2016

Mitigating Interference During Virtual Machine Live Migration through Storage Offloading

Morgan S. Stuart
Virginia Commonwealth University

MITIGATING INTERFERENCE DURING VIRTUAL MACHINE LIVE MIGRATION THROUGH STORAGE OFFLOADING

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at Virginia Commonwealth University.

by

MORGAN S. STUART

Director: Dr. Xubin He,

Professor, Department of Electrical & Computer Engineering

Virginia Commonwewalth University

Richmond, Virginia

December, 2016

Acknowledgements

I would like to first and foremost acknowledge my advisor Dr. Xubin He for encouraging me to pursue graduate work, fostering a collaborative research environment, and his tireless effort to be as effective in the classroom as he is in the lab. I want to also acknowledge the members of STAR Lab, especially Tao Lu, Ping Huang, and Pradeep Subedi, for their candid advice, insights, and sense of humor. I also want to thank the many professors who've challenged me and inspired me, including Dr. Mike McCollum, Dr. Wei Zhang, and Dr. Robert Klenke. For my collaborators on projects outside of school, I will always appreciate your flexibility and encouragement. Finally, I would like to thank my parents, Bess and Andy Stuart, who instilled in me the creativity and confidence that got me this far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

MITIGATING INTERFERENCE DURING VIRTUAL MACHINE LIVE MIGRATION THROUGH STORAGE OFFLOADING

By Morgan S. Stuart

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at Virginia Commonwealth University.

Virginia Commonwealth University, 2016.

Director: Dr. Xubin He,
Professor, Department of Electrical & Computer Engineering

Today's cloud landscape has evolved computing infrastructure into a dynamic, high utilization, service-oriented paradigm. This shift has enabled the commoditization of large-scale storage and distributed computation, allowing engineers to tackle previously untenable problems without large upfront investment. A key enabler of flexibility in the cloud is the ability to transfer running virtual machines across subnets or even datacenters using live migration. However, live migration can be a costly process, one that has the potential to interfere with other applications not involved with the migration. This work investigates storage interference through experimentation with real-world systems and well-established benchmarks. In order to address migration interference in general, a buffering technique is presented that offloads the migration's read, eliminating interference in the majority of scenarios.

CHAPTER 1

INTRODUCTION

1.1 Overview

Full machine virtualization [1][2][3], which decouples the hardware and operating system to allow disparate environments atop a single physical machine, has become standard for large scale datacenters. Enterprises leverage virtual machines (VMs) to create infrastructures that are easier to manage and efficiently share resources, while still supporting a variety applications. The highly dynamic nature of these topologies has afforded the 'cloud' computing label that many are familiar with today [4].

In an effort to maximize performance, consolidate utilization, and avoid downtime, cloud providers can leverage live VM migration to transfer a customer's running VM to separate hardware resources with no discernible downtime [5]. This popular capability [6] helps create a highly dynamic topology, where the provider focuses on responding to the hardware and customer demands, and customers prioritize building and maintaining applications with little concern for the underlying hardware.

A primary struggle for cloud environments is maintaining the performance of hosted virtual machines, as any computation that shares resources may have a tendency to interfere with one another. The work presented here focuses on migration interference - the performance degradation caused by live migration in the cloud. Experimentation on real-world systems is used to guide the design of the proposed solutions as well as evaluate each approach.

1.2 Motivation

Sharing resources invites scarcity and contention, and the cloud environment is no different. A growing body of work has studied the issue of VM interfence - scenarios where co-located VMs compete for access to shared hardware resources, ultimately degrading the performance of all virtual machines involved [7]. The challenge of approaching this issue is further elevated by the strong abstractions provided by full machine virtualization, often referred to as the *semantic gap*. Any method that breaks this gap and requires introspection into a customer's VM, for any purpose, undermines core security and privacy drivers for cloud use-cases.

Using shared storage migrations can complete rapidly by only moving the VM's working memory, otherwise the migration must move the VM's large backing store. Research on live migration has largely focused on methods of improving migration performance, for both memory [8][9][10][11] and full-machine migrations [12][13][14]. However, as migration has become more prevalent, researchers have begun to recognize and address issue of interference caused by VM migration [15]. Methods tend to focus on reducing the overall data transfer [11][16] or optimally placing VMs [17][18] to avoid inter-machine interference on network, CPU, and memory resources.

1.3 Problem Statement

Full machine live migrations that require a full read of the VM's backing disk are burdensome to this already limited resource. Techniques for migrating a full virtual machine should make an effort to mitigate the interference that is likely to impact unsuspecting co-located workloads. Presented here is a dynamic buffering technique that uses an interference classifier to offload the VM's virtual disk at opportune times during a migration.

1.4 Contributions

1. Experimentation illustrating the mutual storage interference between a migrating VM and co-located workloads.

2. Online detection of interference given only the host machine's I/O metrics.

3. Design and implementation of host-level processes that leverage interference classification as means to offload virtual disk data when the probability of interference is low.

1.5 Organization

The remainder of this thesis is organized as follows. Details on the background needed to frame this work are provided in Chapter 2. Chapter 3 examines interference and introduces the design for *Migration Buffering* as means to more directly mitigate storage interference during full machine migration. Chapter 4 examines an implementation of this design, including an experimental evaluation. In Chapter 5, prior works in this area are outlined. Conclusions and future work are presented in Chapter 6.

CHAPTER 2

BACKGROUND

2.1 Datacenter Challenges

A traditional enterprise datacenter houses a variety of software applications integrated with lower-level storage, security, and network hardware. While maintaining performance of these systems is paramount, so is the need to keep the operation flexible and low-cost [19]. In these traditional datacenter models, a single application or a group of related applications may run inside a standard operating systems, atop a single physical machine. Larger applications may require multiple machines for improved performance or fault tolerance. However, with these configurations, applications are unlikely to utilize all of their resources at all times, potentially leaving costly resources idle for extended periods of time. Ideally, applications would be arranged across the datacenter's infrastructure such that when one application is idle, another is ready to make use of the now underutilized resources. However, sharing these resources in a traditional, non-virtualized environment is complex and potentially application-specific. High performance computing (HPC) centers typically approach this issue using monolithic batch scheduling systems to allocate jobs/applications to machines. For transactional workloads, such as web applications, batch scheduling does not deliver the on-demand low latency response that's typically required.

Ensuring high utilization of resources in an HPC environment does not present the same challenges. Instead, HPC applications are long-running and performance intensive, typically hosting bleeding edge technology with potentially complex configurations [20] [21]. This includes custom operating systems and low-level profiling.

These challenges are difficult to address in the more traditional static physical topologies [22].

2.2 Traditional Computer Architecture

Conventional organization of a physical computer can be broken into several layers, each abstracting complexities for the components that it interfaces [23] [24]. At the lowest level, the hardware layer encapsulates various microelectronic components and power systems, such as a CPU, main memory, storage, and networking interfaces. These components are generally highly complex and do not readily interoperate with each other. Instead, these components must be orchestrated in tandem using the instructions and control signals that they are fed. The operating system kernel is the low-level arbiter and controlling software that engages these components in order to make them functional at a more practical level [25]. The kernel houses drivers for the hardware that can translate general commands to the specific piece of hardware implementing the functionality. Different scheduling and monitoring mechanisms within the kernel ensure that the hardware is utilized to it's fullest while still avoiding errors or other faults. With the kernel in place providing a means to interact with different hardware configurations in a standard way, the final computing layer can be implemented as the user-space environment. This final layer describes what many are familiar with as the modern day computer, hosting potentially hundreds of processes simultaneously, such as internet browsers, word processors, or multimedia applications. User-space applications are scheduled by the kernel to have their instructions execute on the CPU. These instructions may be simple operations such as arithmetic, or they may instead be requests for a lower-level operation, which can only be facilitated on behalf of the application by the kernel.

In combination, these layers faithfully describe most non-virtualized computing

systems, including desktop computers, servers, and even most 'smart' mobile devices. As time-tested as this architecture is, in many scenarios it has it's limitations and difficulties as discussed in 2.1.

2.3 Virtualization

Specification of virtualization and virtual machines as they are commonly known today was outlined in [1]. Virtual machines are managed by a virtual machine monitor (VMM or hypervisor), which must provide an isolated and efficient means to run a machine, with only negligible difference when compared to the machine's execution on real hardware. Importantly, these criteria generally cannot simply be fulfilled by common interpreters or machine simulators. Thus, a VMM decouples the hardware from the OS kernel, allowing multiple systems to be run simultaneously, efficiently, and in isolation. This added layer of abstraction provides valuable utility and flexibility in a variety of scenarios, leading to an entire market of virtualization software targeted at both enterprises and home users.

Today's hypervisor's [2] [3] can be separated into two distinct categories - Type I hypervisors and Type II hypervisors. A Type I hypervisor is a VMM that acts both as the primary OS kernel for the overall system as well as the VM hypervisor. By placing the hypervisor at such a low-level, expensive context switching can be avoided in hopes of improving overall performance. In contrast, a Type II hypervisor operates in user-space like other applications, only making lower-level calls when necessary. The distinction between Type I and Type II can indeed become blurred when a non-VMM kernel loads modules at runtime that allow user-space VMMs more direct access to virtualization functionality contained within the kernel.

Whether a hypervisor is Type I or Type II, both can present either a virtual hardware interface that is entirely faithful to it's physical counterparts or instead

provide the VM with optimized virtual components. The former is referred to as full virtualization and is often utilized when virtualizing older systems that cannot be made virtualization-aware. The latter methodology is known as paravirtualization, and can greatly simplify the hypervisor's duties and even bring further optimization. It is now common to use paravirtualization to present optimized virtual devices to a more modern kernel ready with supporting drivers.

Hypervisors must expose a storage system to the VM - the virtual analog of a hard-disk drive or solid-state drive known as a virtual disk. There are numerous formats for storing virtual disk, but in general, the virtual disk can be understood as a file on the physical backing storage system. The hypervisor exposes this file through it's I/O device driver, ensuring that the VM cannot write outside this region.

Finally, it's important to impart that the hypervisor's narrow but low-level arbitration allows it near complete complete control over the systems that it's virtualizing. Hypervisor's can redirect and inspect I/O operations or even completely pause a VM. Possibly inspired by these properties, it's typical to refer to a hypervisor's VMs as *guests* to the hypervisor *host*, all running atop a *physical host* hardware system.

2.4 Live Migration

The hypervisor separates the OS from the underlying hardware in a well-defined, or narrow, way. With this layer in place Clark et. al. showed that a running virtual machine could be transferred to a hypervisor located on a separate physical machine, without shutting down the VM [5]. This operation is known as VM live migration. Variants requiring that the VM shutdown before transferring are known as offline migration. During live migration, the VM's working data is copied to the destination and any dirtied data is iteratively retransmitted. Once sufficient working data resides at the destination, the hypervisor can suspend the VM and transfer the small amount

of remaining state to the destination in what is known as the *stop-and-copy* phase of migration. A migration process that is able to reach the stop-and-copy phase with the desired downtime though iterative copying is said to have *converged*. If network bandwidth is low or the dirty rate of the VM workload is high, it's possible for the migration to not converge. Depending on the implementation, the migration may simply fail in error or be forced to prematurely stop-and-copy, resulting in a longer downtime.

The live migration implemented by [5] only moves the working memory and CPU state of the VM - the backing virtual disk must be accessible from both the source and destination hypervisors since it is not transferred. Now known as a memory-only migration, this technique limits the use-cases for live migration to those that only require movement locally, keeping the VM within reach of the host's shared storage.

Following the development of memory-only migration was extensive research into full machine migration, or migrations that uplift both the working memory and backing virtual disk of a VM. Three methodologies for this type of migration arose: *pre-copy* [5] [26], *post-copy* [27], and hybrid *pre+post-copy* [28] migration. During pre-copy migration, the hypervisor begins transferring the VM's backing virtual disk to the destination as soon as migration is begun, but the VM continues to run on the source system. Since the VM continues to run, it may write to regions that have already been transferred, requiring that the updated data be resent to the destination. Therefore, either the hypervisor must implement dirty block tracking in order to later resend the dirtied data, or the hypervisor must support write-mirroring. The hypervisor monitors the transfer speed and the amount of data remaining to be transferred. Once the amount of data is small enough to be transferred within the configurable downtime (typically only 10s or 100s of milliseconds to prevent disruption), the hypervisor pauses the VM's execution and transfers all remaining data and VM state to

the destination where the VM is immediately resumed. The initial work of [5] could be considered a memory-only form of pre-copy storage migration.

A post-copy migration inverts this process [27]. The basic VM state is transferred first to the destination and resumed, while the source begins sending the bulk data that remains at the source machine. If the VM requests data at the destination that has not yet been transferred, the data can be prioritized and sent immediately to minimize the delay inflicted on the I/O operation. This continues until all of the data at the source machine is transferred to the destination.

Combining pre- and post-copy migration techniques helps to compromise between the two techniques [28]. Under this hybrid scheme, data is initially copied in a manner similar to a pre-copy migration. However, rather than iteratively retransmit data until convergence, the pre-copy phase is bounded by some criteria, at which point the running state is transferred and a post-copy phase begins in order to transfer the remaining data.

While many schemes exist to perform full machine migration, all methods must read the entirety of the backing virtual disk at least once. This is intuitive given that the entire virtual disk must be transferred.

2.5 Cloud Computing

With rapid research and development of modern virtualization techniques came the growth of cloud computing. Though the term is often used ambiguously in marketing material, the term 'cloud' has concrete meaning for practitioners and researchers. Leveraging virtualization, public cloud providers allow customers to construct topologies in minutes, with costs often calculated by the hour. Virtualization allows the provider to expose an elastic infrastructure to their customers, atop the provider's own static physical datacenters [4] [29].

There are three types of clouds: *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS), and *Software-as-a-Service* (SaaS) [30]. IaaS providers allow customers to directly instantiate VMs, typically allowing for a variety of different OS options and pre-configured software. Customers can often further define there topology by configuring virtual LANs, load balancers, and firewalls. PaaS removes some of the complexity of IaaS by enabling customers to directly instantiate a supporting piece of software, such as a web server or database. The customer still has to provide content or make use of the newly created platform, such as providing the content for a web site or reading/writing to a database with their application. Finally, SaaS further removes the intricacies of IaaS and PaaS by allowing the user to purchase access to an instance of a software that's hosted by the provider. The provider no longer has to distribute updates, and can maintain a more consistent application code-base.

The work presented herein focuses on applications for IaaS providers and customers, but designs and implementations are valuable for any infrastructure that makes use of VMs.

CHAPTER 3

DESIGN OF STORAGE MIGRATION BUFFERING

In this era of expansive data collection and analysis, cloud infrastructures are commonly used to house massive datasets. In a study of private cloud data centers consisting of eight thousand physical machines and 22 petabytes of data, Birke et. al. [31] report that the average size of an individual VM's virtual disk usage is upwards of 60GB in size since 2012. In 2013, VMs used 76GB on average, with some using as much as 200GB. They found that a typical physical machine hosts 11 VMs on average, with I/O rates capable of exceeding well beyond 10MB/s per VM. Clearly cloud environments and substantial data usage go hand-in-hand.

Researchers and practitioners have historically focused on improving performance of VM storage migrations. This includes leveraging compression, deduplication, and optimizing the order of block transmissions. Only more recent work has approached issues arising from the resource contention caused by a live migration. Still, this work largely focuses on non-storage resources, such as CPU, memory, and network subsystems, largely ignoring storage.

In this chapter *Migration Buffering* is presented, a method of storage migration that prioritizes the storage I/O of co-located workloads, while still maintaining migration performance. Migration buffering uses an interference classifier to detect the likelihood of storage interference. When the likelihood is low, data is prefetched off of the backing storage system for use when interference becomes more likely. We first motivate our approach by demonstrating the severity of interference and then present our design.

3.1 Investigating Storage Interference

Migration of an entire Virtual Machine, including its virtual disk, risks inflicting considerable storage performance degradation on both the migrating machine and those machines co-located with it. This interference is due to the contention between I/O-bound workloads and the large read required to transfer the migrating machine's virtual disk. Existing work in the area of live VM migration has largely focused on improving migration performance. The limited work addressing interference stemming from migration has not considered storage contention. For these reasons this work must empirically demonstrate the issue of storage interference between storage migration and co-located workloads.

3.1.1 Environment and Workloads

We use **4** distinct I/O-bound workloads throughout our experimentation: Filebench's Fileserver and Videoserver [32], YCSB [33], and Fio [34]. These workloads, their descriptions, and our configurations are outline in the following subsections along with a description of our environment.

3.1.1.1 Base Environment

All physical machines are equipped with an Intel Xeon E5-2630, 64 GB of RAM, and are connected via both a 1 Gbps Ethernet and Infiniband networks. Migrating VMs have a 20GB virtual disk and each workload VM has a 40GB virtual disk. All VMs are configured to use raw storage with write-through caching, 2 VCPUs, and 2GB of RAM running atop QEMU(2.1.3)/KVM(CentOS 6, Kernel 2.6.32) [35][3]. Additional configuration details are provided in the test subsection.

3.1.1.2 Filebench Fileserver & Videoserver

Filebench [32] is a file system benchmarking application that uses a *wokload model language* to describe the storage I/O characteristics of an application. This allows researchers to describe complex storage applications, including multiple threads, file operations, and simulated events using only a plain text configuration file. The *filebench* project provides base configurations that imitate common real-world applications such as the *fileserver* and *videoserver* models used in this work.

The *filebench* ***fileserver*** simulates common file operations performed concurrently, including creation, deletion, and mode changes. The result is a benchmark that can closely simulate a shared file store such as an FTP or NFS server. The base *fileserver* configuration is determined empirically to over-utilize the test VMs, so the default number of threads is reduced from 50 to 5 in the model.

The *filebench* ***videoserver*** simulates the concurrent uploading and streaming of video files. Many streaming threads sequentially read various files, simulating a large user-base viewing content. In addition, a smaller number of uploading threads sequentially write new files and delete old ones, simulating privileged tasks that are accepting new submissions and managing old ones. The default *videoserver* model requires excessive storage space without providing significant load, so the base configuration is altered to increase the event rate to 200 per second, using 100 threads, across only 20 active videos and 16 passive videos.

3.1.1.3 Yahoo! Cloud Serving Benchmark (YCSB)

YCSB [33] provides a program suite that can be used to compare the relative performance of database management systems, with specific focus on key-value and cloud serving storage systems. *YCSB* is capable of interfacing with a wide variety

of storage software, including caching systems. With a storage system in place to test, *YCSB* loads a workload configuration that simulates clients utilizing the chosen system. In these experiments, *Cassandra* [36], a popular open-source key-value store, is loaded with 12 million entries for *YCSB*'s simulated clients to query. *WorkloadA* is used for execution, which consists of an equal amount of read and write operations that are issued continuously until the test is ended.

3.1.1.4 Flexible I/O (Fio)

Fio [34] is a highly configurable utility that can be used to simulate a variety of I/O patterns across a data file. It's typically used as a micro-benchmark to efficiently stress I/O systems since it only requires a filesystem to issue requests against. *Fio* doesn't attempt to imitate application I/O patterns on its own, though it can be used to replay pre-recorded traces, but this functionality is not used in these experiments.

We use *Fio* in several of our initial experiments due to it's simplicity and flexibility. Unless otherwise noted, *Fio* is configured to perform synchronous random read/write across a 4GB file at 1MB/s using 2 threads. This makes for a fairly low throughput workload, but one that demands consistency through the use of synchronization. These parameters were chosen to compliment the other workloads, which are generally higher throughput, but don't enforce expensive synchronization.

3.1.2 Investigating Possible Compromises

A hypervisor typically facilitates some level of configuration when migrating, and most modern operating systems allow users to tune process-level priorities. These initial experiments explore how these simple, yet naive techniques, impact a migration and it's tendency to interfere with a workload. More thorough experimental investigation follows in section 3.1.3.

3.1.2.1 Migration Speed Trade-off

We first demonstrate an intuitive trade-off between migration speed and guest storage I/O performance. A full machine migration must read the entirety of migrating virtual disk at least once. Furthermore, it's expected that the VMM's migration process, in this case QEMU/KVM's pre-copy method, will only read the virtual disk data at a rate that is aligned to the configured migration speed. In other words, a faster migration will result in more disk I/O from the hypervisor's migration procedure, while a slower migration will demand less throughput from the storage device. We use a workload VM running our *fio* configuration and an idle VM that is migrated at increasing speeds to demonstrate the trade-off between migration speed and guest VM performance. Note that this tests uses local hard-disk drives. The max-normalized results of these experiments are provided in Figure 1, with maximums of 263 IOPS and 321 Mbps.

From these results we can make several observations. Most important is the clear ability for a migration to degrade a storage-bound workload. Additionally, a faster migration will have a larger impact on a co-located workload, while a slower migration will have less of an impact. This is clear from the decrease in IOPS achieved as the configured migration speed is increased. These result also suggest that a possible solution may be to simply migrate at lower speeds and accept the reduced migration performance. But even with this compromise, a migration may fail to complete due to block dirty rates [37].

3.1.2.2 Adjusting I/O Priority

Storage I/O issued to the host machine must first be enqueued for dispatch by the kernel I/O scheduler. Our systems use the default Linux CFQ scheduler,

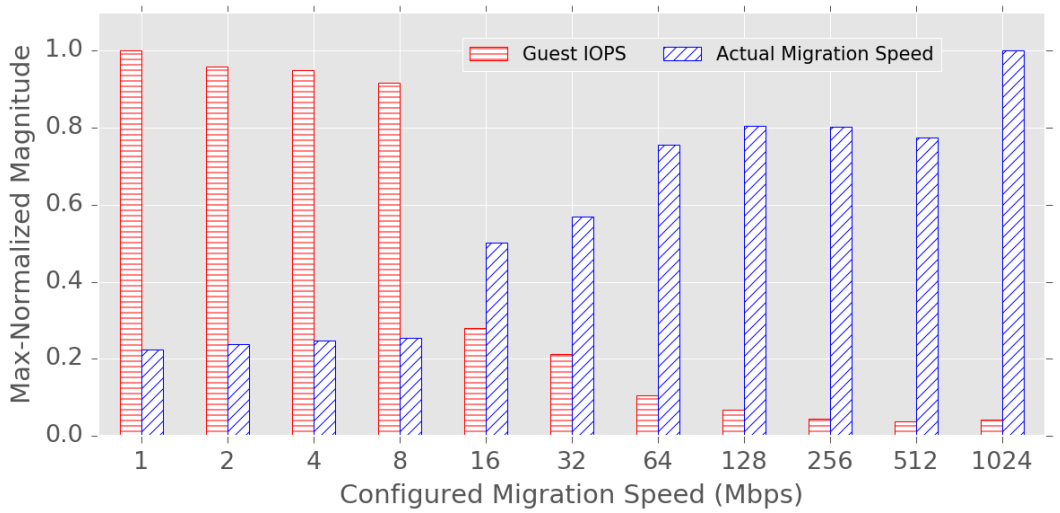

Fig. 1. Trade-off between Guest IOPS (*fio* random read/write) and increasing migration speed. Higher migration speed will decrease migration latency, but reduce guest performance.

which is capable of linking I/O priority to a specific process. A possible approach to abating interference caused by storage migration is to decrease the I/O priority for the migration process. The goal being to encourage the I/O scheduler to prioritize other workloads over the expensive migration process. We configure the migration for 128Mbps and adjust the migration processes' priority to low (4) *best effort* priority. With migration at the lowest I/O priority, *fio* can double it's IOPs when co-located with a migration - but this is still only 36% of *fio*'s nominal performance. Although not tested, reducing the migration's process priority is likely to impact the migration's ability to converge in much the same way throttling would. Therefore, adjusting the system's scheduler cannot provide the fine-grained control needed to approach interference.

3.1.3 Workloads Co-located with Migration

The previous experiments have demonstrated basic storage-level interference and shown that tuning the OS scheduler cannot address the issue. We now take a more systematic approach of investigating interference across a wider variety of workloads.

As discussed at the start of this chapter, cloud environments are capable of scaling to thousands of machines and often demand high storage throughput. While environments supporting these clouds are both complex and expensive, our experiments must only focus on recreating the pertinent attributes and use-case of a live storage migration. A full machine migration is a process involving a VM and its virtual disk, which subsequently involves the storage subsystem hosting the virtual disk. In cloud environments, this must naturally be a high performance shared storage volume that can accommodate the workloads of many machines. A migration of any kind requires both a source and destination physical machine. Moving groups of VMs may involve many more physical machines, but any single VM still has only one pair of physical hosts. These properties support the validity of our experiments described in the this section and their ability to fairly represent cloud environments.

Two physical machines are used to demonstrate interference, one as the migration source host and another as the destination host. In order to apply reasonable utilization to the physical storage system, identical workloads are run in 2 VMs on the source machine simultaneously. Meanwhile, a third VM with no active workloads migrates from the source to the destination. No workloads or other significant software are run at the destination - only a hypervisor ready to receive the incoming VM. This configuration produces notable interference only at the source machine's storage system. Therefore, all storage metrics presented in this work are collected from the source machine. A diagram of the configuration is provided in Figure 2.

In practice, a migrating VM may be running its own workloads when undergoing migration, and the destination host may already be under heavy utilization. However, our tests keep the migrating VM idle in order to isolate the impact of storage-level interference. This is needed because workloads executed within a migrating VM will degrade migration in two different ways - (1) writes from within the migrating machine will force repeated transmissions of dirtied blocks [37] and (2) I/O from the migrating machine's workload will contribute to the storage-level interference. While (2) is a shared property of any workload co-located with migration, (1) is unique to workloads running inside a migrating guest. This work is focused on characteristics of (2), so we control for (1). Therefore, by keeping the migrating machine idle, the experiments can directly show the impact of co-located workload interference on migration latency, without mixing the impact of dirty block re-transmissions. Of course, if these workloads under test were instead run inside the migrating VM, migration latency would likely see further degradation.

In an effort to create an analog of enterprise-level systems, all virtual disks at the source are stored on a 12 TB RAID-6 volume. The destination machine stores the incoming VM's virtual disk to a local HDD. Metrics and other collected data are stored to a single HDD on the source machine. Each experiment is repeated three times and the results are averaged.

All migrations, including both the QEMU/KVM's pre-copy methodology and our buffering methodology, target a 40MB/s live VM transfer. This is above the default configuration of 32MB/s in order to simulate a high performance migration - the preferred case in most scenarios.

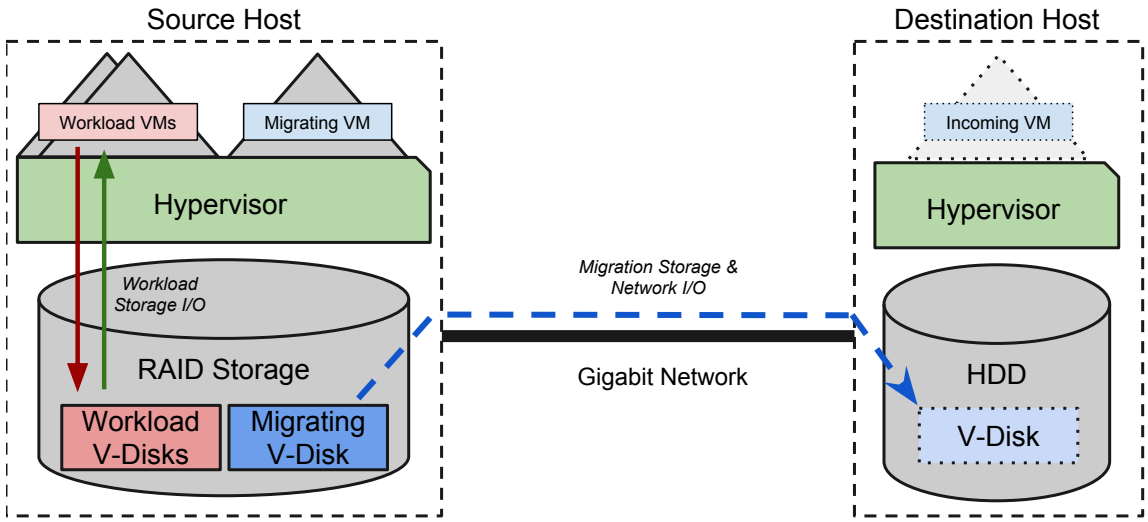

Fig. 2. Configuration of physical machines, the VMs they host, and the experimental process. Matching workload VMs issues I/O requests while a third idle VM migrates to the second physical host.

3.1.3.1 Nominal Workload & Migration

Initial experiments examine the storage I/O characteristics of each workload in isolation. Figure 3 illustrates sector-level patterns for each workload's I/O, collected from within the guest VM during the first 100 seconds of nominal execution. We find that the *filebench fileserver* has a consistent operating region, with over 98% of I/O occurring within a contiguous 15% of the virtual disk's sectors. In addition, *fileserver* completes just under 1.5 million I/O operations, which is second only to the *videoserver*'s 5 million completed operations. The *videoserver*'s sector activity in Figure 3 shows that the workload develops "hotpots" in small regions of the disk as threads compete to sequentially stream video files. This creates two layers of locality in which highly local I/O is randomly dispersed across the disk. In contrast, with it's configuration set to random R/W, *fio* sensibly exhibits one of the least sequential patterns, with operations scattered across it's working file. With our low

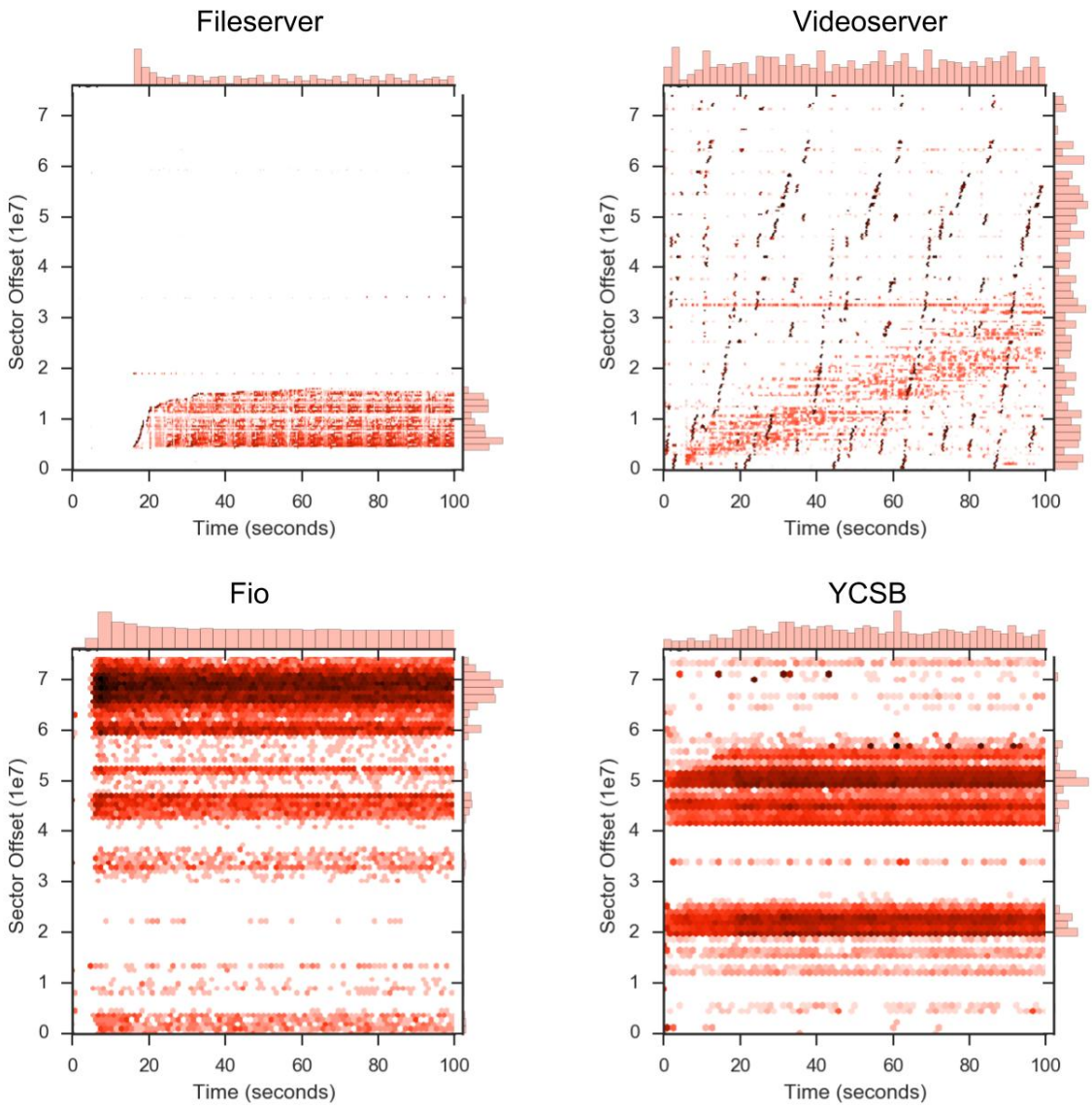

Fig. 3. Guest-level sector usage over time for each workload under test. Regions colored by the log normal (w.r.t the individual workload) of completed I/O operations. Distribution over time and across sectors are given along the top and bottom edges, respectively.

bandwidth configuration, *fio* completes just over 25 thousand I/O operations during execution. However, it'll be seen that even though *fio* executes fewer operations, the burden of forcing a synchronization coupled with large seek times makes *fio* particularly contentious. Finally, *YCSB* demonstrates a reasonable locality with 54% of its operations taking place in 10% of the disk, but with only 100K I/O operations overall. The characteristics of *YCSB* complement the previous three workloads by combining a moderate amount of I/O with a new pattern of locality not yet demonstrated by *fio* nor the *filebench* workloads.

Figure 4 includes a chart for host-level utilization, R/W disk bandwidth, and the number of I/O operations (IOPS) for a static VM migration. The migrating machine runs idle, with no co-located workloads. The same host-level metrics for the *Videoserver*, *Fileserver*, *Fio*, and *YCSB* workloads can also be seen in Figure 4. Each workload is run in isolation within a VM, with no co-located workloads or other VMs. Both the migration and the workloads demonstrate relatively distinct I/O patterns, both in average and variance in resource usage.

The migration itself has the most consistent storage I/O pattern - unsurprising given the fundamentally sequential nature of an idle machine's pre-copy live migration. Disk utilization remains low with only minor fluctuations near the start. Bandwidth stays near the expected 40MB/s with IOPS at a consistent rate. This data hints at an underlying simplicity in the process of a live migration's storage access, one that we intend to leverage in our design. Throughout the remainder of this work, pre-copy migration as demonstrated here is also referred to as *static* migration.

The I/O workloads under test demonstrate a wide-array of characteristics, from chaotic, to consistent and predictable. Both the *Fileserver* and *Fio* workloads exhibit relatively low variance, but with noticeable disruptions to their patterns in their initial stages. In addition, *fio* launches with a more variable level of IOPS and storage

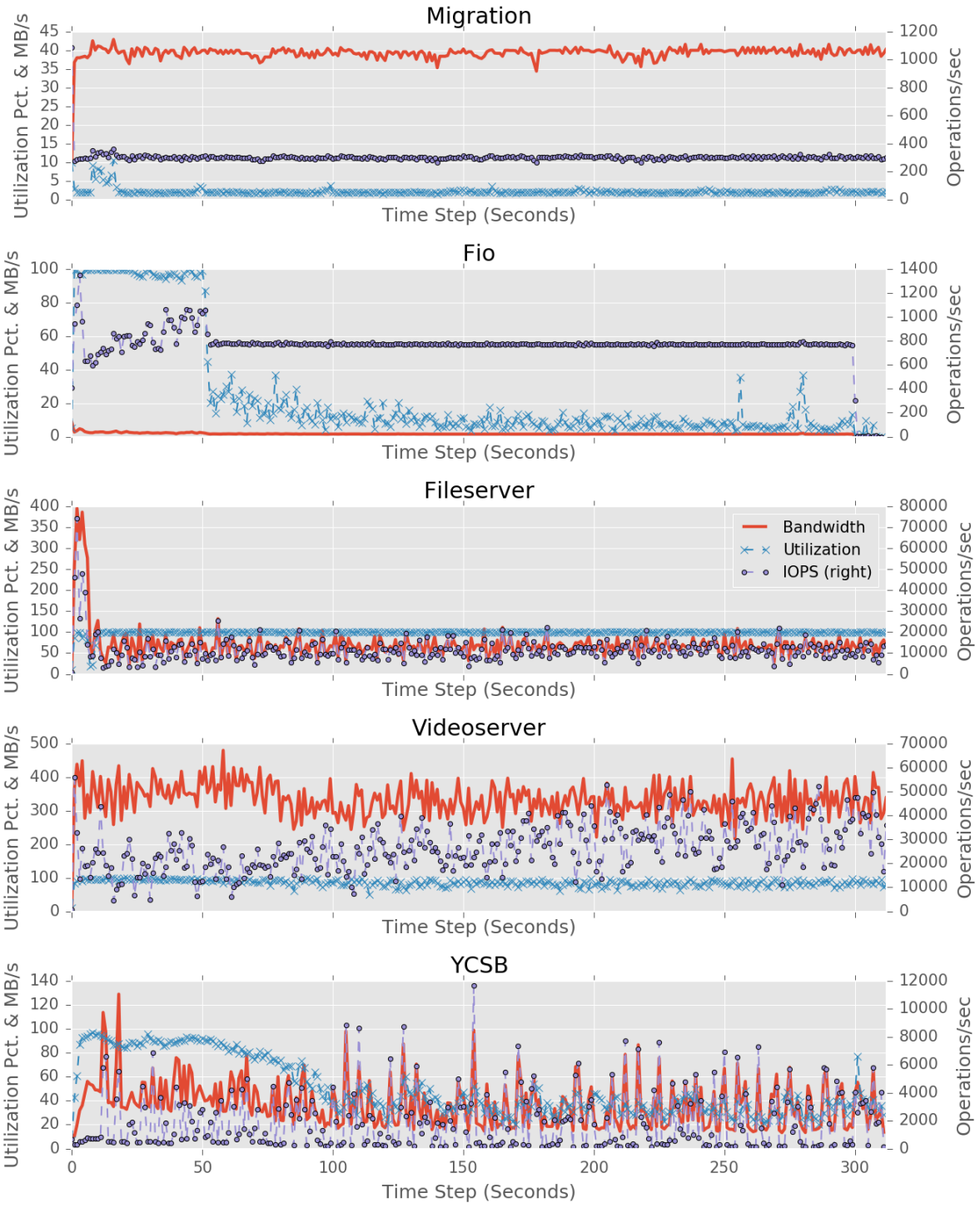

Fig. 4. Characteristics of I/O over time for the nominal execution of each workload, including pre-copy storage migration

utilization, though this abates within a minute of starting. The *Videoserver* has the most sporadic I/O footprint, with 2.2x the standard deviation of bandwidth and 5.7x the standard deviation of IOPs when compared to the *Fileserver*. Finally, *YCSB* undergoes high intensity I/O near the start, but gradually settles to a lower utilization and throughput. This suggests aggressive application-level optimization and caching by the back-end, which will lead to less interference.

These workloads give a diverse cross-section of I/O workloads possible in today's large systems, making them good candidates for use in exploring migration interference and evaluating our approach.

3.1.3.2 Workload-Migration Interference

Next, we examine static migration performance when executed alongside these workloads. These experiments introduce two important new variables - the time offset into the migration at which a workload is launched and the duration of the workload's execution. Workloads launched earlier during the migration process have more time to cause interference. Similarly, longer workloads will also impact performance for a longer period, resulting in larger degradation. We run tests with each workload launching at the start of migration (0 seconds) and at 100, 200, and 300 seconds into the migration process. Note that the migration process takes approximately 540 seconds on average without interference. Each test is performed with workloads lasting 60 seconds, 240 seconds, or 1920 seconds (1, 4, and 32 minutes, respectively). These lengths are chosen to demonstrate the contention of shorter bursts in I/O activity, medium-length application updates, as well as more persistent long running storage usage. In these experiments, the worst-case scenario are those where a workload lasting 1920 seconds is launched at the start of migration.

Under our experimental conditions, the impact of storage contention is severe for

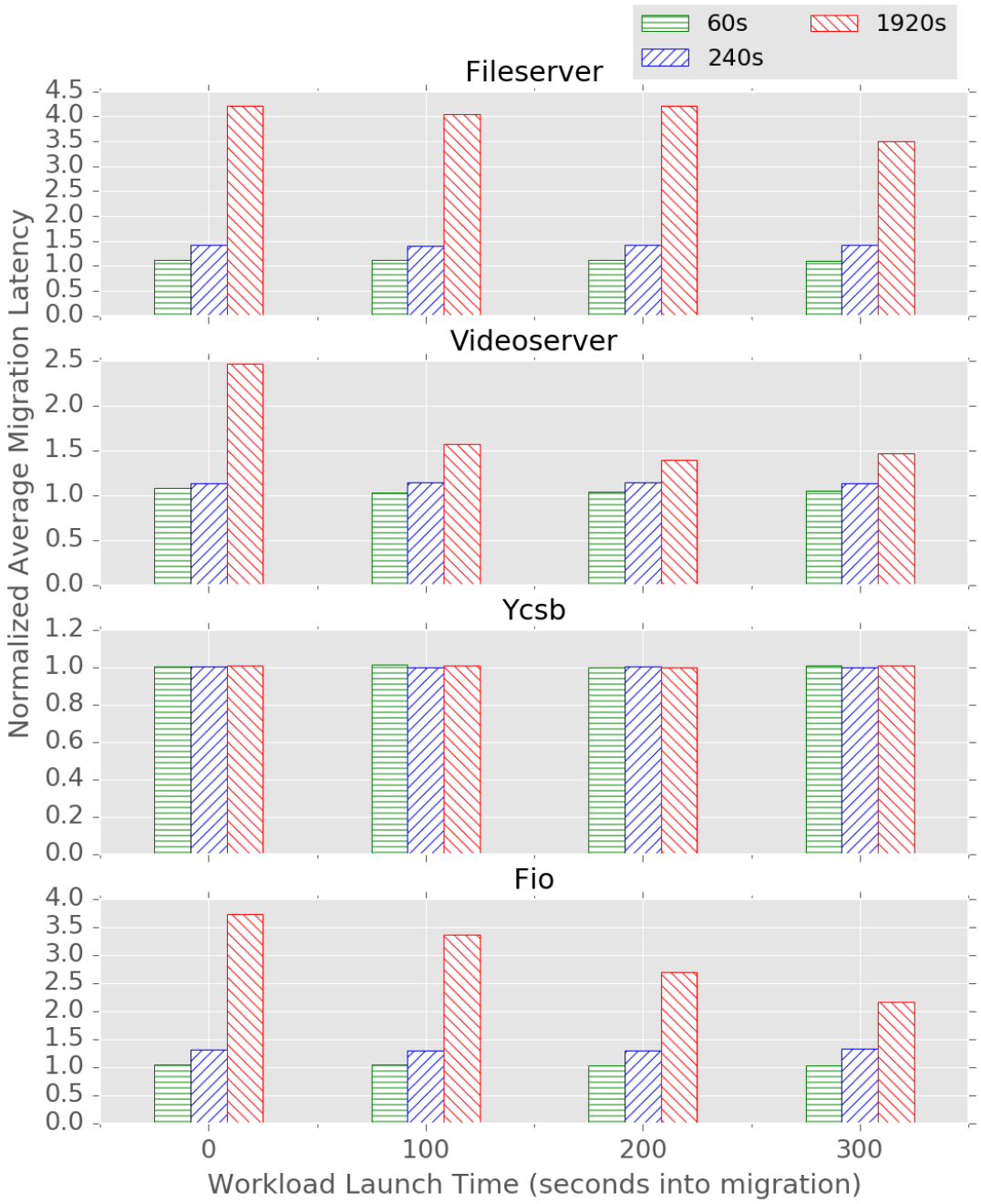

Fig. 5. Comparison of static migration latency for workloads under test. Values are computed as the normalized to the nominal migration latency (lower is better).

three of the four workloads tested. The average migration latency, normalized to the expected latency, for each workload's test set is shown in Figure 5. From these results, we can see the relationship between the workloads' start time and the final impact on performance. Intuitively, the longer these workloads are allowed to run co-located with migration, the longer migration will require to complete. The largest contention is imposed by the *fileserver*, which quadruples the length of the migration in the extreme cases. This specific workload forces the migration to take over 36 minutes, when it normally takes only 9 minutes. The interference caused by both the *fileserver* and *fio* is consistent enough to double migration latency even when launched well into the migration. Even though it's workload is moderate, *YCSB* manages to inflict very little interference on the migration performance. We believe this to be in-part due to the application-level I/O management that's merging and caching I/O requests, which is supported by *YCSB*'s high number of blocks per I/O on average.

Of course, the migration is not the only piece impacted - except for *fio*, all workloads undergo degradation when accounted for via guest-level I/O metrics. The results shown in Table 1 quantify these changes as the ratio of the mean metric during static migration over the mean metric during nominal execution. Both the *fileserver* and *videoserver* see the largest reduction in their ability to maintain I/O. Interestingly, both *fileserver* and *fio* appear mostly unaffected by the migration's sequential read.

From these results, we see that interference can occur at varying degrees across a range of workload characteristics. The experiments show that even a lower bandwidth workload such as *fio* can greatly reduce migration performance, while higher bandwidth applications such as YCSB can have almost no affect, yet itself be degraded by the migration. This implies a complex relationship between the access patterns and how each workload chooses to issue its own I/O.

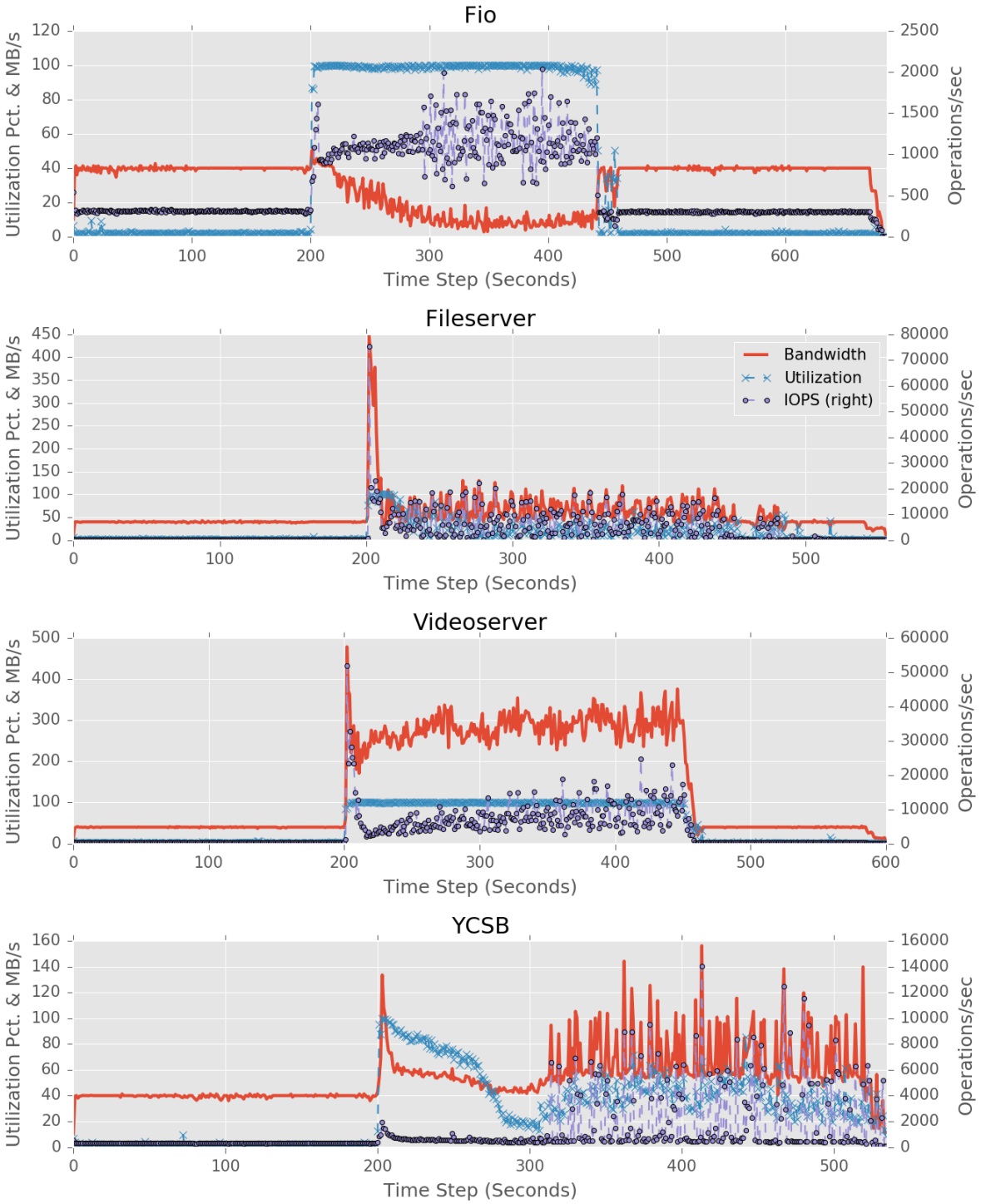

Fig. 6. Characteristics of I/O over time for workoads co-located with pre-copy migration

Table 1. Workload performance degradation when co-located with static migration

	fileserver	*fio*	*videoserver*	*ycsb*
Throughput	0.953	0.973	0.577	0.637
IOPS	0.946	0.998	0.679	0.725

3.2 Design

The experiments conducted in section 3.1 clearly demonstrate the potential for varying degrees of interference between a VM storage migration and co-located VM workloads. We address this scenario in order to find a non-invasive, yet effective strategy for avoiding this type of performance degradation.

Our approach to mitigating storage migration interference makes use of several key observations deduced from our experiments and prior work. First, we recognize that by decreasing migration speed, which reduces the storage I/O rate of the migration process, we can nearly eliminate interference, but at the cost of higher migration latency. Second, pre-copy migration is a predictable process in which the backing virtual disk will be sequentially read at least once. Any further dirtying of data from the migrating VM can be cached in the host's page cache or mirrored to the destination. Finally, we recognize that interference is a case of extremes, and that even simple algorithms can recognize host-level patterns that are indicative of ongoing interference.

We combine these three notions to design *Migration Buffering*, a method of offloading a migrating machine's virtual disk as it is migrated. When no interference is detected, un-migrated virtual disk data can be prefetched into a secondary, low-interference storage buffer. If interference conditions arise, only the prefetching I/O

must subside to reduce contention, but the migration can continue at full speed until the buffer is exhausted or the migration has completed. This technique is conceptually similar to the video and audio content buffering many experience daily.

3.2.1 Adaptive Transfer

A cohort of processes can eliminate or greatly reduce mutual resource contention if one or more applications can reduce its utilization of the shared resource. We've shown this to be true for migration in section 3.1. We believe a comparatively long-running I/O process such as VM migration must be capable of adapting to the demands of other applications, especially applications that may have no view or knowledge of the migration. Such an online adaption allows the migration to prioritize more sensitive applications in exchange for reduced speeds, and thus prolonged migration times. However, this facility must be carefully balanced to ensure that the migration itself does not fail to meet any timing requirements.

In this design, we employ an adaptive transfer as means to adjust the migration's read rate in concert with the availability of data in the buffer. This may be implemented through lower-level I/O throttling of the migration process or by directly interfacing with the VMM, if it allows.

3.2.2 Buffering

Storage migration requires a full read of the virtual disk, no matter where it resides. Furthermore, this read should preferably have high throughput, both to satisfy any latency restrictions, but also to ensure migration convergence. These types of large and intensive I/O patterns are not unique to modern environments, but the predictable pattern of a full read across a large dataset makes storage migration a special case. These properties of storage migration I/O inspires the prefetching aspect

of our design - when a machine is being migrated, we can accurately anticipate what data will be required from the storage system. Bringing non-migrated data off of the storage system, when the likelihood of interference is low, and storing the data in low-interference storage creates a *buffer* for use when interference probability is high. Therefore, when the system determines that interference is probable, the migration process can continue at desired speeds by sourcing non-migrated data from the buffer. During these periods, only the rate of prefetching must be reduced in order to prevent storage I/O interference. The migration performance, the rate at which data is being transferred to the destination, must only be reduced when the buffer is depleted and interference is still occurring.

Importantly, the migration buffer can be implemented in a variety of ways, including secondary local disks, RAM, or even idle disks from other machines across the data center [38]. The only requirement is that continued usage of the buffer will not impact critical storage applications. Ideally, the buffer can also act as a storage cache for the VM undergoing migration. This way I/O requests from the migrating VM can serve two purposes - to deliver data into and out of the VM, but also to pull data from the primary storage and into the buffer.

3.2.3 Interference Classification

An important input to adaptive transfer and prefetching is the accurate identification of I/O interference as it occurs. Intuitively, such a system can either monitor application performance directly, through some known metrics such as client request latency, or indirectly, in a *black box* fashion. It may even be feasible to allow an administrator or user to directly provide feedback on the levels of interference from their viewpoint. In the case of cloud computing, where providers and customers are separated by a strong semantic gap, a black box method is preferable in order to

maintain customer privacy and abstraction. An important property for the flexibility of the design is the need for interference classifiers output to be a probability of interference, rather than a binary classification. Outputting a ratio that indicates the likelihood of interference enables the other components to respond fluidly. A binary output would require integration of the output over time to simulate degrees of interference, which we'll show is important to allow a practitioner to configure their desired level of performance compromise.

CHAPTER 4

IMPLEMENTATION AND EVALUATION

In order to demonstrate the effectiveness of our Migration Buffering, as well as to explore its parameters, we implement a prototype of the design describe in section 3.2. After detailing the implementation, we evaluate our approach and demonstrate its ability to abate interference in the majority of scenarios presented earlier.

4.1 Components and Features

Our prototype is a host-level, black-box version of the proposed design as a set of three primary components: the *Migration Controller*, *Storage Monitor*, and *Storage Prefetcher*. These components are loosely coupled by design, allowing each to be run on separate hosts if need be, or as a single application process. In addition to these components, the Migration Controller utilizes an *Interference Classifier* to identify interference during the process of migration. A summary of these components can be seen in Figure 7.

4.1.1 Migration Controller

The Migration Controller is responsible for collecting the metrics retrieved by the Storage Monitor and forwarding them to the Interference Classifier. This classification algorithm reports the probability of interference, which the Migration Controller then uses to determine a new migration speed. The Migration Controller communicates with the hypervisor directly, in our case QEMU, to both configure migration speed and retrieve the amount of data transferred. The flow chart in Figure 8 outlines the

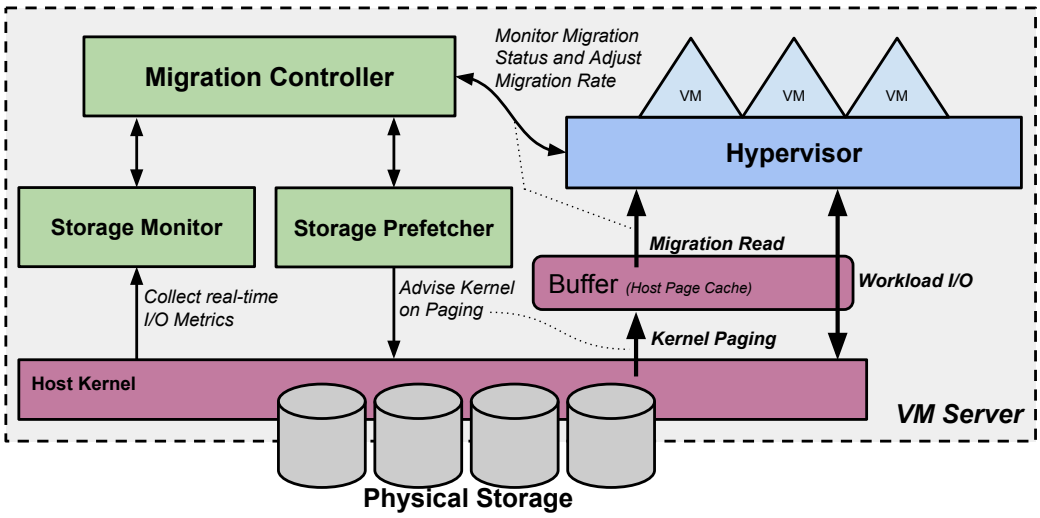

Fig. 7. Overview of our method of migration using adaptive transfer and prefetching

Migration Controller's basic steps, computed every second during a migration.

The Migration Controller uses a Proportional Integral Derivative (PID) feedback loop to manage the migration and prefetching rates. This *interference-PID* is given an interference probability as its set-point, and after each step returns a migration rate delta, or suggested change in migration rate. This delta therefore represents the PID's suggested change in migration speed in order to approach the configured max allowable interference probability, which we refer to as the *interference threshold*. A higher interference threshold weights the controller's priority in favor of the migration process. A lower threshold prioritizes co-located workloads by forcing the migration to reduce speed in the face of lower interference probability. Note that the migration rate is capped at the configured rate - if the delta values suggest higher speeds, the migration will not exceed this target.

The interference-PID's delta is also provided to the prefetcher. That is, the migration rate change needed to achieve the desired interference probability is passed to the prefetcher for use in scaling the amount of data to be prefetched. Therefore, if

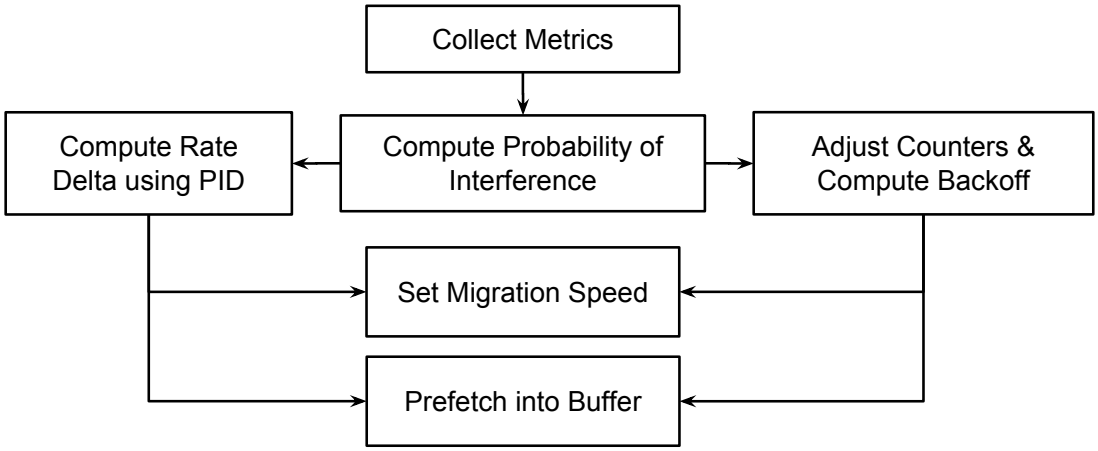

Fig. 8. Migration Controller's steps and their dependencies.

the interference-PID delta is positive, but small, the prefetcher fetches comparatively fewer pages than if the delta was larger. If the delta is negative or zero, no pages are prefetched. The rate delta provided to the prefetcher is first converted from bytes to number of pages by dividing by page size (4KB in our system). Then, the number of pages is scaled by a configurable constant referred to as the *prefetching factor*. This allows us to decouple the PID gains from the aggressiveness of the prefetcher. For example, by keeping the interference-PID gains small, the system does not "over-react" to variations in interference probability by making large adjustments to the rates over time. However, these small gains would result in smaller prefetches during each time-step. To counter this, and maintain an aggressive prefetching process, we can allow an administrator to increase this prefetching factor to a larger value, giving the interference-PID a wide-range of page amounts to prefetch per step.

Additionally, as long as the amount of data in the buffer is larger than the migration rate per step time, the controller maintains the migration speed at the target rate. Once the buffer's size shrinks to below the amount to transfer in a time-

step, the migration rate is dictated by the interference-PID's delta much like the prefetcher. This logic is key to allowing the migration to maintain it's speed when it's able to source from the buffer and not cause storage interference.

Assuming our method of eliminating interference is effective, then once we reduce interference, the interference classifier will begin outputting lower probabilities. In turn, our system will respond by increasing speeds, which will subsequently cause interference to occur again. These oscillations could potentially be combated by fine-tuning the interference-PID but we empirically find that a more brute force count-and-lock approach to be more effective. The controller keeps a counter of how long the probability of interference has been above the interference threshold. Once this count grows, the controller begins to adjust a *backoff* variable. This backoff value is adjusted proportionally to the error in interference probability. Each iteration, the backoff is used to reduce both prefetching and positive movement in migration speed. As prefetching is reduced through this backoff, the probability of interference begins to decrease since I/O pressure is being reduced. Once the probability is no longer above this upper interference threshold, the lock counter stops increasing and prefetching remains in this degraded state. A second low-end threshold dictates at what interference probability the controller is willing to release the prefetcher from its degraded backoff state. Once the probability is below this low interference threshold for a short period, the backoff is reset, and the rate of migration will begin to grow.

4.1.2 Storage Monitor

The interference classifier housed within the migration controller requires storage metrics, updated every time-step, in order to determine the probability of interference at that time. The storage monitor process collects metrics via the counters exposed in */proc/diskstats* on the physical machine hosting the storage subsystem. In order to

reduce the volatility of the metrics reported, the storage monitor computes a moving average on the current and past three time-steps, with more recent values weighted more heavily. This filtering has the benefit of transparently adding a temporal aspect to the interference classifier, without the need for time to be explicitly considered.

4.1.3 Storage Prefetcher

During periods of low interference probability, the storage prefetcher marks the next un-migrated pages of the migrating virtual disk as *WILLNEED* using the Linux kernel's *fadvise* interface. Cached pages are determined through the kernel's *fincore* system call. Due to the sequential nature of a migration, the next n pages are prefetched from offset where data has neither been migrated nor cached. The number of pages prefetched is determined by the probability of interference and the resulting interference-PID's output. A positive delta from the interference-PID is converted from bytes to pages and scaled by a configurable factor.

Using the host's page cache as the buffer in this way has both benefits and drawbacks. The implementation is fairly simple and requires no modifications to the hypervisor or operating system, improving on the practicality of the system. Furthermore, if the migrating VM were to modify storage blocks that have already been transmitted, the host's page cache will naturally capture these modifications. The modified data has therefore been buffered transparently, and our prefetching process does not need to explicitly fetch this data. The same can be said of data read by the migrating VM. Of course, if memory usage is high, or the block dirty rate is high, using the page cache may not be wise. By paging large chunks of the virtual disk, we potentially evict critical data and cause memory interference. Furthermore, there is no guarantee that the OS will honor the suggestions conveyed through the calls to *fadvise* at all. Still, we find this prefetching technique suitable for demonstrating our

design - real-world implementations may opt for different mechanisms.

4.1.4 Interference Classifier

The proposed design requires an *interference classifier* - an algorithm capable of computing the likelihood, or probability, of interference occurring. As discussed earlier, it is important for practical applications that this algorithm not need VM introspection or any application-specific knowledge. Many clouds and VM providers enforce strong isolation, breaking this isolation considerably restricts the use-cases of the proposed design. For these reasons, the interference classifier in this work uses only host-level metrics from user-space to provide the probability of interference.

The problem of identifying interference can be simplified to a binary classification problem: at any point, the system can either be undergoing interference or not. Many algorithms for binary classification exist, but we seek those that are easily interpreted, fast to compute, and widely known. For these reasons, we evaluate both *regularized logistic regression* and *decision trees* as offered by the SciKit-Learn Python library [39] for use in classifying interference.

We produce samples by labeling the data presented in section 3.1.3.1 and 3.1.3.2. For experiments where obvious interference degradation occurs, we label all time ranges where the workload is executing as positive for interference. Therefore, we mark all pairings of workloads and static migration, except for *YCSB*, as interference. All other collected samples are labeled negative. This includes samples of an idle machine, nominal migration, and nominal workload execution. Additionally, we record metrics for nominal migration with fixed prefetching at each step in an effort to prevent misclassification of the paging process's characteristics.

The points in Figure 9 demonstrate the natural separation of the workloads within the dimensions of some of our notable host-level metrics. These contain only

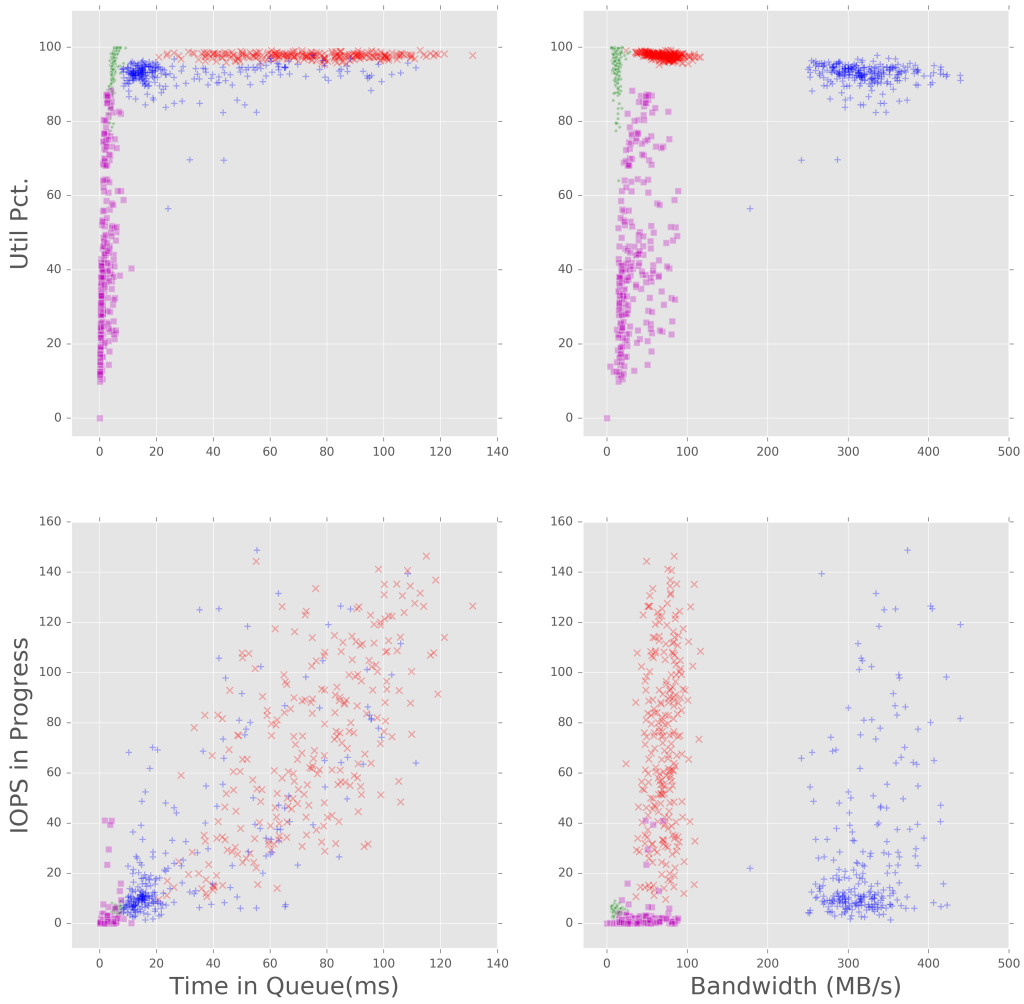

Fig. 9. (Best viewed in color) Sample of resulting host-level metrics when workloads are co-located with live migration (i.e. interference test). *Fileserver* marked as red 'X', *Videoserver* as blue '+', *fio* as green points, and *YCSB* as magenta squares.

points where both the workload and migration are running, including negatives provided by *YCSB*. It's clear that some metrics, such as percent utilization are strong indicators for interference.

With labeled samples generated from our tests, the predictive power of the collected attributes is evaluated by constructing and testing models. Numerous versions of each model with varying hyperparameters are constructed and tested. We use 5-fold cross validation on 75% of the samples in order to evaluate the classification power of different combinations of our collected metrics. Included in these trials are various transformations of each metric, intended to enhance separability for the model. Models selected by cross-validation were then judged on their ability to maximize both accuracy and F-score on the hold-out 25% test set. Under these circumstances, both logistic regression and decision perform well with their default hyperparameters. However, we favor logistic regression for its more continuous output, which applies well given our use of the output probability and not simply discrete prediction. Our final logistic regression model is made up of the following coefficients and features: $-2.27 \times 10^{-5}(IOPS\ Comp.)^2$, $-7.80 \times 10^{-1} \log(IOPS\ Comp.)$, $2.10 \times 10^{-2}(IO\ Ticks)^2$, $0.55 \cdot \log(Throughput)$, $6.87 \times 10^{-4}(Util.\ Pct.)^2$, and an intercept of -0.296. These learned coefficients provide some insight into the characteristics of interference. Our model defines interference as periods where the host is spending a large portion of time performing I/O, but the number of I/O operations remains small. Thus, for our data, storage interference is a special case of extremes - when more is demanded than can be given across a suspiciously low number of requests.

Combining our three primary components at the host level, as described above and illustrated in Figure 7, yields a practical version of our proposed design. With this prototype, we can compare our method directly against static migration.

4.2 Results

In order to investigate our prototype implementation, we evaluate our method under the same conditions as our initial interference experimentation in section 3.1. In addition to these experiments, we also adjust the interference threshold of our implementation to demonstrate the prototype's ability to vary the degree of compromise in migration performance.

For all experiments the interference-PID is empirically configured with the gains $p_k = 2 \times 10^6$, $i_k = 5 \times 10^4$, $d_k = 6 \times 10^4$ bytes and the prefetching factor is set to 50. Unless otherwise noted, the high interference threshold is set to 0.5 and the low interference threshold is set to 0.15. The lock count threshold is set at two with an unlock count threshold of three. Functionally, this means that migration will begin adapting when the probability of interference has remained above 0.5 for two time-steps, and will only fully rebound if the probability remains under 0.15 for 3 steps. The results of our evaluation are shown in Figure 10 for each combination of workload, workload length, and workload launch time.

Our methodology is designed to page data yet-to-be-migrated into host memory when the probability of interference is low. This prefetched data can then be used to avoid storage interference later in the process. For this design, the worst case scenario occurs when a high-interference I/O workload is already executing once the migration is launched. In this situation, our implementation may have to reduce speed until the I/O pressure resides, since no prefetched data exists in the buffer. Conversely, the best case scenario for our methodology is when the entire virtual disk can be prefetched before any interfering workloads launch. More precisely, if enough data is prefetched to last a period of interference contention, our method will help prevent degradation to both the I/O workloads and the migration.

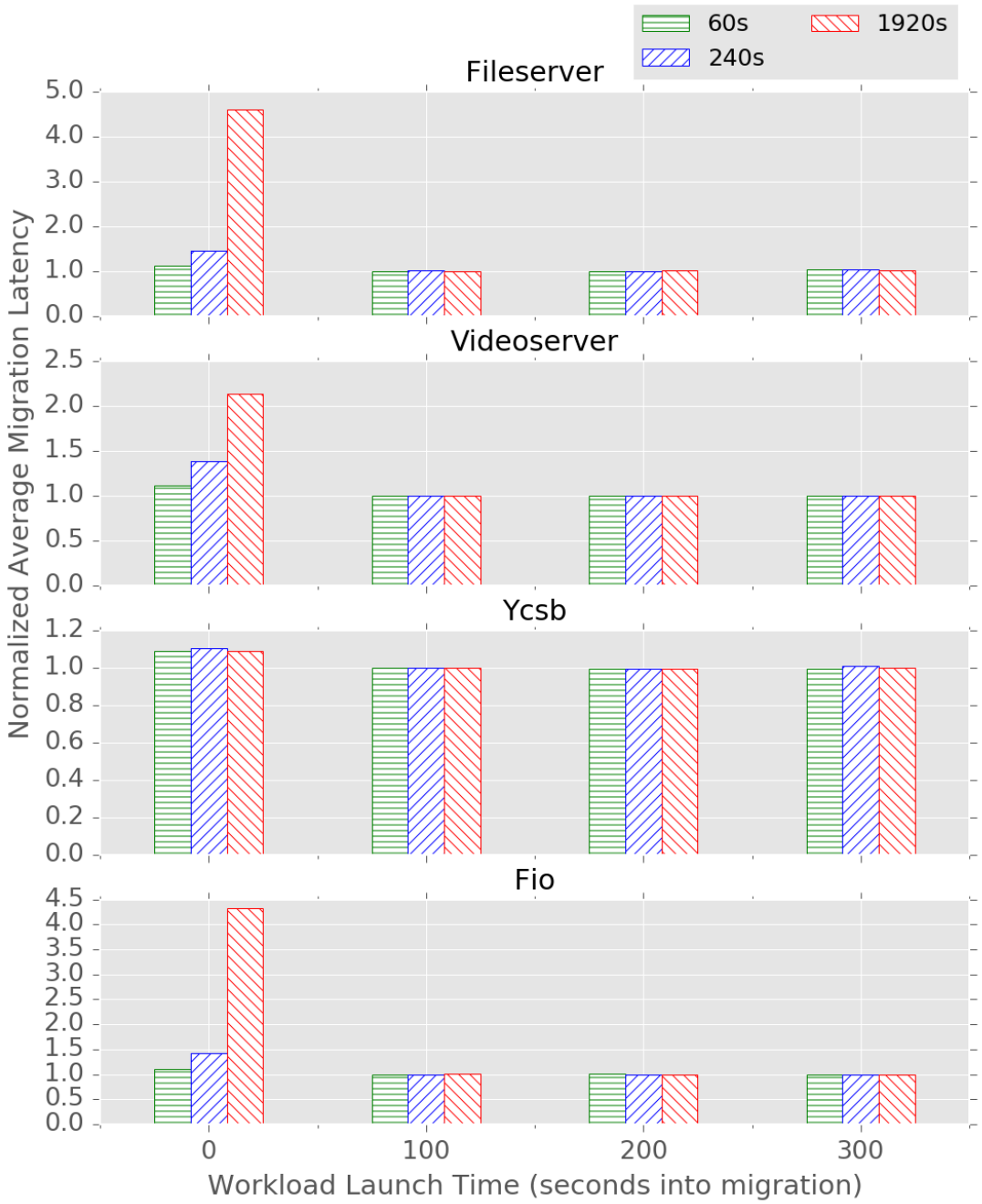

Fig. 10. Comparison of migration latency when using *Migration Buffering*. Values are computed as the normalized to the nominal migration latency (lower is better).

Table 2. Workload degradation when co-located with adaptive migration

	Fileserver	*Fio*	*Videoserver*	*YCSB*
Throughput	1.01	1.01	0.923	0.855
IOPS	1.02	1.01	0.965	0.884

The results in Figure 10 illustrate our implementation's ability to prevent degradation in migration latency in the majority of scenarios. As expected, our method only sees degraded performance when a contentious workload is executing at the start of migration. When contentious workloads are launched later in the migration, data prefetched into the buffer allows the migration to maintain it's transfer rate.

A detailed example of a worst case scenario for our design is illustrated in Figure 11. In this experiment, *fio* is launched alongside our migration system for 240 seconds, causing interference before our method is able to prefetch any amount of the virtual disk. Still, our system recognizes the interference and reduces both prefetching and migration transfer speed in order to avoid impacting the workload. Thus, even in the worse case, when the migration latency is lengthened, the workload still performs near nominal levels. In Figure 12 fio is launched 100 seconds into the migration, and our configuration is able to prefetch almost all of the virtual disk before *fio* launches. Thus, our implementation preserves the low migration latency as well as the co-located workload's performance. Intuitively, under this configuration with workloads on or after 100 seconds, the migration and workload perform near nominal levels. The results in Figure 12 also illustrate how our configuration pushes the prefetcher to aggressively increase its rate and move the entire virtual disk into memory. Of course, loading this much data into memory may cause unwanted memory pressure

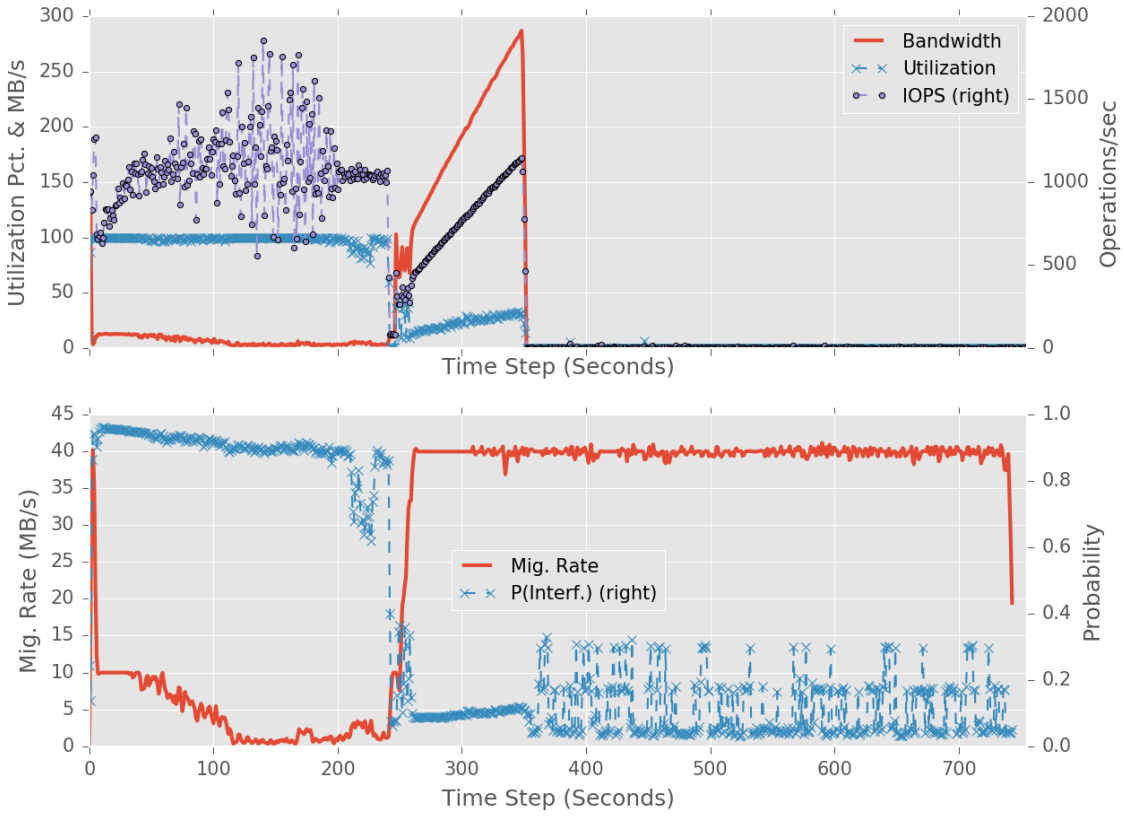

Fig. 11. Live migration using Migration Buffering against 240 second *fio* launched at the start of the migration. The migration resumes at full speed, with prefetching at approximately 250 seconds.

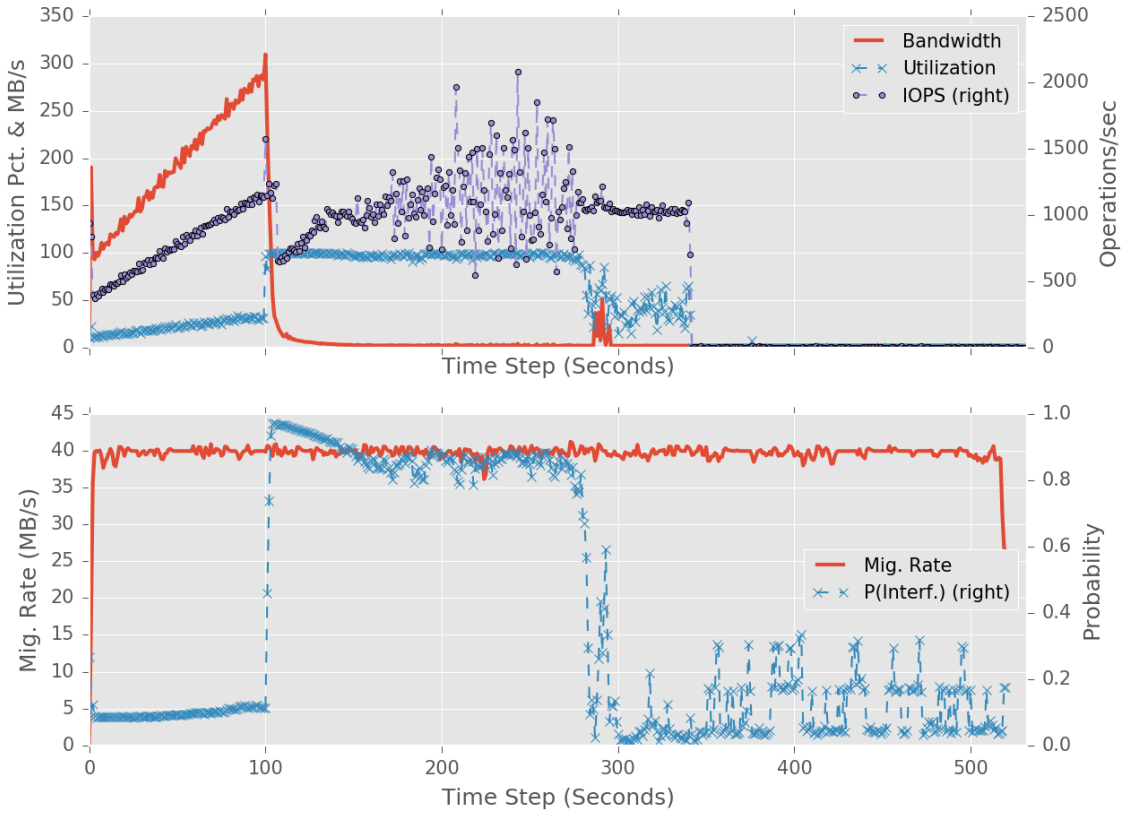

Fig. 12. Live migration using Migration Buffering against 240 second *fio* launched at 100 seconds into an ongoing migration. The migration is able to maintain speed during the contentious workload, allowing the migration to achieve a nominal latency.

and disk utilization, but at the risk of later vulnerability to interference should an I/O burst arrive. A trivial preventative measure for unwanted memory pressure is to simply cap the amount of prefetched data.

In both Figures 11 and 12 some characteristics of our design are clearly exemplified. The interference-PID attempts to move the interference probability to the configured interference threshold (0.5), and it does this by increasing the number of pages prefetched each step. This can be seen between approximately 250 and 350 seconds in Figure 11 and 0 and 100 seconds in Figure 12. The interference threshold combined with the prefetching factor is what we use to control this growth, though the PID configuration also has an impact.

Varying the interference threshold allows one to vary the desired level of migration performance compromise. This is illustrated in Figure 13 - as expected, the higher threshold value of 0.75 consistently results in faster migrations. Although not explored, this higher threshold will inevitably lead to more interference.

Presented in Table 2 are the average IOPS and throughput of each workload when run against our buffering methodology, normalized to the nominal results for each workload. For simplicity, we only show results for the worst case - these results are computed from experiments where each workload is run for 1920 seconds from the start of migration. Shorter versions of the workload distort these values due to the asymmetry of their metrics, while evaluating at later launch points would clearly favor our method. This simple evaluation of the worst-case demonstrates the effectiveness of simply reducing the migration speed - maintaining migration speed while sourcing from the buffer is similarly effective.

Our evaluation has demonstrated our approach's ability to mitigate storage interference - in the majority of scenarios we maintain migration speed with significantly reduced impact to workload performance.

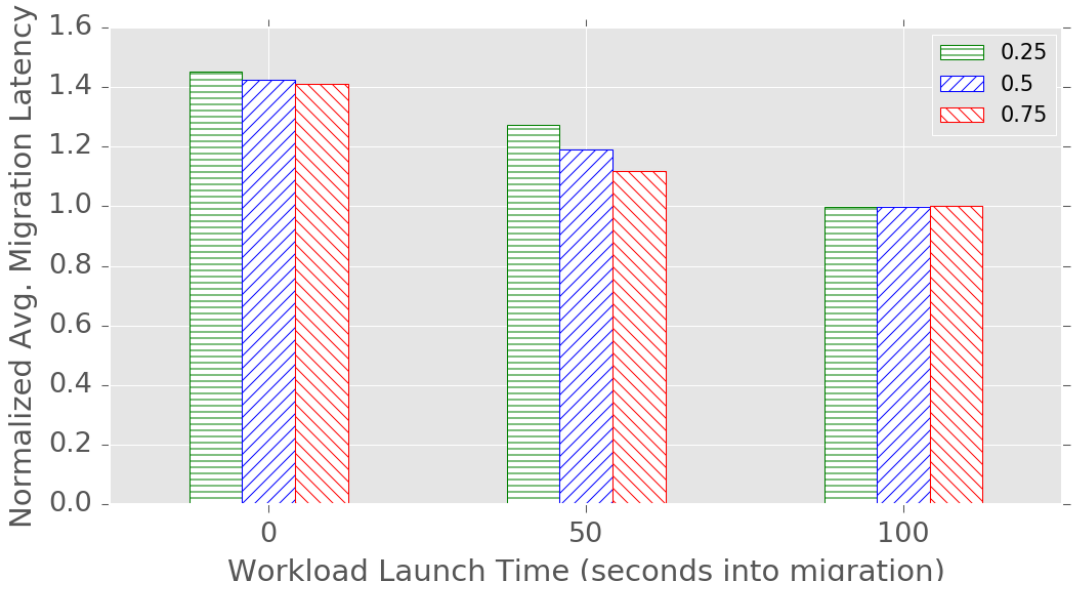

Fig. 13. Effect of interference threshold on migration latency for 240 second *fio*. Higher thresholds correspond to a more aggressive migration, allowing an administrator to tune the process to their needs.

CHAPTER 5

RELATED WORK

5.1 Virtual Machine Live Migration

Work in the area of live migration has largely focused on methods of improving migration performance, for both memory [8] [9] [10] [11] and full-machine migrations [12] [13] [14]. However, as migration has become more prevalent, researchers have begun to recognize and address issue of interference caused by VM migration [15]. Methods tend to focus on reducing the overall data transfer [11][16] or optimally placing VMs [17][18] to avoid inter-machine interference on network, CPU, and memory resources. iAware [17] treats a VM live migration as a CPU and network I/O demanding task. Given a group of VMs to be migrated along with potential destination hosts, iAware can make an interference-aware migration decision based on the measurements to jointly minimize VM migration and co-location interference. However, iAware focuses on interference as it pertains to memory-only live migration, which differs from the challenges unique to VM storage migration. Along with taxing CPU and network usage, storage migration also introduces a burdensome disk read. As we've shown, this additional I/O competes for the backing storage resource, negatively impacting co-located VMs through interference. Work in DeepDive [18] uses host-level metrics to identify when running VMs are interfering. Similar to the work in iAware[17], DeepDive use live migration to move culprit VMs.

In [8] the authors implement a new termination criteria for KVM's pre-copy migration. In pursuit of this goal, the authors explore the impact a migration has on application performance in order to motivate the need to avoid pursuing counter-

productive downtime for stop-and-copy. Their algorithm focuses on the optimal moment that the migration should switch between iteratively copying the VMs working memory to the stop-and-copy phase.

Migrating a full virtual machine, including it's virtual disk, requires the transfer of a considerable amount of data. Work in Shrinker [13], VMFlock [14], and CloudNet [40] focus on reducing the amount of data to transfer to the destination. Still, these works do not address storage interference nor do they detect the affinity between related VMs in order to avoid application degradation. The work in LIME [41] instead migrates the entire network of related VMs, but this method cannot scale when VM clusters contain thousands of machines. Furthermore, the limited bandwidth of a WAN connection may make this technique impractical. The authors of Pacer [42] perform a synchronized migration of VMs, but again don't propose a grouping mechanism to handle large clusters.

5.2 Resource Contention

VM migration in general can be considered an abberation, which requires intense utilization of almost all system components. Performance interference due to resource contention, including CPU, cache, memory, and IO, has attracted significant research.

Thread slow-down caused by L2 cache contention has been reported in [43]. Classification-based thread scheduling has been proposed to address CPU time and last level cache (LLC) contention in multicore processors [44]. In ordered to increase the utilization of warehouse scale computers, Bubble-Up [45] proposes sensible co-location of applications based on the prediction of performance degradation that results from LLC and main memory bandwidth contention. VM storage performance degradation caused by I/O device contention has also been discussed in [46] [47].

Although many methods have been proposed to address storage interference,

they're generally not well-suited as a solution in the domain of storage migration. Storage I/O bursts can cause a significant increase in request latency. Caching layers are often used to absorb sudden bursts in read requests, helping to further reduce disk contention. Everest [38] proposes write off-loading to dampen peak loads. However, write off-loading does not address the long read burst caused by a storage migration. Unfair storage resource allocation may violate the application service level objective (SLO). Soundararajan et. al. [48] propose quanta-based proportional resource allocation via coordinated learning and throttling-based I/O scheduling to enforce application SLO in Associateshared server farms. Stay-Away [49] also proposes proactively throttling the execution of batch applications to protect the performance of latency sensitive applications. In storage migration, I/O throttling can be employed to reduce the migration speed so as to mitigate the interference. However, this will considerably prolong the migration time, which is a key metric when evaluating VM migration performance. Contention for the I/O device itself dramatically degrades VM storage performance. To avoid I/O interference between servers, TRACON [46] proposes task-VM mapping using interference prediction to minimize the runtime and maximize the I/O throughput of data-intensive applications in a holistic way. TRACON assumes the storage of each physical machine is independent. However, our scenario is based on deployments in which VMs are backed by a centralized storage system, which is common in cloud environments [50]. Moreover, in TRACON the task and VM are independent, but in our case the VM migration task cannot be isolated from the migrating VM. To speedup MapReduce applications, ILA [47] proposes nonlinear interference prediction and adaptive delay scheduling. ILA targets batch processing applications and assumes jobs can be delayed and scheduled at a later time, while our work targets maintaining the performance of delay-sensitive applications. DeepDive [18] monitors low-level system metrics to pinpoint the culprit resources undergoing

interference, then employs VM migration to balance the system loads and mitigate interference. However, the interference caused by the migration activity is not discussed. DeepDive primarily focuses on system management within a local area, but storage migration typically occurs across wide area environments.

Fundamental to our interference avoidance mechanism are I/O burst off-loading and prefetching. The I/O burst caused by storage migration is predominately a series of sequential reads. Similar to Everest [38], which redirects write requests to a low-load volume, we use a buffer to serve the read requests of storage migration so as to redirect the read requests out of the primary storage. In order to better utilize periods of low contention, prefetching is used to pull un-migrated data into the buffer. Correlation-directed prefetching [51] employs frequent sequence mining to achieve fine-grained data preloading. Adaptive feedback-directed prefetching [52] employs counter based feedback to achieve prefetching aggressiveness control. However, since the storage migration can be treated as a sequential stream, sophisticated prefetching algorithms to discover and recognize block correlations are simply not needed. Instead, our method focuses on a rate-controlled prefetching mechanism. Empirically, we employ logistic regression on a handful of I/O features to estimate interference severity in order to dynamically adjust prefetching speed, as well as migration rate.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusion

An entire industry, and with it countless new applications, has been spawned from the rapid growth and success of virtualization technology. At the heart of the modern cloud infrastructure is full machine virtualization, with its strong isolation and administrative perks it has proliferated nearly all corners of modern computing. Live migration, a key enabler of virtualization's flexibility, has seen significant attention from both researchers and practitioners.

In order to mitigate migration storage interference, we've proposed a migration approach that leverages the known sequential pattern of a pre-copy migration, in combination with interference prediction and adaptive transfer. Our approach is successful at abating workload degradation due to interference in nearly every scenario. In some experiments, our methodology must compromise the performance of the migration itself, though we show that the degree of this trade-off is configurable. Still, in the majority of tests where both the migration and workload suffer, our prototype is successful at eliminating any reduction in storage performance, for both the workload and migration.

6.2 Future Work

Our design and prototype invite several future enhancements and areas to explore. Most important is the need for a hypervisor-level implementation of migration buffering, which would provide more granular control of the prefetching process and

introspection into the migrating machine's I/O. An implementation within the hypervisor would allow prefetching at the block-level rather than page-level, while also removing the uncertainty of the operating system's paging decisions.

To reduce complexity, we opted to not address the migrating machine's I/O pattern's in this work, though it would intuitively impact both the buffering we propose and the retransmissions typical of pre-copy migrations. New work in this area should address this challenge more extensively.

In order to combat our design's worst-case scenario, buffering of the virtual disk can begin prior to the actual migration process - much the same way an application buffers a portion of streaming media before beginning playback. For VM migration, an administrator or placement algorithm may indicate their desire to perform migration before it actually occurs. Thus giving time to prefetch and warm the migration buffer as means to mitigate interference.

The control mechanisms centered around our interference-PID may be more complex than required. An exploration into tuning this system, or even reworking it entirely, may make the approach more effective while being easier for others to grasp.

Finally, while successful, our interference classifier is rudimentary - additional work should investigate this component closely as its output controls many aspects of the design. New features should be explored along with additional learning algorithms, with success contingent on correctly classifying entirely unseen workloads.

Appendix A

ABBREVIATIONS

CPU	Central Processing Unit
FTP	File Transfer Protocol
GB	Gigabyte
HPC	High Performance Computing
IOPS	I/O Per Second
KB	Kilobyte
KVM	Kernel Virtual Machine
LAN	Local Area Network
MB	Megabyte
MPI	Message Passing Interface
NFS	Network File System
OS	Operating System
PID	Proportional Integral Gain
QEMU	Quick Emulator
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
R/W	Read/Write
VCU	Virginia Commonwealth University
VM	Virtual Machine
VMM	Virtual Machine Manager
WAN	Wide Area Network
YCSB	Yahoo! Cloud Serving Benchmark

REFERENCES

[1] Gerald J Popek and Robert P Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[2] Paul Barham et al. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462. URL: http://doi.acm.org/10.1145/945445.945462.

[3] Avi Kivity et al. "kvm: the Linux Virtual Machine Monitor". In: *Proceedings of the Linux Symposium*. Vol. 1. Ottawa, Ontario, Canada, June 2007, pp. 225–230. URL: http://linux-security.cn/ebooks/ols2007/OLS2007-Proceedings-V1.pdf.

[4] Michael Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[5] Christopher Clark et al. "Live Migration of Virtual Machines". In: *NSDI'05*. CA, USA, 2005.

[6] Robert Birke et al. "When Virtual Meets Physical at the Edge: A Field Study on Datacenters' Virtual Traffic". In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '15. Portland, Oregon, USA: ACM, 2015, pp. 403–

415. ISBN: 978-1-4503-3486-0. DOI: 10.1145/2745844.2745865. URL: http://doi.acm.org/10.1145/2745844.2745865.

[7] Younggyun Koh et al. "An Analysis of Performance Interference Effects in Virtual Environments". In: *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*. 2007, pp. 200–209. DOI: 10.1109/ISPASS.2007.363750.

[8] Khaled Z. Ibrahim et al. "Optimized Pre-copy Live Migration for Memory Intensive Applications". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 40:1–40:11. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063437. URL: http://doi.acm.org/10.1145/2063384.2063437.

[9] Constantine P. Sapuntzakis et al. "Optimizing the Migration of Virtual Computers". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 377–390. ISSN: 0163-5980. DOI: 10.1145/844128.844163. URL: http://doi.acm.org/10.1145/844128.844163.

[10] Wei Huang et al. "High performance virtual machine migration with RDMA over modern interconnects". In: *Cluster Computing, 2007 IEEE International Conference on*. 2007, pp. 11–20. DOI: 10.1109/CLUSTR.2007.4629212.

[11] Petter Svärd et al. "Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '11. Newport Beach, California, USA: ACM, 2011, pp. 111–120. ISBN: 978-1-4503-0687-4. DOI: 10.1145/1952682.1952698. URL: http://doi.acm.org/10.1145/1952682.1952698.

[12] Jie Zheng, T. S. Eugene Ng, and Kunwadee Sripanidkulchai. "Workload-Aware Live Storage Migration for Clouds". In: *VEE'11*. Newport Beach, USA, 2011.

[13] Pierre Riteau, Christine Morin, and Thierry Priol. "Shrinker: Improving Live Migration of Virtual Clusters over WANs with Distributed Data Deduplication and Content-Based Addressing". In: *Euro-Par'11*. Bordeaux, France, 2011.

[14] Samer Al-Kiswany et al. "VMFlock: Virtual Machine Co-Migration for the Cloud". In: *HPDC'11*. San Jose, USA, 2011.

[15] David Breitgand, Gilad Kutiel, and Danny Raz. "Cost-aware Live Migration of Services in the Cloud". In: *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. SYSTOR '10. Haifa, Israel: ACM, 2010, 11:1–11:1. ISBN: 978-1-60558-908-4. DOI: 10.1145/1815695.1815709. URL: http://doi.acm.org/10.1145/1815695.1815709.

[16] Akane Koto et al. "Towards Unobtrusive VM Live Migration for Cloud Computing Platforms". In: *Proceedings of the Asia-Pacific Workshop on Systems*. APSYS '12. Seoul, Republic of Korea: ACM, 2012, 7:1–7:6. ISBN: 978-1-4503-1669-9. DOI: 10.1145/2349896.2349903. URL: http://doi.acm.org/10.1145/2349896.2349903.

[17] Fei Xu et al. "iAware: Making Live Migration of Virtual Machines Interference-Aware in the Cloud". In: *TRANSACTIONS ON COMPUTERS* PP (99 2013).

[18] Dejan Novaković et al. "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 219–230. URL: http://dl.acm.org/citation.cfm?id=2535461.2535489.

[19] Krishna Kant. "Data center evolution: A tutorial on state of the art, issues, and challenges". In: *Computer Networks* 53.17 (2009), pp. 2939–2965.

[20] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q". In: *Supercomputing, 2003 ACM/IEEE Conference*. IEEE. 2003, pp. 55–55.

[21] Ken Koch. "How does ASCI actually complete multi-month 1000-processor milestone simulations". In: *Proceedings of the conference on high speed computing*. 2002, pp. 22–25.

[22] Wei Huang et al. "A case for high performance computing with virtual machines". In: *Proceedings of the 20th annual international conference on Supercomputing*. ACM. 2006, pp. 125–134.

[23] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[24] James E Smith and Ravi Nair. "The architecture of virtual machines". In: *Computer* 38.5 (2005), pp. 32–38.

[25] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[26] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. "Fast Transparent Migration for Virtual Machines." In: *USENIX Annual technical conference, general track*. 2005, pp. 391–394.

[27] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. "Post-copy live migration of virtual machines". In: *ACM SIGOPS operating systems review* 43.3 (2009), pp. 14–26.

[28] Aidan Shribman and Benoit Hudzia. "Pre-Copy and post-copy VM live migration for memory intensive applications". In: *European Conference on Parallel Processing*. Springer. 2012, pp. 539–547.

[29] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities". In: *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*. Ieee. 2008, pp. 5–13.

[30] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. "A taxonomy and survey of cloud computing systems". In: *INC, IMS and IDC* (2009), pp. 44–51.

[31] Robert Birke et al. "(Big)Data in a Virtualized World: Volume, Velocity, and Variety in Cloud Datacenters". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies*. FAST'14. Santa Clara, CA: USENIX Association, 2014, pp. 177–189. ISBN: 978-1-931971-08-9. URL: `http://dl.acm.org/citation.cfm?id=2591305.2591323`.

[32] V. Tarasov, E. Zadok, and S. Shepler. "Filebench: A Flexible Framework for File System Benchmarking". In: *;login: The USENIX Magazine* 41.1 (2016), pp. 6–12.

[33] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: `10.1145/1807128.1807152`. URL: `http://doi.acm.org/10.1145/1807128.1807152`.

[34] Jens Axboe and Aaron Carroll. *fio(1) Linux User's Manual*. http://linux.die.net/man/1/fio.

[35] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=1247360.1247401.

[36] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[37] Wenjin Hu et al. "A Quantitative Study of Virtual Machine Live Migration". In: *CAC'13*. Miami, USA, 2013.

[38] Dushyanth Narayanan et al. "Everest: Scaling Down Peak Loads Through I/O Off-loading". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 15–28. URL: http://dl.acm.org/citation.cfm?id=1855741.1855743.

[39] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[40] Timothy Wood et al. "CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines". In: *VEE'11*. Newport Beach, USA, 2011.

[41] Eric Keller et al. "Live Migration of an Entire Network (and Its Hosts)". In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. HotNets-XI. Redmond, Washington: ACM, 2012, pp. 109–114. ISBN: 978-1-4503-1776-4. DOI: 10.1145/2390231.2390250. URL: http://doi.acm.org/10.1145/2390231.2390250.

[42] Jie Zheng et al. *Pacer: Taking the Guesswork Out of Live Migrations in Hybrid Cloud Computing*. Tech. rep. TR13-01. Rice University Technical Report, Jan. 2013.

[43] Dhruba Chandra et al. "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture *". In: *HPCA05*. Washington, DC, USA, 2005.

[44] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". In: *ASPLOS10*. Pittsburgh, USA, 2010.

[45] Jason Mars et al. "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: ACM, 2011, pp. 248–259. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155650. URL: http://doi.acm.org/10.1145/2155620.2155650.

[46] R.C. Chiang and H.H. Huang. "TRACON: Interference-aware Scheduling for Data-intensive Applications in Virtualized Environments". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. 2011, pp. 1–12.

[47] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. "Interference and Locality-Aware Task Scheduling for MapReduce Applications in Virtual Clusters". In: *HPDC13*. New York, USA, 2013.

[48] Gokul Soundararajan and Cristiana Amza. "Towards end-to-end quality of service: controlling I/O interference in shared storage servers". In: *Middleware'08*. Leuven, Belgium, 2008.

[49] Navaneeth Rameshan et al. “Stay-Away, protecting sensitive applications from performance interference”. In: *Middleware14*. Bordeaux, France, 2014.

[50] Vasily Tarasov et al. “Virtual Machine Workloads: The Case for New Benchmarks for NAS”. In: *FAST’13*. San Jose, USA, 2013.

[51] Zhenmin Li et al. “C-Miner: Mining Block Correlations in Storage Systems”. In: *FAST’04*. San Francisco, USA, 2004.

[52] Ahsen J. Uppal, Ron C. Chiang, and H. Howie Huang. “Flashy Prefetching for High-Performance Flash Drives”. In: *MSST’12*. Pacific Grove, USA, 2012.