7-2016

ON OPTIMIZATIONS OF VIRTUAL MACHINE LIVE STORAGE MIGRATION FOR THE CLOUD

Yaodong Yang
University of Nebraska-Lincoln, yangyaodong88@gmail.com

ON OPTIMIZATIONS OF VIRTUAL MACHINE LIVE STORAGE MIGRATION FOR THE CLOUD

by

Yaodong Yang

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Engineering
(Computer Science - Computer Engineering)

Under the Supervision of Professor. Hong Jiang

Lincoln, Nebraska

July, 2016

ON OPTIMIZATIONS OF VIRTUAL MACHINE LIVE STORAGE MIGRATION FOR THE CLOUD

Yaodong Yang, Ph.D.

University of Nebraska, 2016

Adviser: Hong Jiang

Virtual Machine (VM) live storage migration is widely performed in the data centers of the Cloud, for the purposes of load balance, reliability, availability, hardware maintenance and system upgrade. It entails moving all the state information of the VM being migrated, including memory state, network state and storage state, from one physical server to another within the same data center or across different data centers. To minimize its performance impact, this migration process is required to be transparent to applications running within the migrating VM, meaning that applications will keep running inside the VM as if there were no migration operations at all.

In this dissertation, a thorough literature review is conducted to provide a big picture of the VM live storage migration process, its problems and existing solutions. After an in-depth examination, we observe that a severe IO interference between the VM IO threads and migration IO threads exists and causes both types of the IO threads to suffer from performance degradation. This interference stems from the fact that both types of IO threads share the same critical IO path by reading from and writing to the same shared storage system. Owing to IO resource contention and requests interference between the two different types of IO requests, not only will the IO request queue lengthens in the storage system, but the time-consuming disk seek operations will also become more frequent. Based on this fundamental

observation, this dissertation research presents three related but orthogonal solutions that tackle the IO interference problem in order to improve the VM live storage migration performance.

First, we introduce the Workload-Aware IO Outsourcing scheme, called WAIO, to improve the VM live storage migration efficiency. Second, we address this problem by proposing a novel scheme, called SnapMig, to improve the VM live storage migration efficiency and eliminate its performance impact on user applications at the source server by effectively leveraging the existing VM snapshots in the backup servers. Third, we propose the IOFollow scheme to improve both the VM performance and migration performance simultaneously. Finally, we outline the direction for the future research work.

ACKNOWLEDGMENTS

My first and sincere appreciation goes to my advisor, Dr. Hong Jiang, for his continuous guidance, support and encouragement throughout my five-year study in University of Nebraska - Lincoln. I first met Dr. Jiang in Huazhong University of Science and Technology in China, where I was a master student at the time. I was impressed by his knowledge and personality during his talk, and was fortunate to become his Ph.D. student in UNL later. Dr. Jiang not only teaches me how to conduct research in storage system, but also helps me to improve my presentation, logic and writing skills. He holds a high standard for the research work all the time and continuously gives me constructive comments and advise for my research, which eventually gives birth to this dissertation. The training I received from Dr. Jiang is vital for my future career development.

I gratefully acknowledge my advisory committee members Dr. Lisong Xu, Dr. Witawas Srisa-an and Dr. Li Tan for their advice, supervision and review of this dissertation. Their valuable input and support give me a chance to do a better job in my dissertation research work.

I would like to thank Dr. Lei Tian and Dr. Bo Mao for their support on my research. Lei helped me a lot when I was in a hard time identifying research problems. Bo has contributed a lot in my research work. I feel very lucky to know them and learn from them.

I also would like to thank all the students and visiting scholars from the ADSL group. These people include Jian Hu, Dongyuan Zhan, Lei Xu, Hao Luo, Ziling Huang, Junjie Qian, Yinjin Fu, Jie Yao, Fang Liu, Zhan Shi, Hua Wang, Wei Tong, Xin Liu, Zhe Zhang, Wen Xia, Yujuan Tan and Zhichao Yan. All of them were very helpful in our weekly group meeting. I'm fortunate to share good memories with

them in the past five years.

Thanks to my brother, Yuekun Yang, for his company in the past 4 years in UNL. Thanks to my parents for their support and love. They always encourage me when I feel discouraged and frustrated.

Most of all, I am profoundly grateful to my wife, Wei Wang, for her endless love and encouragement. We came to Lincoln at the same time and have been good friends since we started our study in UNL. We cherish our shared memories during these five years. I feel doubly and triply blessed to not only earn a Ph.D. degree, but also become a husband and a father (expecting a lovely daughter in this November). I'm very grateful to having them in my life.

Yaodong Yang,

May 2016,

University of Nebraska - Lincoln.

Table of Contents

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

1.1 Challenges in the VM Live Storage Migration 4

1.2 Contributions of the Dissertation 8

1.3 Organization of the Thesis 10

2 Related Work **11**

2.1 Live Memory State Migration 11

2.2 Live Storage State migration 17

2.3 Live Multiple Concurrent Migrations 21

3 WAIO: Workload-Aware IO Outsourcing Live Storage Migration **24**

3.1 Background and Motivation 24

3.1.1 IO Interference Problem 26

3.1.2 Feasibility Analysis of IO Outsourcing 28

3.2 System Design and Implementation 29

3.2.1 WAIO Architecture 29

3.2.2 WAIO Algorithm 32

3.2.3 Data Consistency Issues 34
3.3 Performance Evaluation 35
3.3.1 The Prototype Implementation 36
3.3.2 The Experimental Setup 36
3.3.3 Trace Driven Evaluations 38
3.3.4 Sensitivity Study 40

4 SnapMig: Snapshot-based VM Live Storage Migration 46
4.1 Background and Motivation 46
4.1.1 VM Snapshot 49
4.1.2 VM Snapshot Migration 50
4.1.3 VM Snapshot Backup 51
4.1.4 Motivation 53
4.2 System Design and Implementation 54
4.2.1 SnapMig Architecture 54
4.2.2 SnapMig Workflow 57
4.3 Performance evaluation 60
4.3.1 The Experimental Environment 60
4.3.2 Performance Metrics and Experimental Setup 61
4.3.3 Results Analysis 62
4.3.3.1 Migration of A Single VM 62
4.3.3.2 Simultaneous Migrations of Multiple VMs 63
4.3.4 Sensitivity Studies 65

5 IOFollow: Improving the VM Live Storage Migration by IO Following 68
5.1 Background and Motivation 68

5.1.1 Sequential IO Property 68
5.1.2 Threads Model in Virtualized Systems 70
5.1.3 Motivation 72
5.2 System Design and Implementation 74
5.2.1 Design Objectives 74
5.2.2 IOFollow architecture overview 75
5.2.3 IOFollow Migration Blocks Scheduling 77
5.2.4 Migration-aware Block Cache Manager(MABCM) 79
5.2.5 Data consistency 81
5.3 Performance Evaluation 82
5.3.1 Evaluation Methodology 82
5.3.2 Workload Analysis and Trace Replay 84
5.3.3 Result Analysis 85
5.3.4 Sensitivity Studies 86
5.3.4.1 Chunk Size 87
5.3.4.2 Resource Allocation Policy 88
5.3.4.3 Concurrent VM Migrations 89

6 Directions for Future Research Work 93
6.1 Semantic-Aware Live-Block Migration 93
6.2 Redundancy/Similarity-Based Data Elimination 96

7 Conclusion 102
7.1 WAIO: Workload-Aware IO Outsourcing Live Storage Migration . . . 102
7.2 SnapMig: Snapshot-based VM Live Storage Migration 103
7.3 IO Follow: Improving the VM live storage Migration by IO Following 104

Bibliography **105**

List of Figures

1.1 The VMware vSphere vMotion System 5

3.1 The WAIO system architecture. 30
3.2 Data structure of the WAIO system 34
3.3 VM IO performance during the migration 39
3.4 VM IO performance during the VM live storage migration with two virtual disk images 40
3.5 VM IO performance comparison under different surrogate devices 41
3.6 VM IO performance comparison under different cache modes 42
3.7 VM IO performance comparison under different migration speeds 43
3.8 VM IO performance comparison with different virtual disk image sizes . 44

4.1 The structure of VM snapshots and the workflow for read/write requests 48
4.2 The VM IO performance comparison during the live storage migrations . 52
4.3 The distribution of VM snapshots among production servers and the backup server 53
4.4 The architecture of SnapMig System. 55
4.5 The workflow of the SnapMig scheme. 59
4.6 The VM Performance Comparison in Single VM Migration 63
4.7 The VM Performance Comparison in Multiple VM Migrations 64
4.8 The Total Migration Time in Different VM Migration Approaches 65

4.9 The VM IO Throughput under different Number of Running VMs (Read-Only Workload) 66
4.10 The VM IO Throughput under different Number of Running VMs (Read-Write Workload) 67
4.11 The VM Migration Speed under different Number of Running VMs . . . 67

5.1 The IO threads model of virtualized system during different phases of VM live storage migration 71
5.2 The architecture of IOFollow System 77
5.3 The Migration Performance Comparison in Single VM Migration 86
5.4 The VM Performance Comparison in Single VM Migration 87
5.5 The Migration Performance Comparison in Single VM Migration among Different Migration Chunk Size 88
5.6 The VM Performance Comparison in Single VM Migration among Different Migration Chunk Size 89
5.7 The Migration Performance Comparison in Single VM Migration among Different Storage Allocation Policy 90
5.8 The VM Performance Comparison in Single VM Migration among Different Storage Allocation Policy 91
5.9 The Migration Performance Comparison among Multiple VMs Migration 92
5.10 The VM Performance Comparison among Multiple VMs Migration . . . 92

6.1 An illustrative example of the Redundancy/Similarity-Based Data Elimination scheme. 99

List of Tables

3.1 Comparison between WAIO and The state-of-the-art schemes 26
3.2 VM IO performance during the migration 26
3.3 VM IO performance under different migration speed 27
3.4 Hardware Specifications in Our Experimental Platform 37
3.5 Trace characteristics 38

4.1 An example of VM snapshots distribution and placement during the migration process 58
4.2 Hardware Specifications in Our Experimental Platform 61

5.1 Hardware Specifications in Our Experimental Platform 83
5.2 Trace characteristics 83

Chapter 1

Introduction

The cloud computing technology is revolutionizing how businesses are conducted in the Enterprise IT departments [1]. Enterprises rent IT resources from the cloud providers on an on-demand and pay-as-you-go basis, which brings in numerous benefits ranging from cost savings, to higher level of reliability, availability and scalability. Netflix, a leading video service company in the world, has shutdown all of its own data centers and run all the video services in the Amazon AWS cloud platform [2]. In the last quarter of 2014, five of the largest cloud computing providers, including Amazon, Microsoft, IBM, Google and Salesforce, saw their revenue of the cloud infrastructure service surge by between 37% and 96%. Given its total market of only $16 billion, cloud computing as an industry is still considered in its infancy, since $16 billion is only a tiny fraction of the almost $4 trillion of IT spending for companies globally [1].

Through the virtualization technology and resource consolidation, as well as the usage-based billing model, cloud computing is able to provide on-demand access to computing, data and software utilities as a service that does not impose any constraints on the end-user physical locations and system configurations [3]. Cloud computing has become one of the most important technologies that is poised to fundamentally change people's lives and IT ecosystems in the near future. Cloud

computing's witnessed success and promising prospect can be attributed in part to its underlying computing and storage infrastructure: virtual machine (VM).

As one of the most popular products in the cloud computing market, VMs have been extensively deployed to run different services for customers, such as web service, mail service, database service and printing service [4]. The underlying hypervisors for the virtualized environment are responsible for the management of physical devices and provision of all sorts of virtual devices to VMs. At the same time, hypervisors are supposed to guarantee the isolation and fairness among different VMs on top of a single shared physical server, while improving the overall performance for all the VMs with the accessible physical resources [5].

The *VM live migration* is a built-in module in modern hypervisors, which can migrate a running live VM from one physical server to another, either within the same cluster or across different data centers globally. The primary purpose of VM live migration is to meet the increasing need for load balancing and server consolidation, system maintenance and upgrade, VM mobility and manageability in cloud data centers. This process entails moving the entire state information of a VM being migrated, which includes the synchronization of CPU states, memory states, network interfaces of the target VM between the source and the destination of migration, and VM virtual disk images and snapshots (for reliability and recovery purposes), while the VM is still executing its workload. At the same time, this process is transparent to the applications running within the migrating VM and other co-scheduled VMs on the same physical server. With the advent and wide deployment of the VM-based cloud computing infrastructure, VM live migration, as an essential functional component of the hypervisors, like ESX, XEN, QEMU-KVM and HyperV, is becoming more and more important for several important reasons.

First, the VM live migration feature in hypervisors enables fast and transparent

workload rescheduling among unevenly utilized physical nodes, for energy efficiency and efficient utilization of computer server resources. As shown in a recent study by Gartner Group [6], 61% of the total 518 respondents were conducting server consolidation projects currently while 28% were planning server consolidation in the near future. It is also a known fact that load balancing among hundreds of thousands of servers is a big concern in data centers [7, 5]. With the support of the VM live storage migration, the mapping between VMs and the hosting physical servers can be dynamically adjusted, so that a better level of load-balancing and energy efficiency can be achieved at the runtime.

Second, due to the increasing requirements for system maintenance and upgrade, such as replacing defected components, enhancing system performance, and expanding data storage capacity, data servers and storage subsystems are routinely experiencing system upgrades [8, 9]. VM live migration can migrate all the running VMs out of the servers that are to be repaired or upgraded.

Third, each running VM has its own resource requirement at the runtime, e.g., the memory footprint, the network bandwidth and the storage throughput. If the physical server cannot provide such resources to the VM, the VM will be live migrated to another server that has sufficient resource available.

Fourth, the problem known as *vender lock-in* forces customers to be dependent on the providers for cloud services and prevents them from changing to another provider without substantial switching costs. A flexible and portable VM live migration approach can play an important role in alleviating the vendor lock-in problem.

Finally, hybrid cloud computing, where VMs run in both private and public cloud sites and are live migrated back and forth as requested, is growing as the most popular infrastructure. Nearly half of large enterprises will have deployed hybrid cloud infrastructure in data centers by the end of 2017 [10]. VM live migration becomes

critical to a wider acceptance and deployment of hybrid clouds. In a typical cloud infrastructure, data storage can be either shared or distributed depending on whether the data are stored in a centralized environment, where all servers share the physical storage, or distributed (share-nothing) environment, where each server has its own dedicated storage. In the shared-storage environment, VM live migration only involves the synchronization of CPU states, memory states and network interfaces of the target VM between the source and the destination of migration. VM virtual disk images remain in the shared-storage that is accessible from both the source and the destination. With the growing trend of shared-nothing architectures in cloud data centers and the need for VM live migration across different clouds over WAN, it is increasingly important to consider VM live migration in a distributed, or share-nothing storage environment, where VM live storage migration must also migrates the state of VM virtual disk images and snapshots from the source to the destination. In fact, *VM live storage migration* has become an integral part of VM live migration in modern hypervisors [11, 12, 13].

It is for these reasons, *this dissertation focuses on VM live migration in a distributed storage environment, namely, VM live storage migration.* Figure 1.1 shows an example system of the vMotion System from the VMware company. As the running VM is migrating from one node to another, both the VM's in-memory states and virtual disk images are migrating from the source server to the destination server.

1.1 Challenges in the VM Live Storage Migration

Given that the capacity of the VM virtual storage, which includes the VM's virtual disk images and snapshots, is much larger than that of other VM state information, such as memory state, CPU state and network state, it is crucial to improve the

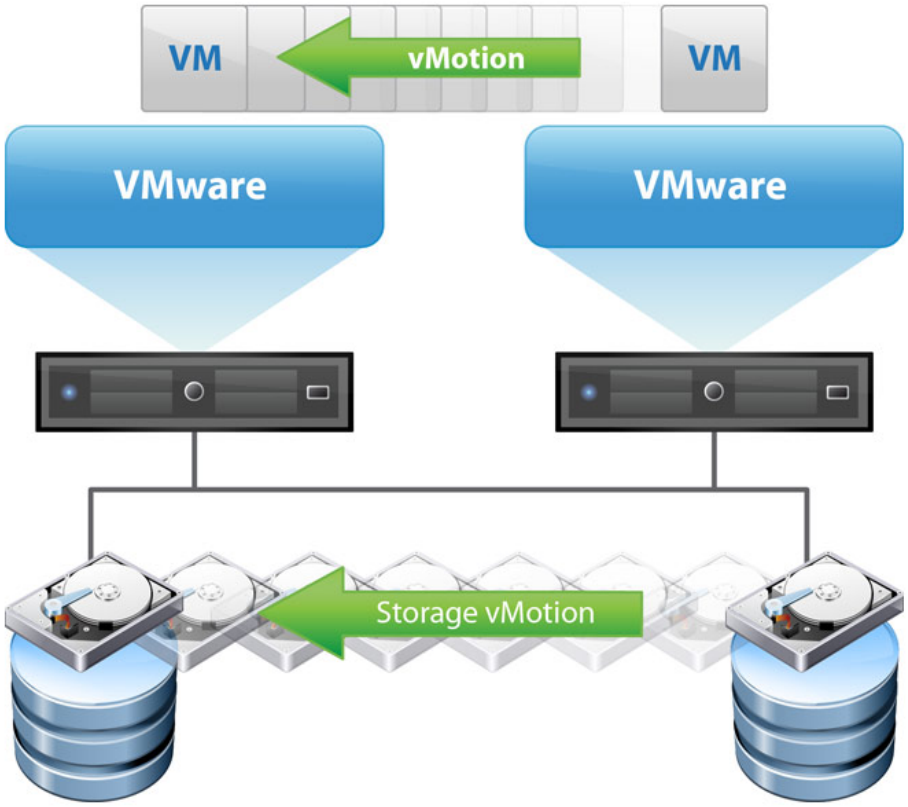


Figure 1.1: The VMware vSphere vMotion System

performance of VM live storage migration. To be specific, any sound and desirable VM live storage migration scheme must posses the following properties:

- ***Short Migration Time:*** In modern data centers, VMs are running 7×24 hours to serve customers globally. The time window for the system maintenance and upgrade is short, so that it is crucial to live migrate a VM quickly. However, the size of a VM's storage image is commonly several to tens of GBs and it may take minutes or even hours to complete a VM live storage migration, which is likely to curtail the capabilities of system management in the cloud data centers. For instance, in one use case from Aliyun [14], the largest cloud service provider by Alibaba in

China, each VM has about 40GB of storage usage and each server holds 25 VMs on average. Assume that every VM evenly shares the 10Gbps network bandwidth attached to a single physical server, from the network's perspective, it still needs about 13 minutes transmit the data required of a single VM live migration. In addition, these VM images share the same storage resource, so that each VM can only obtain a fraction of the total storage bandwidth within the physical server. The limited storage bandwidth is further shared by two types of IO threads: the VM IO threads serving the application and the migration IO threads carrying out VM live migration. Considering that these two types of IO threads interference with each other significantly as they share the same limited resources, the real storage throughput for the migration thread is much smaller than the network bandwidth. Moreover, new updates induced by the VM IO thread during the migration process also need to be migrated to the destination server, which will further lengthen the migration time. Therefore, efficient and effective migration approaches are desired to speed up the VM live storage migration.

- ***High VM IO Performance:*** The IO performance within running VMs must comply with the Service Level Agreement (SLA) all the time. During the VM live storage migration, applications within the migrating VM are still running and they should be oblivious of the migration process. However, the available storage resource is severely stretched due to the additional resource hungry migration threads. In one of our experiments, the VM IO throughput drops from 94.59 MB/s to 65.80 MB/s. If the migration thread consumes the storage resource aggressively, the IO performance of the migrating VM is substantially reduced, thus violating the predefined SLA. In the extreme case, the VM will be stalled and applications cannot run at all. At the same time, the co-located VMs, which reside in the same physical server and assume the same importance as the migrating VM, are also subject to

the reduced storage resource and degraded IO performance. Therefore, the cloud service provider should provide the same performance guarantee for all the co-located, concurrently running VMs in the physical server.

- ***Capability to Migrate Multiple VMs Simultaneously:*** Given the wide deployment of VMs in the data centers, it is common to migrate multiple VMs out of a single server, or migrate multiple VMs into a single server. For the best resource utilization, the mapping from VMs to physical servers should be dynamically adjusted, according to the VM's workload characteristics and priorities [15]. In such cases, multiple resource hungry migration threads will be introduced to the physical server, whose overall storage resource remains unchanged during the migration period. This leads to the available storage resource for a single VM to fall sharply. Therefore, it's much more challenging to achieve acceptable VM IO performance for all the running VMs, while migrating multiple VMs to their destinations at reasonable migration speed.
- ***Capability to Migrate Multiple VM Snapshots:*** As indicated previously, VM snapshots are extensively employed to recover the VM from system crash and data loss, but it comes at a cost. The VM snapshots also need to be migrated to the destination, besides the virtual disk images and other VM state information, in the VM live storage migration. More importantly, the size of each VM snapshot is not negligible and varies significantly, and it largely depends on the write traffic to the running VM. Take the Aliyun cluster use case [14] as an example, the size of each VM is 40GB, and each VM snapshot is 8GB on average, which means 20% of the virtual disk images have been changed since the last snapshot. Therefore, VM snapshots will further deteriorate the VM live storage migration performance.

A number of approaches have been proposed to improve the live storage migration

performance, including optimizing the data block transmission sequence [12] and the migration workflow [11], reducing redundant data transmission [16], and leveraging heterogeneous storage devices [17]. After an in-depth examination, we find that all these approaches fail to address the vital problem of IO interference between the VM IO process and migration IO process, because both types of IO processes share the same critical IO path by reading from/writing to the same shared storage device. Owing to IO resource contention and requests interference between the two different types of IOs, not only will the IO request queue lengthens in the disk, but the time-consuming disk seek operations will also become more frequent. As a result, the performance of the VM IO process will be seriously degraded. Our experimental results show that the VM IO throughput decreases by a factor of up to 6.42 (see Section 3.1.1).

1.2 Contributions of the Dissertation

By addressing the aforementioned problems, we strive to make the following contributions in this dissertation:

- ***WAIO:*** In Chapter 3, a Workload-Aware IO Outsourcing (WAIO) framework is proposed to improve both the VM IO performance and migration performance during the live storage migration. The main idea is to temporarily capture the working set of the target VM and outsource this working set data to a surrogate device during the migration period. By doing so, the VM IO process can access the surrogate device during migration, while the migration IO process accesses the original disk most of the time. As a result, the IO interference between both types of IO processes can be reduced significantly and the overall live storage migration performance can be improved. The surrogate device can be a spare SSD, a spare

hard drive (HDD) or available space in another storage node. This framework is orthogonal to existing optimization approaches, including the DBT [12] and IO Mirroring [11] approaches, and it is a performance booster layer for most, if not all, VM live migration schemes. Further, this framework can also be used to improve the performance of other VM tasks such as VM replication [18], since these tasks will encounter the same IO interference problem as VM live storage migration does. The empirical evaluation of our prototyped system shows that our WAIO framework can improve the VM IO performance by up to 11.83 times, compared with the DBT approach. On the other hand, our system can migrate a VM in a higher speed, without sacrificing the VM IO performance significantly.

- ***SnapMig:*** Motivated by the observation of the VM snapshots-backup process and its resulting snapshots of VM state information available in the backup servers, we propose a novel VM live storage migration scheme, called *SnapMig*, to improve both the VM performance and migration performance simultaneously in Chapter 4. By leveraging the backup servers to transfer migrating VMs' base images and previous snapshots, the source servers only need to migrate the latest VM state changes to the destination servers. Consequently, the otherwise severe interference between the I/O traffic generated by the user applications within the VMs (including the migrating VMs) in source servers and the I/O traffic induced by the VM migration process is significantly reduced, leading to substantial performance improvements to both the VM threads and migration threads. In addition, after the migration, recent VM snapshots made available in the destination servers by the backup servers allow the users to roll back their VMs to previous states freely. Moreover, SnapMig is orthogonal to existing migration approaches, and it is regarded as a performance optimization layer for the existing migration approaches. Finally, we observe that

the performance advantages of SnapMig become more pronounced with the concurrent live storage migrations of multiple VMs.

- ***IOFollow:*** In Chapter 5, we present the IOFollow system that is designed to boost the VM live storage migration performance. Motivated by the fact that the order of the blocks being migrated, sequential or random, does not impact the performance significantly because the migrating VM's virtual disks can not be recontructed until all the data blocks are available in the destination server, while the order of the VM IO requests does since each IO request is consumed by applications in the runtime, the IOFollow system schedules the block migration sequence according to the VM IO requests, so that the time-consuming disk head movements can be reduced significantly. In addition, by selectively caching data blocks from migration threads, incoming VM IO requests can be served in the memory, thus minimizing the number of memory accesses at the same time. In this way, both VM IO performance and migration performance can be improved significantly. Our experimental results show that the performance improvement is much more compelling in the multiple concurrent VM migration scenarios.

1.3 Organization of the Thesis

The rest of the dissertation is organized as follows. In Chapter 2, we introduce the relevant background about VM live storage migration systems and algorithms. From Chapter 3 to Chapter 5, we present our solutions to solve the problems of VM live storage migration identified in Chapter 1. Chapter 6 and 7 explore the future research work and conclude the dissertation.

Chapter 2

Related Work

In this chapter, we introduce the basics of VM live migration, the prevailing VM migration schemes addressing various issues, of which some represent the state of the art, and their key features.

2.1 Live Memory State Migration

Virtual Machine (VM) migration, in principle, is the transferring of the CPU state, memory state, device state, network state, storage state, and other states of VMs from one physical node to another over LAN or WAN [19, 20]. One straightforward way to perform a VM migration operation is to suspend a VM at the source, then transfer the VM states, and finally resume the VM at the destination. The advantage of this approach is simple and easy to implement. However, it needs to interrupt the running OS and applications in the VM, whose long downtime is intolerable for applications on non-stop services. Therefore, VM live migration, which migrates VMs on the fly, becomes an imperative feature of virtual machine monitors (VMM) and most, if not all, modern VMMs support VM live migration. Among the transfers of various VM states, that of VM's memory state usually takes a major portion of the migration time and a number of approaches have been proposed to accelerate the live memory state migration [21, 22].

In general, memory migration approaches can be classified into two categories: pre-copy memory migration and post-copy memory migration. Pre-copy memory migration copies all the memory pages from the source to the destination. Because many memory pages can be modified by OS and applications during the transfer, re-transfers of modified (dirty) pages are required until the retransfer rate is equal to or higher than the modifying rate. Once this point is reached, VMM will suspend the VM on the source node, transfer the remaining dirty pages, and resume the VM on the destination node. The downtime ranges from milliseconds to seconds depending on the amount of dirty memory pages. In post-copy memory migration, in contrast, the VMM suspends the VM on the source node, transfers a minimal subset of the VM states (e.g., the CPU state, device state, and network state), resumes the VM on the destination node although most of the memory state still resides on the source node. Then a background copying process is initiated to transfer the remaining memory pages from the source to the destination. When the VM tries to access pages that have not been transferred, page faults will be generated and trapped by VMM on the destination node, and redirected to the source node over the network. Comparing with pre-copy memory migration, post-copy memory migration has a much shorter downtime while significantly degrading the performance of user applications during migration. The pre+post copy memory migration aims to strike a balance between user performance and downtime. In Pre+post copy memory migration, upon the completion of the transfer of the original memory state, the VMM suspends the VM at the source and resumes it at the destination[23, 12].

Since the access latency of persistent storage systems is still several orders of magnitude slower than that of volatile memory chips, modern operating systems aggressively cache data from the storage system in memory in order to hide the long access latency. Therefore, there is a large portion of data cached in memory with a

duplicated copy in the storage system. Changyeon *et al.* [24] have observed that the amount of data duplication between memory and storage device is higher than 55% or 2.2 GB of data in a Linux server. When it comes to VM live migration, it is not only time consuming but also unnecessary to transfer these duplicated memory pages from the source server to the destination server. Based on this observation, they propose to track the duplicated memory pages in the source server at the runtime. When migrating, instead of migrating these duplicate pages over the rate-limited connection to the destination, the destination server directly fetches these pages from the shared storage server. Therefore, the total data transmission is reduced significantly, and the live migration performance is improved as well [24, 25].

Hai *et al.* [26] propose to classify memory pages into several types according to different characteristics, such as high word similarity, low word similarity, a large number of zero bytes, and then adopt different compression algorithms to compress memory pages with different properties. As a result, a better tradeoff of computation and compression ratio can be achieved. Because of the smaller amount of data transmission and low compression overhead in the live migration period, the total migration time and downtime are both significantly decreased.

Petter *et al.* [27] propose to employ a delta compression algorithm to improve the VM live migration performance, e.g., shortening total migration time and VM downtime. In this approach, a delta memory page is computed for each dirty page, and then the delta page is compressed and migrated to the destination server, rather than the corresponding raw memory page. In the destination server, the raw memory page can be reversed from the delta page and the previous copy. As there are lots of zero bits in the delta page, it is much easier and more efficient to compress delta pages than raw memory pages. Therefore, the total data transmission during the VM live process is largely reduced [27].

Haikun *et al.* [28] design a new live VM migration scheme based on checking-pointing/recovery and trace/replay techniques. Execution trace files are generated in the source node, which contains enough information to replay a long-term execution of the VM instruction-by-instruction. By iteratively transferring trace files from the source node to the destination node, all the VM state information will be synchronized in the destination server. Due to the smaller size of trace files, compared with dirty memory pages, VM migration downtime and network bandwidth consumption is greatly diminished.

Michael *et al.* [29] design and implement a post-copy based VM live migration scheme in the XEN hypervisor. In their implementation, adaptive pre-paging and dynamic self-ballooning are adopted to reduce the total migration time and VM down-time in the VM live migration. With adaptive pre-paging, VMs' memory working set can be analyzed by the sequence of previous memory page requests, so that memory pages can be proactively pushed to the destination server before VMs issue the access requests on them. Under the dynamic self-ballooning scheme, the VM can return the free memory pages to the underlying hypervisor before the VM live migration starts. Therefore, the total memory footprint of the migrating VM is significantly reduced.

Kai-Yuan *et al.* [30] argues that the VM live migration can be improved by exploiting the application's assistant. For instance, many Java applications are running within VMs and there are plenty of dirty pages waiting for the Garbage Collection of JVM to reclaim. The removal of these dirty pages from the VM live migration process will not only speed up the VM live migration process, but also have no impact on the applications themselves. The experimental results show that the completion time, network traffic of transferring memory pages and application downtime, all improved by up to 90%, compared with conventional VM live migration scheme.

In the virtualized environment, physical memory pages are under the control of

hypervisors, which then provide individual VMs with virtual memory pages as requested. All the virtual memory pages that are allocated to and not used by VMs reside in the free memory pool. The content of these free memory pages is irrelevant to the VM, so that they can be treated as zero pages. The VM live migration performance will be improved significantly if these free pages within migrating VMs can be detected and removed from the live migration process. Jui-Hao *et al.* [31] design a novel introspection scheme that can effectively identify the free memory pool without the byte comparing of memory pages. With such Virtual Machine Introspection scheme, both the VM live migration and VM memory deduplication can be improved significantly.

PMigrate [32] is a framework that aims at parallelizing VM live migration, as the increasing amount of resources allocated to individual VMs also offers opportunities to leverage such resource for parallelization in the VM live migration. Not only can the dozens of vCPUs, but also the dozens of NIC ports can be leveraged to migrate the VM's state information. In addition, they design a dynamic fine-grained lock abstraction, the range lock, to increase the parallelism of concurrent mutation in a shared memory address space.

Senthil *et al.* [23] analyze four popular optimization techniques for VM live migration, including Delta Compression, Page Skip, Subpage deduplication and Data Compression. They demonstrate that the performance gain of any optimization technique is closely related to the application's characteristics, by conducting VM live migration experiments with different optimization techniques and different applications. Furthermore, several guidelines have been provided in the selection of suitable optimization technique and the possible combination of different optimization schemes.

Enlighted Post-Copy [33] is an optimization scheme for the VM live migration,

which exploits the VM's runtime information as enlightenment during the VM live migration process. Prior to the VM live migration, the enlightenment information that contains the VM's runtime information, such as the VM's working set and free pages, is passed to the hypervisor. After the VM is resumed at the destination server, memory pages within the VM's working set are transferred to the destination server first. At the same time, memory pages that belong to the free pool of the VM are discarded in the source server. Therefore, the total migration time is reduced significantly and the running VM's performance is improved as less page fault occurred in the destination server.

SRVM [34] is a hypervisor support mechanism for VM live migration with pass-through SR-IOV network devices. SR-IOV pass-through can provide much better VM performance compared with para-virtualization schemes. However it introduces challenges for the VM live migration, as hypervisors cannot freely save/restore pass-through devices like para-virtualization devices. SRVM solves these challenges by providing hypervisors support for tracking dirty memory pages and provisioning VFs after VM live migration. At the same time, it does not require any modifications in guest OS or driver. With the SRVM scheme, the virtualization system can gain both high VM performance (by SR-IOV device) and flexible VM live migration capability (with SRVM support) at the same time.

All the research works above improve the VM migration for different scenarios. For instance, Kai's approach [30] can only applicable to the VMs that are running Java Applications inside. Meanwhile, they are different from our migration approaches (WAIO, SnapMig and IOFollow), as they focus on main memory layer of the VM live migration with the shared storage, while our approaches are mainly improving the VM live storage migration in the storage layer without shared storage. However, they provide us with a big picture of the state-of-the-art research works and inspire us to

design new solutions to improve the VM live storage migration performance.

2.2 Live Storage State migration

In a typical VM live migration process in the non-shared storage environment, the transferring of memory state and virtual disk images accounts for the most of the migration time and the network bandwidth consumed. Because migrating the virtual disk images usually takes much longer time than migrating the memory state, the research problem of how to deliver an effective and efficient VM live storage migration has attracted a great deal of attention from academia and industry [35, 36, 37].

In the early generation of VMware vSphere vMotion [11], the Snapshot scheme leverages the virtual machine snapshots and iteratively consolidates a series of snapshots from the source to the destination. Each snapshot not only preserves the VM's power state, like powered-on, powered-off or suspended, but also contains all the files that make up this VM, including virtual disks, memory footprint, virtual network interface and other devices [38]. Along soon as the size of the last snapshot drops blew a predefined threshold, the iteration is stopped; the VM is suspended in the source, and then resumed in the destination. Due to the performance and consistency issue, this scheme is rarely used in current systems. Different from Snapshot, vMotion's DBT(Dirty Block Tracking) [11] performs updates in place, instead of logging writes in a snapshot file, and uses a dirty block table to keep track of the writes during the last iteration of storage migration. Both Snapshot and DBT share a major problem: the downtime between the suspension and resumption of the VM can be relatively long under write-intensive workloads. To address this problem, IO Mirroring [11] is proposed to mirror every write request to both the source and destination nodes during the live migration. In the background, the virtual disk images of the VM(except

the data in the new write requests) are being migrated to the destination. Once this migration is done, the virtual disk images in the source node are identical automatically to those in the destination node, since all the data in the new write requests have already been mirrored to the destination. Therefore, when the VM is suspended in the source, only the VM's memory state and other states need to be transferred to the destination before the resume of VM in the destination node. IO Mirroring decreases the downtime significantly at the expense of the increased network traffic(the mirroring of every write request).

Luo *et al.* [39] propose a three-phase migration algorithm that provides minimal downtime and keep the system consistency. Also, they present an incremental migration scheme to facilitate the migration back to the source node. The evaluation result shows that the downtime of this algorithm is around 100 milliseconds, close to the the migration in shard storage environment. The migration time is reduced largely by avoiding the unnecessary data transmission as well. However, the reduction of migration time can be achieved only when the VM is going to migrate back to the server that holds the VM's previous virtual disk images.

In order to avoid unnecessary retransfer of frequently-updated blocks during iterations of dirty block transfers, Zheng *et al.* [12] proposed a scheme that distinguishes frequently-updated areas from infrequently-updated areas, and transfers infrequently-updated data blocks prior to frequently-updated data blocks. By doing so, the migrated data blocks will be less likely to become dirty and request a re-transmission before the completion of VM live migration. Therefore, both the total amount of data transferred and the migration time is decreased significantly. However, this scheme can not improve the VM IO performance during the live migration. The VM IO process accesses the popular region of the virtual disk images; while the migration IO process accesses the unpopular region of the virtual disk images. Therefore, the disk

head has to seek back and forth, thus of significantly IO performance drop during the migration.

Shrinker [16] is a distributed system that is capable of migrating a virtual cluster over WAN. It has two built-in services: Coordination Service (in the source site) and Indexing Service (in the destination site). The Coordination Service tracks the hash values of memory pages and virtual disk blocks that have already been transferred to the destination site, so that hypervisors in the source side can perform data deduplication by replacing duplicated transmission of memory pages and disk blocks with their hash values of them. The Indexing Service records the hash values and the location information of the memory pages and disk blocks in the destination side. Hypervisors in the destination can reconstruct the VM's memory and virtual disk images with the communication between Indexing Service and other hypervisors that hold the real data. As a result, the total data transmission and migration time is reduced substantially. This scheme can reduce the redundant data transmission, rather than the the amount of data read from the disk. Every data block in the virtual disk images is read into the memory, and then the fingerprint is calculated. Based on the fingerprint, Shrinker can decide whether to transfer the data block or its fingerprint. Therefore, the IO interference between VM IO requests and migration IO requests still exists in the disk. In addition, this scheme increases the computation overhead(generating fingerprint for each data block).

Zhou *et al.* [17] take the speed discrepancy between HDDs and SSDs, and the wear-out issue of SSDs into consideration in order to optimize the live storage migration. They propose the three optimizations of 1) low redundancy storage migration designed to reduce the total data transmission; 2) improving the VM IO performance when migrating a VM from SSDs to HDDs by leveraging SSDs higher performance and the Source-based Low Redundancy Storage Migration; and 3) an asynchronous IO

Mirroring mechanism to significantly reduce the IO response time for each request in the running VM during the migration. In the WAN environment, the VM IO performance will degrade since the VM needs to fetch the latest updates form the destination server(across WAN), that is usually much slower than that in the LAN environment.

Ali *et al.* [40] build a VM live storage migration system, called XvMotion, to migrate VM over long distances across heterogeneous systems, with performance similar to that of VM migration in Local Area Network. In order to achieve this goal, they proposed several techniques and optimizations, including streams transport framework, asynchronous IO mirroring, memory and disk coordination, stun during page send and so on.

Eventually, the performance of the storage system in the virtualized environment plays a vital role for the VM live storage migration performance. Many other research works also indirectly improve the VM live storage migration performance by the contributions in IO scheduler [41, 42], file systems [43, 44, 45], Solid State Drives [46, 47, 48] and data deduplication techniques [49, 50].

In the VM live migration, the VM is running applications most of the time(except the downtime window), so the IO performance of the running VM is critical in the cloud environment. All the schemes mentioned above mainly focus on the reduction of total data transmission, migration time and VM downtime, but fail to improve the VM's IO performance. The goal of this dissertation is to improve the VM IO performance and the migration performance during the VM live storage migration, by addressing the IO-interference problem.

In principle, Our schemes share the same goal as previous research works, which is to speed up the VM live storage migration performance and provide reasonable IO performance for the migrating VM. However, they achieve this goal through different

ways. Our schemes try to minimize the serious IO interference problem by leveraging the existing idle base images and previous snapshots in backup servers, outsourcing the VM's working set to surrogate storage device and scheduling the migration sequence. Previous research works focus on the performance optimization techniques of the aggregate IO streams in the source server, such as leveraging the fast read performance of SSD, removing the redundant data transmission and so on. Our schemes can be combined with these research works together to achieve better VM live storage migration performance.

2.3 Live Multiple Concurrent Migrations

Given the widely deployment of VMs in current cloud data centers, it's not uncommon to migrate multiple VMs from/to a single server simultaneously. The resources contention between migrating threads and VM threads surges as the number of VM involved in the migration process at the same time, so that it's more challenging to perform multiple VM migrations fast and guarantee the SLA for all customer applications. A number of research works have been targeting this problem and we highlight several representative research work as follows:

VMScatter [51] is a multicast-based VM live migration system, which can efficiently migrate a group of VMs from one shared source server to multiple destination servers. Given that there are plenty of identical memory pages across VMs [52, 53], VMScatter can transfer these identical pages to many different servers simultaneously in a single transmission from the source host, instead of transferring each page to each destination server individually. All the unique or dirty pages will be unicasted to the related server separately. Therefore, both the total data transmission and network traffic will be reduced significantly.

Timothy *et al.* [15] argue that manually-initiated VM live migration holds several disadvantages: 1) lack the agility to response to the sudden workload changes or hotspots; 2) is error-prone in the decision of new mapping from VMs to physical servers, since there are many factors involved, such as CPU, memory and network for each application and each physical server. They design and implement the Sandpiper system that contains two components: hotspot detection algorithm and hotspot migration algorithm. The hotspot detection algorithm will decide when to migrate VMs, while the hotspot migration algorithm will determine where to migrate and how much resources to allocate after the migration. In addition, black-box and gray-box strategies are proposed to identify hotspots in the virtualized system.

Live Gang Migration [52] is inspired by the fact that co-located VMs often have many identical memory pages, such as the same operating systems, same applications and libraries, same Java Virtual Machines. Existing migration approaches will transfer these duplicated memory pages for every VM involving in the live migration process, which not only slow down the migration process, but waste the previous network resources. In the live gang migration approach, identical memory pages will be identified and duplicated prior to the transmission of VM state, so that only a single memory page copy needs to be migrated. They also exploit the benefits of different granularities for the identical pages detection algorithm, like whole memory pages and subpages.

Tao *et al.* [54] identify the synchronization problem in the multiple VM migration scenarios. Given that many applications are deployed in multiple VMs with different purposes. For instance, one VM run the web server and the other one run database server. From the performance perspective, these VMs should reside in the same server; otherwise, they will encounter heavy communication overhead within application logics. In the migration of co-located VMs, it is crucially to coordinate

their migration process so that these VMs will reside in the same physical server.

Jie *et al.* [13, 55] investigate the live migration of multi-tier applications, and they indicate that the coordination of multiple VM migration plays a crucial role for the overall performance of user applications. Suppose a user application consists of three VMs: web server VM, application server VM and database VM. All these three VMs reside in the same physical server, and need to migrate to another server, due to system maintenance or load balancing requirement. There would be a high network latency that will drag down the overall application performance, if any one or two VMs have reached the destination server, while the other ones are still running in the source server. In order to solve this problem, they design the COMMA system that coordinate the migration progresses based on the communication impact among VMs, so that the migration impact on multi-tier applications' performance is effectively minimized.

Solely Relying on the approaches above can not provide the satisfiable VM live storage migration performance, as the core IO interference problem is not addressed sufficiently by the existing approaches. Our WAIO, SnapMig and IOFollow are designed to tackle this interference from the beginning and minimize this interference from different perspectives. They are orthogonal to the existing solutions, so that they can be combined with previous approaches to further improve the VM live storage migration.

Chapter 3

WAIO: Workload-Aware IO Outsourcing Live Storage Migration

3.1 Background and Motivation

As introduced in Chapter 2, there are many approaches to optimize the VM live storage migration [11, 38, 11, 11, 39, 12, 16, 17, 40]. After an in-depth examination, we find that all these approaches fail to address the vital problem of IO interference between the VM IO process and migration IO process, because both types of IO processes share the same critical IO path by reading from/writing to the same shared storage device. Owing to IO resource contention and requests interference between the two different types of IOs, not only will the IO request queue lengthens in the disk, but the time-consuming disk seek operations will also become more frequent. As a result, the performance of the VM IO process will be noticeably degraded. Our experimental results show that the VM IO throughput decreases by a factor of up to 6.42 (see Section 3.1.1).

In this work, we propose a Workload-Aware IO Outsourcing (WAIO) framework to improve both the VM IO performance and migration performance during the live storage migration. The main idea is to temporarily capture the working set of the target VM and outsource this working set data to a surrogate device during the migration period. By doing so, the VM IO process can access the surrogate device during migration, while the migration IO process access the original disk most of

the time. As a result, the IO interference between both types of IO processes can be reduced significantly and the overall live storage migration performance can be improved. The surrogate device can be a spare SSD, a spare hard drive (HDD) or available space in another storage node. This framework is orthogonal to existing optimization approaches, including the DBT [12] and IO Mirroring [11] approaches, and it is a performance boost layer for most, if not all, VM live migration schemes. Further, this framework can also be used to improve the performance of other VM tasks such as VM replication [18], since these tasks will encounter the same IO interference problem as VM live storage migration does. The empirical evaluation of our prototyped system shows that our WAIO framework can improve the VM IO performance by up to 11.83 times, compared with the DBT approach. On the other hand, our system can migrate a VM in a higher speed, without sacrificing the VM IO performance significantly.

Both our WAIO and Zheng *et al.*' work [12] exploit the application's IO access characteristics, but they improve the VM live migration performance in different ways. Zheng's scheme aims to reduce the total amount of data transferred significantly, by exploiting the VM's workload locality. Through the analysis of the workload locality, infrequently updated data blocks are distinguished from frequently updated data blocks in virtual disk images. The infrequently updated data blocks are transferred prior to frequently updated data blocks in the migration, so that the re-transmissions of data blocks are minimized, thus reducing total amount of data transmission. While WAIO also exploits the workload locality, its methodology is completely different from Zheng's. WAIO makes use of workload locality to capture and outsource the VM's working set to a surrogate device during the migration, which does not affect the transmission sequence of data blocks. Importantly, WAIO is orthogonal and complementary to the above approaches and can further improve these techniques.

Table 3.1 briefly compares WAIO with the above approaches.

Table 3.1: Comparison between WAIO and The state-of-the-art schemes

Features	DBT	IO Mirroring	Zheng[6]	WAIO
Migration Time Reduction	✓	✓	✓	✓
VM IO Performance				✓
Workload Locality			✓	✓

3.1.1 IO Interference Problem

As aforementioned, we hypothesize that the IO interference between migration IOs and VM IOs is the root cause of slow migration speed and VM IO performance degradation. On one hand, the VM migration process reads data blocks from the virtual disk images at the source and writes them to the virtual disk images at the destination. On the other hand, VM IOs are serviced by hypervisors and directed to the virtual disk images at the source. Two concurrent but independent streams of IOs result in a contention of the hard disk head at the source, since only one disk head can perform an IO operation at any given time. Not only do the seek operations of the disk head become more frequent, but the queue of IO requests also gets longer. Therefore, it is difficult to achieve a better VM IO performance and shorter migration time simultaneously during the VM live storage migration. This IO interference problem clearly remains in the existing approaches, because of the co-existence of VM IOs and migration IOs at the source [56].

Table 3.2: VM IO performance during the migration

Throughput (MB/s)	Sequential IOs	Random IOs
No Migration	58.44	5.46
Live Migration	19.43	0.85

In order to validate our hypothesis, we conduct experiments on the QEMU-KVM system in a LAN environment. A running VM is migrated back and forth between two servers over a 1Gbps Ethernet. The IO requests are generated by IOMeter [57] with 60%/40% read/write requests, in a Windows XP VM of 15GB. The size of each sequential request is 32KB, while that of each random request is 4KB. The migration speed is 20.8MB/s. Table 3.2 shows the VM IO performance degradation during live storage migration. As indicated in this table, the throughput decreases by a factor of 3.01 for sequential requests and 6.42 for random requests during the live migration, compared to the scenarios without migration. Such a significant IO performance degradation can potentially cause the running applications inside the VM to be suspended, thus violating SLA. Table 3.3 shows the migration performance under different migration speeds. In this experiment, we evaluate the performance of sequential IO requests under different migration speeds. The migration speed is set through the QEMU-KVM system. As shown in this table, by setting a higher migration speed of 40.14MB/s, the total migration time is reduced to 25.5% of that under the lower speed of 10.26MB/s, while the VM performance is degraded by a factor of 4.55. Clearly, given the limited resources (e.g., storage bandwidth) and IO interference, there is a tradeoff between the migration and VM performances and one or another must give.

Table 3.3: VM IO performance under different migration speed

Migration Speed (MB/s)	10.26	40.14
VM IO Throughput (MB/s)	42.25	9.29
Migration time (s)	1498	382

3.1.2 Feasibility Analysis of IO Outsourcing

In order to alleviate the IO interference problem, one possible solution is to "outsource" VM IOs so that migration IOs can occupy the disk head and perform sequential read operations if possible. This point is also consistent with the known fact that the offline storage migration approach (with no external IO requests) is much faster than its live counterpart (with concurrent external and internal IO requests). Before we can leverage the IO outsourcing to optimize VM live storage migration, there are two key questions to answer: What IOs to outsource and where to outsource.

To answer the first question, we need to take workload characteristics into considerations [58, 59]. First, reads must be synchronous while writes can be asynchronous. This means that if writes can be accommodated by a surrogate storage device for IO outsourcing, the completion of a write can be immediately returned to the user without any interference to the virtual disk images. Second, a read may access the storage medium of the virtual disk image and cause a difficulty of IO outsourcing. Fortunately, previous studies indicate that access locality is one of the key web workload characteristics and observe that 10% of files accessed on a web server account for approximately 90% of the requests and 90% of the bytes transferred [60]. In the virtualized environment, the IO requests of virtual disks also have strong temporal and spatial locality. For instance, another study [12] indicates that, in a file server, 72% of the blocks that are read during the migration process were also read before the start of the migration. Among these blocks, 96% are read for more than three times during migration. Strong spatial locality is also confirmed in this study. Based on the strong locality of IO requests, the working set of the migrated VM is expected to be reasonably small, so that it is viable to outsource the popular read requests and all the writes to the surrogate device.

To answer the second question, we recognize and leverage the ubiquitous spare/free storage resources in data centers. In the cloud environment, a temporary virtual disk can serve to hold the VM's working set, as long as this temporary virtual disk does not compete for the storage bandwidth with the original virtual disks. The placement of the temporary virtual disk is quite flexible, including hard disks, SSDs or RAIDs. In case there is no spare device, free storage space on the storage systems under light loads can also hold the temporary virtual disk.

3.2 System Design and Implementation

In this section, we first present the architecture overview of the WAIO system, and then we introduce the algorithm of WAIO in details. The data consistency issue of WAIO is discussed at the end of this section.

3.2.1 WAIO Architecture

Figure 3.1 shows the architecture overview of the WAIO system. WAIO is an augmented module in the IO stack layer of hypervisors, and it includes five functional modules: Popular Data Identification, Surrogate Space Manager, IO Redirector, Space Reclaimer, and Administrator Interface. They all reside in the source node of VM live storage migration. The responsibilities of the five modules are elaborated as follows:

Popular Data Identification tracks the popularity of read requests from the VM itself in the virtual disk images. Only the popular data blocks that will be read are outsourced to the surrogate device. Since the surrogate device serves all write requests, it is unnecessary to track the popularity of write requests. Each virtual disk image of the running VM is divided into fixed size chunks, and the Popular

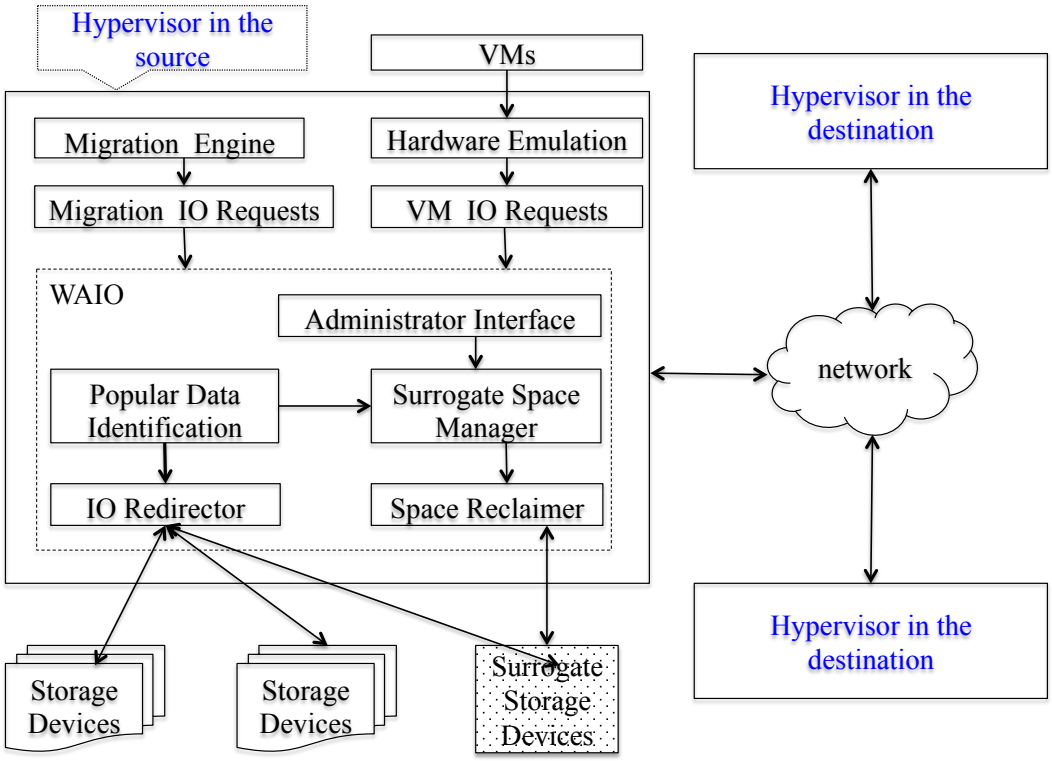


Figure 3.1: The WAIO system architecture.

Data Identification module records the access frequency for each chunk. If the access frequency for a particular chunk exceeds a predefined threshold, the whole chunk will be outsourced to the surrogate device. All the subsequent accesses to this chunk will be served by the surrogate device, which removes their IO interference with the migration process.

The migration module normally scans the whole virtual disk images by sending read-only requests. Most of these requests are only issued once, with the exception of the requests that read dirty data blocks. Therefore, the popularity of data chunks is not affected by the requests from the migration module.

IO Redirector redirects the appropriate IO requests from the running VM itself to the surrogate device. It redirects all the write requests on the surrogate device, in order to reduce the IO traffic to the original storage device. Meanwhile, all the popular read requests, identified by the Popular Data Identification module, are redirected to the surrogate device by the IO Redirector. When the surrogate device only has partial data for a request, the IO Redirector will issue read requests to the original storage device, and merge the data from the original device with that in the surrogate device. The non-popular read requests will be directed on the original storage device and the data chunk will be outsourced to the surrogate storage device only if the access frequency exceeds a predefined threshold.

The read requests from the migration module can be redirected to either the original storage device or the surrogate device. While the original storage device provides the bulk of the virtual disk images, the surrogate device supplies the updated data chunks. Most of the requests will be redirected to the original storage device, due to the VM workload locality.

Surrogate Space Manager is responsible for managing the surrogate device and perform garbage collection within the surrogate device. The IO Redirector needs to request free storage space from the Surrogate Space Manager, before the data is stored on the surrogate device.

Space Reclaimer is used to reclaim all the blocks in the surrogate device after the migration is completed, so that the surrogate device is available to other storage services. It is also responsible for freeing the data structures in memory.

Administrator Interface provides an interface for system administrators to configure design options. Particularly, WAIO collects the information about the surrogate storage device through this interface.

The design of WAIO is quite flexible. First, the Popular Data Identification

module can be implemented with different algorithms that capture the locality of read requests for different applications. We employ the access frequency to evaluate the popularity of data chunks, and there are many other algorithms for the same purpose [12]. WAIO can easily implement any combination of them to capture the popular read requests efficiently for different workloads. It may be preferable to capture and analyze the IO access patterns before the start of migration, and then choose the most suitable algorithm in the WAIO during the live migration. Second, the surrogate storage device can be a spare SSD, HDD, RAID or free storage space on other nodes, as long as there is free comparable storage bandwidth. Finally, WAIO can be incorporated into various migration approaches (e.g., DBT and IO Mirroring) and other VM functionalities (e.g. VM Replication). In this paper, we focus on the VM live storage migration scenario.

3.2.2 WAIO Algorithm

WAIO exploits the VM's working set and outsources it to the surrogate storage device during the migration period. For this purpose, WAIO needs to track the following information in memory, as depicted in Figure 3.2.

1. The popularity of each chunk in the virtual disk images: WAIO employs a Hash Table to track the accessed data chunks during the migration. When a data chunk is accessed for the first time, WAIO adds an entry in the Hash Table for this data chunk. The key is the data chunk's logical chunk id, and the value contains the access count (frequency) and a pointer to the chunk entry. For subsequent accesses, we need to keep updating the corresponding entries in the Hash Table.

2. The status of each chunk: During the migration, data chunks of the virtual disk images have different states. In order to track such state information, WAIO employs several fields in the chunk entry, including Cache Bit, Dirty Bit,

Dirty Bitmap Pointer and Physical Address. The Cache Bit records whether the corresponding chunk is already cached in the surrogate storage (Cache Bit is set to 1) or not (Cache Bit is set to 0). The Dirty Bit indicates whether new data has been written to this chunk during the migration period. If so, the Dirty Bit is set to 1; otherwise set to 0. When the Dirty Bit is set, a Dirty Bitmap is required to track the updates to this chunk in details (e.g., which bytes in this chunk have been overwritten) and its address is stored in the Dirty Bitmap Pointer field of the chunk entry. Finally, The starting address of the data chunk in the surrogate storage is recorded in the Physical Address field of the Chunk Entry.

In the VM live storage migration, both the running VM and migration module in the hypervisor send IO requests to the virtual disk images of this VM. If an IO request needs to access multiple consecutive data chunks, the request will be divided into multiple sub-requests that will access their corresponding data chunks concurrently. The original request will not complete until all the sub-requests finish their IO operations. If an IO request only needs to access one data chunk, then the request itself is also regarded as a sub-request. All sub-requests will be serviced by the original storage device and the surrogate storage device.

An IO sub-requests from the running VM can be either a read and write request. For the read sub-requests, WAIO first looks up the Hash Table for the corresponding data chunk. If there is no such an entry (the data chunk is not in the surrogate device), WAIO serves this sub-request by reading the data chunk from the original disk. In addition, the whole data chunk is outsourced to the surrogate storage device, and a new entry is inserted to the Hash Table, which indicates that this data chunk is in the surrogate storage. If the entry exists, WAIO will check the chunk entry to determine the status of this data chunk. When the Cache Bit is set (the whole chunk is cached in the surrogate storage), this sub-request will be directly served by the

surrogate storage device. Otherwise, WAIO will perform the following operations: 1) read the whole chunk from the original disk, 2) merge them with the dirty data in the surrogate storage, 3) return data to the sub-request, 4) update the chunk entry information. For the write sub-requests, WAIO only needs to write the data to the surrogate storage device and update the corresponding chunk entry in the Hash Table.

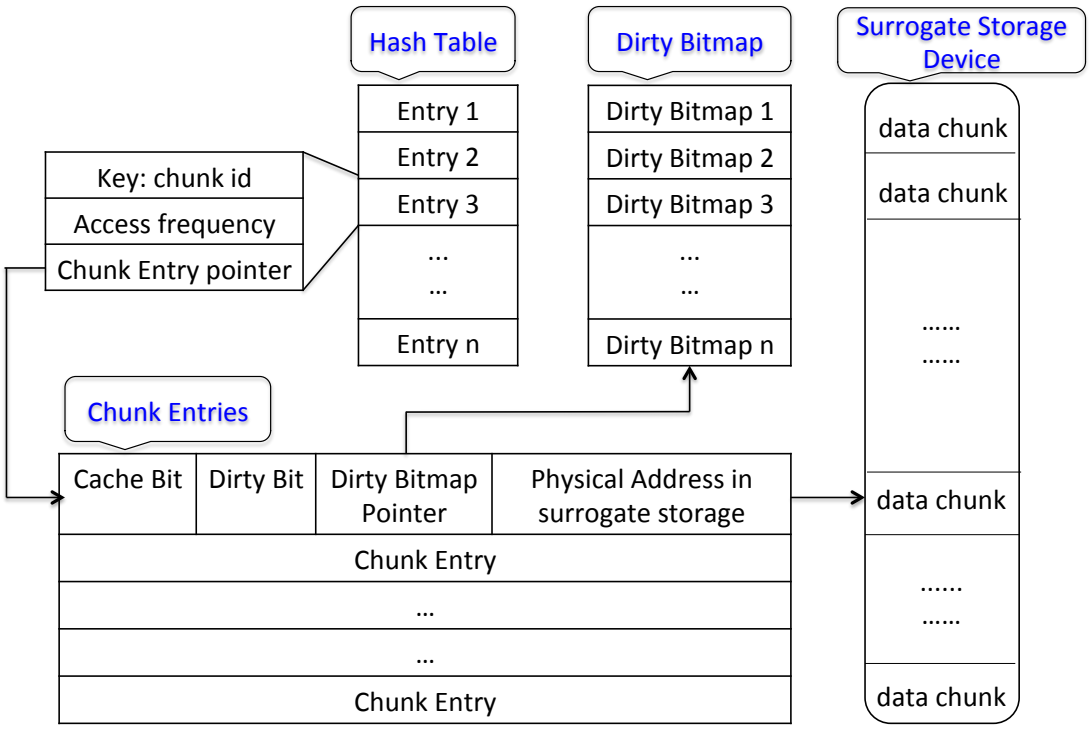


Figure 3.2: Data structure of the WAIO system

3.2.3 Data Consistency Issues

Data consistency is a key issue for the design of systems ranging from single node to large scale distributed systems [61, 62, 63]. In WAIO system, two aspects deserve to

consider carefully: 1) all the in-memory data structures must be safely stored, and 2) the data outsourced to the surrogate storage device must be reliably stored until they are reclaimed by WAIO.

First, in order to prevent unexpected data loss of the in-memory data structures, including the Hash Table, Chunk Entries and Dirty Bitmaps, WAIO stores them in a non-volatile RAM (NVRAM). The total size of these in-memory data structures is very small, so that it will not incur significant extra hardware cost. For instance, given a 1TB virtual disk image with 1MB chunk size, the space consumption in the worst case is only 25MB, when each chunk is outsourced to the surrogate storage device. In addition, the NVRAM is already widely deployed in the storage servers in the cloud data centers, for the purpose of system reliability and write performance improvement. Therefore, it is feasible to make use of the existing NVRAM to store these in-memory data structures.

Second, we can rely on the already built-in reliability mechanism (e.g. RAID, ECC, Replicas) of the surrogate storage device to protect the data outsourced from WAIO. Moreover, these resources are only necessary during the migration period. Once the VM live migration completes, they will be reclaimed and available to other services.

3.3 Performance Evaluation

We implement a lightweight prototype of WAIO in the QEMU-KVM system. Since WAIO is orthogonal and complementary to existing VM live storage migration approaches, we evaluate its effectiveness by conducting performance comparisons between DBT, a representative state-of-the-art approaches, and DBT enhanced with WAIO in different migration scenarios. The evaluations are driven by three real

world block level traces that represent different workload characteristics.

3.3.1 The Prototype Implementation

In the QEMU-KVM system, the IO functionality is provided by the QEMU system that is an open-source virtual machine emulator [64]. QEMU can emulate many virtual devices for virtual machines, including virtual disk drives and virtual network interface cards. The QEMU driver captures all the IO requests from VMs, and then passes them to the KVM kernel module. The KVM kernel module will dispatch these requests to the corresponding QEMU application and return the results to the VM. The QEMU application processes the IO requests on behalf of the VM [65]. WAIO is implemented in the QEMU application and therefore is able to capture the working set of the running VM.

When the VM live migration command is received at the hypervisor, WAIO is initiated and the in-memory Hash Table is created. WAIO redirects IO requests from both the hardware emulation layer (VM IOs) and the migration module (Migration IOs) in the *bdrv_co_readv_em* and *bdrv_co_writev_em* functions. The popular read data and newly written data are outsourced to the surrogate device (configured by WAIO administrator interface) and the in-memory Hash Table and chunk entries are updated at the same time. When the migration process completes, both the in-memory Hash Table and storage space on the surrogate device are reclaimed by the space reclaimer of the WAIO. WAIO is implemented in a lightweight manner, which only requires 638 lines of code added or modified in the QEMU application.

3.3.2 The Experimental Setup

The experimental platform consists of two servers as the source and the destination for the VM live storage migration. Each server is configured with an Intel Xeon X3440

processor, 8GB DDR memory and two 1TB hard drives, 12.10 Ubuntu system, QEMU 1.5.1 system with KVM enabled. The server in the source side of the migration has an 80GB SSD as the surrogate device. These two servers are connected by a 1Gbps Ethernet. The hardware information is described in details in Table 5.1.

Table 3.4: Hardware Specifications in Our Experimental Platform

CPU	Intel(R) Xeon(R) CPU, X3440@2.53GHz
MotherBoard	Winbond Electronics 0V52N7
Memory	8GB, AMI CMX8GX3M2A1333C9
Hard Drives	1TB Seagate ST31000524AS, SATA
SSD	80GB, Intel SSDSA2CW08
Network	1Gbps Ethernet

In order to measure the performance of WAIO, we migrate a running VM between the two servers. The VM is configured with 1 virtual CPU, 2GB memory, 1 virtual disk, 1 virtual network interface card and Ubuntu 12.10 system. During the migration, we replay block level traces and collect the IO performance within the VM. The Storage Performance Council [66] has published several block level traces for research purposes, and these traces have been widely employed to evaluate the storage system performance [67, 68, 69]. The Financial1 and Financial2 traces are collected from OLTP applications in a large financial institution and the WebSearch1 trace is collected from a web search engine. The key characteristics of these traces are summarized in Table 3.5. In our experiments, we implement a trace replay tool that will read the trace file and keep sending the IO requests to the VM's virtual disk. The IO throughput, as the VM IO performance metrics, are reported by the trace replay tool during the live storage migration. In a typical virtualized environment, there are multiple running VMs within a single server. In our experiments, there are two more running VMs in each server. One VM is running the fileserver benchmark and the other one is running the webServer benchmark.

Table 3.5: Trace characteristics

Trace Name	Read Ratio	IOPS	Avg. Req. Size (KB)
Financial1	32.8%	69	6.2
Financial2	82.4%	125	2.2
WebSearch1	100%	113	15.1

3.3.3 Trace Driven Evaluations

As indicated earlier, DBT is one of the fundamental approaches for the VM live storage migration. We integrate our WAIO into the DBT scheme and run experiments to evaluate the performance improvement. In this experiment, the VM is configured with 2GB memory, one 15GB virtual disk image attached with a virtio driver. The cache mode is set by default (writethrough) in the QEMU application, which enables the host page cache and disables the VM disk write cache. One spare SSD in the source server is employed as the surrogate device, and we set the migration speed of the running VM at 40MB/s. We replay the three traces during the VM live storage migration and set the number of outstanding IOs as one. The IO throughput within the VM is reported when the migration completes.

Figure 3.3 shows that, compared with DBT, WAIO increases the throughput by a factor up to 1.30, 2.61 and 11.10 for the Financial1, Financial2 and WebSearch1 traces respectively. The reasons why WAIO achieves significant improvement on IO throughput are threefold. First, most of the IO requests (more than 90% for all of the three traces) are outsourced to the surrogate SSD, so that they are not affected by the migration IO requests. The average request size is several KBs (see Table 3.5), while the chunk size is 1MB. By outsourcing a single chunk, the surrogate SSD can serve many incoming IO requests with a high possibility. This is why WAIO can outsource such an amount of IO requests to the surrogate SSD. Second, the surrogate SSD has better IO performance than the original hard disk. Even if we use hard

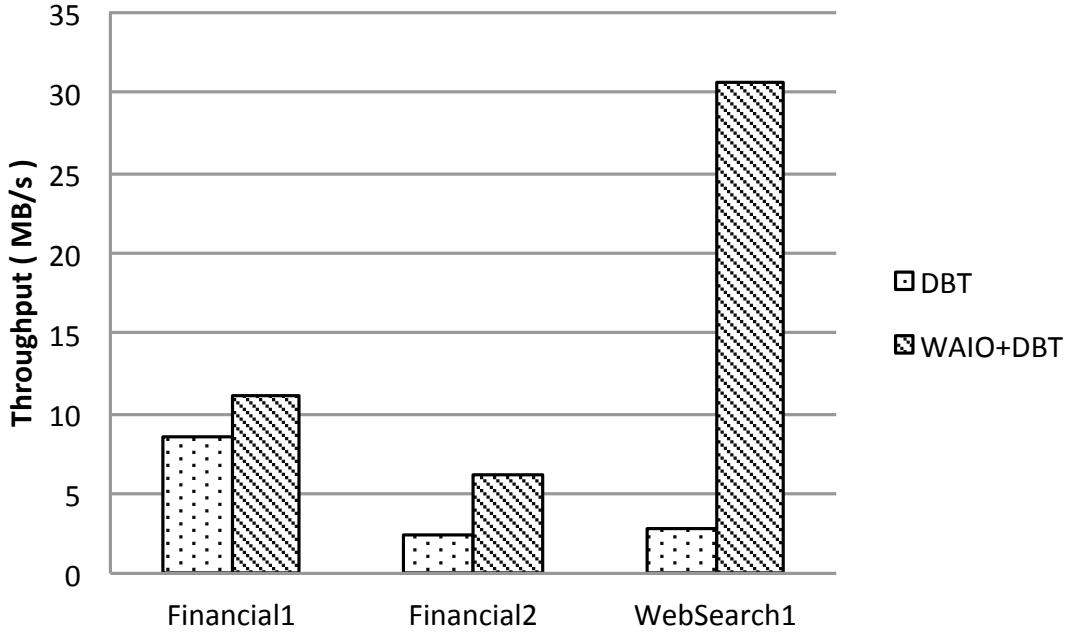


Figure 3.3: VM IO performance during the migration

drive as the surrogate device, WAIO can still improve the throughput to some extent. We will show this result in the sensitivity study. Third, since many IO requests are outsourced to the surrogate SSD, the IO queue in the original device is shortened accordingly, thus increasing the throughput of the remaining IO requests served by the original device.

In addition, WAIO only consumes 3.99%, 5.57% and 10.01% of its virtual disk space in the surrogate device for the Financial1, Financial2 and WebSearch1 traces respectively. Such storage space overhead in the surrogate storage device is negligible and will be reclaimed immediately when the live migration completes.

Figure 3.4 shows a comparison of the DBT and WAIO's average user response time every 10 seconds. In this test, the migrating VM has two virtual disk images: system disk image and data disk image, and these two virtual disk images reside in

different storage devices. The running application within the VM keeps read/write data from the data disk images. From the figure, we can see that the user response time performance of DBT and WAIO during the migration of the system disk is almost the same. During the migration of the data disk, it is clear that WAIO significantly improves user response time performance of the baseline DBT, by 91% on average.

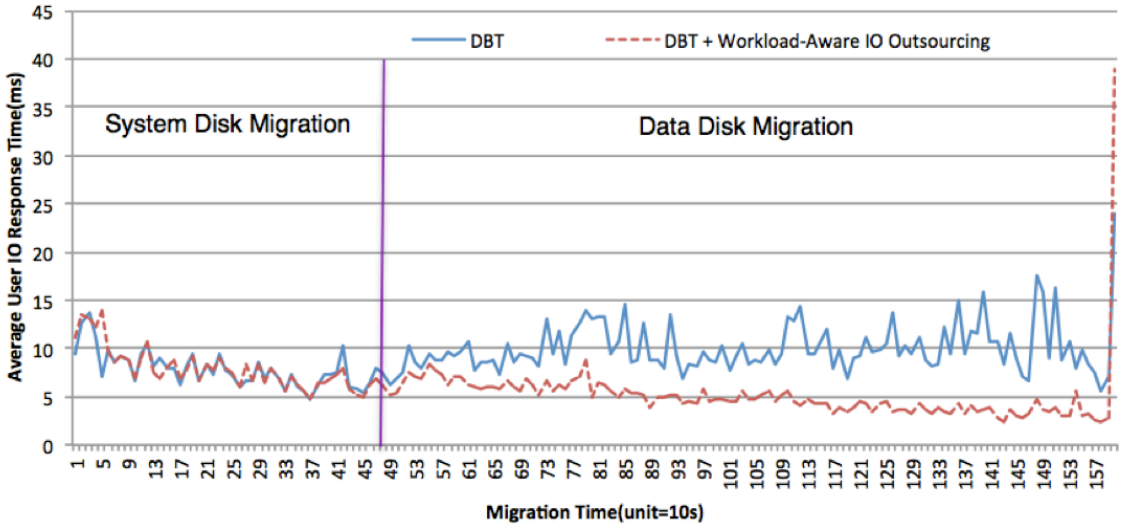


Figure 3.4: VM IO performance during the VM live storage migration with two virtual disk images

3.3.4 Sensitivity Study

WAIO's performance is likely influenced by several important factors, including surrogate device type, cache mode in the QEMU application, migration speed and the virtual disk image size. Due to the space limitation, we only discuss the sensitivity studies on the Financial2 trace. It also shows similar trends for the other traces.

Surrogate Device: To evaluate the impact of surrogate devices on the VM IO performance during migration, we employ an SSD and an HDD as the outsource target and conduct experiments that measure VM IO performance during migration. The

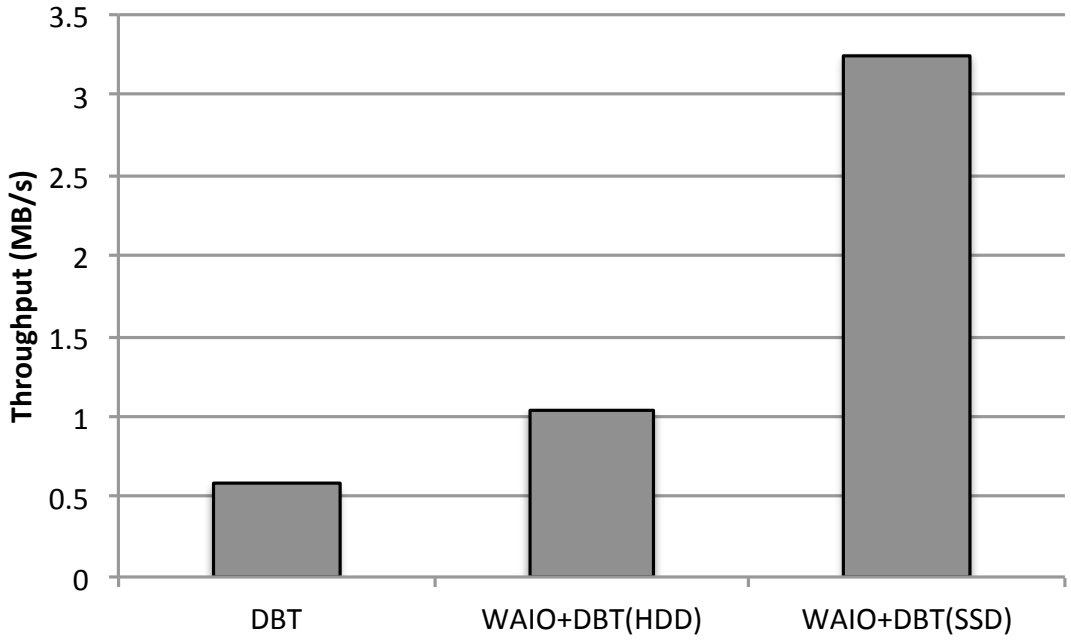


Figure 3.5: VM IO performance comparison under different surrogate devices

virtual disk image size is 15GB and the migration speed is 57MB/s. Figure 3.5 shows the evaluation result. When the spare HDD is used as the surrogate device, WAIO improves the VM IO throughput by 79.3% (from 0.58MB/s to 1.04MB/s). Only 528MB space overhead is introduced on the spare HDD. Moreover, the consumed space can be reclaimed after the live migration completes, about 6 minutes later in our experiments, driven by the Financial2 trace. The VM IO performance can be further improved by 213% with a spare SSD as the surrogate device. We can see that with an SSD-based surrogate device, the overall performance improvement will be better. The reason is that the SSD has better IO performance than the HDD. Given the wide deployment of hybrid storage system, it is preferable to employ a spare SSD as the surrogate device for WAIO.

VM Cache Mode: There are three modes of VM cache management in the

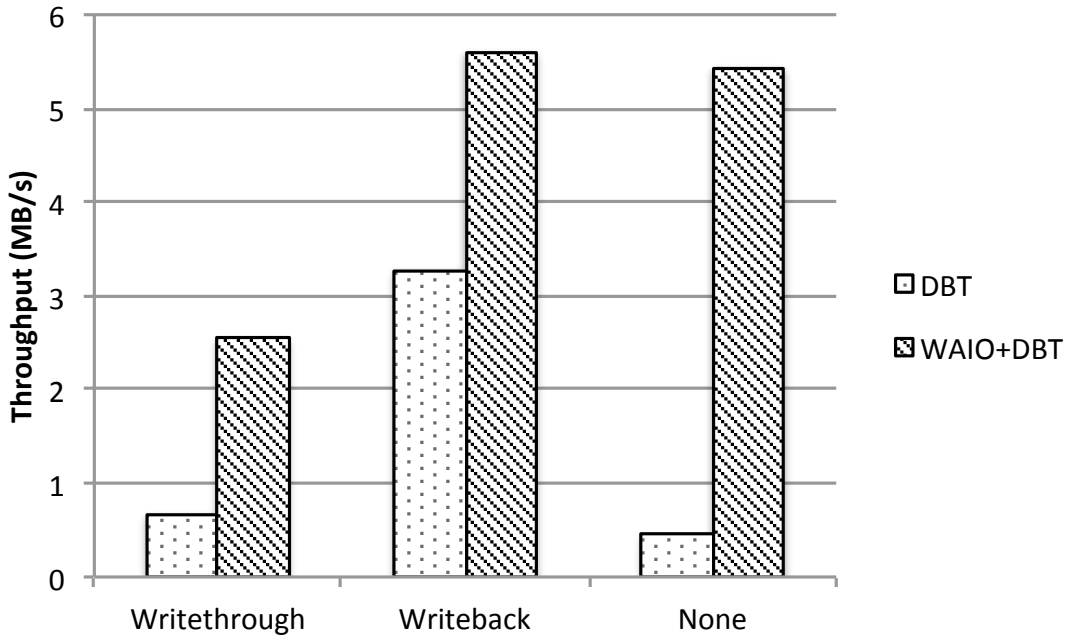


Figure 3.6: VM IO performance comparison under different cache modes

QEMU application: writethrough (default), writeback, and none. The writethrough mode enables the host page cache and disables the VM disk write cache. The writeback mode enables both the host page cache and the VM disk write cache. The none mode disables the host page cache and enables the VM disk write cache. Figure 3.6 shows the performance improvement of WAIO under all of the three cache modes. We can see that WAIO improves the VM IO performance during migration by a factor of 3.75, 1.71 and 11.83 for the writethrough mode, writeback mode and none mode, respectively. This result also demonstrates that there is still tremendous access locality observed at the physical block device level, even after the filtering of the two level buffer caches (VM cache and host cache). Thus, by making good use of such access locality, WAIO is able to improve the VM IO performance during migration. In the none cache mode, the cache space in memory is relatively small (only VM disk write

cache is enabled). In this scenario, WAIO is extremely useful in boosting the VM IO throughput, since it can capture the VM's working set more effectively.

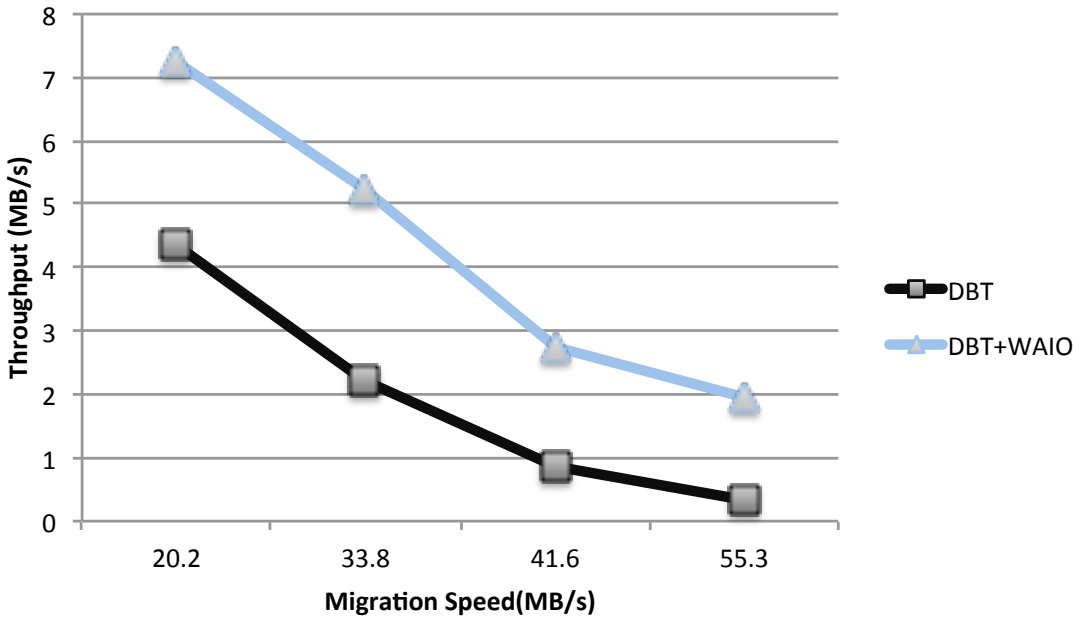


Figure 3.7: VM IO performance comparison under different migration speeds

Migration Speed: To examine the sensitivity of WAIO to the VM migration speed, we conduct experiments to migrate the VM at different migration speeds and measure the VM IO performance. The VM IO throughput is reported in Figure 3.7. We draw three key observations. First, the VM IO throughput drops significantly as the migration speed increases. This is because there are more migration IO requests waiting in the queue of original storage device at a higher migration speed, taking away more storage bandwidth available for the VM IO requests. Second, WAIO improves the VM IO performance by a factor of up to 6.09 at different migration speeds, compared with DBT. In WAIO, once the VM's working set is outsourced to the surrogate device, the VM IO requests are mainly served by the surrogate device.

As a result, their throughput is less affected by the increased migration IO requests. Third, WAIO provides more flexibility to the cloud-computing infrastructure. Compared with DBT, WAIO can either achieve similar VM IO throughput (5.25MB/s) with a higher migration speed (41.6MB/s over 33.8MB/s), or achieve higher VM IO throughput (5.25MB over 2.24MB/s) at a similar migration speed (33.8MB/s). System administrators can define different policies for live storage migration, based on the service level agreement and system management requirement.

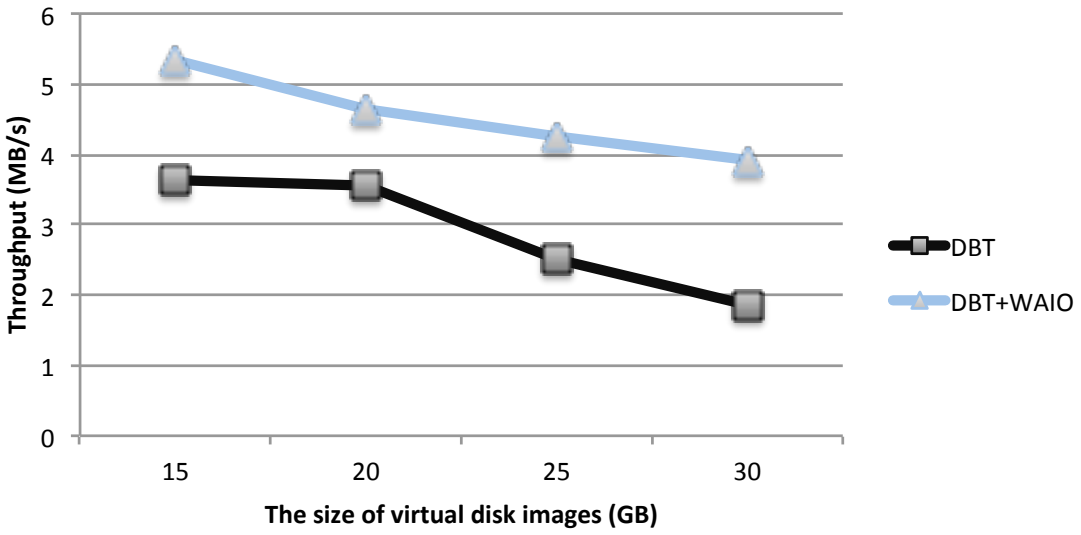


Figure 3.8: VM IO performance comparison with different virtual disk image sizes

Virtual Disk Image Size: To evaluate WAIO with different virtual disk image sizes, we create four VMs with different virtual disk images (i.e., 15GB, 20GB, 25GB, and 30GB), and then measure the VM IO performance during migration. Also, we scale the address space of the trace file to cover the address space of the virtual disk image in our experiment. Figure 3.8 shows that the VM IO throughput in both WAIO and DBT decreases as the virtual disk image size increases, due to the increased time-consuming disk seek operations. At the same time, WAIO outperforms DBT by a

factor up to 2.10, in terms of VM IO throughput during migration. Such improvement is mainly attributed to the reduced IO interference in the original device. In addition, since the outsourced data chunks are allocated sequentially in the surrogate device, WAIO is less sensitive to the virtual disk image sizes than the DBT mechanism. Therefore, WAIO becomes more effective than DBT in the VM live storage migration with larger virtual disk images.

Chapter 4

SnapMig: Snapshot-based VM Live Storage Migration

4.1 Background and Motivation

In this section, we provide the necessary background information for the SnapMig research, including VM snapshots, live migration of VM snapshots and VM snapshot backup, which then helps motivate this research.

In the production environment, VMs may encounter two types of failures: system crash and data loss. VM snapshots, which preserve the VM status at a previous time stamp, can be employed to roll back the VM to a previous consistent state before the system crash happened. Snapshot backup that transfers previous VM snapshots to backup servers is routinely leveraged to restore VM states when hypervisors hit data-loss failures [70, 71, 72]. Both of these two schemes are extensively deployed in the cloud industry products [73].

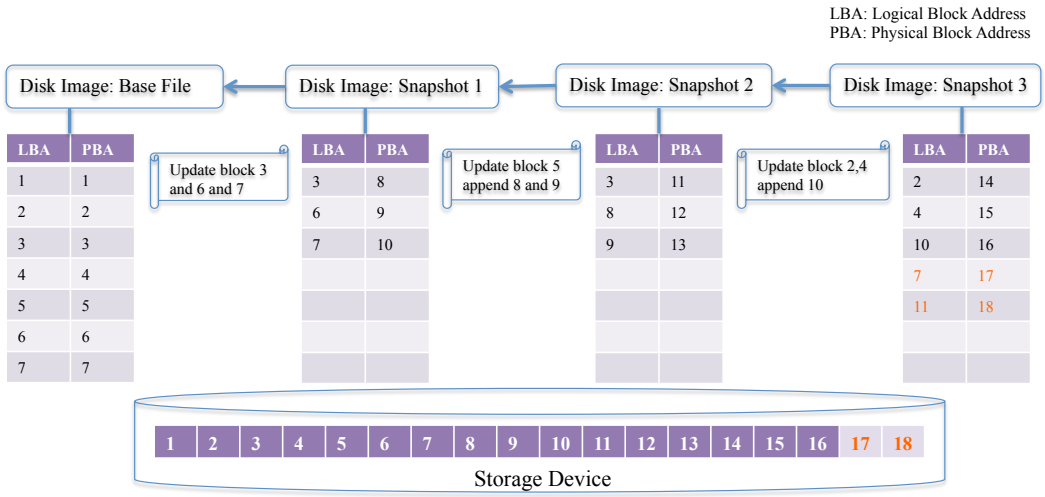
As indicated in previous chapters, the VM live storage migration is a nontrivial job. All the state information of the migrating VMs, including the memory footprint, the states of the virtual network, the virtual disk images and snapshot information [74], must be migrated to destination servers [75]. The total size of such state information is usually dozens of GB's, of which the largest part is the virtual disk images and snapshots that account for about 80% to 95%. In addition, as migrating VMs are running applications for customers during the migration period, VM states

keep changing in the migration period and all these updates must be migrated to the destination servers as well. More importantly, when multiple VMs live migrate at the same time, it is more challenging to migrate VMs quickly and provide reasonable IO performance for all the VMs simultaneously [51, 13].

During the VM live storage migration, additional migration threads are introduced in the source servers and they consume significant amount of storage resource. However, the overall system resource is not increased at all. The migration threads and VM threads interfere, instead of cooperate, with each other, which leads the performance of all these threads to degrade significantly [76]. There are several VM migration schemes proposed in the literature and industry products, such as Dirty Block Tracking [11], IOMirroring [11], workload-aware [12] and redunfdant data reduction [77, 78] approaches. However, all of them ignore the fact that migrating VMs' base images and previous snapshots are already in the backup servers (through the regular backup operations), and backup servers can help migrate this resource-hungry (i.e., network and storage bandwidth) information to the destination servers on behalf of the source servers. In this paper, *we argue that by leveraging the VM backup snapshots in the VM migration process, the overall migration efficiency can be improved significantly.*

Motivated by the observation of the VM snapshots-backup process and its resulting snapshots of VM state information available in the backup servers, we propose a novel VM live storage migration scheme, called *SnapMig*, to improve both the VM performance and migration performance simultaneously. By leveraging the backup servers to transfer migrating VMs' base images and previous snapshots, the source servers only need to migrate the latest VM state changes to the destination servers. Consequently, the otherwise severe interference between the I/O traffic generated by the user applications within the VMs (including the migrating VMs) in source

servers and the I/O traffic induced by the VM migration process is significantly reduced, leading to substantial performance improvements to both the VM threads and migration threads. In addition, After the migration, recent VM snapshots made available in the destination servers by the backup servers allow the users can roll back their VMs to previous states freely. Moreover, SnapMig is orthogonal to existing migration approaches, and it is regarded as a performance optimization layer for the existing migration approaches. Finally, we observe that the performance advantages of SnapMig become more pronounced with the concurrent live storage migrations of multiple VMs.



Requests	Step_1	Step_2	Step_3	Step_4	Step_5	Step_6
Read block 1	Search Snapshot 3	Search Snapshot 2	Search Snapshot 1	Search Base File	Get PBA 1	Read PBA 1 from Storage Device
Read block 6	Search Snapshot 3	Search Snapshot 2	Search Snapshot 1	Get PBA 9	Read PBA 9 from Storage Device	
Update block 7	Allocate block 17 in Storage Device	Write data to physical block 17	Add an entry to Snapshot 3(7->17)			
Append block 11	Allocate block 18 In Storage Device	Write data to physical block 18	Add an entry to Snapshot 3(11->18)			

Figure 4.1: The structure of VM snapshots and the workflow for read/write requests

4.1.1 VM Snapshot

For the VM reliability and availability purposes, VM snapshots are widely employed to restore VMs for customers upon system crash or data loss [70, 71, 79, 80, 74]. Currently most modern virtualization platforms, including VMware, HyperV, and KVM, support snapshots in their products.

There are two types of snapshot: disk snapshot and system snapshot. Disk snapshot retains the state of the corresponding VM's virtual disk image at a specific time stamp. Given a disk snapshot, the user can roll back the VM to a previous consistent state freely, but the VM needs to reboot and applications are required to restart. The creation process for a disk snapshot includes two phases: 1) flushing only the in-memory buffer cache data to virtual disks, and 2) taking a snapshot for each virtual disk. Therefore, there is negligible performance overhead for running applications within in the VM. Disk Snapshot is largely adopted for VM backup and disaster recovery. System snapshot contains the state information of the RAM and other virtual devices of the VM, besides virtual disk images. With the support of system snapshots, users can roll back the VM to a previous running state. Applications will be resumed at the last execution point, so that it is unnecessary to reboot the VM in the restore period. However, a system snapshot takes a longer time to create than a disk snapshot, since the state of the memory footprint and all virtual devices will be recorded in the snapshot. In addition, the running VM will be stunned during the creation phase, which will cause significant performance degradation for the running applications inside the VM [38]. System snapshots are mainly used to perform risky operations in the testing environment. In this work, we only focus on the disk snapshots, because they is widely used in the VM environment. Moreover, a scheme based on disk snapshots is also easy to be extended to be based on system

snapshots.

For the purpose of storage efficiency, every VM snapshot only records the state changes of the VM image made since the most recent snapshot in the implementation, assuming that it has the access to all the previous snapshots and the base image. As shown in Figure 4.1, each snapshot and the base image maintain a mapping table from the Logical Block Address (LBA) to Physical Block Address (PBA) as their metadata. Each table (except the one in the base image) records the updates and new writes since the most recent snapshot. For the update (block 7) / write (block 11) requests, new entries are added to the mapping table of snapshot 3 (the current snapshot). For the read requests (block 1 and 6), older snapshots (snapshots 1, 2 and 3) and/or the base image have been queried in order to get the latest version of the accessed data blocks. Normally, production servers hold several snapshots for each VM, so that the VM can be restored to any of the stored previous snapshots quickly upon system crash or data corruption. Besides the snapshot support in modern virtualization platforms, major storage vendors like EMC also provide storage optimization for VM snapshots [73].

4.1.2 VM Snapshot Migration

As indicated in the previous subsection, there are several existing snapshots for each VM created by the user or system automatically, and each snapshot holds the changes of the VM image since the last snapshot or base image (if this is the first snapshot). The size of each VM snapshot varies significantly, and it largely depends on the write traffic to the running VM. Take the Aliyun cluster use case [81, 14] as an example, the size of each VM is 40GB, and each VM snapshot is 8GB on average, which means 20% of the virtual disk images have been changed since the last snapshot. When it comes to the VM live migration, these snapshots will be either migrated to the destination

server (named FullMig) or discarded at source server (named SelectiveMig). If we migrate these snapshots to destination server (FullMig), the user can make use of these snapshots for VM restore at the destination server, as they did before the migration [38]. However, the total amount of data transmission increases and a longer migration time is unavoidable. If we discard the snapshots, and just migrate the current VM state to the destination server (SelectiveMig), the total amount of data transmission is much smaller than the FullMig option [82]. However, users can not roll back to any of the previous snapshots at the destination server.

Our experimental results show that the VM IO performance drops from $\frac{285}{65} = 4.38\times$ (reduction) to $\frac{279}{35} = 7.97\times$ (reduction), from migrating one VM to migrating 2 VMs, as indicated in figure 4.2. At the same time, the migration time increases for more than $3\times$ for both FullMig and SelectiveMig scheme, when we compare the 2 VMs concurrent migrations with single VM migration. Ideally, we would like a solution that can deliver faster VM live storage migration and preserve all the previous snapshots simultaneously. More importantly, we would like to reduce the traffic for the source server, as the performance degradation for the migrating/co-located VMs in the source server in the migration period is crucial for the overall system performance.

4.1.3 VM Snapshot Backup

For the VM reliability and availability purposes, there are many mechanisms employed to protect VMs, such as snapshots of the datastores via storage systems, replication of storage volumes/LUNs, snapshots of virtual machines and replication within virtualized applications [73]. Among all these mechanisms, the most widely employed one is VM snapshot backup. In the runtime, a series of VM snapshots are created, and then these snapshots will be transferred to backup servers in the backup window regularly. Within backup servers, these snapshots will be further processed to reduce

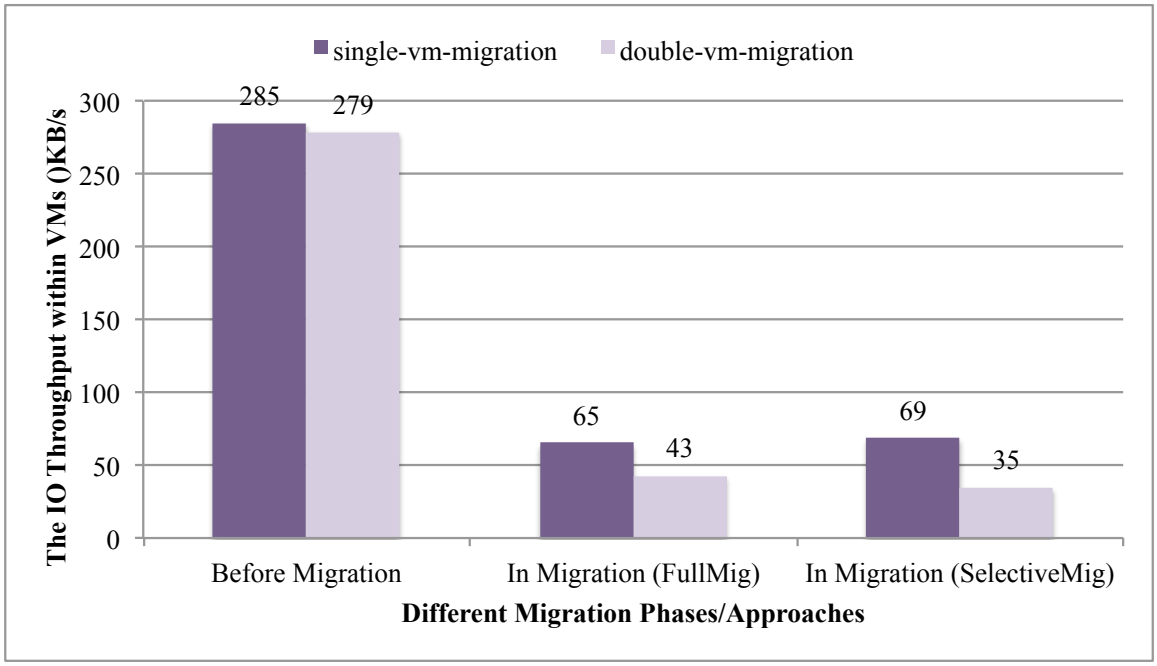


Figure 4.2: The VM IO performance comparison during the live storage migrations

the storage space consumption [83]. For instance, by employing the deduplication techniques, the redundant data blocks will be identified and removed. As indicated in Figure 4.3, for a particular VM (JohnVM), the production server holds several most recent snapshots (snapshots 5, 6 and 7) for a single VM, while the backup server holds all the previous VM snapshots (snapshots 1-6) and the base image (JohnVM.img) at the time of the last backup operation. The newly created snapshot (snapshot 7) after the last backup operation will only reside in the production server. The previous snapshots (1-4) are merged into the base image (JohnVM.img') in the production server. In this scenario, if the production server encounters any data loss or power outage issue, there is still an extra copy of the VM and its snapshots in the backup server. Another production server can resume the VM with the support from the backup server. When there is no data corruption occurring in the production server, the snapshots in the backup server will remain idle most of the time.

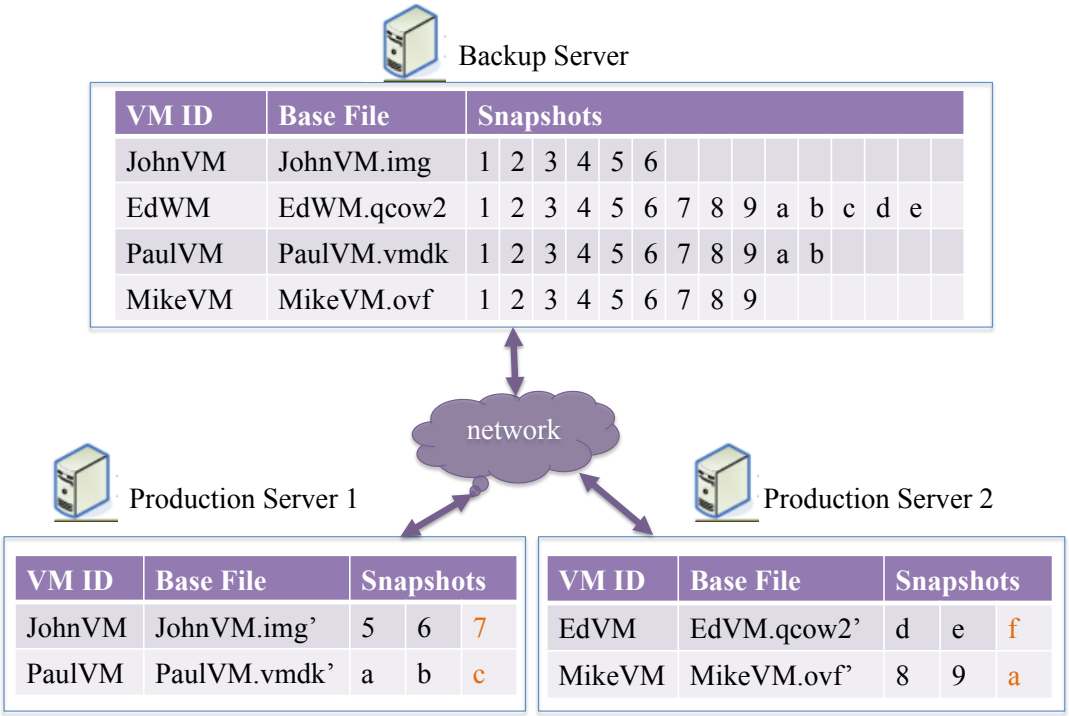


Figure 4.3: The distribution of VM snapshots among production servers and the backup server

4.1.4 Motivation

During the VM live storage migration, the source server will be quite busy, e.g., executing the scheduled maintenance task, running many co-located VMs, or waiting to shutdown soon. In general, whenever VM live storage migration is invoked, IO-intensive migration threads will be introduced to the source server. In order to achieve satisfactory VM IO performances for all the migrating/co-located VMs in the source server and migrate VMs to the destination servers quickly, it is vital to eliminate the unnecessary IO traffic to in the source server. Motivated by the observation

and analysis above about the idle snapshots in the backup server(s), we propose the SnapMig scheme that can achieve these goals by leveraging these idle snapshots in the backup servers. In the SnapMig scheme, backup servers transfer migrating VMs' base images and previous snapshots to the destination server(s), while the source server only migrates the latest changes to the migrating VMs. The migrating VMs will be resumed once their states are reconstructed successfully in the destination server(s). The benefits of SnapMig are fourfold: 1) Better VM IO performance during the migration, because the much reduced, if not completely eliminated, migration traffic involved in the source server allows the migrating/co-located VMs to achieve much better IO performance, as if there were no migration at all for most of the time; 2) Shorter migration time, because the backup server(s) that are idle most of the time can transfer the VM images to the destination server(s) at a much higher speed than any source server that has a very heavy running workload; 3) All the previous snapshots are available in destination server(s), so that the users can freely roll back VMs to any of the previous states; and 4) The performance improvement will be much more pronounced in the scenarios where multiple VMs are live migrated concurrently.

4.2 System Design and Implementation

In this section, we present the design and implementation of the proposed SnapMig by introducing the SnapMig architecture, its key functional modules and workflow.

4.2.1 SnapMig Architecture

Figure 4.4 shows the architecture overview of the distributed virtualization system that includes three parts: **management clients**, **production servers** and **backup servers**, connected by a high speed network. **Management clients** provide a console

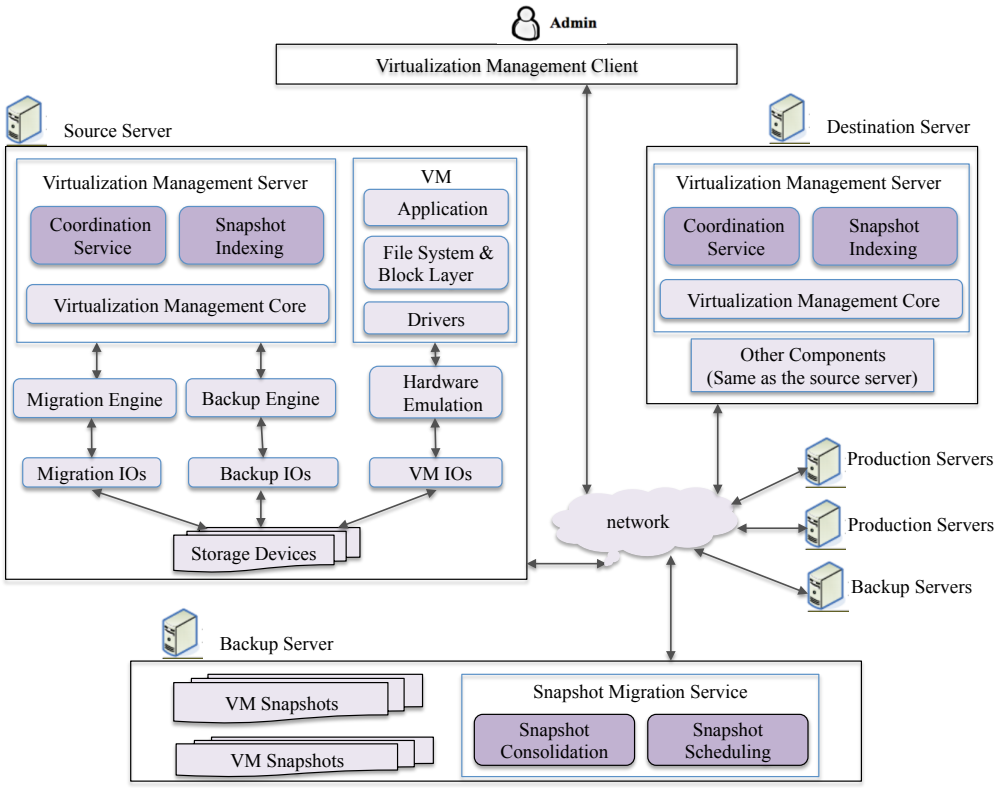


Figure 4.4: The architecture of SnapMig System.

for system administrators to perform various management jobs, such as VM creation, VM snapshot backup and VM live migration. They can reside anywhere as long as the network connection to other servers is available. **Production servers** host many live VMs and perform jobs as requested by the management clients. There are two layers in the production servers: virtualization management server (VMS) and the hypervisor module. The VMS listens to the incoming requests from the management clients and perform jobs by calling the corresponding functionality within the hypervisor module. By design, the VMS, such as open source libvirt [84], can support most modern hypervisors. **Backup servers** store VM snapshots from the

production servers through regular backup operations, and they are usually equipped with various functionalities for the purpose of reliability and space efficiency, such as redundant data elimination and data compression [83].

The SnapMig scheme is integrated into this cluster and it consists of four functional modules. **Snapshot Indexing** and **Coordination Service** are within the VMS component in the production servers, while **Snapshot Consolidation** and **Snapshot Scheduling** are in the backup servers, as shown in Figure 4.4. The responsibilities of these four modules are elaborated below.

Snapshot Indexing tracks the distribution and placement of VM snapshots among the backup servers. There may be multiple copies of a single VM snapshot in several backup servers for increased reliability, especially for VMs with higher priorities. In addition, VM snapshots may be migrated among the backup servers for the purposes of load balancing and reliability. Once the VM live migration starts, this module will query backup servers for the distribution and placement of the current snapshots of the migrating VM and pass it to the Coordination Service module.

Coordination Service controls the VM live migration workflow. With the snapshots distribution and placement information from Snapshot Indexing, Coordinate Service will instruct the corresponding backup servers to migrate VM base image and previous snapshots to the destination server. Once these backup servers finish the transmission, this module will invoke the native migration engine within the source server, which will live migrate the latest snapshots and in-memory state to the destination server. Meanwhile, this module will re-configure the VM and its snapshots in the destination server. Finally, it will log the overall migration progress, so that the migration can be resumed upon migration failures.

Snapshot Consolidation is designed to eliminate the unnecessary data transmission from the backup servers to the destination servers. It merges some older

snapshots to the base image or a single snapshot, before migrating them to the destination server. As shown in Table 4.1, snapshots 1-4 are unnecessary for the destination server, it would be better to merge them to the base image before the snapshot migration.

Snapshot Scheduling is an advanced feature designed for the further reduction of the total migration time. With the analysis of the migrating VM's working set, the sequence of blocks in the base image and snapshots can be scheduled, so that "hot" blocks (i.e., frequently accessed) that are within the VM's working set are migrated before others. Once these hot blocks are ready in the destination server, the source server starts the live migration of the latest state changes, and then the VM can be restarted in the destination before all the blocks are migrated to the destination server. Although this module is under implementation and thus not shown in the evaluation results of Section 4, we expect it will further reduce the total migration time significantly.

The design of SnapMig is quite flexible. First, it supports all kinds of modern hypervisors, as SnapMig communicates with hypervisors through the standard Virtualization Management API. Second, the SnapMig scheme is orthogonal to most state-of-the-art VM live storage migration approaches included in the Migration Engine in Figure 4.4, and it can complement with these existing approaches to further improve the VM live storage migration performance. Finally, the SnapMig scheme is scalable to the architecture of the cluster. Backup servers and production servers can be freely added to or removed from the cluster in the runtime.

4.2.2 SnapMig Workflow

Figure 4.5 shows the workflow of the VM live storage migration in the SnapMig scheme by way of an example. In this example, the migrating VM has a single

Table 4.1: An example of VM snapshots distribution and placement during the migration process

Server Name	VM ID	Base File	Snapshots					
Source Server	JohnVM	JohnVM.img'	5	6	7			
Destination Server	JohnVM	JohnVM.img'	5	6	7			
Backup Servers	JohnVM	JohnVM.img	1	2	3	4	5	6

disk image (named JohnVM.img) and 7 disk snapshots (1-7) in total, as indicated in Table 4.1. The snapshot 7 is created after the last daily backup operation, so it only resides in the source server. The base image and all other snapshots, 1-6, are already stored in the backup servers through the regular backup operations. There are several copies for the base image and some important snapshots in the backup servers for the reliability purpose. In order to save storage space in the source server and allow users to roll back to recent snapshots, the older snapshots (1-4) are merged to the base image. Only newer snapshots (5-7) are retained in the source server.

As indicated in Section 4.1.2, FullMig and SelectiveMig are the state-of-the-art migration approaches. In the FullMig approach, the VM base image and snapshots 5-7 are migrated to the destination server first, and then a conventional migration approach within the migration engine, such as Dirty Block Tracking or IOMirroring [11], is called to migrate the in-memory state and the latest VM state changes. Different from the FullMig approach, SelectiveMig only migrates the latest virtual disk image to the destination server and discards all the existing snapshots, such as snapshots 5-7. For instance, if a specific data block is updated both in snapshot 5 and snapshot 7, SelectiveMig only migrates the latest version of this data block (from snapshot 7).

The workflow of our SnapMig scheme is as follows. At the beginning of the migration, the Coordination Service queries the Snapshot Indexing for the latest distribution and placement of the migrating VM's snapshots. Then it will notify

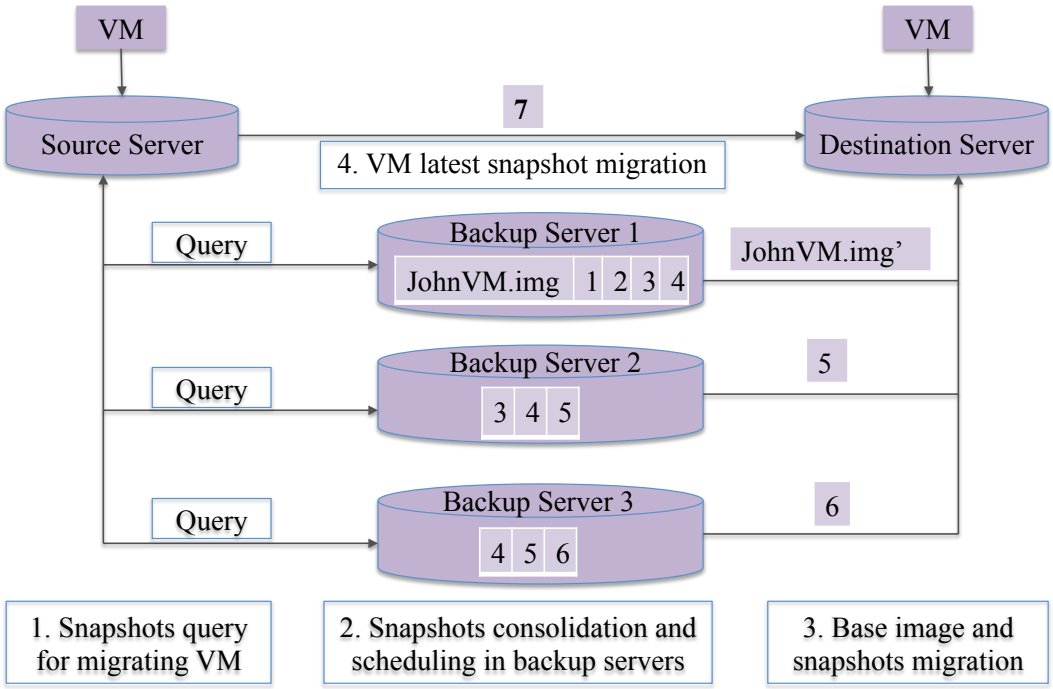


Figure 4.5: The workflow of the SnapMig scheme.

the corresponding backup servers to start the base image and snapshots migration. These backup servers will consolidate all the unnecessary snapshots first and start the migration to the destination server. Once the base image and snapshots are available in the destination server, the source server will migrate the latest state changes and the in-memory state to the destination. The VM will be resumed in the destination server finally.

Compared with FullMig and SelectiveMig, SnapMig introduces negligible migration traffic in the source server, including only the read requests of storage system and network transmission. For instance, if a specific data block is updated in the

base image and snapshots 5-6, SnapMig does not need to migrate this data block to the destination, since this data block will be migrated by the backup servers. Such significant traffic elimination in the source server will improve the overall system performance in several aspects: shorter migration time, better VM performance, and better multiple VM migrations, which will be evaluated in Section **??**.

4.3 Performance evaluation

In this section we present a detailed evaluation of our SnapMig scheme in comparison to two state-of-the-art VM migration schemes, FullMig and SelectiveMig. We focus on two key measures of VM migration efficiency, the VM migration performance (migration time) and VM IO performance (IO throughput of user applications within migrating and co-located VMs in the source server), under different configurations (e.g., single vs. multiple co-located VMs in the source server, single vs. multiple migrating VMs).

4.3.1 The Experimental Environment

In order to evaluate the performance of our SnapMig scheme, we implement a light-weight prototype of SnapMig in a cluster to conduct VM live storage migration experiments. The cluster consists of three servers, a source server, a destination server and a backup server. Each server is configured with an Intel Xeon X3440 processor, 8GB DDR memory and two 1TB hard drives, 12.04 Ubuntu system, QEMU 2.4.50 system with KVM enabled and libvirt 1.2.20. The SnapMig prototype is embedded in the libvirt platform [84]. In the source and destination servers, the host system and software are installed in one disk drive and all the VM virtual disk images are stored in a hardware RAID set. The backup server only stores and transfers VM virtual

disk images and snapshots. These three servers are connected by a 1Gbps Ethernet. The hardware information is described in details in Table 5.1.

Table 4.2: Hardware Specifications in Our Experimental Platform

CPU	Intel(R) Xeon(R) CPU, X3440@2.53GHz
MotherBoard	Winbond Electronics 0V52N7
Memory	8GB, AMI CMX8GX3M2A1333C9
Hard Drives	1TB Seagate ST31000524AS, SATA
RAID	4*160GB, RocketRaid 2240
Network	1Gbps Ethernet

4.3.2 Performance Metrics and Experimental Setup

From the user's perspective, the performance of the running VMs, including migrating VMs and co-located VMs, should not be affected by the migration process, at least to the extend of not violating the Service Level Agreement (SLA). Meanwhile, from the cloud service provider's perspective, the resource consumption for the live storage migration should be minimized, for the purposes of the overall performance and energy efficiency. For instance, to meet the SLA of user applications running on the migrating VM and/or co-located VMs in the source server, the migration should be completed within a reasonable time period and consume reasonable amount of network/storage bandwidth of the source server. In this work, we focus on the following metrics to compare SnapMig system with the state-of-the-art solutions: 1) the IO performance of migrating VMs, 2) the IO performance of co-located VMs, 3) the migration time. We compare the performance of our SnapMig scheme with two state-of-arts schemes (FullMig and SelectiveMig) in different migration scenarios.

Two common VM migration scenarios are evaluated in our experiments: migration of a single VM and simultaneously migrations of multiple VMs. Each VM is created with 1 virtual CPU, 2GB memory, 20GB disk image, 1 virtual network interface

and Ubuntu 12.10 system. There are a number of disk snapshots for each VM, and each snapshot contains some updated data blocks to the VM since its last snapshot. The base image and older snapshots resident in the backup server through the daily backup operations, while newer snapshots sit in the source server only. During the live storage migration, IO requests are issued from the Fio tool [85] within both migrating VMs and co-located VMs. The performance of the three migration schemes (FullMig, SelectiveMig, SnapMig) are compared under different user workloads generated by the Fio benchmark.

4.3.3 Results Analysis

4.3.3.1 Migration of A Single VM

In this scenario, there is only one VM (mig-vm-1) migrating out of the source server, and the other three co-located VMs (co-located-vm-1, co-located-vm-2, co-located-vm-3) are running in the source server. As indicated in Figure 4.6, the IO throughput of the migrating VM in SnapMig is about 4.61× higher than FullMig and SelectiveMig. At the same time, the IO throughput of the co-located VMs under SnapMig is also significantly higher that that under FullMig and SelectiveMig, by about 4.33×, as shown in Figure 4.6. Furthermore and importantly, the total migration time of the SnapMig scheme is significantly reduced from that of the FullMig and SelectiveMig schemes, from about 619 seconds to 313 seconds, as shown in Figure 4.8. From these results we have the following observations: 1) Once we leverage the backup server to migrate the bulk of the VM images, the pressure for the storage device in the source server drops significantly. Therefore, both the migrating VMs and co-located VMs can have better IO performance, as if there were no migration at all. 2) Since the backup server has virtually no running workloads outside of its backup windows, thus

much less busy than the source server, the VM images can be migrated much faster than from the source server.

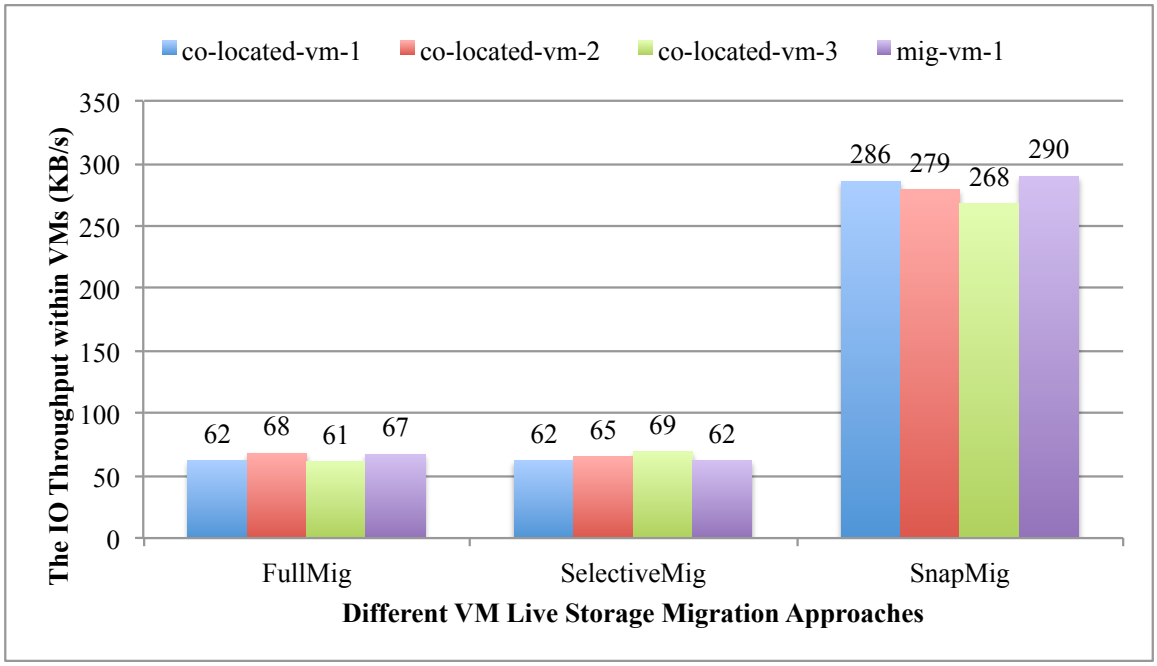


Figure 4.6: The VM Performance Comparison in Single VM Migration

4.3.3.2 Simultaneous Migrations of Multiple VMs

In this VM migration scenario, two VMs (mig-vm-1 and mig-vm-2) are migrating from the source server to the same destination server at the same time. As shown in Figures 4.7 and 4.8, the throughput of all the four VMs in our SnapMig scheme, two migrating VMs and two co-located VMs are about 8× higher than that of these VMs under the FullMig and SelectiveMig schemes. Similar to but much more pronounced than the case of single VM migration, the total migration time of SnapMig is drastically reduced that of FullMig and SelectiveMig, from 2028 seconds to 694 seconds. From these results, we notice that SnapMig's performance advantages, in both the VM performance and migration performance, become more pronounced as more

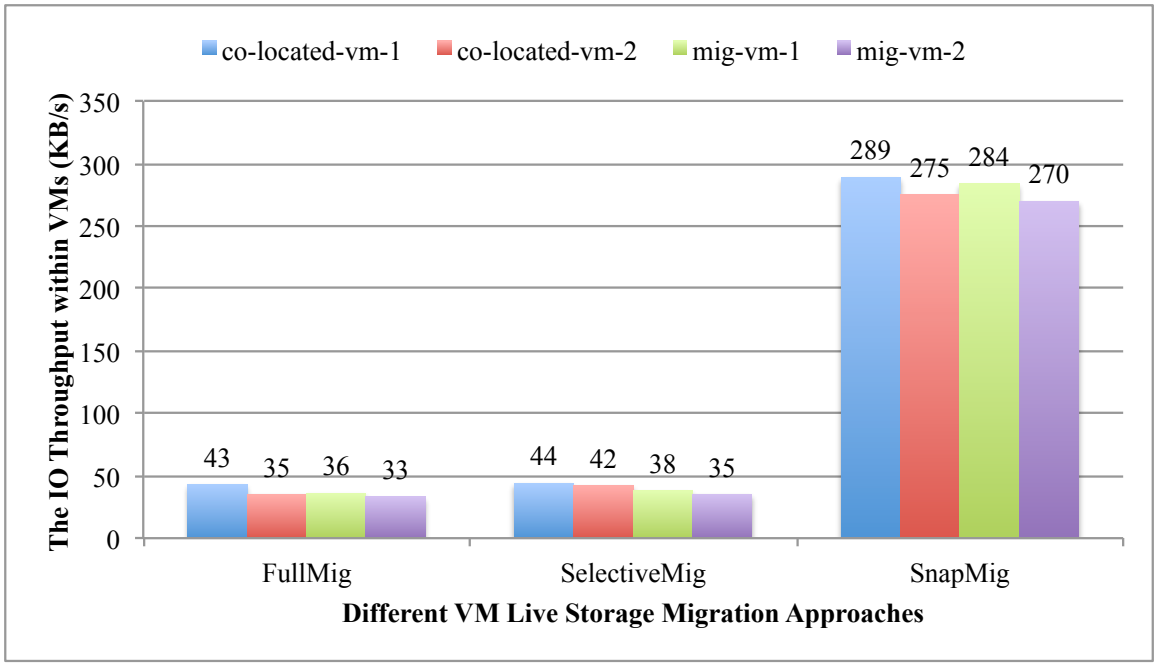


Figure 4.7: The VM Performance Comparison in Multiple VM Migrations

VMs are being migrated simultaneously. For example, its IO throughput advantage over FullMig/SelectiveMig increases from 4.61× to 8× and migration time advantage over FullMig/SelectiveMig increases from $\frac{663}{351} = 1.89\times$ (reduction) to $\frac{2028}{694} = 2.92\times$ (reduction), from migrating one VM to migrating 2 VMs. The main reason is that, with more migrating VMs, there will be more hungry migrating threads competing for the same resources in the source server, which leads to more severe interference between application IO traffic and migration IO traffic in the source server in both the FullMig and SelectiveMig schemes and results in more serious degradation of both IO throughput and migration time. The SnapMig scheme, in contrast, almost completely avoids this traffic interference in the source server since the bulk of the migration traffic is diverted or outsourced to the backup server. In other words, while both FullMig and SelectiveMig are very sensitive to the increase in the number of migrating VMs, SnapMig is relatively insensitive to such increase.

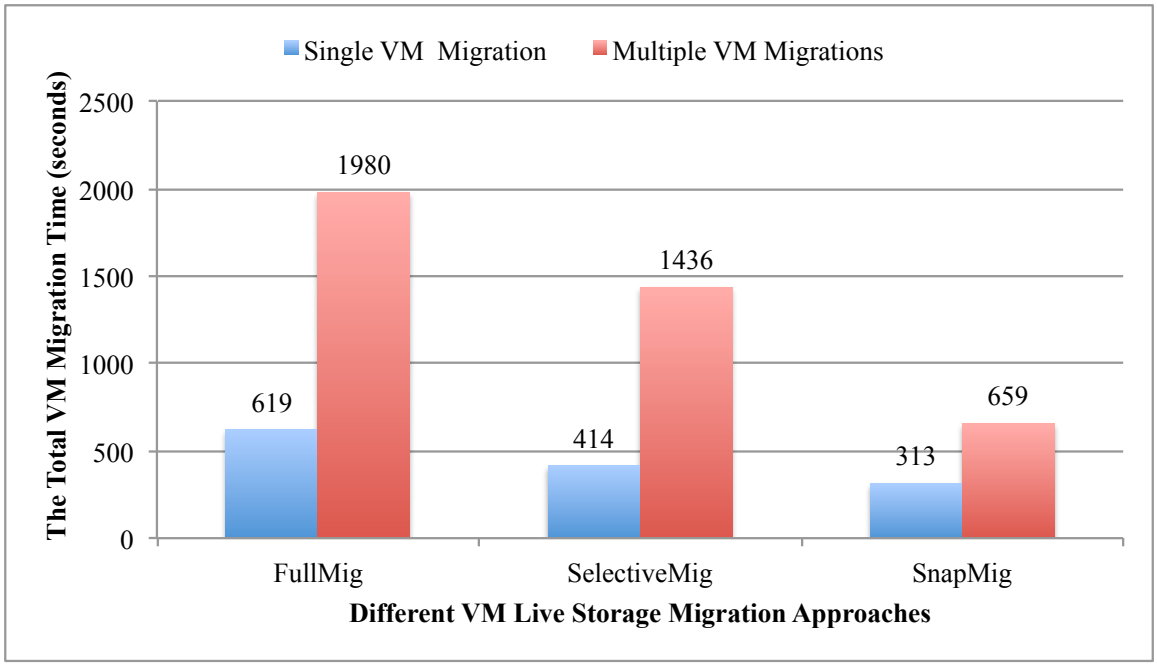


Figure 4.8: The Total Migration Time in Different VM Migration Approaches

4.3.4 Sensitivity Studies

In order to investigate how the number of co-located VMs and the VM workload characteristics affect the performance of the SnapMig system, we conduct a single-VM migration experiment with the number of co-located VMs increasing from 1 to 7. In addition, each VM runs two types of workload: ReadOnly workload and ReadWrite workload during the migration. Due to the space limitation, we only discuss the comparison between SnapMig and SelectiveMig schemes. It also shows similar trends for the comparison between SnapMig and FullMig schemes. The performance results, normalized to that of a single VM running in the source server without any VM migration, are shown in figure 4.9, 4.10 and 4.11. From these results, we draw the following observations: 1) As the number of running VMs increases, the average VM throughput decreases. Since the overall storage resource is fixed, the more running

VMs are involved, the less storage resource will be available to each VM. 2) In both workloads, the average VM IO performance under the SnapMig migration is very close to that in the No Migration scenario. However, the average VM performance drops significantly in the SelectiveMig scenario. 3) The migration time in the SnapMig scenario is less sensitive to the number of running VMs, while that in the SelectiveMig scenario soars as the number of running VMs increase. This further confirms that SnapMig introduces negligible extra traffic to the source server.

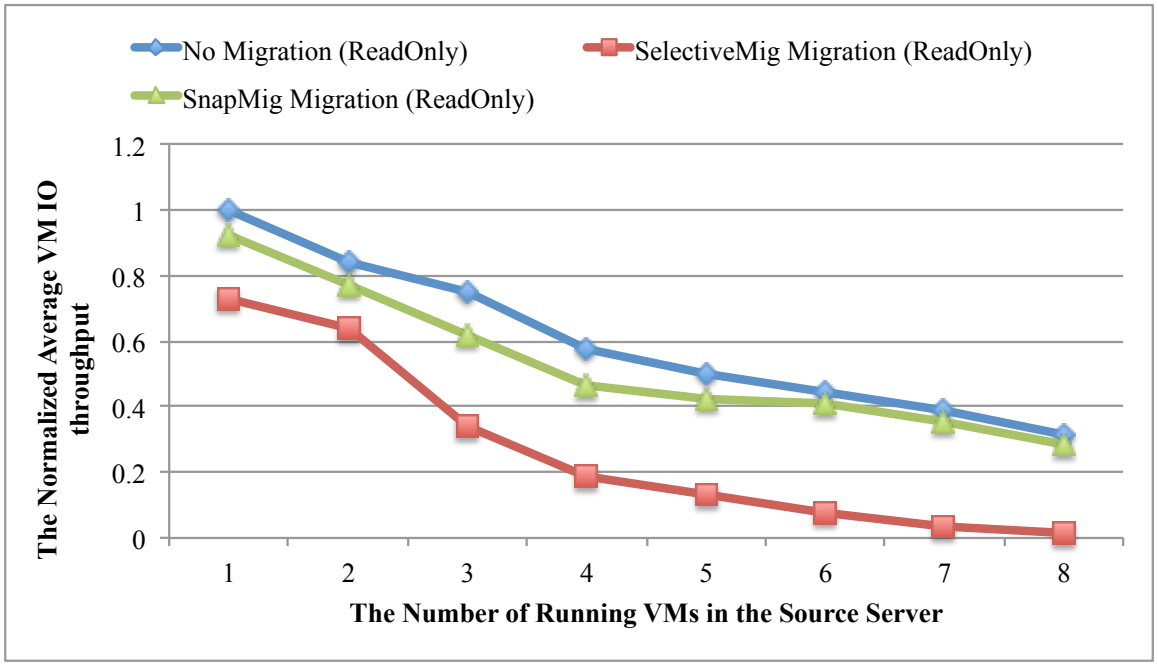


Figure 4.9: The VM IO Throughput under different Number of Running VMs (Read-Only Workload)

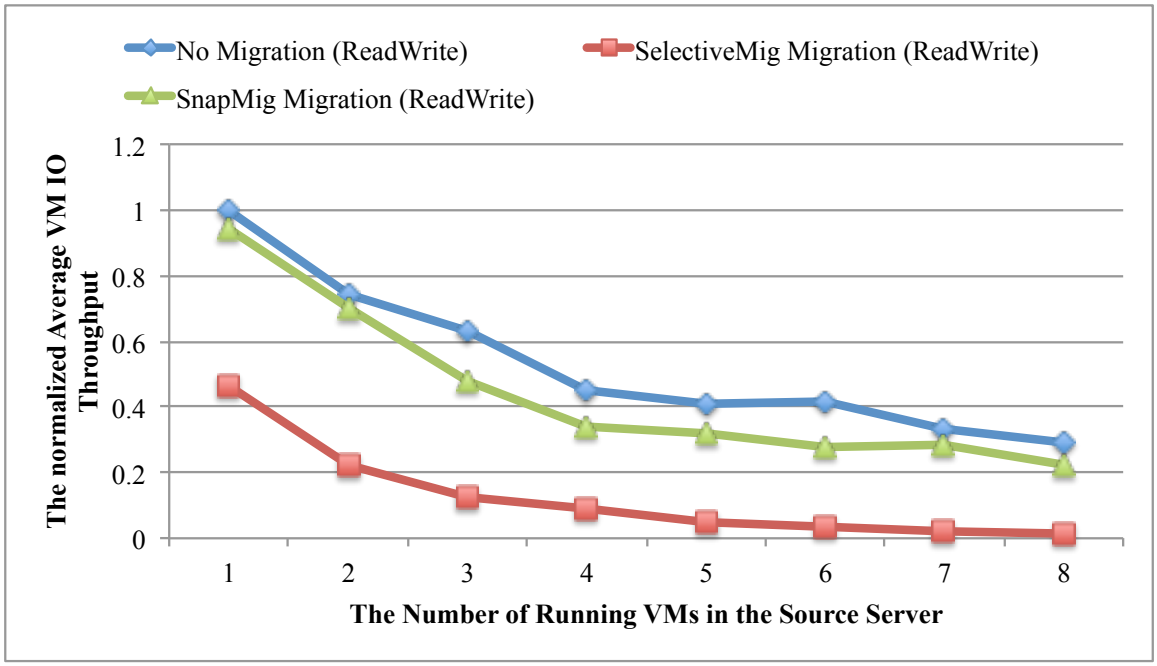


Figure 4.10: The VM IO Throughput under different Number of Running VMs (Read-Write Workload)

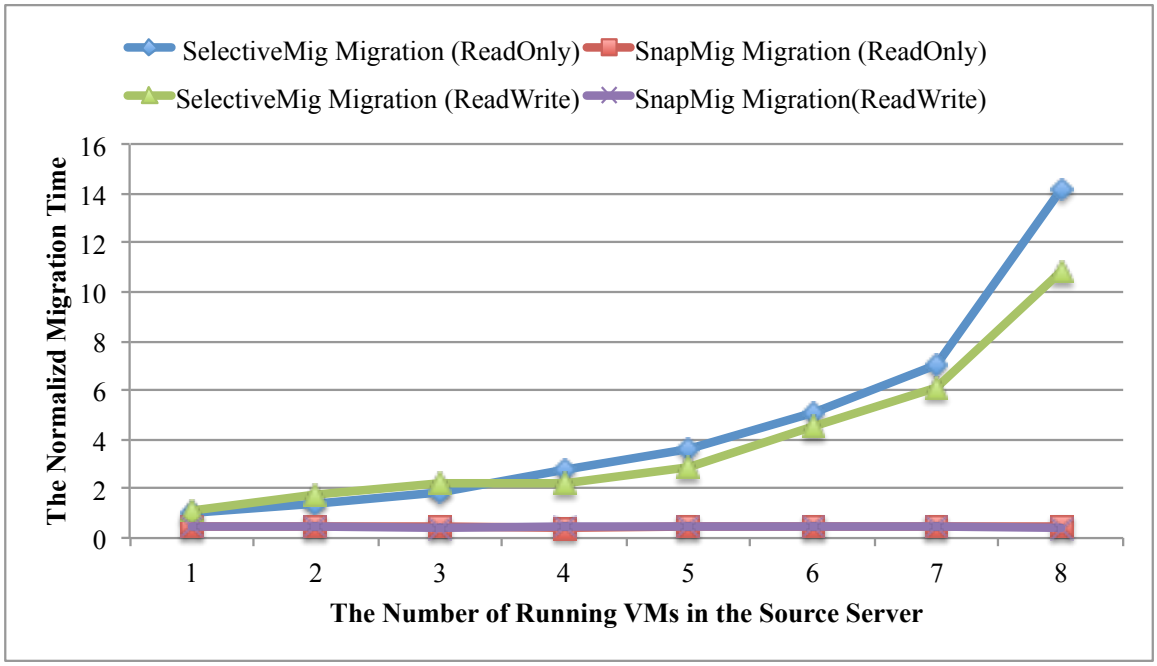


Figure 4.11: The VM Migration Speed under different Number of Running VMs

Chapter 5

IOFollow: Improving the VM Live Storage Migration by IO Following

5.1 Background and Motivation

In this section, we provide the necessary background information for the IOFollow research, including Sequential IO property and threads model, which then helps motivate this research.

5.1.1 Sequential IO Property

Sequential IO property has been one of the most fundamental concepts in system research area [86], mainly because of the performance disparity between Sequential IOs and Random IOs in storage systems. Over the past few decades, the data transfer bandwidth has increased a great deal, due to the more density of bits in the surface of a disk drive. However, the costs of seek and rotation delay have reduced slowly, since it's much more challenging to speed up the mechanical movements of disk head and the spinning speed of the platters. Therefore, the performance of Sequential IOs is much better than that of Random IOs with frequent disk seeks and rotations [87].

The Sequential IO property is determined by many metrics from both Spatial and Temporal dimensions [88, 89, 90, 91, 86]:

- **Spatial Dimension:**

  - **Consecutive Addresses:** the difference between the Logical Block Addresses (LBA) of consecutive IO requests is within a predefined threshold. It could be further classified as *Strictly consecutive* (no gap between the LBAs of consecutive IO requests) and *Strided access* (bounded gap between the LBAs of consecutive IO requests).
  - **Consecutive Bytes Accessed:** the size of data to read or write in an IO request on average.

- **Temporal Dimension:**

  - **Interleaved Streams:** the mixture of IO requests from multiple threads, applications or VMs. The Sequential IO property for individual IO streams may be affected by the interleaving with other IO streams. For instance, two consecutive IO requests (request 1 and 2) with consecutive addresses from stream A may experience poor IO performance, as a long disk head movement involved during which the disk serves another IO request (request 3) from stream B and the LBA of request 3 is far away from the LBA of either request 1 or 2.
  - **Inter-arrival Time:** the time interval between consecutive IO requests. Once there is a long waiting time between two IO requests, some other background IO requests may be issued in between, which will influence the Sequential IO property of the original IO stream.

The interleaving of multiple IO streams will not only affect the Sequential IO property, but also decrease the IO performance significantly for each participating stream. The experimental results from Xing *et al.* [9] indicate that a Random Write IO stream is destructive to all other kinds of interleaving IO streams, such as Sequential

Read / Write stream. The performance of the Random Write IO stream itself will also be degraded significantly. More use cases of performance degradation for multiple interleaving IO streams are presented in their work [9].

Given the nature of disk access characteristics and Sequential IO property of interleaving IO streams, modern file systems, both local file systems [92, 93, 94, 95, 96] and distributed file systems [97, 98], improve the IO performance by aggressively sending sequential IO requests to underlying disk drives. For instance, in the Log-based File system [92], a sufficient number of updates are buffered in memory before they are sent to disks as a large sequential segment request, so that the disk throughput is improved significantly compared with individual small random write requests. Read requests are served by a similar manner, which will read a whole segment from disks at once. In the Google File System(GFS) [98], the minimal size for each request is 64KB by default, so that high IO throughput in disks can be achieved with sequential IO requests. Unfortunately, file systems can only affect, but not determine the Sequential IO property of IO streams that are produced by user applications themselves. Many optimization techniques have been invented to improve the Sequential IO property by leveraging the semantic hints from the applications [99, 100].

5.1.2 Threads Model in Virtualized Systems

In the virtualized environment, the IO stream at the gate of the storage system is a multi-layer interleaving of individual IO streams: First, an application-level IO stream is produced with the interleaving of multiple thread-level IO streams within the same application. Second, for each running VM, one VM-level IO stream is generated with the interleaving of multiple application-level IO streams and the VM Operating system IO stream. Finally, the interleaving of all VM-level IO streams and the hypervisor IO stream will become the final IO stream for the storage system. Considering

all the metrics related to the Sequential IO property, VM-level IO Streams (named as VM IOs) are mostly determined by user applications and guest Operating Systems. Figure 5.1 shows the IO threads model of virtualization systems.

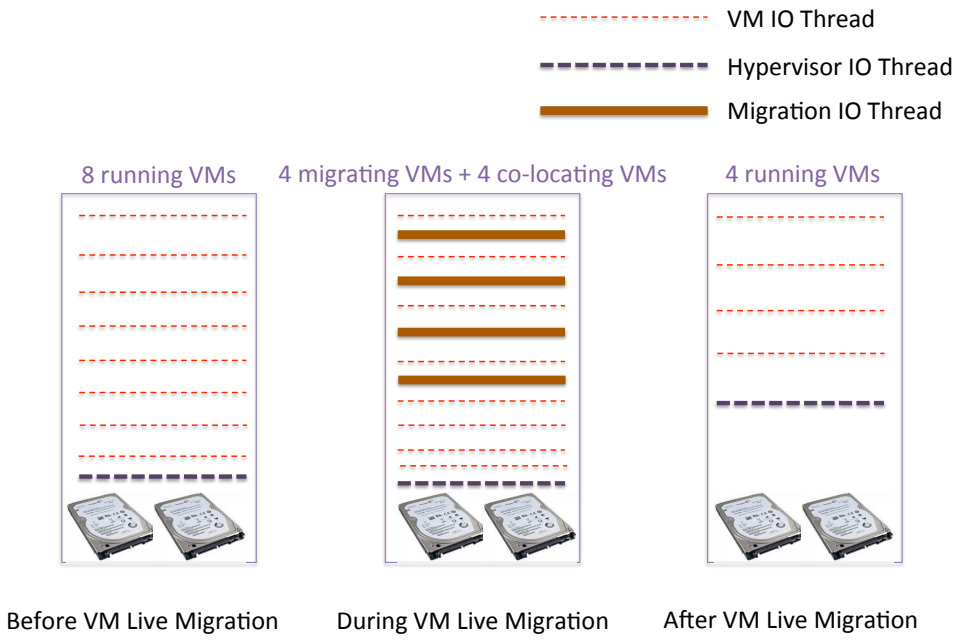


Figure 5.1: The IO threads model of virtualized system during different phases of VM live storage migration

When it comes to the VM live storage migration, one migration thread will be assigned for each migrating VM, and it will migrate all the state information of the migrating VM from the source server to the destination server. Most of the time, the migration thread will read data of the VM's virtual disk images from the beginning to the end, and sync the updated data to the destination server as well. From the

perspective of the Sequential IO property, the migration thread is a perfect sequential workload, as it will read data within the space of the virtual disk images purely sequentially and deserve good performance. Unfortunately, that's not the reality.

It is the interleaving with other IO streams, including VM IOs and other migration IOs, that undermines the Sequential IO property of migration threads. Disk head has to seek and rotate to other places in order to serve other IO requests between two consecutive read requests for the migration thread. After the serving of other IO requests, disk head has to seek and rotate back to the neighbor of previous location, in order to serve the next read request for the migration thread. Therefore, the IO performance of all participating IO streams is degraded significantly and longer VM migration time and low VM performance can not be avoided.

During the VM live storage migration, hypervisors are already overloaded because of the additional bandwidth hungry migration threads. By further destroying the Sequential IO property of migration threads, more weight will be put on the migration engine, so that it's much more challenging to migrate VMs fast and provide SLA for the IO performance of all participating VMs.

5.1.3 Motivation

After carefully exam the state-of-the-art research works and the internal mechanism of migration workflow, we observed that 1) Individual IO requests from the migrating VM are generated by applications, so that we have limited capabilities to manipulate the VM IO stream in order to improve the VM IO performance. 2) Only the total migration time and the accuracy of the VM state transmission matters for migration threads, while the individual request size, the starting address of each request or the sequence of migration IO requests does not matter at all. Therefore, the migration engine has the full flexibility to generate different kinds of migration IO requests,

as long as all the state information of the migrating VM arrives in the destination correctly within a reasonable migration time window.

Inspired by these observations, we propose our novel VM live storage migration scheme, named *IOFollow*, that will improve both the VM IO performance and migration performance by generating and scheduling the migration sequence according to the IO stream of VM requests. Essentially, we will select the next block migration candidate based on the two criteria: 1) this data block has not been migrated to the destination; 2) the address of this data block is close to the current access region or the position of the disk head. In this way, we expect to get rid of the unnecessary disk head movements, so that the interleaved IO requests stream at the gate of the storage system becomes more sequential. The performance of all the VM IO threads, hypervisor IO thread and migration IO threads climbs significantly during the VM live storage migration process. Further, we can selectively cache in memory data blocks read by migration threads from the storage system, and use these data to serve the predicted incoming VM IO requests, so that these VM IO requests will not need to touch the storage system at all. Therefore, both the VM IO performance and migration IO performance can be improved significantly.

At first glance, IOFollow, WAIO and Zheng's work [12] all improve the VM live storage migration performance by exploiting the workload characteristics within VMs, but their fundamental ideas are different from each other. In the WAIO system, the VM's working set is identified and outsourced to another surrogate storage device temporally, so that the VM thread is served by the surrogate device and the migration thread accesses the original storage device most of the time. The IO interference between these two kinds of threads are solved by the isolation of these threads to different storage devices. In Zheng's work, the goal is to diminish the repeated transmissions of frequently updated data blocks by migrating the infrequently updated

data blocks first. In other words, the VM threads will access the hot disk zones (the VM's working set) while the migration threads access the cold disk zones (infrequently accessed data blocks). This scheme will reduce the total data transmission at the cost of intensifying IO interference between VM threads and migration threads, as storage devices have to seek back and forth to serve the interleaved IO requests both in the hot zones and cold zones. Finally, IOFollow aims to improve the VM migration performance by letting migration threads and VM threads cooperate with each other. Moreover, IOFollow improves the cache hit ratio for VM threads by intelligently cache data blocks in memory.

5.2 System Design and Implementation

In this section, we first outline the main design objectives of IOFollow. Then we present the architecture overview of the IOFollow system, followed by a description of migration blocks scheduling and IOFollow Block Cache Manager. The data consistency issue of IOFollow is discussed at the end of this section.

5.2.1 Design Objectives

The design of IOFollow aims to achieve the following three objectives:

- *Accelerating the VM live storage migration performance:* - By removing most of the unnecessary and time consuming disk seek operations, the VM live storage process can be significantly accelerated.

- *Improving the VM IO performance:* - By improving the block cache hit ratio and reducing disk seek operations, VM IO requests can be either served by the block cache, or served by storage device faster because of the less disk seek operations.

- *Providing high flexibility:* - IOFollow is quite flexible and can be tuned with different parameters, such as cache replacement algorithms and migration chunk sizes, for different applications.

5.2.2 IOFollow architecture overview

File based virtual disk images have been extensively adopted in the virtualized environment [64, 101]. From the VM's perspective, the virtual disk image is the same as the physical disk that supports all kinds of block layer's API, such as ISCSI commands. It's compatible with main stream Guest Operating Systems, such as Windows series and Linux series. From the storage system's point of view, virtual disk images are nothing but regular large files that can reside in most file systems. Therefore, all the IO requests from user applications of the running VM become the IO requests for the underlying large file that hold the virtual disk image. Similarly, the migration thread will read this large file to migrate the VM's storage state information. The performance of virtual disk images is crucial for the running VM's performance and the migration agility. To improve the performance of virtual disk images, several dedicated file/storage systems have been invented for virtual disk images only, such as VMWare's VMFS and Tintri VMStore [102, 103].

Figure 5.2 shows the architecture overview of the IOFollow system. IOFollow is a simple module that can be incorporated into any modern hypervisors, and its parameters, such as the migration chunk size, the block cache replacement algorithm, can be tuned to different application workloads. For the VM live storage migration jobs, only the server in the source side needs to incorporate the IOFollow module, while the server in the destination side remains intact. IOFollow is a performance boost layer that can be combined with conventional live migration approach, including Dirty Block Tracking and IO Mirroring, to further improve both VM IO performance

and migration performance. IOFollow can be applied to live migrate VMs between servers within the same cluster or across different data center globally.

In the current virtualized systems, there are two level caches for each running VM: guest disk write cache (within the VM), and host page cache (within the hypervisor). According to the characteristics of different applications, users can choose to enable or disable either of these two level caches or both, before the creation of a VM or during the lifecycle of the running VM. In the normal execution period, only the VM IO stream from the running applications and guest operating system visits these two level cache systems, which could save a lot of storage access for applications or guest operating system. When the live storage migration starts, an additional IO hungry stream of IO requests from migration thread comes in, which will occupy a large portion of dedicated host page cache for the migrating VM. As traditional cache can not improve performance for streaming IO requests (e.g. online streaming video applications), a better design of cache system is necessary for the overall system performance.

IOFollow contains two major components: Migration Blocks Scheduler and Migration-Aware Block Cache Manager. Migration Blocks Scheduler will analyze the VM IO requests traffic, identify the current VM access zone, predict the later IO access region, and then select the right data chunk to migrate. Once the data chunk is migrated to the destination server, it will be handed over to the Migration-aware Block Cache Manager. The selection of migration data chunks is based on two parts: 1. shorter seek time of the storage system for the migration IO request; 2. this data block fetched by the migration thread may serve the later VM IO requests with a higher possibility. Migration-aware Block Cache Manager is to manage the memory resource and intelligently cache data blocks for later VM IO requests. As the migration thread only scan the virtual disk images once, it will not access the same data blocks for

more than once, except the updated data blocks. However, these data blocks may be accessed by VM IO requests. Therefore, by caching hot migration data blocks in memory, many incoming VM IO requests can be served by Migration-aware Block Cache Manager in memory directly. Once the block cache is full, a cache replacement algorithm will take actions and cold data blocks will be evicted.

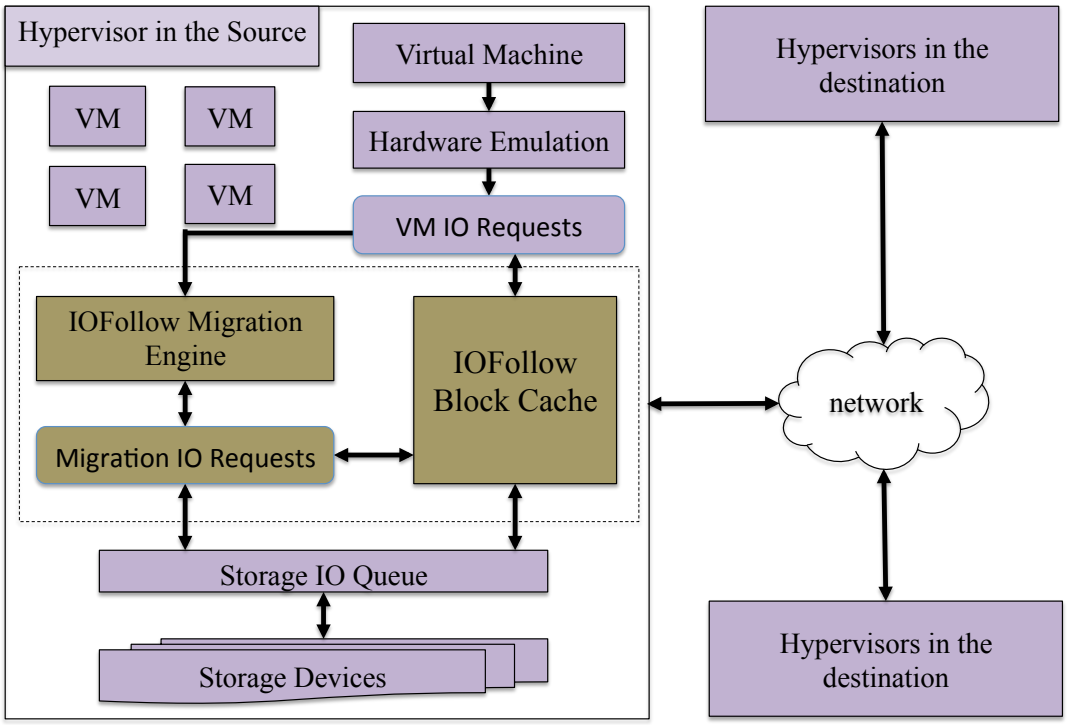


Figure 5.2: The architecture of IOFollow System

5.2.3 IOFollow Migration Blocks Scheduling

In the Migration Blocks Scheduling component, there are three decisions to make for the VM live storage migration purpose:

The first decision system administrators need to make is how much storage resource to allocate for the migration thread. Given the total storage resource remains unchanged, the more resource migration thread gets, the less storage bandwidth VM threads have. Essentially, this is a tradeoff between VM IO performance and migration performance, and it can be adjusted for different user cases or preferences.

After the storage resource allocation is determined, Migration Blocks Scheduling will select the next chunk to read from the storage system, and this chunk will be migrated to the destination server. Ideally, we prefer to data chunks satisfying the following conditions: 1) this chunk has not been migrated to the destination server yet. 2) the starting address of this data chunk is close to current VM IO access region. 3) the incoming VM IO requests will read partial or full of this data chunk with a high possibilities. In order to intelligently pick the right data chunk candidate, the spatial and temporal locality of the VM IO stream needs to be analyzed online, and the VM's working set needs to be predicted by the Migration Blocks Scheduling component.

The last decision is the migration chunk size. While both fixed and dynamic chunk size are applicable for IOFollow system, the overall VM and migration performance can be noticeably affected by the sizes of individual migration data chunks. This is an old and common problem in many aspects of system design, which require different techniques, such as online profiling and trace analysis, to tackle this problem. For instance, in Zheng's work, dynamic chunk size has been applied to reduce the number of repeated data chunk migration. In our IOFollow system, we start from fixed chunk size and evaluate its performance improvement. There is no doubt that IOFollow can be combined with other chunk size determination mechanisms for different workloads. The full algorithm of Migration Blocks Scheduling is presented in Algorithm 1.

Algorithm 1 IOFollow Migration Blocks Scheduling

```
1: Initialization:
2: setMigVM: the set of concurrent migrating VMs
3: position: the disk head location of the storage system
4: function IOSCHEDULING(setMigVM)
5:     for vm1 in setMigVM do
6:         serve IO requests for vm1
7:         update position with IO requests' addresses and sizes
8:         vmrange = the block range of vm1 that needs to migrate
9:         SelectChunk(vmrange, position)
10:    end for
11: end function
12: function SELECTCHUNK(vmrange, position)
13:    if vmrange is not empty then
14:        select the data chunk from vmrange whose address is close to position
15:        remove this chunk from vmrange
16:        issue the IO request for this data chunk
17:        update the position to the current chunk address
18:    else
19:        Return Complete
20:    end if
21: end function
```

5.2.4 Migration-aware Block Cache Manager(MABCM)

In the runtime, each running VM is assigned a number of memory pages by the hypervisor, so that the VM can cache whatever it wants in memory, rather than access the storage device every time. Normally these memory pages are classified as guest write disk cache and host page cache [104], as described in previous section. When it comes to the VM live storage migration, the whole virtual disk images will be read from storage system to memory and then migrated to the destination server. *A straightforward question comes in: Do we need to cache these data blocks in memory or not?* Apparently, we can not cache all these data blocks in memory, as the memory allocated to single running VM is usually much smaller than the size of the virtual disk images. If we do not cache all these data blocks in memory at all, we end up

wasting a lot of storage bandwidth. Considering the cost it takes to read in data blocks from storage system to the memory and additional read requests will be issued to the storage system by the VM thread, even if the target data blocks have already been read in once by the migration thread. Therefore, caching a set of data chunks with a higher access possibility by VM thread later in memory will improve the VM IO performance by the reduction of storage accesses. The challenging is how to identify these data chunks and how to evaluate and replace data chunks once the cache is full. As we discussed in the previous section, the Migration Blocks Scheduling component takes charge of the first challenge, while the MABCM solves the second one.

In MABCM, each entry is a data chunk of the virtual disk image in a specific address, and there are a number of such entries in the MABCM. For each entry we can evaluate its liveness value based on the answers to these questions: 1. Is this block has already been migrated to the destination server or not? 2. How long has this block been in the cache? 3. What's the possibility that this data block will be accessed by the VM IO threads in the near future? With such information, MABCM can sort these entries and replace the entry with the least liveness value when the cache is full. As such information is closely related to the spatial and temporal locality of workloads, such cache management algorithm has to be tuned to cater different applications. The more accurate locality we learn from workloads, the better cache hit ratio and IO performance we can achieve. Compared with the baseline approach, in which traditional two level cache management algorithm is employed, our migration-aware cache management scheme can significantly improve the VM live storage migration performance. The skeleton algorithm of Migration-aware Block Cache Manager is introduced in Algorithm 2.

Algorithm 2 Migration-aware Block Cache Manager

```
1: Initialization:
2: Apply memory pages from the hypervisor.
3: initalize metadata for the cache manager
4: Start to take requests from VM threads or migration threads
5: function ReadBlock(blockAddr)
6:     if blockAddr is in cache then
7:         return the block and update liveness info for this block
8:     else
9:         Read data block from storage, return data block to the client
10:        Migrate this block to the destination if has not migrated yet
11:    end if
12: end function
13: function WriteBlock(blockAddr, dataBuf)
14:    if blockAddr is in cache then
15:        update data block in memory and the liveness info
16:    else
17:        Read data blocks from storage, and update data in memory
18:        Migration data blocks to destination if has not migrated yet
19:    end if
20: end function
21: function PutBlock(blockAddr, dataBuf)
22:    if cache is not full then
23:        put data blocks to the cache and update the liveness info
24:    else
25:        evict data blocks with least liveness information
26:        put data blocks to the cache.
27:    end if
28: end function
```

5.2.5 Data consistency

System failures or migration crash can be caused by many factors, such as hardware/software bugs, power failures, wrong operations or attacks from outside. Our IOFollow system embraces these failures with the proactive design for system consistency and robustness. Specifically, IOFollow stores the key data structures in a non-volatile RAM (NVRAM), in order to prevent the sudden loss of power supply or system crash. Since the size of these data structures is generally very small, it will

not pose any significant cost for the system. For data in the blocks cache, we can store them directly in DRAM, as there is always another copy in the storage system.

5.3 Performance Evaluation

In this section, we present the performance evaluation of the IOFollow scheme through extensive trace-driven experiments.

5.3.1 Evaluation Methodology

As discussed before, both VM performance and migration performance are important for the overall system performance. The migration performance is determined by the network performance and storage performance, while the VM performance is mainly determined by the storage system. In this evaluation, we focus on the storage performance evaluation for both VM threads and migration threads.

Specifically, we create a virtual disk image in the disk drive, and replay the published storage block level traces on top the virtual disk image. At the same time, we generate the migration IO requests on top of the same virtual disk image, and the migration IO requests will read in all the data blocks within this virtual disk images. For both VM IO performance and migration performance, we use average IO response time as the performance metrics. A shorter response time for the VM IO requests means higher IO performance for the applications within the running VM. Also, a shorter response time for the migration IO requests indicate faster VM live storage migration. We compare our IOFollow scheme with the standard migration scheme that will ignore the characteristics of the VM IO stream and migrate the virtual disk images from beginning to the end.

In the experiments, we need to consider several parameters: First, we employ fix

Table 5.1: Hardware Specifications in Our Experimental Platform

CPU	Intel(R) Xeon(R) CPU, X3440@2.53GHz
MotherBoard	Winbond Electronics 0V52N7
Memory	8GB, AMI CMX8GX3M2A1333C9
Hard Drives	1TB Seagate ST31000524AS, SATA
Network	1Gbps Ethernet

Trace	Read Ratio	IOPS	Avg. Req. Size(KB)
Fin2	82.4%	125	2.2
WebSearch1	100%	113	15.1
WebSearch2	100%	100.3	14.9
WebSearch3	100%	63.52	15.2

Table 5.2: Trace characteristics

migration chunk size in the experiments, and the IOFollow system performance can be further improved by more sophisticated dynamic chunk size migration approaches. Second, we allocate the storage resource between VM threads and migration threads with pre-determined ratio. For instance, 2:1 means storage system will serve every two VM IO requests before perform one migration IO request, unless there is no VM IO request waiting in the IO queue. This approach is simple but effective to achieve a tradeoff between the VM performance and migration performance. Our IOFollow system can also improve the VM live storage migration performance under other storage resource allocation policies. Finally, we evaluate the IOFollow system under different number of concurrent VM live storage migrations.

The experimental platform consists of a single server configured with an Intel Xeon X3440 processor, 8GB DDR memory and two 1TB hard drives, 12.10 Ubuntu system. The hardware information is described in details in Table 5.1.

5.3.2 Workload Analysis and Trace Replay

In order to measure the performance improvement of IOFollow scheme, we replay block level traces and collect the IO performance during the migration process. The Storage Performance Council [66] has published several block level traces for research purposes, and these traces have been widely employed to evaluate the storage system performance [67, 68, 69]. The Financial2 trace is collected from OLTP applications in a large financial institution and the WebSearch traces are collected from a web search engine. The key characteristics of these traces are summarized in Table 5.2. In our experiments, we implement a trace replay tool that will read trace files and generate the migration IO requests according to the VM IO stream. Algorithm 3 introduces the workflow of our trace replayer in details.

Algorithm 3 IOFollow Trace Replayer

Initialization:
vmSize: the logic size of the VM's virtual disk images in total
chunkSize: the pre-determined chunk size for migration threads
storageRatio: serve storageRatio VM IO requests from *vmIORequestsQ* before serve one migration request
vmIORequestsQ all the IO requests from the trace file in order
function REPLAYING(vmSize)
    migChunkAddrSet: the starting address of each data chunk in migration
    position: current position of storage system
    while *migChunkAddrSet* is not empty **do**
        serve *storageRatio* VM IO requests, and update *position* accordingly
        record individual response time for each VM IO request
        find the chunkAddr from *migChunkAddrSet* that is close to *position*
        remove this chunkAddr from *migChunkAddrSet*
        serve this migration request
        update *position*, and record the IO response time for the migration request
    end while
    report average response time for VM IO requests
    report average response time for migration requests
end function

5.3.3 Result Analysis

In this scenario, there is only one VM migrating from the source to the destination. The migration chunk size is 1MB and the storage resource allocation ratio is 2:1 between VM threads and migration threads. As indicated in figure 5.3, compared with the standard migration scheme, the average IO response time for the migration thread in our IOFollow scheme are reduced by $\frac{172-129}{172} = 25.0\%$, $\frac{166-107}{166} = 35.5\%$, $\frac{185-99}{185} = 46.5\%$ and $\frac{169-112}{169} = 33.7\%$ for different traces. The main reason is that, with the simple dynamic scheduling of migration block sequence, individual blocks are migrated when the disk head is moving close to it, so that unnecessary seek and rotation operations are reduced noticeably for migration IO requests. Moreover, such scheduling makes easier for the disk controller to apply internal optimizations, such as the requests merge, in order to further improve the IO performance. Therefore the overall migration IO performance is improved significantly.

The performance evaluation result for the VM IO thread is shown in figure 5.4. We have the following two observations: 1. The average IO response time increases by $\frac{60-35}{35} = 71.4\%$, $\frac{52-22}{22} = 136.4\%$, $\frac{68-25}{25} = 172.0\%$, $\frac{62-37}{37} = 67.6\%$ for individual storage traces. This clearly indicates that the VM IO performance is significantly affected by the live storage migration jobs. Not only more IO requests generated by the migration thread, but also the access locality of the applications is destroyed but the new coming migration IO requests. For instance, two consecutive read requests from the application become non-consecutive requests when a migration request is served in between, and the address of the migration request is far from that of the application requests. 2. Compared with the standard migration approach, the average IO response time in our IOFollow scheme is decreased by $\frac{60-42}{60} = 30.0\%$, $\frac{52-31}{52} = 40.4\%$, $\frac{68-45}{68} = 33.8\%$ and $\frac{62-49}{62} = 20.9\%$ for different traces. The reason behind such

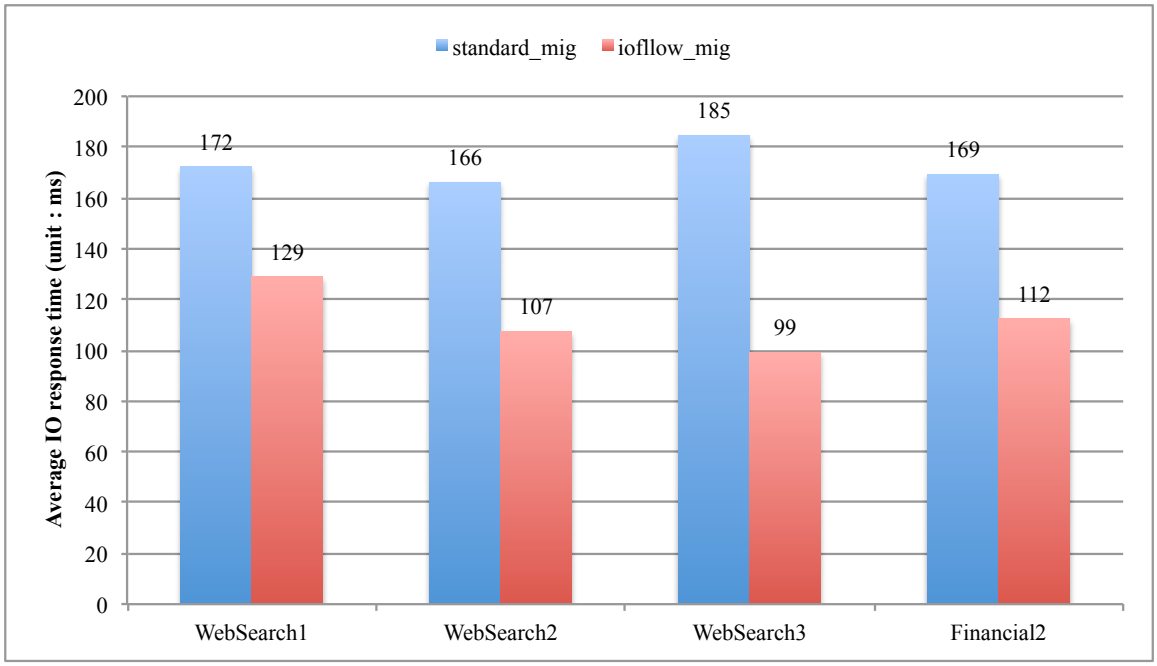


Figure 5.3: The Migration Performance Comparison in Single VM Migration

improvement is that the interleaved IO stream at the gate of the storage system is becoming more sequential in the IOFollow scheme than that in the standard scheme. Therefore, the access locality of the applications can be reserved.

5.3.4 Sensitivity Studies

In order to investigate how the migration chunk size, the storage allocation policy and the number of concurrent VM migrations affect the performance of IOFollow system, we conduct a series of trace driven experiments. We report the normalized response time of the IOFollow scheme based on that of the standard scheme under the same experiment configuration.

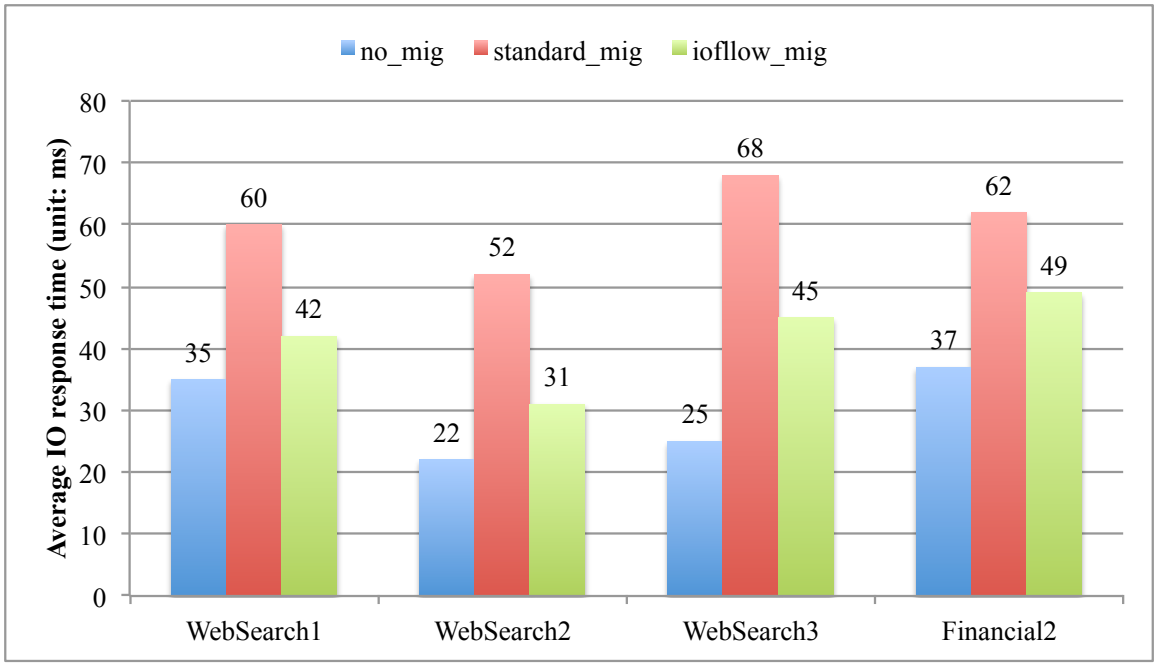


Figure 5.4: The VM Performance Comparison in Single VM Migration

5.3.4.1 Chunk Size

In this group of experiments, we compare the migration performance and VM performance between IOFollow and standard migration approach, for the four traces under different migration chunk sizes ranging from 128KB, to 256KB, 512KB, 1MB and 2MB. We normalize the IO response time in IOFollow based on that in the standard migration. As figure 5.5 shows, compared with standard migration, IOFollow reduces the average IO response time for the migration thread from 45% to 65%, for all the traces under different chunk sizes. Meanwhile, IOFollow decreases the IO response time for the VM thread from 70% to 88%, as indicated in figure 5.6. Therefore, IOFollow can improve the VM live storage migration performance under different migration chunk sizes.

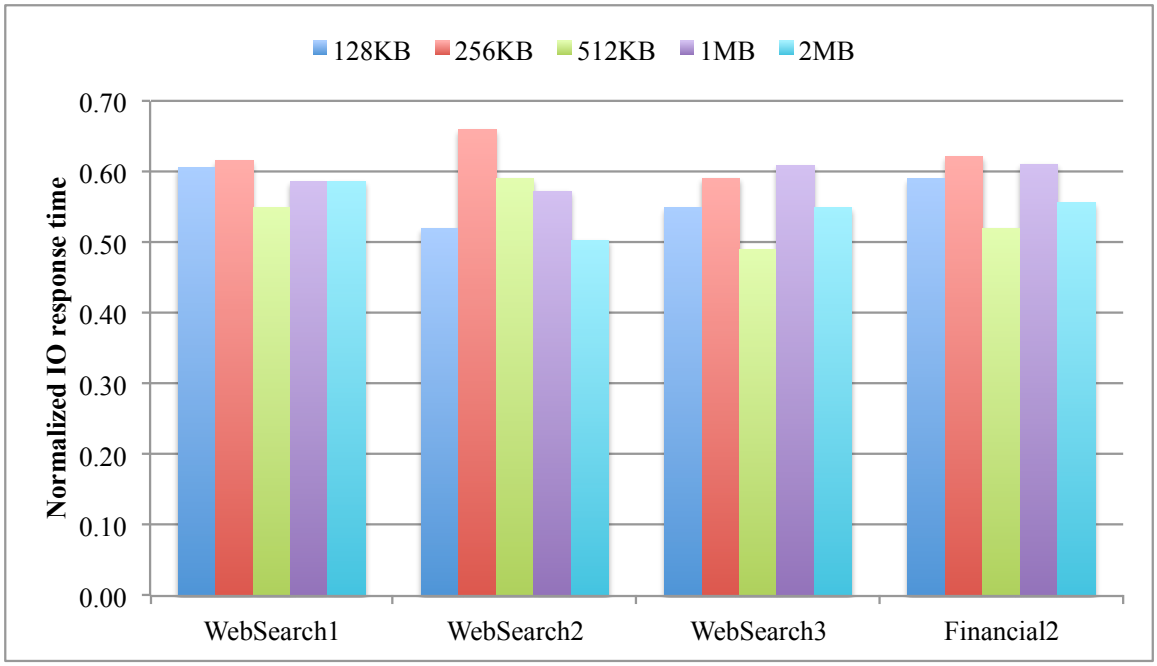


Figure 5.5: The Migration Performance Comparison in Single VM Migration among Different Migration Chunk Size

5.3.4.2 Resource Allocation Policy

Before the start of the VM live storage migration, the policy of storage resource allocation between VM IO threads and migration threads is determined by the system administrators or automatically. In this group of experiments, we evaluate the performance improvement of IOFollow scheme over standard migration under different storage allocation ratio (VM IO resource:migration resource) from 1:1, to 2:1, 3:1 and 4:1. As figure 5.7 and 5.8 show, IOFollow scheme can improve the VM IO performance by up to 74% and reduce the IO response time for the migration thread by up to 50%. When many running VMs share a single server and the server can not satisfy the requirement of storage IO bandwidths for each individual VMs, one or more VMs will need to be live migrated to other servers. However, there is very

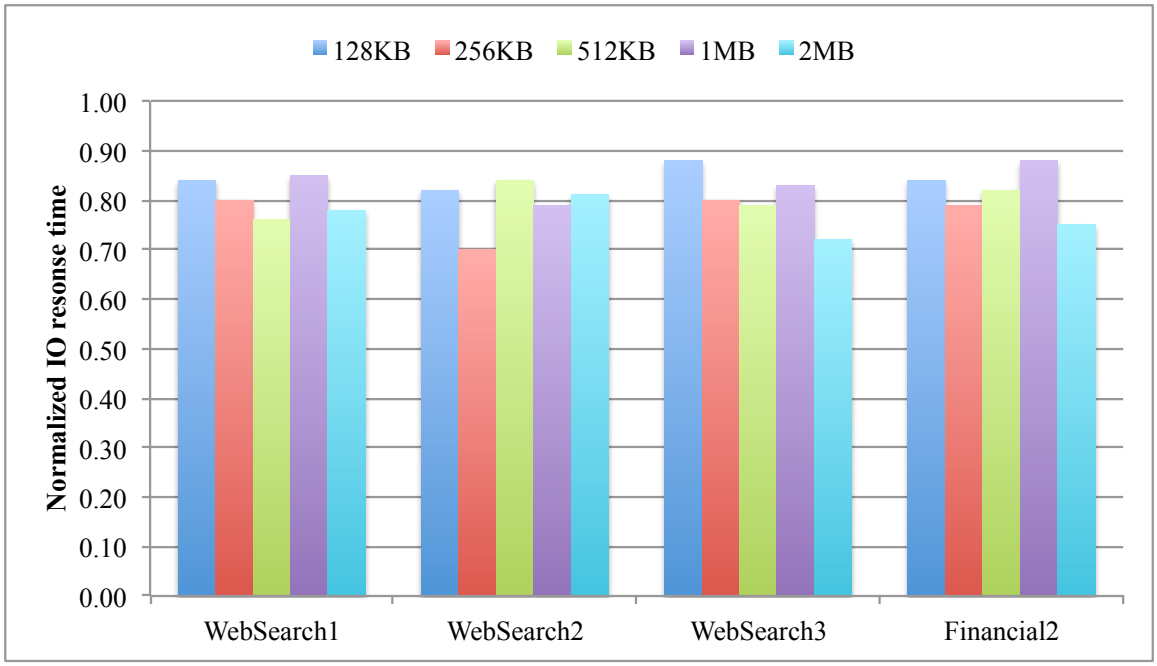


Figure 5.6: The VM Performance Comparison in Single VM Migration among Different Migration Chunk Size

limited storage IO bandwidth available for the new migration threads, as the total storage IO bandwidth is fixed. In this scenario, IOFollow is far more important for the VM IO performance, as it introduce less bandwidth consumption for the storage server, compared with the standard migration.

5.3.4.3 Concurrent VM Migrations

As discussed in previous sections, multiple concurrent VM live storage migration is not uncommon in the current data center. In such scenarios, the source server will undergo bigger pressure as multiple IO hungry migration streams are introduced and the overall storage capacity for the source server remains the same as before. In order to investigate how much can IOFollow scheme improve the VM live storage migration performance compared with standard migration scheme in these scenarios,

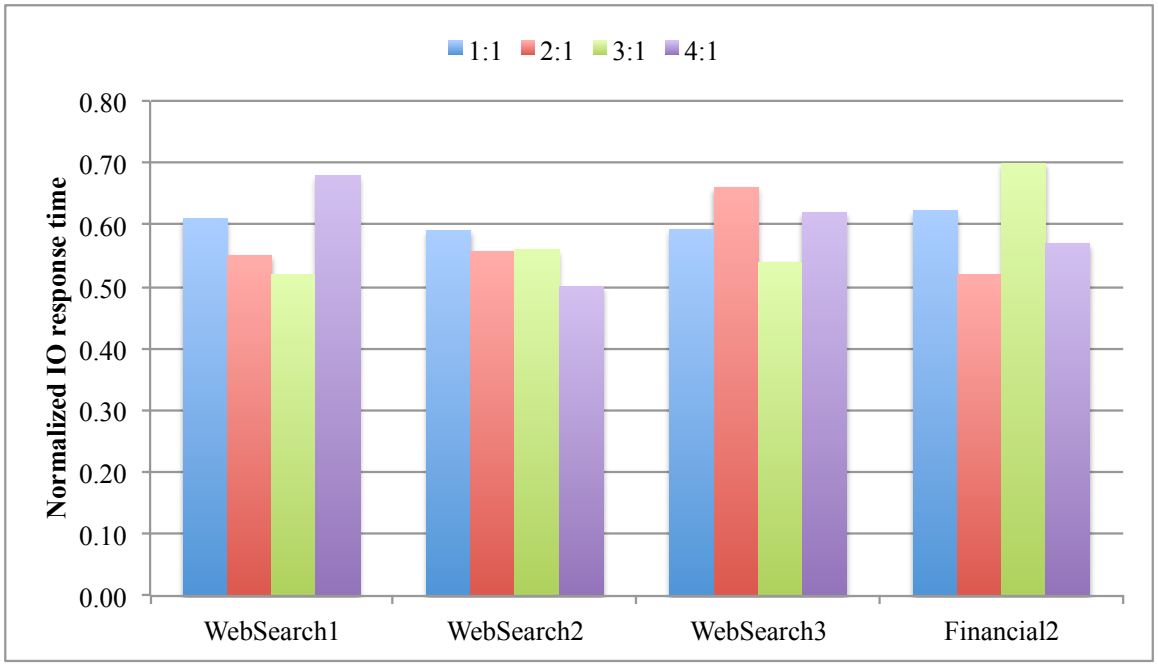


Figure 5.7: The Migration Performance Comparison in Single VM Migration among Different Storage Allocation Policy

we conduct a series of experiments to evaluate such performance improvement under different traces and different concurrent VM live storage migration.

As multiple migration threads are introduced, it's better to schedule the VM IO requests and migration IO request of a single VM as a unit, and then round robin between multiple concurrent migrating VMs. The reason is that the targeting addresses of VM IO requests and migration IO requests for a single VM are close to each other, which will makes the final IO request stream in the disk control more sequential. Therefore, in this set of experiments, we let the the storage system round-robin among several migrating VMs and each time it will serve a number of VM IO requests and a single migration request for a particular VM. We report the normalized IO response time based on the standard migration approach. As indicated from figure 5.9 and 5.10, IOFollow reduces the average IO response time by 55% for

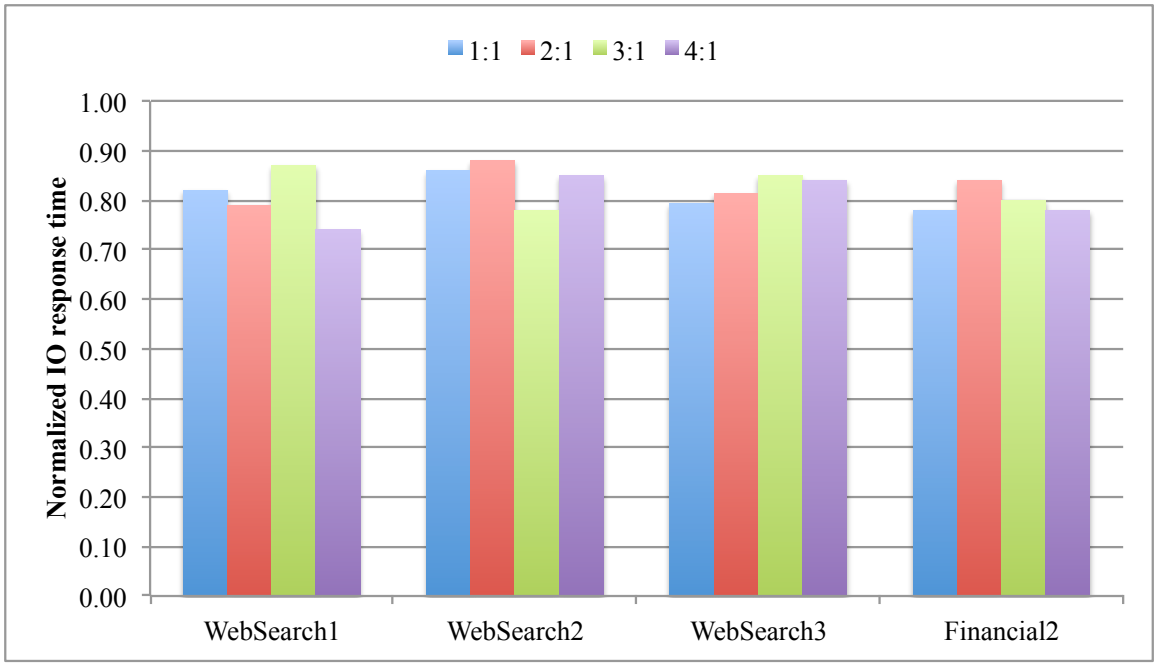


Figure 5.8: The VM Performance Comparison in Single VM Migration among Different Storage Allocation Policy

migration thread and 45% for VM IO thread, compared with standard migration. In addition, as the number of the concurrent migrating VMs increase, the performance improvement by IOFollow becomes more important.

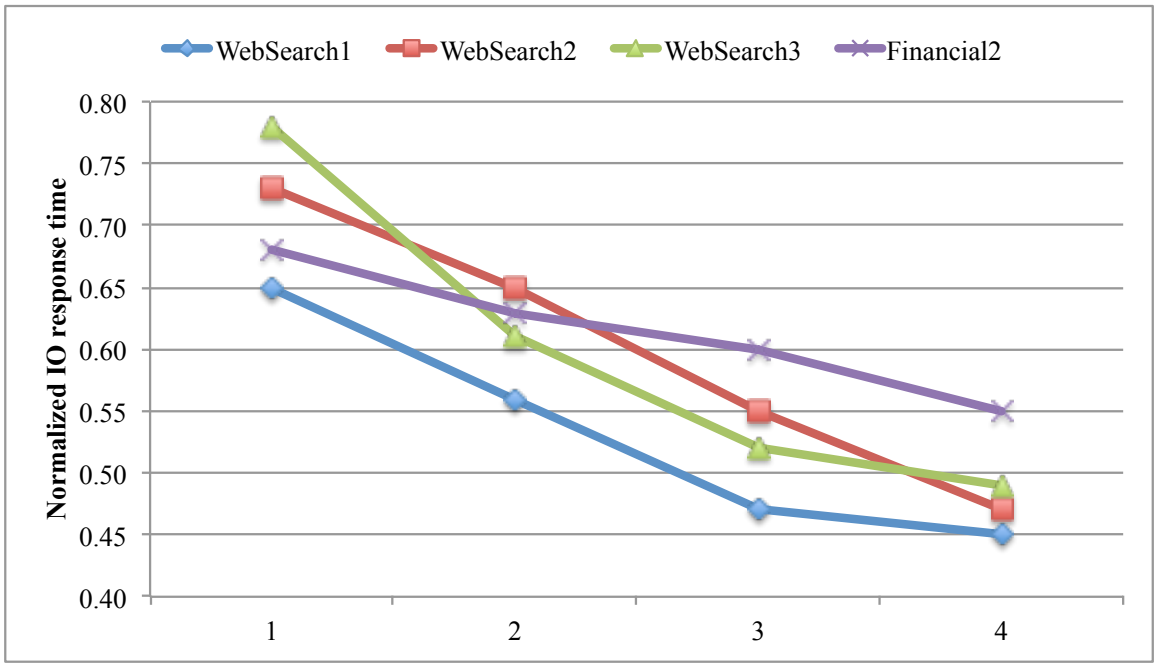


Figure 5.9: The Migration Performance Comparison among Multiple VMs Migration

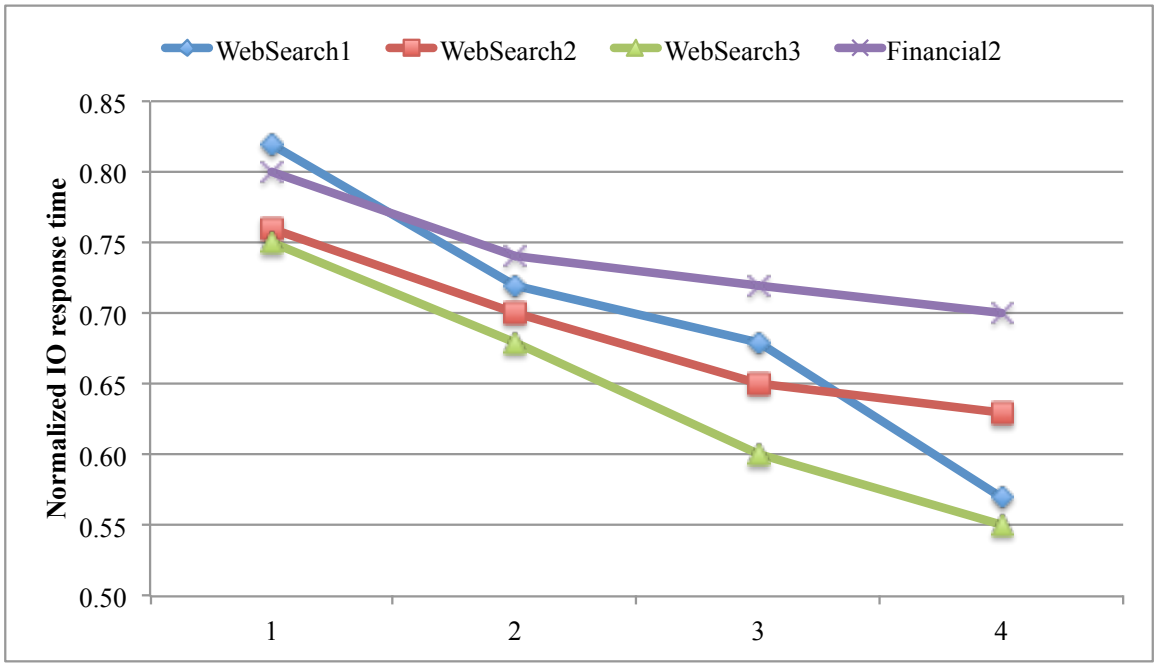


Figure 5.10: The VM Performance Comparison among Multiple VMs Migration

Chapter 6

Directions for Future Research Work

So far, we have designed and implemented three novel live storage migration schemes to improve the migration efficiency. Our trace driven evaluation results indicate that all these schemes can significantly improve the overall migration performance. In this section, we introduce two more ideas that also have the potential to improve the live migration efficiency. The motivation and design of these two ideas are presented here, while the system implementation and evaluation can be done in the future research work.

6.1 Semantic-Aware Live-Block Migration

Intuitively, the key to the reduction of storage migration time is how to maximize the effective migration bandwidth and minimize the amount of data transferring. File systems provide us with such an opportunity to reduce the total data transmission during the VM live storage migration process, as there is an abundance of free storage space in storage systems. Based on the data collected from extensive data-center benchmarking studies over the past 16 years, Mark Levin points out that on average the disk storage utilization is $56.6\% - 75.5\%$ for UNIX environments, and $46.6\% - 55.8\%$ for Windows environments [105]. In a five-year study of file system metadata from more than 60,000 Windows PCs in a large corporation [106], it clearly shows

that the mean file system fullness has dropped from 49% in 2000 to 45% in 2004, and the aggregate fullness of the population, computed as total consumed space divided by total file system capacity, has been steady 41% overall all the sample years. Furthermore, Symantec's 2008 State of the Data Center Survey [107] found that data centers utilize 50% of their storage capacity. The low utilization of storage systems may stem in part from the fact that the cost of hard disk has been declining continually relative to that of management, and storage capacity is always over-provisioned for peak performance and the ever-increasing demand for storage capacity.

In the virtualized environment, virtual disk images emulate physical disks and provide storage service for the running VMs. Inspired by the facts of ubiquitous and rich free space in storage systems, we propose to leverage filesystem semantics for less amount of data transferring. The basic idea of *Semantic-Aware Live-Block Migration* (SALM) is to extract the liveness information of filesystem blocks of virtual disks, and migrate the live blocks only. By doing so, it can significantly reduce the storage migration time. SALM involves the extraction of block liveness information and the process of live-block migration.

Block Liveness Extraction. By definition, block liveness is about whether or not a filesystem block on the storage device is valid. In general, a file system uses bitmap-like data structures in filesystem metadata (e.g., block bitmap and inode bitmap in the Linux Ext2 file system) to keep track of allocated data and metadata blocks. When a file system allocates data blocks for a new file, the corresponding bits in the block bitmap will be set and the data blocks will be written. When deleting a file, only the block bitmap will be updated. Since virtual disk image is unaware of the block liveness, it may result in lack of storage intelligence and incapability of semantic exploitation at the device level. If a disk can be made aware of the filesystem block liveness at the block level, it is able to perform better data layout optimiza-

tion [108], prefetch and caching, secure delete [109], intrusion detection [110], RAID reconstruction [111], energy-efficient logging [112], SSD garbage collection [113].

To this end, explicit and implicit approaches to extract the block liveness information within storage devices are proposed [114]. With the explicit approach, a file system conveys the filesystem semantic information to the underlying storage device directly and explicitly. This approach requires modifications to the file system, the standard block interface and storage device firmware. One typical example is the TRIM command [115, 116]. TRIM has been proposed by the ATA standard Technical Committee T13 to make SSDs aware of invalid/deleted data blocks more effectively, and SSD can utilize the block liveness information to perform the garbage collection operations more efficiently. Although the explicit approaches are able to easily convey semantic information between the file systems and the underlying storage devices, many existing computer systems cannot benefit from them due to the need to modify file systems, device drivers as well as the block interface. On the contrary, implicit approaches convey semantic information implicitly while keeping the standard block interface unmodified. Sivathanu *et al.* propose SDS [117], a semantically-smart disk system that can infer metadata structure fields of FFS-like file systems by performing a series of semantics extraction operations. SDS attempts to provide a generic semantics extraction method with the help of a user-space assistant software tool.

In the context of live storage migration, Both explicit and implicit approaches can be employed to extract the block liveness information from the file systems in VMs. For explicit approaches, one possible method is to enhance virtual block device drivers of VMM to support the TRIM command as modern SSD products do. It enables VMM to have opportunities to interpret every TRIM command and obtain the block-liveness information of file systems in the guest OS. Another possible method is to develop and run a daemon program in the guest OS that can directly extract

the filesystem semantics and communicate the block liveness information with VMM. For implicit approaches, the key is to uncover the relationship between data blocks and files, and the relationship among files within the storage device. These semantics enable the block device to identify which file the specific data block belongs to, and, further, which directory the specific file belongs to.

Live-Block Migration. With the block liveness information, the migration of live blocks is easy. Instead of migrating every block of the file system in conventional live storage migration approaches, our proposed live-block migration approach transfers only valid file system blocks. When an operation of VM live storage migration is triggered, VMM initiates a migration process. The migration process then requests the list of live blocks, and starts to transfer live blocks of the file system from the source to the destination on the background until all the live blocks have been transferred. Ideally live-block migration can halve the amount of data transferring and the migration time. Live-block migration can work for any pre-copy, post-copy, pre+post-copy live storage migration, or even offline storage migration approaches, and it can also work with any virtual machine disk image formats (e.g., VMDK, VHD, VDI, QCOW).

6.2 Redundancy/Similarity-Based Data Elimination

It is a well-accepted fact that there is rich data redundancy in storage datasets and workloads. Data redundancy stems from a variety of sources. First, in virtualized and consolidated storage environments, it is quite likely to run similar OSes and software and thus create data redundancy across virtual disks. A very recent study of the workloads obtained from a virtual machine running two web servers ("web"), an email server ("mail"), and a file server ("homes") shows that the unique writes

account for 42.35%, 7.83%, and 66.37% of the total writes under the web, mail, and homes workloads respectively [118, 119]. Second, version-control systems and the versioning schemes embedded in development and office software promote duplication of files for the sake of rollback and recovery. Another study of the workloads obtained from 7 disks used in experimental systems for kernel development and 4 disks in Office systems shows that the percentage of duplicate blocks ranges from 7.9% to 85.9%, and $5.8\% - 28.1\%$ of writes are duplicated [120]. Therefore, eliminating data redundancy with data deduplication during live storage migration has been proposed as an effective and efficient means to eliminating unnecessary data transferring of duplicate data blocks [16, 121].

However, the existing deduplication-based VM live storage migration approaches cannot work efficiently for the VMs under write-intensive workloads because data blocks after updates may be unique of any data blocks. It is necessary to design a new live storage migration approach that can not only eliminate the transferring of duplicate data, but also lower the amount of transferring of dirty unique data blocks.

Fortunately, real-world workload analysis clearly show the ubiquitous existence of data similarity: the data content of a block to write is always similar to the original content of the same block. For example, Yang *et al.* [122] conducted experimental studies, and found that usually only $5\% - 20\%$ bits inside a data block is changed by an update operation under a wide set of typical workloads. Motivated by this observation, they propose to log all previous versions of changed data blocks in time sequence in a delta-compressed format to save storage overheads in their proposed TRAP-Array architecture. Likewise, Wu and Xu [123] propose ΔFTL is to store the compressed delta in SSD upon the write operation, instead of the new data for the purpose of write reduction. More recently, VeloBit implements a content locality caching technique in its SSD caching software product by combining content-based

caching with high-speed delta compression.

Inspired by the observations and exploitation of ubiquitous data redundancy and similarity in storage workloads and datasets, we propose the *Redundancy/Similarity-based Data Elimination* (RSDE) scheme for the VM live storage migration of VMM to eliminate unnecessary transferring of redundant and similar data with the judicious combination of data deduplication and delta compression. Essentially, both data deduplication and delta compression are to trade computation for the reduction of data transferring. Therefore, the latency of transferring any given data block consists of the latencies of compressing the data content of the block at the source, transferring the compressed data (fingerprint or delta), and uncompressing the data at the destination. Higher compression ratio can accelerate the transfer process, but at the cost of longer uncompressing time. It implies that it is necessary to strike a good balance between compression ratio and uncompressing speed.

To this end, RSDE is designed to compress the original disk images of VMs to transfer with the data deduplication module, while compressing the data blocks that are being updated during the previous iteration of transferring compressed dirty data blocks with delta compression module. In doing so, it can leverage the scalability advantage of the data deduplication approaches to identify and eliminate duplicate blocks along the space dimension, while easily capturing and compressing similar blocks by delta compression along the time dimension. Moreover, in order to simplify the data deduplication process and ensure high deduplication throughput, we deploys fixed-size data chunking and fingerprint generation in the data deduplication module, as the same as those used in deduplication-based primary storage systems [50, 124, 120]. In order to deliver high-throughput delta compression performance, RSDE, like TRAP-array [122] and ΔFTL [123], generates the delta by XORing the new and old versions of the same data blocks and compresses the delta using LZF [125]. In doing

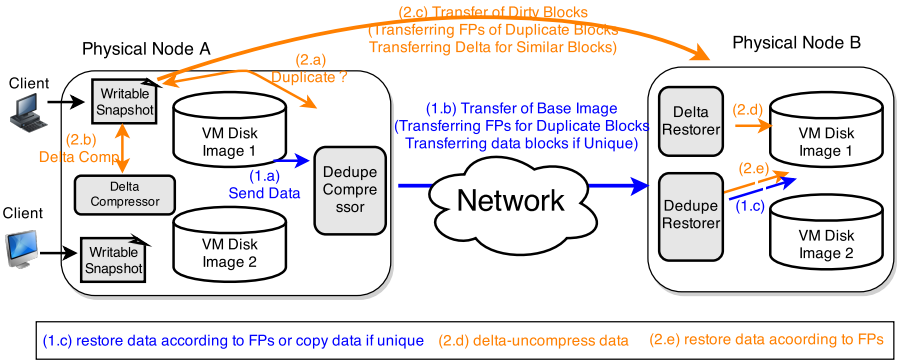


Figure 6.1: An illustrative example of the Redundancy/Similarity-Based Data Elimination scheme.

so, it can save the computation and space overheads of maintaining and looking up a global index structure to match a reference data block.

Figure 6.1 shows an illustrative example of the proposed RSDE scheme. On the source node, RSDE integrates three important functional modules: *Writable Snapshot*, *Dedupe Compressor* and *Delta Compressor*. Writeable Snapshot is a dedicated write logger for each VM disk image to accommodate all the writes of users that are originally targeted at the corresponding VM disk image during storage migration. The responsibilities of Dedupe Compressor include: (1) generating fingerprints(FPs) using MD5/SHA1 hashing functions given a data block to transfer; (2) looking up the fingerprint index and identify the uniqueness of the specific data block; (3) updating the fingerprint index after returning the fingerprint value and reference count of the data block to RSDE. The main function of Delta Compressor is to generate the compressed delta between the new content and the old content of a data block given a write operation. On the destination node, RSDE integrates two functional modules: *Dedupe Restorer* and *Delta Restorer*. Dedupe Restorer is used to restore the data content for a given fingerprint while Delta Restore is to restore the data block for a given compressed delta.

The workflow of VM live storage migration consists of two steps: Step 1 is to migrate the gang of the original VM images and Step 2 is to iteratively migrate dirty blocks until the amount of dirty blocks is less than a predefined threshold. On Step 1, as shown in Figure 6.1, VMM sends all the data blocks of VM images to Dedupe Compressor (Step 1.a). Dedupe Compressor generates the corresponding fingerprints and looks up the fingerprint table. If the data block to transfer is redundant, VMM transfers the address and fingerprint of the block to the destination. If found unique, VMM transfers the address, data content and fingerprint of the block to the destination (Step 1.b). When VMM at the destination receives the packet, it will check the flag of the packet. If it contains a unique block, VMM directly writes the data content to the target block and adds the fingerprint and target block address in its fingerprint index. Otherwise, VMM will look up the fingerprint index and find the corresponding block containing the data content it belongs to. VMM copies the content from the found data to the target block and updates the fingerprint index accordingly (Step 1.c). To further improve performance, the copy operation in Step 1.c can be performed in an asynchronous way: VMM updates and locks the fingerprint index as usual, and appends the copy operation to a "TODO" list instead of copying immediately. VMM can perform all the copy operations on the "TODO" list periodically or when the system is idle. This asynchronous IO optimization offers great opportunities to coalesce and reorder IO requests and mitigate costly disk head seeks.

On Step 2, VMM is to iteratively transfer data blocks dirtied by users, logged in Writable Snapshot, during the last iteration until the amount of dirty blocks is less than a predefined threshold. In each iteration, VMM firstly sends the dirty blocks in Writable Snapshot to Dedupe Compressor to determine whether it is a duplicate or unique block (Step 2.a). If it is unique, VMM sends the content and address of the block to Delta Compressor for delta compression (Step 2.b). The next step

is to transfer the address and fingerprint of the dirty block for duplicate blocks, or transfer the address and compressed delta of the dirty block for unique blocks (Step 2.c). When the packet is received at the destination, VMM will check the flag of the package and route it to Dedupe Restorer or Delta Restorer accordingly. For unique blocks, Delta Restorer will uncompress the delta, read the old content of the target block, and generate the new content by XORing the old content and delta data. Finally, the new content will be written to the target block (Step 2.d). The restoring of duplicate blocks on Step 2.e is as the same as Step 1.c. When the amount of dirty blocks is less than a threshold, VMM suspends the VMs at the source, performs one iteration of Step 2, and resumes the VMs on the destination.

Chapter 7

Conclusion

In this dissertation, we focus on the VM live storage migration performance and identify the fundamental and serious IO interference problem. The insights and understanding obtained from this study motivate us to design and implement three novel schemes to improve the VM live storage migration performance from different and orthogonal perspectives, as follows:

7.1 WAIO: Workload-Aware IO Outsourcing Live Storage Migration

Conventional migration approaches, such as Dirty Block Tracking (DBT) and IO Mirroring, do not address the problem of IO interference between VM IO requests and Migration IO requests during the migration period, which degrades both the VM IO performance and the migration performance. In this work, we propose a Workload-Aware IO Outsourcing scheme, short for WAIO, to improve the VM live storage migration efficiency. WAIO effectively outsources the VM's working set to a surrogate device during the migration and creates separate IO path for servicing the VM IO requests. By outsourcing VM IO requests from the original storage to the surrogate device, the VM live storage migration process can be performed on the original storage, no longer interfered, while the outsourced VM IO requests are serviced separately and thus much more quickly. Our lightweight prototype implementation

of WAIO and extensive trace-driven experiments demonstrate that, compared with the existing migration approach DBT, WAIO significantly improves the VM's IO performance during the migration process. Moreover, WAIO allows the hypervisor to migrate a VM at a higher migration speed, without sacrificing the VM's IO performance.

7.2 SnapMig: Snapshot-based VM Live Storage Migration

Most existing VM migration approaches can not solve the IO interference problem, as they induce significant extra storage and network traffic to the source server that is already heavily loaded or scheduled for upgrade or repair. As a result, both the VM performance perceived by the application/user and the migration performance are degraded significantly. In this work, we aim to address this problem by proposing a novel scheme, called SnapMig, to improve the VM live storage migration efficiency and eliminate its performance impact on user applications at the source server by effectively leveraging the existing VM snapshots in backup servers. By delegating backup servers to transfer VM base image and snapshots to the destination server, the source server only needs to migrate the latest state changes to the destination server, leading to simultaneously improved VM performance, shortened migration time and more efficient multiple-VM migration. Our lightweight prototype implementation of the SnapMig scheme demonstrates that, compared with the state-of-the-art approaches, SnagMig can significantly reduce migration time and improve the source server VM performance at the same time. Moreover, the performance improvement provided by SnapMig becomes much more pronounced with multiple concurrent VM migrations.

7.3 IO Follow: Improving the VM live storage Migration by IO Following

Current VM live storage migration approaches ignore the Sequential IO property of the interleaving IO streams at the gate of the storage server, by simply migrate the VM's virtual disk images sequentially, regardless of the concurrent VM IO streams and other migration streams. Therefore, many unnecessary time consuming disk head seek and rotation operations are introduced, which degrades both the VM IO performance and migration performance. Inspired by these observations, we propose our novel VM live storage migration scheme, named *IOFollow*, that will improve both the VM IO performance and migration performance by generating and scheduling the migration sequence according to the IO stream of VM requests. In this way, we expect to get rid of the unnecessary disk head movements, and the overall VM live storage migration performance can be improved significantly. Our trace-based experiments indicate that our IOFollow system can improve the overall performance significantly.

Bibliography

[1] "Amazon still dominates the 16 billion dollars cloud market." [Online]. Available: http://www.businessinsider.com/synergy-research-amazon-dominates-16-billion-cloud-market-2015-2

[2] "Netflix shuts down its last data center, but it still runs a big it operation," http://arstechnica.com/information-technology/2015/08/netflix-shuts-down-its-last-data-center-but-still-runs-a-big-it-operation/.

[3] "Computer systems research program guidelines." [Online]. Available: http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503306

[4] "The virtualization techniques." [Online]. Available: https://www.fluxlabs.net/solutions/virtualization

[5] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end performance isolation through virtual datacenters," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 233–248.

[6] "Server consolidation." [Online]. Available: http://searchdatacenter.techtarget.com/definition/server-consolidation.

[7] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: Cloud scale load balancing," in

ACM SIGCOMM Computer Communication Review, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[8] R. Nathuji, C. Isci, and E. Gorbatov, "Exploiting platform heterogeneity for power efficient data centers," in *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*. IEEE, 2007, pp. 5–5.

[9] X. Lin, Y. Mao, F. Li, and R. Ricci, "Towards fair sharing of block storage in a multi-tenant cloud," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 15–15.

[10] "Gartner special report." [Online]. Available: http://www.gartner.com/newsroom/id/2599315

[11] A. Mashtizadeh, E. Celebi, T. Garfinkel, M. Cai *et al.*, "The design and evolution of live storage migration in vmware esx."

[12] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai, "Workload-aware live storage migration for clouds," in *ACM Sigplan Notices*, vol. 46, no. 7. ACM, 2011, pp. 133–144.

[13] J. Zheng, T. S. E. Ng, K. Sripanidkulchai, and Z. Liu, "Comma: Coordinating the migration of multi-tier applications," in *ACM SIGPLAN Notices*, vol. 49, no. 7. ACM, 2014, pp. 153–164.

[14] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng, "Multi-level selective deduplication for vm snapshots in cloud storage," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 550–557.

[15] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Black-box and gray-box strategies for virtual machine migration." in *NSDI*, vol. 7, 2007, pp. 17–17.

[16] P. Riteau, C. Morin, and T. Priol, "Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 431–442.

[17] R. Zhou, F. Liu, C. Li, and T. Li, "Optimizing virtual machine live storage migration in heterogeneous storage environment," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 73–84.

[18] "Vmware vsphere replication." [Online]. Available: http://www.vmware.com/products/vsphere/features/replication

[19] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 13.

[20] N. Li, H. Jiang, D. Feng, and Z. Shi, "Pslo: enforcing the x th percentile latency and throughput slos for consolidated vm storage," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 28.

[21] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[22] M. Nelson, B.-H. Lim, G. Hutchins *et al.*, "Fast transparent migration for virtual machines." in *USENIX Annual technical conference, general track*, 2005, pp. 391–394.

[23] S. Nathan, U. Bellur, and P. Kulkarni, "On selecting the right optimizations for virtual machine migration," in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2016, pp. 37–49.

[24] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *ACM Sigplan Notices*, vol. 48, no. 7. ACM, 2013, pp. 41–50.

[25] R. Birke, M. Bjoerkqvist, L. Y. Chen, E. Smirni, and T. Engbersen, "(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)*, 2014, pp. 177–189.

[26] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[27] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," *ACM Sigplan Notices*, vol. 46, no. 7, pp. 111–120, 2011.

[28] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *Proceedings of the 18th ACM inter-*

national symposium on High performance distributed computing. ACM, 2009, pp. 101–110.

[29] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 51–60.

[30] K.-Y. Hou, K. G. Shin, and J.-L. Sung, "Application-assisted live migration of virtual machines with java applications," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 15.

[31] J.-H. Chiang, H.-L. Li, and T.-c. Chiueh, "Introspection-based memory de-duplication and migration," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 51–62.

[32] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen, "Parallelizing live migration of virtual machines," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 85–96.

[33] Y. Abe, R. Geambasu, K. Joshi, and M. Satyanarayanan, "Urgent virtual machine eviction with enlightened post-copy," 2015.

[34] X. Xu and B. Davda, "Srvm: Hypervisor support for live migration with passthrough sr-iov network devices," in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2016, pp. 65–77.

[35] C. Tang, "Fvd: A high-performance virtual machine image format for cloud." in *USENIX Annual Technical Conference*, 2011.

[36] I. Ahmad, A. Gulati, and A. Mashtizadeh, "vic: Interrupt coalescing for virtual machine storage device io," in *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, 2011, p. 45.

[37] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, "Multilanes: providing virtualized storage for os-level virtualization on many cores," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 317–329.

[38] "Understanding virtual machine snapshots in vmware esxi and esx." [Online]. Available: http://kb.vmware.com

[39] Y. Luo, B. Zhang, X. Wang, Z. Wang, Y. Sun, and H. Chen, "Live and incremental whole-system migration of virtual machines using block-bitmap," in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 99–106.

[40] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty, "Xvmotion: unified virtual machine migration over long distance," in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 97–108.

[41] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-level i/o scheduling," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 474–489.

[42] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: a decade of wasted cores," in *EuroSys 2016*, 2016.

[43] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *ACM Transactions on Storage (TOS)*, vol. 10, no. 1, p. 3, 2014.

[44] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha, "A logic of file systems." in *FAST*, vol. 5, 2005, pp. 1–1.

[45] D. Le, H. Huang, and H. Wang, "Understanding performance implications of nested file systems in a virtualized environment." in *FAST*, 2012, p. 8.

[46] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–11.

[47] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on nvm," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–14.

[48] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 257–270.

[49] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design tradeoffs for data deduplication performance in backup workloads," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 331–344.

[50] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "idedup: latency-aware, inline data deduplication for primary storage." in *FAST*, vol. 12, 2012, pp. 1–14.

[51] L. Cui, J. Li, B. Li, J. Huai, C. Hu, T. Wo, H. Al-Aqrabi, and L. Liu, "Vm-scatter: migrate virtual machines to many hosts," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 63–72.

[52] U. Deshpande, X. Wang, and K. Gopalan, "Live gang migration of virtual machines," in *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 135–146.

[53] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," *Communications of the ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[54] T. Lu, M. Stuart, and X. He, "Slm: Synchronized live migration of virtual clusters across data centers."

[55] J. Zheng, T. Ng, K. Sripanidkulchai, and Z. Liu, "Pacer: A progress management system for live virtual machine migration in cloud computing," *Network and Service Management, IEEE Transactions on*, vol. 10, no. 4, pp. 369–382, 2013.

[56] K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble, "qufiles: The right file at the right time," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 12, 2010.

[57] "Iometer." [Online]. Available: http://www.iometer.org

[58] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, and A. Warfield, "Characterizing storage workloads with counter stacks," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 335–349.

[59] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter, "How to get more value from your file system directory cache," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 441–456.

[60] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 24, no. 1. ACM, 1996, pp. 126–137.

[61] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential consistency: measuring and understanding consistency at facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 295–310.

[62] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown, "Recon: Verifying file system consistency at runtime," *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, p. 15, 2012.

[63] A. Thomson and D. J. Abadi, "Calvinfs: consistent wan replication and scalable metadata management for distributed file systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 1–14.

[64] "Qemu system main page." [Online]. Available: http://wiki.qemu.org/Main_Page

[65] Q. Yang and J. Ren, "I-cash: Intelligently coupled array of ssd and hdd," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 278–289.

[66] "Umass storage trace repository." [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[67] A. Miranda and T. Cortes, "Craid: online raid upgrades using dynamic hot data reorganization." in *FAST*, 2014, pp. 133–146.

[68] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao, "Workout: I/o workload outsourcing for boosting raid reconstruction performance." in *FAST*, vol. 9, 2009, pp. 239–252.

[69] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song, "Pro: A popularity-based multi-threaded reconstruction optimization for raid-structured storage systems." in *FAST*, vol. 7, 2007, pp. 301–314.

[70] L. Cui, B. Li, Y. Zhang, and J. Li, "Hotsnap: A hot distributed snapshot system for virtual machine cluster." in *LISA*, 2013, pp. 59–74.

[71] J. Li, H. Liu, L. Cui, B. Li, and T. Wo, "irow: An efficient live snapshot system for virtual machine disk," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE, 2012, pp. 376–383.

[72] W. Xiao, Q. Yang, J. Ren, C. Xie, and H. Li, "Design and analysis of block-level snapshots for data protection and recovery," *Computers, IEEE Transactions on*, vol. 58, no. 12, pp. 1615–1625, 2009.

[73] "Emc recoverpoint for virtual machines gives administrators more agility and control." [Online]. Available: https://www.emc.com/collateral/analyst-reports/esg-wp-emc-recoverpoint-for-vms.pdf

[74] Q. Chen, L. Liang, Y. Xia, H. Chen, and H. Kim, "Mitigating sync amplification for copy-on-write virtual disk," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*. USENIX Association, 2016, pp. 241–247.

[75] R. Zhou, F. Liu, C. Li, and T. Li, "Optimizing virtual machine live storage migration in heterogeneous storage environment," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 73–84.

[76] Y. Yang, H. Jiang, B. Mao, L. Tian, Y. Yang, and J. Qian, "Waio: Improving virtual machine live storage migration for the cloud by workload-aware io outsourcing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 314–321.

[77] P. Riteau, C. Morin, and T. Priol, "Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 431–442.

[78] Y. Abe, R. Geambasu, K. Joshi, and M. Satyanarayanan, "Urgent virtual machine eviction with enlightened post-copy," in *ACM SIGPLAN Notices*. ACM, 2016.

[79] L. Cui, T. Wo, B. Li, J. Li, B. Shi, and J. Huai, "Pars: A page-aware replication system for efficiently storing virtual machine snapshots," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2015, pp. 215–228.

[80] L. Cui, J. Li, T. Wo, B. Li, R. Yang, Y. Cao, and J. Huai, "Hotrestore: a fast restore system for virtual machine cluster," in *Proceedings of the 28th USENIX conference on Large Installation System Administration*. USENIX Association, 2014, pp. 1–16.

[81] W. Zhang, T. Yang, G. Narayanasamy, and H. Tang, "Low-cost data deduplication for virtual machine backup in cloud storage," in *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems, HotStorage*, vol. 13, 2013, pp. 2–2.

[82] "Kernel based virtual machine," http://www.linux-kvm.org/page/Main_Page.

[83] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Fast*, vol. 8, 2008, pp. 1–14.

[84] "libvirt: The virtualization api toolkit," http://libvirt.org/index.html.

[85] "Fio," https://github.com/axboe/fio.

[86] C. Li, P. Shilane, F. Douglis, D. Sawyer, and H. Shim, "Assert (! defined (sequential i/o))." in *HotStorage*, 2014.

[87] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 0th ed. Arpaci-Dusseau Books, May 2015.

[88] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 183–197.

[89] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality," in

Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, vol. 4, 2005, pp. 8–8.

[90] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.

[91] A. Wildani, E. L. Miller, and L. Ward, "Efficiently identifying working sets in block i/o streams," in *Proceedings of the 4th Annual International Conference on Systems and Storage*. ACM, 2011, p. 5.

[92] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[93] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system." in *USENIX Annual Technical Conference*, vol. 15, 1996.

[94] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[95] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[96] "Three reasons why reiserfs is great for you," accessed: 2016-03-14. [Online]. Available: https://reiser4.wiki.kernel.org/index.php/X0reiserfs

[97] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[98] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[99] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao, "Second-tier cache management using write hints." in *FAST*, vol. 5, 2005, pp. 9–9.

[100] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Database-aware semantically-smart storage." in *FAST*, vol. 5, 2005, p. 18.

[101] "Vmware vsphere platform," accessed: 2016-03-14. [Online]. Available: http://www.vmware.com/products/vsphere

[102] "Vmware virtual machine file system overview." accessed: 2016-03-14. [Online]. Available: https://www.vmware.com/pdf/vmfs-best-practices-wp.pdf

[103] "Tintri vmstore storage system," accessed: 2016-03-14. [Online]. Available: https://www.tintri.com

[104] "Qemu storage stack," accessed: 2016-05-10. [Online]. Available: https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_hajnoczi.pdf

[105] "Storage management disciplines are declining," accessed: 2016-03-14. [Online]. Available: http://www.computereconomics.com/article.cfm?id=1129

[106] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," *ACM Transactions on Storage (TOS)*, vol. 3, no. 3, p. 9, 2007.

[107] "50accessed: 2016-03-14. [Online]. Available: http://prsync.com/seagate/-storage-utilization-are-data-centers-half-empty-or-half-full-275/

[108] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson, "Trading capacity for performance in a disk array," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000, pp. 17–17.

[109] S. Bauer and N. B. Priyantha, "Secure data deletion for linux file systems." in *Usenix Security Symposium*, vol. 174, 2001.

[110] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. Soules, G. R. Goodson, and G. R. Ganger, *Storage-based intrusion detection: Watching storage activity for suspicious behavior*. School of Computer Science, Carnegie Mellon University, 2002.

[111] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Improving storage system availability with d-graid," *ACM Transactions on Storage (TOS)*, vol. 1, no. 2, pp. 133–170, 2005.

[112] Y. Yue, L. Tian, H. Jiang, F. Wang, D. Feng, Q. Zhang, and P. Zeng, "Rolo: A rotated logging storage architecture for enterprise data centers," in *2010 International Conference on Distributed Computing Systems*. IEEE, 2010, pp. 293–304.

[113] Y. Lee, J.-S. Kim, S.-W. Lee, and S. Maeng, "Zombie chasing: Efficient flash management considering dirty data in the buffer cache," *Computers, IEEE Transactions on*, vol. 64, no. 2, pp. 569–581, 2015.

[114] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Life or death at block-level." in *OSDI*, vol. 4, 2004, pp. 26–26.

[115] F. Shu and N. Obr, "Data set management commands proposal for ata8-acs2," *Management*, vol. 2, p. 1, 2007.

[116] J. Kim, H. Kim, S. Lee, and Y. Won, "Ftl design for trim command," in *The Fifth International Workshop on Software Support for Portable Storage*, 2010, pp. 7–12.

[117] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-smart disk systems." in *FAST*, vol. 3, 2003, pp. 73–88.

[118] R. Koller and R. Rangaswami, "I/o deduplication: Utilizing content similarity to improve i/o performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.

[119] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds." in *FAST*, 2011, pp. 91–103.

[120] F. Chen, T. Luo, and X. Zhang, "Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives." in *FAST*, vol. 11, 2011.

[121] T. Wood, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe, "Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines," in *ACM Sigplan Notices*, vol. 46, no. 7. ACM, 2011, pp. 121–132.

[122] Q. Yang, W. Xiao, and J. Ren, "Trap-array: A disk array architecture providing timely recovery to any point-in-time," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 289–301.

[123] G. Wu and X. He, "Delta-ftl: improving ssd lifetime via exploiting content locality," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 253–266.

[124] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication—large scale study and system design," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 285–296.

[125] M. A. Lehmann, "Liblzf," accessed: 2016-05-10. [Online]. Available: http://oldhome.schmorp.de/marc/liblzf.html