the predictable service would be the same as (or only slightly higher than) an instance with the same amount of resources on average in the existing predictability-oblivious service.

Client-side throttling does not work. One might think that providing an additional class of predictable service is unnecessary, as delays can be added on the client side to achieve predictable behavior. However, this intuition is incorrect. As the client does not know how bad VM performance may get in the future (the *WorstPerf* performance mentioned above), it cannot target a performance level that is guaranteed to be consistent. In fact, even if the client knew the value of *WorstPerf*, it would have to set its delay to achieve this worst performance. Any other value could be exceeded.

3 Background and Related Work

Disk Drives and I/O Interference. Many recent studies have established that I/O interference prevents VMs from achieving predictable performance [2, 4, 5, 6, 7, 9]. This is primarily due to the mechanical nature of HDDs. Disks make I/O performance highly dependent on the locality of accesses across VMs, variability in I/O sizes, request priorities, and access burstiness [4].

Many previous efforts to address this problem have focused primarily on resource scheduling techniques, seeking to provide proportional allocation of I/O resources with strong isolation [12, 13, 14, 15]. Argon [14] in particular shares common techniques with VirtualFence, such as space partitioning of caches (memory caches in the case of Argon) and time partitioning of I/O access time. However, our focus on predictability instead of isolation leads to fundamental differences, including non-work-conserving allocation policies and static configuration parameters.

Other works seek to provide proportional allocation while supporting latency-sensitive applications [16, 17, 15, 18]. mClock [4] goes further by providing propor-

tional allocation, subject to minimum reservations and maximum limits. In principle, mClock can support predictability when the maximum limit is set equal to the reservation. However, the authors did not consider this scenario. Moreover, mClock does not consider the properties of storage devices and how they impact predictability. VirtualFence does and, hence, combines SSDs and HDDs. For this reason, the two systems cannot be fairly compared quantitatively. Alternate approaches using sophisticated machine learning techniques to meet end-user QoS targets have also been studied [19].

Besides the differences described above, our work identifies predictability, a stricter form of isolation, as desirable, and is the first to study hybrid SSD/HDD systems in this context. We also quantify the lack of predictability across a range of VMM architectures and configurations.

Hybrid SSDs and HDDs. Recent advances in Flash memory SSDs have led to studies exploiting their high-speed random reads, low power consumption, feature size, and shock resistance [20, 21, 22, 23, 24, 25]. Most research on SSD organization has focused on either using SSDs as HDD replacements [21, 26], or using SSDs as a caching layer in the storage hierarchy [23, 24, 27, 28, 29]. The primary focus of these works is on the performance benefits of SSDs. The energy implications of SSDs have also been studied [30, 31].

In comparison to these efforts, our work is the first to quantify the impact of SSDs on VM performance predictability. In fact, we present the first characterization of predictability on SSDs across a spectrum of VMM types. Interestingly, our results show that SSDs do *not* guarantee predictability: in write-intensive workloads, their predictability is actually *worse* than that of HDDs. Moreover, SSDs exhibit cost-per-byte that still cannot compete with those of HDDs. VirtualFence combines the advantages of both storage media to promote predictability.

4 Methodology

In this section, we first define the metric we use to evaluate VM performance predictability. We then describe the virtualized systems we study in Section 5, each of which comprises a different combination of VMM, approach to storage virtualization, and storage device. Finally, we describe the experimental environment, including the workloads and hardware execution platform.

4.1 Performance Deviation Metrics

We measure performance deviation as the percentage performance degradation when a VM runs in the presence of other VMs compared to when it runs alone on the physical host. Specifically, let P_I be the (initial) performance of a VM when running alone, and P_D be the (degraded) performance of the VM when co-located with other VMs. Then, the amount of deviation is given by $\delta = |\frac{P_I - P_D}{P_I}| \times 100\%$. When multiple (say n) VMs executing the same workload are used in an experiment, we report the average deviation $E[\Delta] = \frac{\sum_{i=1}^{n} \delta_i}{n}$, where $\Delta = \{\delta_1, \delta_2, ..., \delta_n\}$.

As we show below, performance deviation is often different for throughput and response time. Thus, throughout the paper, we study deviation for both metrics.

4.2 Virtualized Systems

We measure performance deviation across four well-known VMMs and two types of persistent storage, SSD and HDD. First, we study VMWare's Workstation (8.0), where the host OS is responsible for device I/O. In contrast, we also study VMWare's ESXi Server (5.0), where guest VM I/O operations trap into the VMM, which then directly accesses the I/O device. Third, we study Xen (4.0.1), where a split-driver model is used for device I/O. Here, an isolated device domain (Dom0) runs the device drivers. Therefore, VMs (guest VMs, which are referred to as DomU) pass their I/O requests through to Dom0 on I/O accesses. Like VMWare's ESXi, the Xen VMM runs on the hardware directly. Finally, we study KVM (0.12.5), where the Linux kernel is equipped with native virtualization capabilities. As such, it relies on the Linux kernel to actually accomplish device I/O.

As VMWare Workstation and KVM run on top of Linux, and Xen incorporates drivers from Linux, we also run our workloads as processes on a Linux setup.

Finally, we study both file-based and disk-partition-based storage of VM persistent data in Xen and KVM.

4.3 Workloads

We use workloads from Filebench [32], a popular framework for measuring and comparing file system performance. Specifically, we use Fileserver, Mailserver, and Webserver. Fileserver emulates a server hosting directories owned by multiple users; Mailserver focuses on mail operations and has an I/O mix of a read per sync write; and Webserver emulates a server that services a read-only workload.

To measure deviation, we compare a VM's I/O performance when running alone against that when running with three other VMs. Specifically, we configure workloads of four VMs running concurrently, each of which executes the same Filebench application. One of the VMs is configured to produce a low-intensity I/O load that is approximately 8% of the storage system's saturation load. Each remaining VM is configured to produce approximately 24% of the storage system's saturation

load. (We produce a load of *x%* of saturation by finding the saturation throughput for each application, and adjusting the number of threads and thread I/O rate to achieve *x%* of that throughput.) Overall, the four VMs reach 80% of saturation, representing an aggressive consolidation scenario. We then compare the low-intensity VM's performance to when it runs alone, and the performance of each of the three higher-intensity VM to when it runs alone. Note that we scale the load to maintain a constant utilization level (80%) across storage systems (HDD, SSD, and VirtualFence).

We call the above setup a 4-VM heterogeneous workload and use it as the primary workload for our study for two reasons. First, we want to study how higher-intensity VMs affect the predictability of low-intensity VMs. Second, we want to study deviation when VM consolidation leads to high utilization levels. In Section 7, we also study homogeneous workloads and systems with low-intensity VMs only, resulting in low aggregate loads.

4.4 Experimental Platform

We run our experiments on a server equipped with a 2.4GHz 4-core Xeon CPU (each core supports two hardware threads), 8GB of RAM, a 60GB SSD, and a 160GB 7200RPM SATA HDD. According to its datasheet, the HDD has an average seek time of 11ms and full stroke time of 22ms. The SSD is spec'ed with random read performance >20,000op/s and random write performance >5,000op/s. We measured erasures, including garbage collection, to take approximately 3.5ms-4ms in a write-only benchmark. The guest OS in the VMs is always a Debian installation with Linux kernel version 2.6.32. The host OS for VMWare Workstation and KVM is the same Linux installation.

The Linux 2.6 kernel has 4 commonly used disk schedulers: Noop, Deadline, Anticipatory, and Completely Fair Queuing (CFQ). We set the disk schedulers of the guest and host (including Xen's Dom0) Linux systems to Noop and CFQ, respectively. We choose Noop in the guest OS to isolate the impact of the VMM's I/O scheduling. We choose CFQ for the host OS because it minimizes deviation when not using VirtualFence.

In all experiments, we allocate 512MB of memory to each VM and pin it to a core to minimize the impact of VMM CPU scheduling. We run at most 4 VMs simultaneously so that each VM can be allocated an entire core.

5 VMM-Driven Performance Deviation

Our study begins with characterizing I/O performance deviation for the four different VMMs, when using file-based vs. disk-partition-based storage virtualization, and when using an SSD vs. an HDD device. Figure 1 plots the deviation experienced by the low-intensity VM in the 4-VM workloads described in Section 4.3. We plot deviation for both throughput and response time. In both cases, the lower the measure, the better. The range markers represent the minimum and maximum values from three experiments, whereas the bar represents the average. We do not show the results for the high-intensity VMs because they exhibit similar trends.

Many interesting observations arise from these graphs. First, deviation is endemic across all system configurations for both throughput and response time. For example, Xen's deviation ranges from ∼17-32% for throughput and ∼327-433% for response time when running on an HDD, and ranges from ∼9-28% for throughput and ∼166-479% for response time when running on an SSD.

These high deviations show that performance degrades significantly when running with other VMs. Much of the degradation arises from interference between the 4 VMs. The higher aggregate workload leads to increased queueing times, resulting in higher response times. When running on the HDD, interleaved requests from multiple VMs increase seek overheads, resulting in loss of throughput and higher response times. Interestingly, the response time deviations for workloads with writes (Fileserver and Mailserver) are worse when running on the SSD compared to the HDD. The main reason is that SSD writes sometimes cause expensive Flash block erasures, which may affect accesses from all VMs.

Second, deviations for all workloads and HDD/SSD combinations are worse when a VMM is used vs. standalone Linux. Since many VMMs operate on a Linux base, such as KVM or Xen's Dom0, this further demonstrates that VMMs intrinsically increase deviation.

Third, file-based virtual disks exhibit slightly worse predictability than partition-based virtual disks. Two potential sources for the greater deviation are the file system code running inside the VMMs, and the need to update the metadata of the files implementing virtual disks when the hosted VMs access their virtual disks.

Finally, Figure 1 shows that KVM exhibits the highest deviations. The main reason is that, by default, KVM propagates writes coming from the VMs directly through to the storage device to improve reliability. When this feature is disabled, the KVM deviations become comparable to those of the other VMMs.

Although the results are not shown here, we have also measured performance deviation for a low load scenario, where the aggregate load reaches only 20% of saturation. Throughput deviation is relatively low in this scenario but response time deviation is still significant.

Taken together, these observations mean that current systems lack I/O predictability. Large deviations may arise from the resource allocation policies; specifically, work-conserving policies link VMs' I/O resource alloca-

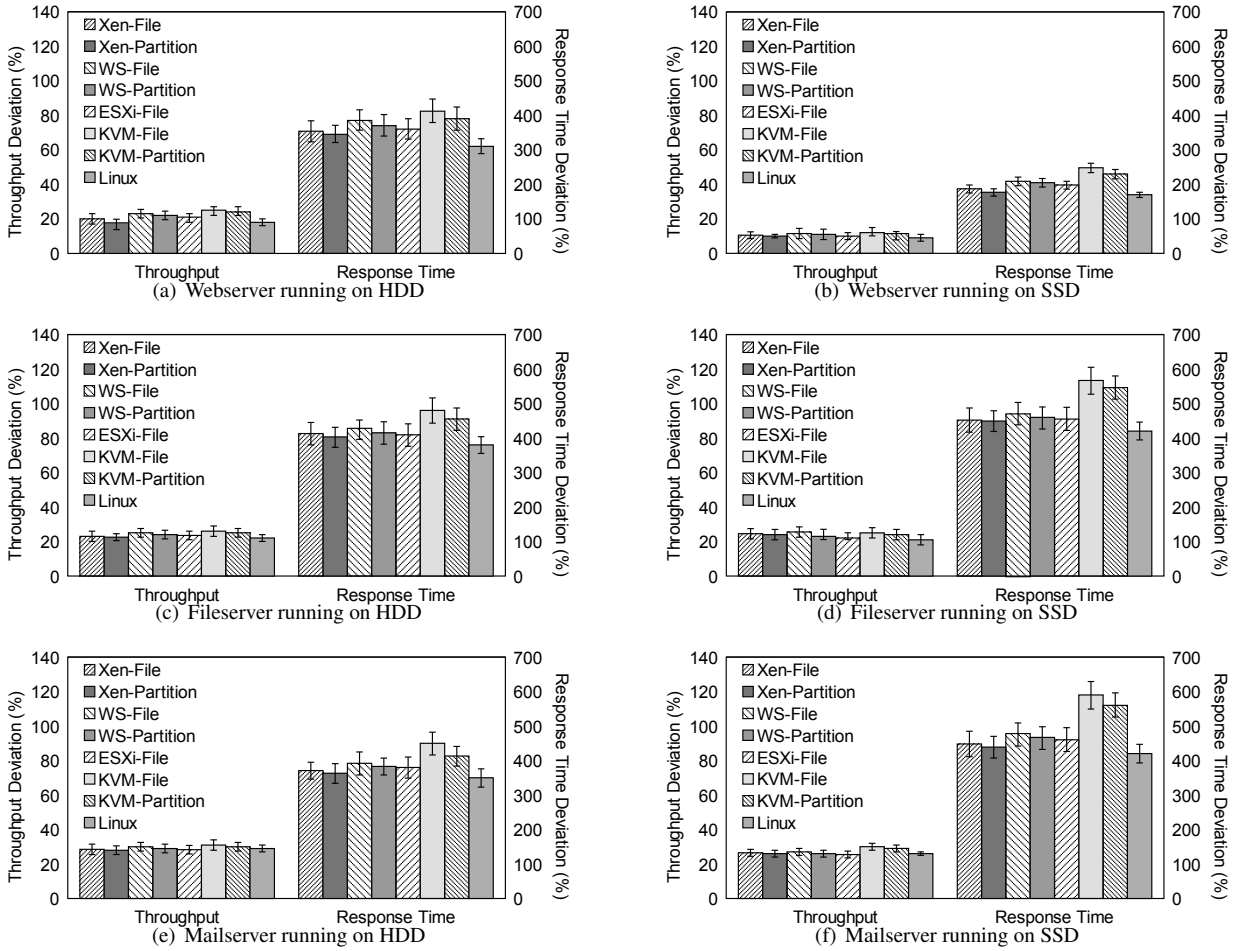


Figure 1: Performance deviation experienced by the low-intensity VM in 4-VM heterogeneous workloads.

tion to the number of co-located VMs, causing deviations as this number changes. Large deviations may also arise from device-specific characteristics. Interleaving HDD requests from different VMs leads to higher seek overheads, whereas SSD erasures initiated by a VM can interfere with operations from other VMs.

6 VirtualFence

VirtualFence uses three techniques to reduce performance deviation between VMs whose virtual disks are stored on the same physical disk: (1) a non-work-conserving time-division I/O scheduling algorithm with relatively coarse-grained time quanta, (2) a small persistent SSD cache in front of a much larger HDD, and (3) space partitioning of both the HDD and the SSD cache.

The non-work-conserving time-division I/O scheduling serves two purposes. First, it ensures that the resources allocated to a VM are (mostly) constant regardless of the number of co-located VMs. Second, it avoids fine-grained interleaving of requests from different VMs to reduce inter-VM interference; for an HDD, this reduces seek overheads between operations from the same VM, whereas for an SSD, it reduces the interference of erasures from one VM on accesses from other VMs.

Despite the non-work-conserving policy, a system with only HDDs would still suffer some performance deviation when multiple VMs are co-located; as the system switches from serving 1 VM to another, the HDD's head will have to move across partitions, leading to higher seek time for the first HDD operation, and so performance deviation. We limit the impact of this deviation by putting the SSD cache in front of the HDD. With a reasonable hit ratio in the SSD cache, we may eliminate some of these expensive HDD accesses. Moreover, the SSD cache significantly increases the performance of the virtual disk, so that the expensive first HDD operation is amortized across many more operations.

Finally, the space partitioning of the HDD limits the seek overheads between operations from the same VM, while the space partitioning of the SSD cache is a non-work-conserving space allocation scheme to ensure constant cache allocation for each VM.

Note that VirtualFence deals solely with storage I/O

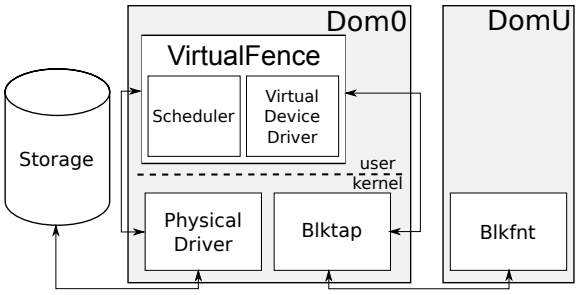


Figure 2: VirtualFence architecture.

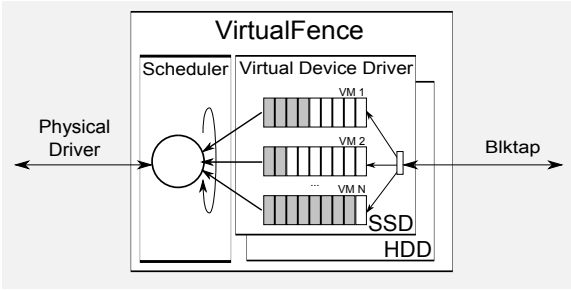


Figure 3: Driver with non-work-conserving time partitioning.

resources, assuming that other resources (e.g., CPU cores and memory) are also managed with predictability-preserving schedulers.

6.1 Prototype

We have implemented a prototype VirtualFence system in the Xen VMM version 4.0.1, using the blktap user-level toolkit [33]. This prototype includes a device driver and a scheduler. The device driver instances—a separate instance of the device driver is used to service each distinct virtual disk—and the scheduler each runs as a user-level process in Dom0 (Figure 2). We have not experimented with multiple SSDs and HDDs but it should be trivial to extend our prototype, as long as the caches for virtual disks co-located on a single HDD are themselves co-located on a single SSD. We discuss multiple SSDs and HDDs again in Section 7.

The SSD cache holds two types of persistent data: (1) blocks cached from the HDD, and (2) metadata describing the state of each cache block (e.g., valid bit, HDD block number). The driver implements the data structures needed to support an LRU replacement policy in volatile memory, including an LRU list of blocks, a write list that points to dirty blocks that need to be written to the HDD and then evicted, and a free list. At start up, the driver scans the SSD for all metadata, and builds all the in-memory data structures. No "last usage times" are kept across system restarts.

The LRU maintenance is simple. A background thread attempts to maintain the size of the free list above a threshold size by evicting the oldest entries in the LRU list as needed. Dirty blocks to be evicted are moved to the write list while the writes to the HDD are outstanding. If the free list ever reaches a low watermark threshold, processing of incoming requests is halted until the free list grows above the low watermark.

The driver uses asynchronous I/O to read and write data from/to both the SSD and HDD.

6.2 Space Partitioning

VirtualFence uses a separate partition of an SSD as a cache for each virtual disk co-located on the same HDD. (Note that while the sizes of the partitions in our evaluation experiments are the same, it is trivial to make the cache size proportional to the size of the virtual disk, so that larger virtual disks also have larger caches.) We have also implemented a variation that uses a single SSD partition as a shared cache across multiple virtual disks to quantify the impact of space partitioning on deviation.

The implementations of the two variants are slightly different. In the space-partitioned version (i.e., VirtualFence), the caching code runs inside the driver process that manages each virtual disk. In the shared-cache variant, the caching code runs in a separate process (that must then interact with the multiple drivers managing the virtual disks sharing the cache). This structure makes the shared cache implementation slightly less efficient than VirtualFence because of the inter-process communication between the cache manager and the drivers.

6.3 Time Partitioning

Our non-work-conserving I/O scheduler assumes that a physical SSD/HDD pair is used to service at most n simultaneous active virtual disks, and so divides access to the physical disks into n equal-sized time slots. When a VM starts running on a host, its virtual disk is allocated one or more I/O slots (depending on how much of the host's I/O resources is assigned to that VM). The scheduler then round-robins between the slots, *leaving a slot idle* when it is unassigned or the assigned virtual disk does not have any I/O activities; utilizing these slots would break the non-work-conserving property of the scheduler. On the other hand, this property of the scheduler also impacts performance, as we discuss extensively in Section 7.3. Figure 3 illustrates our implementation.

The driver translates each user I/O request into requests to the SSD and HDD, and adds each type of requests to the appropriate device I/O queue. Each virtual disk has a distinct set of queues that are serviced dur-

ing the slots assigned to the VM.[1] The scheduler informs a driver instance when its assigned slot is scheduled, at which time it is allowed to forward requests to the SSD and/or the HDD until the slot time expires. A driver can end its slot early (see below), in which case the scheduler will lengthen the slot time appropriately the next time that slot is scheduled. If a driver overruns the slot time, the scheduler will deduct the overrun from the next slot.

To implement accurate time partitioning without losing performance, we need to send as many accesses as possible in a slot without running over the time allocated to the slot. In addition, it is more efficient to batch requests because of effects such as disk scheduling and fixed access overheads. Thus, our approach is to estimate the service times of batches of accesses, and to send the largest batch of accesses that is estimated to complete within the remaining time in the slot.

Our driver dispatches requests to the SSD and HDD in the same manner as follows. If there are no pending requests, then wait until a request arrives or the current slot terminates. If there are pending accesses, and at least the first access is estimated to complete within the remaining time in the slot, find the largest batch that is estimated to fit within the remaining time. (We explain our prediction model below.) After the completion of a batch of requests, if time remains in the slot, then repeat.

The driver will end a slot early if the first pending HDD request is estimated to take longer than the remaining time in the slot. This is because HDD resources are much more constrained than the SSD, and thus, when the remaining slot time cannot be used for accessing the HDD, it is better to "credit" it to the next slot instead of wasting it. On the other hand, if a batch of HDD requests overruns the slot time, while waiting for the batch to complete, slowly send SSD requests to not waste this time while not causing even more slot delay by having to wait for the completion of a large batch of SSD requests.

Predicting HDD request service times accurately can be quite complicated [34, 35]. For our purposes, however, it is sufficient to use a simple piece-wise linear function that predicts the access time of a request based on the distance between the block being requested and the block requested by the immediately preceding request. When predicting the service time of a batch, we order the requests using the block addresses under the assumption that the disk scheduling algorithm includes some form of scanning. We parameterize the prediction function for our specific HDD by benchmarking the service time of a large number of random batches of accesses, each batch with a random mix of reads and writes. Our approach leads to reasonably accurate prediction of batch service time: for a benchmark issuing batches of random sizes averaging 50 accesses and 336ms batch service time, over 70% of our predictions are within $[-5\text{ms},5\text{ms}]$ of the actual batch service times.

[1] We currently assume that each virtual disk is only attached to a single VM and that a VM attaches to at most one virtual disk.

Variant	HDD	SSD	Cache	NWC
HDD+NWC	x			x
SSD+NWC		x		x
Hybrid/Shared	x	x	Share	
Hybrid/Shared+NWC	x	x	Share	x
Hybrid/Partitioned	x	x	Partition	
VirtualFence	x	x	Partition	x

Table 1: Variants of VirtualFence comprising different combinations of predictability-enhancing techniques. The HDD and SSD columns show whether a variant uses an HDD and SSD device, respectively. When both devices are used, the SSD acts as a cache for the HDD. The Cache column shows whether the SSD cache is shared or partitioned. The NWC column shows whether non-work-conserving time partitioning is used.

Measurements of the SSD used in our experiments show that request service times can be approximated using a linear function that depends on the number of requests simultaneously submitted to the asynchronous I/O system. We parameterize the prediction function for our SSD by benchmarking the service times of a large number of batches of accesses, where each batch has a random size (between 1 and 100), a random split between reads and writes, and random target blocks. For a benchmark with an average batch size of 49 accesses and an average batch service time of 19.4ms, over 83% of our predictions were within $[-250\text{us},250\text{us}]$ of the actual batch service times.

7 Evaluation

We now explore VirtualFence's effectiveness in providing performance predictability. All experiments are performed using the workloads and experimental platform described in Section 4. The SSD cache block size is set to 4KB to match the default 4KB block size of the HDD. We also adjust the SSD cache size to explore the impact of different hit rates. We use the notation VirtualFence(X%,Yms) to denote a VirtualFence system with a time-sharing slot size of Yms, and the SSD cache empirically sized to achieve a hit rate of X%. We explicitly set the SSD cache hit rate to systematically isolate its impact; in practice, administrators would set the SSD partition size (and the number of time slots) for each VirtualFence virtual disk based on the QoS/resources promised to the disk's owner and the number of virtual disks to be consolidated on the physical server.

To isolate the contributions of the different features of VirtualFence toward increasing predictability, we also measure deviation for many incomplete variants of VirtualFence. Table 1 lists these variants. The first two

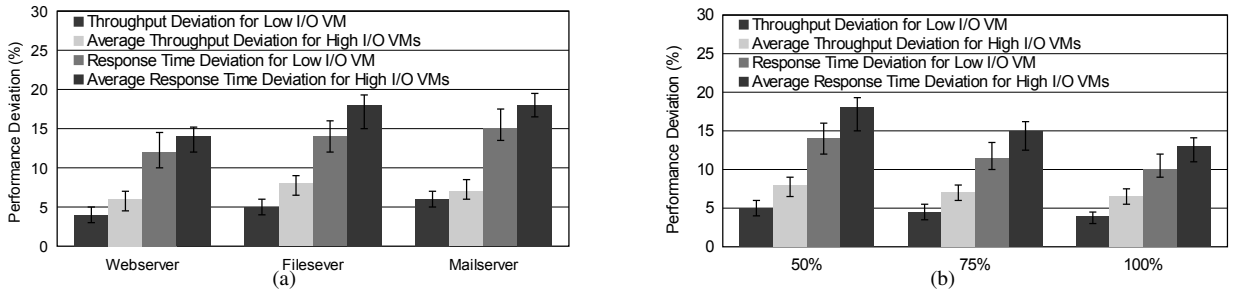


Figure 4: Deviation when running (a) the 4-VM heterogeneous workloads on VirtualFence(50%,20ms), and (b) the 4-VM heterogeneous Fileserver workload on VirtualFence(X%,20ms), with $X \in \{50\%, 75\%, 100\%\}$. The range markers show the minimum and maximum values from three experiments whereas the bar shows the average.

variants, HDD+NWC and SSD+NWC, are designed to isolate the benefits of non-work-conserving time partitioning. The Hybrid/Shared variant uses an SSD cache in front of the HDD, but the entire cache space is shared between multiple virtual disks. Hybrid/Shared+NWC extends this variant with non-work-conserving time partitioning. Hybrid/Partitioned is VirtualFence without non-work-conserving time partitioning, isolating the benefits of space-partitioned SSD caches.

7.1 Performance Deviation

VirtualFence. We begin by showing VirtualFence's effectiveness at reducing performance deviation. Figure 4(a) shows the measured deviation when running the 4-VM heterogeneous workloads on VirtualFence(50%,20ms). Figure 4(b) shows the measured deviation when running the 4-VM heterogeneous Fileserver workload, which experiences the highest deviation, on VirtualFence with hit rates ranging from 50% to 100% and a time-sharing slot size of 20ms.

Figure 4(a) shows that VirtualFence is successful at reducing deviation in both throughput and response time, compared to a system without VirtualFence (Figure 1). In fact, VirtualFence produces lower deviations regardless of storage device or approach to virtualizing storage. For the low-intensity VM, all deviations are $\leq$15%, compared to throughput deviations of $\geq$31% and response time deviations of $\geq$443% without VirtualFence. Furthermore, deviations are always lower than 19% when the SSD cache affords a 50% hit rate. Figure 4(b) shows that deviation decreases as the SSD hit rate increases.

The results are positive in terms of raw performance as well. For example, Fileserver file accesses (97KB on average) by the low-intensity I/O VM take an average of 19ms, when the VM runs in isolation on the HDD configuration. When the same VM runs co-located with 3 high-intensity VMs, the average file access time increases to 98ms. We increase the I/O intensity of each VM by a factor of 3.3x in VirtualFence(50%,20ms) to achieve the same utilization as in the HDD case. Despite the much higher I/O intensity, the low-intensity I/O VM experiences an average file access time of 59ms when running alone, and just 64ms when co-located with 3 high-intensity I/O VMs, under VirtualFence(50%,20ms).

Isolating the contributions of different features. Figure 5 plots performance deviation when the Mailserver workload is run on the variants (including the full VirtualFence implementation) listed in Table 1. Performance deviations for HDD and SSD (from Figure 1) are also shown as baselines. These results are representative of all three Filebench workloads.

First, this figure shows that VirtualFence achieves performance predictability close to that of SSD+NWC. Specifically, SSD+NWC achieves 3% and 10% throughput and response time deviation, respectively, whereas VirtualFence achieves 6% and 12%. SSD+NWC represents the best case scenario since it includes space partitioning (each VM is given a separate SSD partition), non-work-conserving scheduling, and storage completely on the SSD. The fact that these two systems achieve almost the same predictability shows that our caching approach is effective, allowing VirtualFence to extend the predictability benefits of (expensive) SSDs to much larger (and cheaper per byte) HDDs with small SSD caches.

Second, results for HDD+NWC suggest that non-work-conserving time partitioning can also be effective in reducing deviation when not using an SSD cache. However, the movement of the disk head between partitions when changing between time slots assigned to different VMs is sufficiently large that HDD+NWC with a 20ms slot size still incurs a 13% throughput deviation and a 33% response time deviation. As we show in Section 7.3.2, increasing the slot size to attack this source of deviation also increases the response time observed by a VM running alone on a host. As already mentioned, this source of deviation also exists in VirtualFence but is mitigated by the SSD cache.

Third, at this hit rate, non-work-conserving time partitioning achieves higher predictability than using an SSD cache: HDD+NWC has lower deviations than both

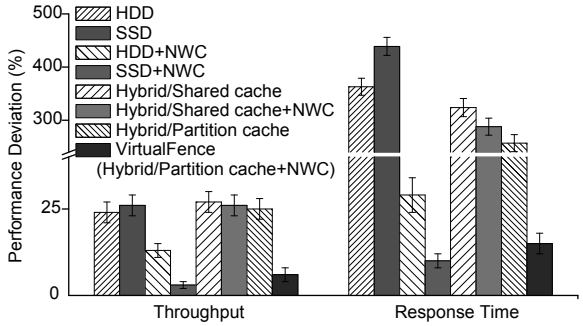


Figure 5: Deviation when running on VirtualFence(50%,20ms) compared to various incomplete variants of it. Each bar shows results for the low-intensity VM in the 4-VM heterogeneous MailServer workload. The cache size for Shared-cache versions is equal to the sum of the caches in the Partitioned-cache cases.

Hybrid/Shared and Hybrid/Partitioned. Interestingly, HDD+NWC is also better than Hybrid/Shared+NWC, implying that the interference at the shared cache negates some of the benefits of NWC. Of course, as the hit rate increases, the relative advantage of using NWC vs. an SSD cache will likely change.

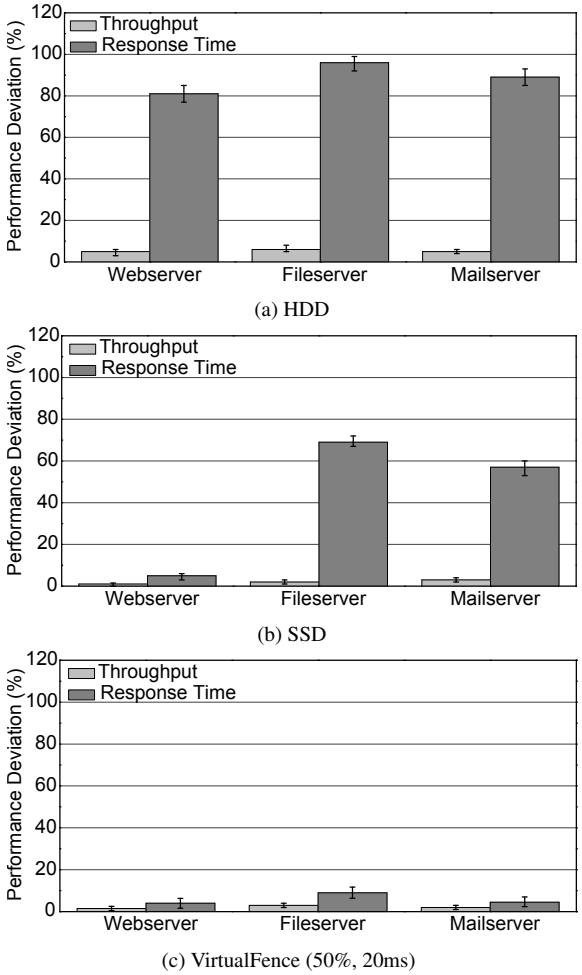
Fourth, as expected, a shared SSD cache produces worse predictability than a partitioned cache. A shared cache can produce higher absolute performance; e.g., it may benefit an I/O-intensive VM running by itself. However, it would hurt predictability when the VM is co-located with other VMs and so must share the cache.

Finally, all three techniques used in VirtualFence contribute to increasing predictability; VirtualFence achieves higher predictability than the other configurations, except for the much more expensive SSD+NWC.

7.2 Performance Deviation at Low Load

The previous subsection shows that VirtualFence is effective in aggressive consolidation scenarios. In this section, we consider what happens when the aggregate load is low, representing more conservative scenarios.

Figure 6 shows the average throughput and response time deviation under low aggregate loads for SSD, HDD, and VirtualFence(50%,20ms). Each workload runs 4 homogeneous VMs, where each VM is configured to generate 5% of the storage system's saturation load (i.e., each VM is less I/O intensive than any VM we have discussed so far). These results show that, even at these low loads, HDD experiences very high deviation. This is because requests from multiple VMs are interleaved, leading to much higher seek overheads. Although throughput deviation for the SSD is low, response time deviation is still significant for the workloads with writes (greater than 50% for both Mailserver and Fileserver). All deviations are below 14% for VirtualFence.



(a) HDD

(b) SSD

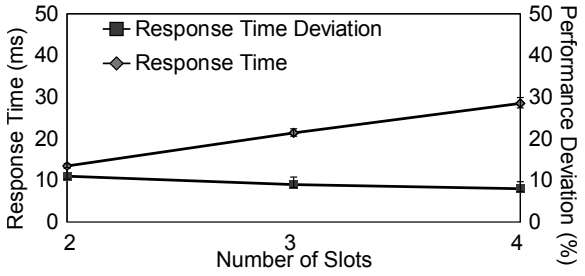
(c) VirtualFence (50%, 20ms)

Figure 6: Deviation when running 4-VM homogeneous, low-rate I/O workloads.

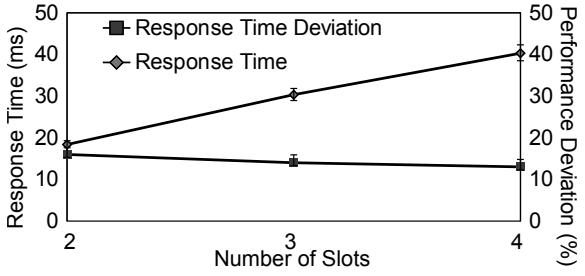
The raw performance results for this low-load workload are interesting as well. For example, Fileserver file accesses by each VM take an average of 21ms, when it runs in isolation on the HDD configuration. When the 4 low-intensity I/O VMs are co-located, the average file access time increases to 42ms. We increase the I/O intensity of each VM by a factor of 2.7x in VirtualFence(50%,20ms) to achieve the same utilization as in the HDD case. Despite the much higher I/O intensity, each VM experiences an average file access time of 35ms when running alone, and just 40ms when it is co-located with the other 3 VMs, under VirtualFence(50%,20ms).

7.3 Performance vs. Predictability

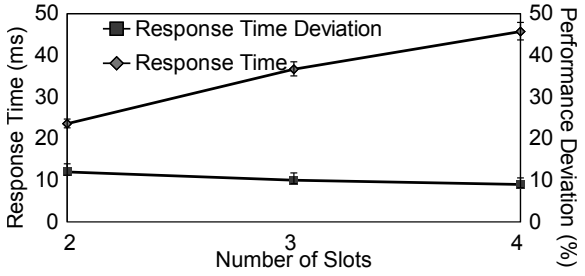
In this subsection, we explore the performance vs. predictability tradeoff that VirtualFence exposes. In particular, we explore the performance deviation and raw performance impact of its two key parameters: the number of VM slots per PM, and the length of each slot.



(a) Webserver workload.

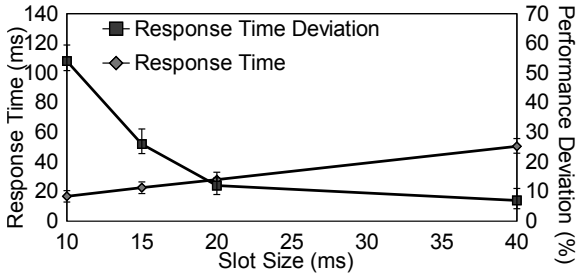


(b) Fileserver workload.

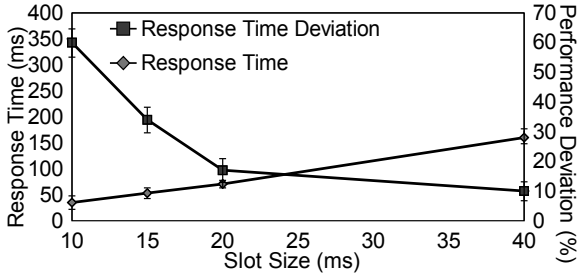


(c) Mailserver workload.

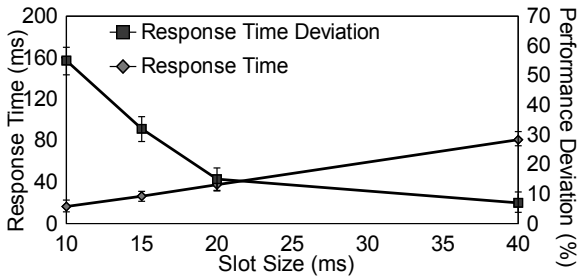
Figure 7: Number of slots and response time trade-off.



(a) Webserver workload.



(b) Fileserver workload.



(c) Mailserver workload.

Figure 8: Slot length and response time trade-off.

7.3.1 Impact of Number of Slots

The number of VM slots determines how aggressively a cloud provider will be able to consolidate VMs onto the same PMs. Figure 7 illustrates the impact of the number of slots on response time deviation and raw response time of VirtualFence(50%,20ms). For each number of slots n, we assume n VMs, and configure each VM to generate only $\sim$5% of the VirtualFence(50%,20ms) saturation load. Such a low aggregate load is a particularly challenging for VirtualFence raw performance-wise, because it may produce significant average waiting times.

As the figure illustrates, VirtualFence produces lower response times as we decrease the number of slots (while keeping the slot length fixed). The reason is that fewer slots also means lower waiting times, as each VM is allotted a higher fraction of time. From the opposite point of view, raw response time worsens linearly with increasing number of slots, because VirtualFence will not service a request until its corresponding slot is scheduled.

Interestingly, response time deviation decreases slowly as we increase the number of slots. With a large number of slots, deviation would approach 0%, because the waiting time would overwhelm the single disk head movement in the first request of each slot.

Clearly, there is a tension between wanting a small number of slots to reduce average response times and wanting to increase the number of slots to improve predictability. Fortunately, VirtualFence makes predictability reasonably good even with only a few slots. Thus, we would like to set the number of slots at the smallest number that will enable enough consolidation.

7.3.2 Impact of Slot Length

The key source of remaining deviation in VirtualFence is the need to move the disk head from one partition to another when changing slots assigned to different VMs. Thus, the slot length directly impacts VirtualFence's predictability: a longer slot better amortizes the inter-partition head movement cost among more requests. However, lengthening the slots also increases response time, because I/O operations issued outside of a VM's slot incur greater delay.

Assuming 4 slots, Figure 8 plots the response time de-

viation and average response time, as a function of slot length for VirtualFence(50%,10-40ms). We use the 4-VM heterogeneous workload (Section 4.3), and focus on the low-intensity I/O VM in Figure 8. This setup is challenging for VirtualFence predictability-wise, because it is almost certain that every first access in the low-intensity VM's slot will cause a disk head movement.

The figure clearly shows the tradeoff between lowering deviation by lengthening the slots against increased response time. For all workloads, lengthening the slots from 10ms to 20ms significantly reduces performance deviation. Further lengthening the slots to 40ms reduces deviation much more slowly at the expense of a further, essentially linear, increase in response time. Thus, a slot length of 20ms is the right tradeoff for our particular SSD and HDD devices. We have chosen this length as our a default for all previous experiments based on these results (and additional ones not shown here).

Again, there is a tension between wanting shorter slots for lower average response times and longer slots for better predictability. The slot length should be the shortest that will produce enough predictability.

8 Discussion

Recall from the Introduction that the goal of VirtualFence is to produce consistent raw performance between $WorstPerf$ and $BestPerf$, the extremes in performance in a predictability-oblivious, work-conserving scenario. However, to achieve this goal, VirtualFence must be properly configured as demonstrated in the previous section. Determining the best configuration involves experimenting with the devices at hand, and understanding how much users value performance vs. predictability. Since we propose VirtualFence for a new class of predictability-conscious cloud service, we expect our users to accept relatively low (but consistent) performance in exchange for good predictability. This would mean a tendency to prefer longer slots. Given that predictability is good even with few slots, the cloud provider can choose to use more slots (as long as performance is still acceptable to users) to enable more aggressive consolidation (lower costs). *As a target for "acceptable performance", the cloud provider can estimate $WorstPerf$ by using its existing (predictability-oblivious) service* and desired amount of consolidation. This value can be used in limiting the number and length of slots.

Clearly, in the predictability-conscious service, resources may go underutilized. The provider may then be tempted to adjust the VirtualFence parameters dynamically to adapt to current workloads and their I/O activities. Unfortunately, doing so could ruin predictability. *A better approach may be to combine the predictability-conscious and predictability-oblivious services onto the same hardware infrastructure.* For example, a VM that requires predictability could be given a fixed fraction of a PM (e.g., a slot of 20ms out of every 100ms, one-fifth of the SSD cache, and a separate disk partition), whereas many co-located predictability-oblivious VMs could fight for the remaining resources.

Importantly, VirtualFence enables cloud users to pay only for the performance that they (consistently) get. If they desire better performance, *they can purchase multiple slots while still retaining predictability.* Given that our system enables the cloud provider to reduce its costs through better resource provisioning and energy conservation, the user may end up paying roughly the same for multiple slots as she would pay for the equivalent of one slot in the absence of VirtualFence.

Finally, it is important to discuss two aspects of our study. First, VirtualFence does not partition *all* I/O resources across VMs. In particular, it does not partition the buffer cache in Xen's Dom0. We made this decision because (1) we find the hit ratio in that cache to be very low; and (2) partitioning the SSD space and the I/O access time is substantially more important for predictability. The low performance deviations that VirtualFence is able to achieve justify our choice.

Second, our evaluation of VirtualFence focuses on a single HDD (and associated SSD). However, our approach extrapolates to RAID or JBOD systems using a single SSD for caching. The reason is that VirtualFence would be built into the driver closest to the disk array and, thus, could partition the SSD space and I/O access time like the array were a single disk.

9 Conclusions

In this paper, we quantified the impact of storage medium, and VMM architecture and configuration on I/O performance predictability. The results showed that unpredictability is pervasive. Based on these results, we proposed VirtualFence, a software/hardware approach for achieving predictability at low cost. VirtualFence combines a small SSD cache in front of a much larger HDD, and non-work-conserving space and time partitioning. Our evaluation showed that VirtualFence can provide high predictability, as long as all of its features are used at the same time. We also identified and quantified the tradeoff between predictability and performance.

We conclude that it is possible to build performance-predictable storage systems with relatively simple software and hardware components, especially for those users that find predictability just as important as (or even more so than) raw performance.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, A. Madhavapeddy, R. Neugebauer, I. Pratt, and A. Warfield, "Xen 2002," *Technical Report of University of Cambridge*, 2003.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, and R. Neugebauer, "Xen and the Art of Virtualization," *SOSP*, 2003.

[3] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments," EPFL, Tech. Rep. 183449, 2013.

[4] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," *OSDI*, 2010.

[5] D. Gupta, L. Checkasova, R. Gardner, and A. Vahdat, "Enforcing Performance Isolation Across Virtual Machines in Xen," *Middleware*, 2006.

[6] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-Aware Virtual Machine Scheduling for I/O Performance," *VEE*, 2009.

[7] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An Analysis of Performance Interference Effects in Virtual Environments," *ISPASS*, 2007.

[8] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds," in *EuroSys*, 2010.

[9] J. Wang, P. Varman, and C. Xie, "Avoiding Performance Fluctuation in Cloud Storage," *HiPC*, 2010.

[10] Apache, "Apache Nutch," http://nutch.apache.org/.

[11] O. Flores and M. Orozco, "NucleR: A Package for Non-Parametric Nucleosome Positioning," *Bioinformatics*, 2011.

[12] L. Huang, G. Peng, and T. Chiueh, "Multidimensional Storage Virtualization," *SIGMETRICS*, 2004.

[13] W. Jin, J. Chase, and J. Kauer, "Interposed Proportional Sharing for Storage Service Utility," *SIGMETRICS*, 2004.

[14] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. Ganger, "Argon: Performance insulation for shared storage servers," *FAST*, 2007.

[15] J. Wu and S. Brandt, "The Design and Implementation of Aqua: An Adaptive Quality of Service-Aware Object-Based Storage Device," *MSST*, 2006.

[16] K. Duda and D. Cheriton, "Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler," *SOSP*, 1999.

[17] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient Guaranteed Disk Request Scheduling with Fahrrad," *Eurosys*, 2008.

[18] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage Performance Virtualization Via Throughput and Latency Control," *ACM TOS*, 2006. [Online]. Available: http://doi.acm.org/10.1145/1168910.1168913

[19] X. Zhang, K. Davis, and S. Jiang, "QoS Support for End Users of I/O-intensive Applications using Shared Storage System," *SC*, 2011.

[20] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *USENIX ATC*, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404019

[21] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-Performance Flash Disks," *SIGOPS Operating Systems Review*, 2007.

[22] C. Dirik and B. Jacob, "The Performance of PC Solid-state Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," *ISCA*, 2009.

[23] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash Based Disk Caches," *ISCA*, 2008.

[24] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," *SIGMOD*, 2008.

[25] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile Main Memory Storage System," *ASPLOS*, 1994.

[26] W. Josephson, L. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," *FAST*, 2010.

[27] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rostron, "Migrating Server Storage to SSDs: Analysis of Tradeoffs," *Eurosys*, 2009.

[28] T. Prichett and M. Thottethodi, "SieveStore: A Highly-Selective, Ensemble-Level Disk Cache for Cost-Performance," *ISCA*, 2010.

[29] Q. Yang and J. Ren, "I-CASH: Intelligently Coupled Array of SSD and HDD," *HPCA*, 2011.

[30] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-Efficient clusters for Data-Intensive Applications," *ASPLOS*, 2009.

[31] H. Tseng, H. Li, and C. Yang, "An Energy-Efficient Virtual Memory System with Flash Memory as the Secondary Storage," *ISLPED*, 2006.

[32] R. McDougall, "Filebench: Application Level File System Benchmark," http://www.solarisinternals.com/wiki/index.php/FileBench.

[33] D. Meyer, "Virtual Disk Backend Driver for Xen," http://wiki.xensource.com/xenwiki/blktap2.

[34] E. Shriver, A. Merchant, and J. Wilkes, "An analytic behavior model for disk drives with readahead caches and request reordering," *SIGMETRICS*, 1998.

[35] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes, "On-line extraction of scsi disk drive parameters," *SIGMETRICS*, 1995.