RICE UNIVERSITY

Workload-Aware Live Storage Migration for Clouds

by

Jie Zheng

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

[illegible]

Dr. T. S. Eugene Ng (Chair),
Assistant Professor,
Computer Science

[illegible]

Dr. Alan L. Cox,
Associate Professor,
Computer Science

[illegible]

Dr. Edward W. Knightly
Professor,
Electrical and Computer Engineering

[illegible]

Dr. Kunwadee Sripanidkulchai
Research Staff Member,
IBM T.J. Watson Research Center

HOUSTON, TEXAS

MARCH, 2010

ABSTRACT

Workload-Aware Live Storage Migration for Clouds

by

Jie Zheng

The emerging open cloud computing model will provide users with great freedom to dynamically migrate virtualized computing services to, from, and between clouds over the wide-area. While this freedom leads to many potential benefits, the running services must be minimally disrupted by the migration. Unfortunately, current solutions for wide-area migration incur too much disruption as they will significantly slow down storage I/O operations during migration. The resulting increase in service latency could be very costly to a business. This thesis presents a novel storage migration scheduling algorithm that can greatly improve storage I/O performance during wide-area migration. Our algorithm is unique in that it considers individual virtual machine's storage I/O workload such as temporal locality, spatial locality and popularity characteristics to compute an efficient data transfer schedule. Using a trace-driven framework, we show that our algorithm provides large performance benefits across a wide range of popular virtual machine workloads.

Acknowledgments

My foremost thank goes to my advisor Professor T. S. Eugene Ng. I thank him for all of his help, inspiration and guidance in my graduate study. He is the best advisor I can ever imagine. I thank him for his patience and encouragement that always carried me through difficult times, and for his insights and suggestions that helped to shape my research skills. His passion for science has influenced me a lot. His valuable feedback contributed greatly to this thesis.

I wish to express my sincere gratitude to my mentor Dr. Kunwadee (Kay) Sripanidkulchai at IBM Research. I had the fortune to work with Kay during my summer internship at IBM. She helped me on every aspect of the research related to this thesis. She taught me a vast amount of knowledge in the areas of cloud computing and machine virtualization and introduced me to many advanced techniques. I really appreciate her sound advice, good company, interesting ideas and suggestions. Without the help from Eugene and Kay, this thesis would not have been possible.

I wish to thank Professor Alan L. Cox who helped me setup the experiment environment and suggested me to use VMmark to explore the I/O patterns. Before this thesis, I had the opportunity to work with Alan on another project. Alan is a very knowledgeable and friendly professor. I am often impressed by his logical thoughts and wise solutions to difficult research questions.

I want to thank Professor Edward W. Knightly for serving on my M.S. thesis committee and asking many insightful questions that helped to shape this thesis.

I also want to thank many friends in our research group. They are Bo Zhang, Guohui Wang, Zheng Cai and Florin Dinu. I enjoyed all the vivid discussions we had on various topics and had lots of fun being a member of this fantastic group. They always gave me instant help when I asked.

Last but not the least, I would like to thank my parents and my best friends who have supported me spiritually throughout my life.

Contents

Illustrations

Chapter 1

Introduction

Cloud computing has recently attracted significant attention from both industry and academia for its ability to deliver IT services at a lower barrier to entry in terms of cost, risk, and expertise, with higher flexibility and better scaling on-demand. Many cloud early adopters have had great successes in leveraging these capabilities to deliver services much faster than any of these users could have achieved if they had to build out their own infrastructure [1, 2]. While these successes have been realized through using a single cloud provider, using multiple clouds to deliver services and having the flexibility to move freely among different providers is an emerging requirement [3]. The Open Cloud Manifesto is an example of how users and vendors are coming together to support and establish principles in opening up choices in cloud computing [4]. A key barrier to cloud adoption identified in the manifesto is data and application portability, particularly once users have implemented their applications using one cloud provider, they ought to be able to migrate that system back in-house or to other cloud providers. Flexibility in migration allows users to have control over business continuity and avoid fate-sharing with specific providers.

In addition to avoiding single-provider lock-in, there are other availability and economic reasons driving the requirement for migration across clouds. To maintain high performance and availability, migrations could be used to move virtual machines from one cloud to another cloud that has better resource availability, to avoid hardware or network maintenance down-times, or to avoid power limitations in the source cloud. Also, moving work out of providers that could be shut down by anticipated natural disasters such as hurricanes or winter storms prior to such disasters is also useful for maintaining high service availability. Furthermore, cloud users may want to move work to clouds that provide lower-cost. The current practice for migration causes significant transitional down time. In order for users to realize the benefits of migration between clouds, we need both open interfaces and mechanisms to enable such migration while the services are running with as minimal service disruption as possible. While providers are working towards open interfaces, in this thesis we look at the enabling mechanisms without which migrations would remain a costly effort.

Live migration provides the capability to move virtual machines from one physical location to another while still running without any perceived degradation. Many hypervisors support live migration within the LAN [5, 6, 7, 8, 9, 10]. However, migrating across the wide area presents more challenges specifically because of the large amount of data that needs to be migrated over limited network bandwidth. In order to enable live migration over the wide area, three capabilities are needed: (i) the running state of the virtual machine must be migrated (i.e., memory migration), (ii) the storage or virtual disks used by the virtual machine must be migrated, and (iii) existing client connections must be migrated while new client connections are directed to the new location. Memory migration techniques have been extensively used in the local area and can be extended to work well in the wide area [11]. Existing client connections can be seamlessly migrated through the use of LAN extension technologies such as L2TP, VPLS and VPNs [12], or layer 3 solutions such as tunneling, MobileIP, and IPv6. New clients can be quickly redirected to the new location using DNS. Neither wide area memory nor network connection migration will result in significant performance degradation. However, storage migration inherently faces significant performance challenges because of its much larger size compared to memory.

The contributions of this thesis are as follows:

- Current solutions for wide-area storage migration incur too much disruption, because they are agnostic to I/O workload. We identify this problem and use quantitative experiment results to show the existence of significant performance degradation in the existing storage migration approaches.

- We diverge from the existing work in storage migration that treats storage as one large chunk that needs to be transferred sequentially. In this thesis, the notion of storage migration scheduling is introduced to orchestrate the sequence in which storage is transferred. Scheduling allows us to take advantage of inherent access patterns such as temporal locality, spatial locality, and access popularity that are found in a wide range of I/O workloads to significantly optimize the data transfer and reduce performance degradation. We develop a novel workload-aware storage migration scheduling algorithm. Our algorithm uses only simple records of a limited number of past I/O operations for workload characteristic inference. It automatically decides proper storage granularity and migration schedule to leverage I/O locality and popularity characteristics while minimizing overhead.

- We use a trace-driven framework to demonstrate how our scheduling algorithm can be leveraged by all proposed migration models to greatly improve storage I/O performance during migration. The benefits are substantial across a wide variety of virtual machine workloads and migration scenarios.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of the existing storage migration technologies and the challenges that they face. Chapter 3 explains how we collect virtual machine storage workload traces for this study. We quantify the locality and popularity characteristics we found in the traces in Chapter 4. Motivated by these characteristics, we present in Chapter 5 a novel storage migration scheduling algorithm that leverages these characteristics to make storage migration much more efficient. In Chapter 6, we explain our evaluation methodology and present results to show that our algorithm is able to provide large performance benefits. Finally, we summarize our findings and discuss future work in Chapter 7.

Chapter 2

Background

2.1 Challenges in Wide Area Migration

A virtual machine (VM) consists of virtual hardware devices such as CPU, memory and disk that are used to run an operating system. Live migration of VMs is a common operation in the local area that involves transferring the running state or *memory migration* of the VM from one hypervisor to another. However, live migration across the wide area entails a few more steps including transferring the running state, persistent storage and network connections associated with a VM.

Migration of the running state, or memory migration, starts with the source hypervisor taking a snapshot of the memory and CPU state. While the snapshot is copied over to the destination hypervisor, the VM continues to run. The source hypervisor tracks and transfers dirty memory pages and CPU state until it freezes the VM for a short time to finish the transfer. The VM is then re-continued at the destination hypervisor. Memory migration across the wide area has been shown to have no impact on running services [11].

To maintain liveness, network connections must also be maintained. Commercially available LAN extension technologies (L2TP, VPLS, VPN) allow the same IP address space to be used across the wide area so the relocated VM can use the same IP address even in its new location [12]. Alternatively, if an IP address change is required, tunneling traffic from the source to the destination or MobileIP/IPv6 can be used to provide seamless handoff [11, 13, 14].

While wide area memory and network connection migration work well, storage migration inherently faces significant performance challenges. Migration of persistent storage, or storage migration, is required because the VM needs access to its disk in its new location. The VM's disk is implemented as a (set of) file(s) stored on the physical disk. Live migration in the LAN may not require storage migration because the virtual disks are often located on shared storage accessible at high speed by both source and destination hypervisors. However, sharing storage across the wide area will bring unacceptable performance. Because of the larger storage size compared to memory, and the limitations in wide area bandwidth, storage migration could impact VM performance if not migrated efficiently.

2.2 Storage Migration Models

Previous work in storage migration can be classified into three migration models: pre-copy, post-copy and pre+post-copy. In the pre-copy model, storage migration is performed *prior* to memory migration whereas in the post-copy model, the storage migration is performed *after* memory migration. The pre+post-copy model is a hybrid of the first two models.

Figure 2.1 depicts the three models. In the pre-copy model [11], the entire virtual disk file is copied block-by-block from beginning to end prior to memory migration. During the virtual disk copy and memory migration, all write operations to the disk are logged and the dirty blocks are retransmitted as necessary. The strength of the pre-copy model is that blocks are copied over prior to when the VM runs in the destination. However, there are two scenarios in which the pre-copy model has weaknesses. First, pre-copying may introduce excessive extra traffic. If we had an oracle that told us when disk blocks are updated, we could come up with an ideal schedule to send only the latest copy of disk blocks rather than transmitting stale copies. Thus, the total number of disk bytes transferred over the network would be the minimum possible which is the total size of the virtual disk[1]. Without an oracle, we will need to transmit some stale blocks resulting in *extra traffic* beyond the size of the virtual disk. Second, if the I/O workload on the VM is write-intensive, write-throttling is employed to slow down I/O to ensure that storage migration can complete. While throttling is useful, it can degrade application I/O performance. We discuss how to

[1]For simplicity, we assume no data compression is performed.

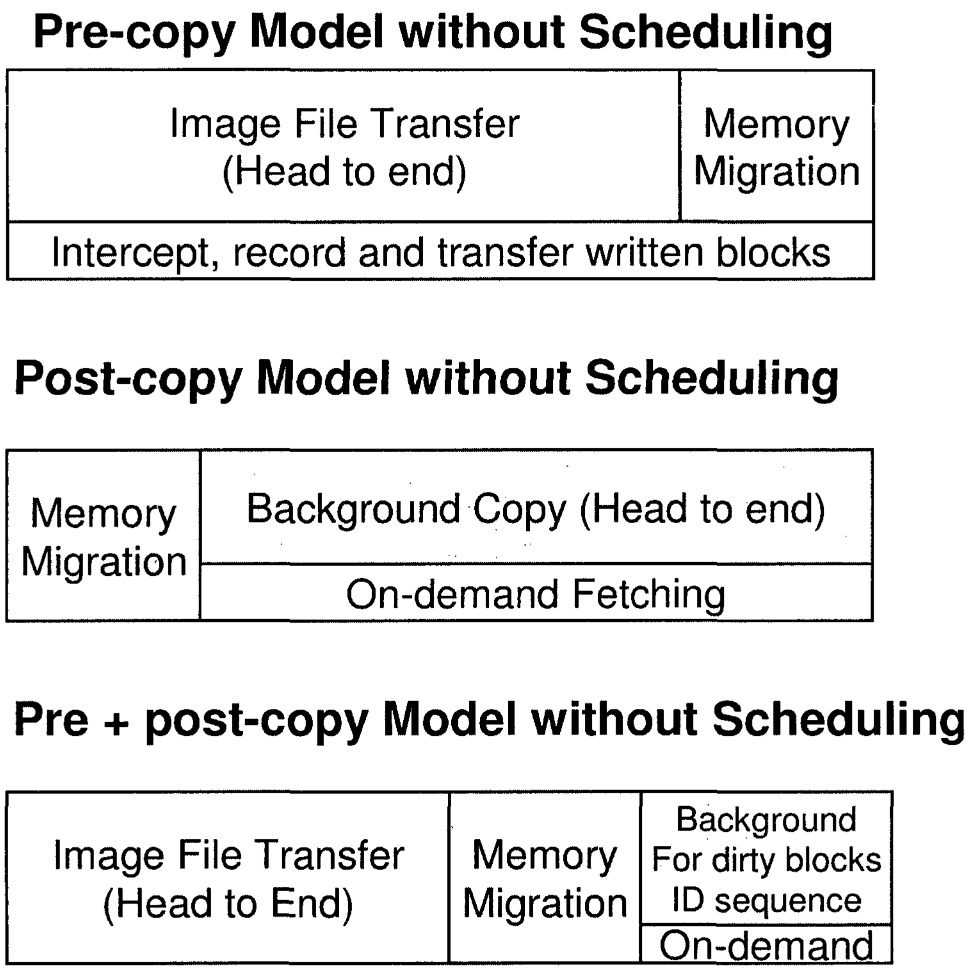

Pre-copy Model without Scheduling
Image File Transfer (Head to end)
Memory Migration
Intercept, record and transfer written blocks
Post-copy Model without Scheduling
Memory Migration
Background Copy (Head to end)
On-demand Fetching
Pre + post-copy Model without Scheduling
Image File Transfer (Head to End)
Memory Migration
Background For dirty blocks ID sequence
On-demand

Figure 2.1 : Models of live storage migration.

improve both weaknesses using our scheduling approach in Chapter 5.

In the post-copy model [15, 16] depicted in Figure 2.1, storage migration is executed after memory migration completes and the VM is running at the destination. Two mechanisms are used to copy disk blocks over: background copying and on-demand fetching. All of the virtual disk blocks are copied in the background from beginning to end. However, during this time if the VM issues an I/O request, it is handled immediately. If the VM issues a write operation, the blocks are directly updated at the destination storage. If the VM issues a read operation and the blocks have yet to arrive at the destination, then on-demand fetching is employed to request those blocks from the source. We call such operations *remote reads*. With the combination of background copying and on-demand fetching, each block is transferred only once ensuring that the total amount of data transferred for storage migration is the minimum which is the virtual disk size. However, remote reads incur extra wide-area delays, resulting in I/O performance degradation.

In the hybrid pre+post-copy model [17], the virtual disk is copied to the destination prior to memory migration. During disk copy and memory migration, a bit-map of dirty disk blocks is maintained. After memory migration completes, the bit-map is sent to the destination where a background copying and on-demand fetching model is employed for the dirty blocks. This model combines the previous two models. While it still incurs extra traffic and remote read penalties, the amount of extra traffic is smaller compared to the pre-copy model and the number of remote reads is smaller compared to the post-copy model. Table 2.1 summarizes these three models.

2.3 Performance Degradation from Migration

While migration is a powerful capability, any performance degradation caused by wide area migration could be damaging. Users are extremely sensitive to latency. For example, every 100 ms of latency costs Amazon 1% in sales and an extra 500 ms page generation time dropped 20% of Google's traffic [18].

In order to better understand the impact of migration on performance, we look at an example migration of a 10 GB MySQL database server that has 160 clients over a 10 Mbps wide area link. The details of the experimental set up are described in Chapter 6. If we were to migrate the server using pre-copying, we would see half of the write I/O operations during the migration postponed due to throttling for an average duration of around 75 minutes. On the other hand, if we were to migrate the server using post-copying, 2 millions blocks requested in the read operations during storage migration would be remote reads across the wide area. I/O performance degradation during migration can be significant. As a result, applications running on the migrated VMs also see degraded performance. Improving the

Model		Pre-copy [11]	Pre+post-copy [17]	Post-copy [15, 16]	w/ Scheduling
Granularity		I/O Operations	Blocks	Blocks	Chunks
Application Performance Impact	Write Operation Degradation	Yes	No	No	No
	Read Operation Degradation	No	Medium	Heavy	Small
	Degradation Time	Long	Medium	Long	Small
	I/O Operations Throttled	Yes	No	No	No
Total Migration Time		>>> Baseline	>> Baseline	Baseline	Slightly> Baseline
Amount of Migrated Data		>>> Baseline	>> Baseline	Baseline	Slightly> Baseline

Table 2.1 : Comparison of VM storage migration methods.

performance degradation is key to making live migration an attractive mechanism to move applications across clouds.

Our approach to improve storage migration relies on the notion of workload-aware storage scheduling. Rather than copying the storage from beginning to end, we compute a schedule to transfer storage at the appropriate granularity which we call *chunks* at the appropriate time to minimize performance degradation. Our schedule is computed to take advantage of the individual I/O locality characteristics of the particular workload to be migrated and can be applied to improve any of the three storage migration models as depicted on the right-hand side of Figure 2.1. To improve the pre-copy model, scheduling is used to group the storage blocks into chunks and send the chunks to the destination in an improved order instead of just blindly sending from beginning to end. Similarly, to improve the post-copy model, scheduling is used to group and order the scheduling of storage blocks sent over using background copying. In the hybrid pre+post-copy model, scheduling is used for both the pre-copy phase and the post-copy phase. The benefits of scheduling are summarized in Table 2.1.

Chapter 3

Workload Trace Collection

To investigate the storage migration scheduling problem, we collect and study a modest set of VM I/O traces. These traces are based on the workloads in the VMware VMmark virtualization benchmark [19] widely used by major computer system vendors to measure system performance.

VMmark includes five types of servers that are representative of the applications run by VMware users, including mail server, file server, web server, Java server, and database server listed in Table 3.1. VMmark also includes a "standby" server which is not included in our study as it has no associated I/O workload. For each server type, we collect traces for multiple client workload intensities by varying the number of VMmark client threads. We refer to the specific traces collected by the workload name and number of client threads, for example, "fs-45" refers to the file server workload with 45 client threads. The default number of client threads specified by VMmark are listed in the table.

Our trace collection platform consists of two physical machines, each configured with a 3GHz Quadcore AMD Phenom II 945 processor and 8GB of DRAM. One machine runs the server application while the other runs the VMmark client. The server is run as a VM on a VMware ESXi 4.0 hypervisor. The configuration of the server VM and the client is as specified by VMmark.

In order to collect the trace of I/O operations, we run an NFS server as a VM on the application server physical machine and mount it on the ESXi hypervisor. The application server's virtual disk is then placed on the NFS storage as a VMDK flat format file. Subsequently, we use ***tcpdump*** to log the NFS requests that correspond to virtual disk I/O accesses. NFS-based tracing has been used in past studies of storage workload [20, 21] and

Workload Name	VM Configuration	Server Application	# Clients
File Server (fs)	SLES 10 32-bit 1 CPU,256MB RAM,8GB disk	dbench	45
Mail Server (ms)	Windows 2003 32-bit 2 CPU,1GB RAM,24GB disk	Exchange 2003	1000
Java Server (js)	Windows 2003 64-bit 2 CPU,1GB RAM,8GB disk	SPECjbb @2005-based	8
Web Server (ws)	SLES 10 64-bit 2 CPU,512MB RAM,8GB disk	SPECweb @2005-based	100
Database Server (ds)	SLES 10 64-bit 2 CPU,2GB RAM,10GB disk	MySQL	16

Table 3.1 : VMmark workload summary.

has the advantage of not requiring any special operating system instrumentation. Note that the I/O requests do not actually go over the network since the NFS server and application server are VMs running on the same physical machine. We trace I/O operations at the disk sector level, which has a granularity of 512 bytes. For convenience, we call each 512 byte sector a block. This is, however, not to be confused with the file system block size, which could vary depending on user configuration.

In the trace file, each I/O access entry includes the time of the access, the offset in the VMDK file, and the data length for read or write operations. In each experiment, we trace the I/O operations for 12 hours. We do not use the first 10 minutes and the last 10 minutes of each trace to avoid effects relating to the ramp up and ramp down stages of the benchmarks.

To confirm that the NFS indirection and tracing does not degrade the application server's performance, we perform each experiment twice, once with the virtual disk located on NFS and once with the virtual disk located on the hypervisor's locally attached disk. We compare the average throughput reported by the application server. We find that by allocating all the DRAM left over by the application server VM to the NFS server VM, the average throughput of the NFS case becomes comparable to or better than the locally-attached disk case. Therefore, no major performance degradation is introduced by the methodology.

Chapter 4

Workload Characteristics

This chapter reports the temporal locality, spatial locality, and popularity characteristics we find in the collected traces. At a high level, our observations corroborate similar observations made in previous studies of other storage workloads [22, 23, 24]. This gives us confidence that the observations are quite general rather than being unique to our traces. What is different is that our analysis is tailored specifically to the time-scales and conditions relevant to storage migration.

4.1 Methodology

In order to understand if the history of past I/O accesses are useful at predicting future accesses, specifically leveraging various types of locality, we analyze the first three hours of our collected traces. Let t denote the start time for the migration. Accesses prior to t can be used as history. Accesses from t onwards up to a maximum migration time are considered as accesses that happen during migration. The maximum migration time is defined as the amount of time to copy the VM to the destination assuming the worst case scenario when during the copy, all the blocks were written to and the entire image needs to be retransmitted, $max\_migration\_time = (2 \times image\_size + memory\_size) \ / \ bandwidth$. We use the image sizes and default number of client threads specified in Table 3.1 and a bandwidth of 100 Mbps throughout this chapter. Note that to simulate complete migrations within the 3 hour segment, the migration start time t is randomly selected from $[3000, 5000]$ seconds. Each analysis is performed 20 times with different migration starting time t. We also use a fixed history period of 3000 seconds before migration starts.

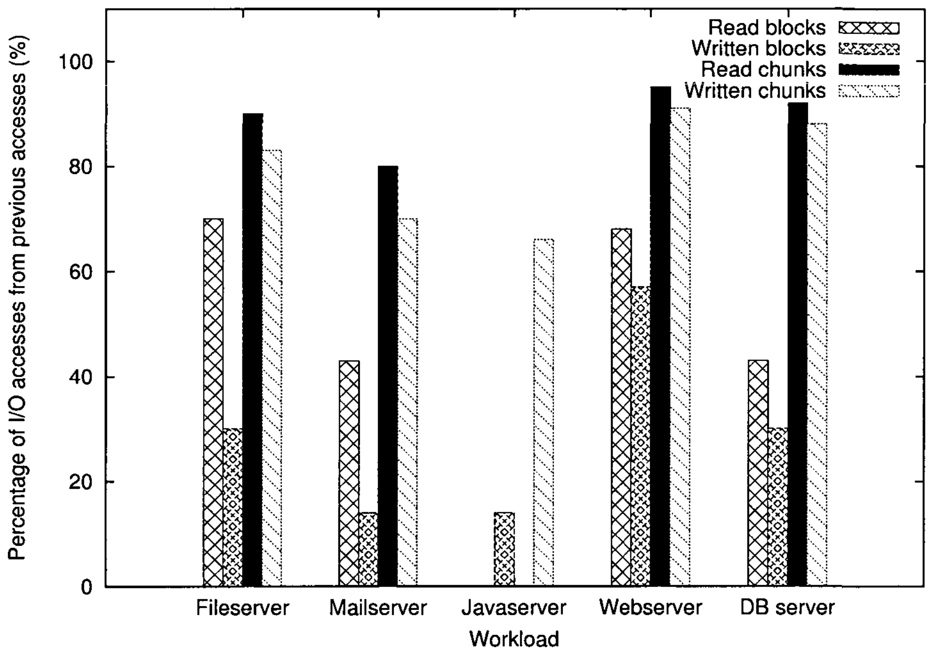


Figure 4.1 : The temporal locality of I/O accesses as measured by the percentage of accesses in the migration that was also previously accessed in the history. The block size is 512 B and the chunk size is 1MB. Temporal locality exists in all of the workloads, but is stronger at the chunk level. The Java server has very few read accesses resulting in no measurable locality.

4.2 Temporal Locality Characteristics

Figure 4.1 shows that, across all workloads, blocks that are read during the migration are often also the blocks that were read in the history. Take the file server as an example, 72% of the blocks that are read in the migration were also read in the history. Among these blocks, 96% of them are blocks whose read access frequencies were $\geq$ 3 in the history. These figures are significant because the file server does not actually read that much data in the disk. As shown in Figure 4.2, less than 15% of the overall storage blocks are read in history and less than 10% of the overall storage blocks are read in the migration. Thus, it is possible to predict which blocks are more likely to be read in the near future by analyzing the recent past history.

However, write accesses do not behave like the read accesses. Write operations tend

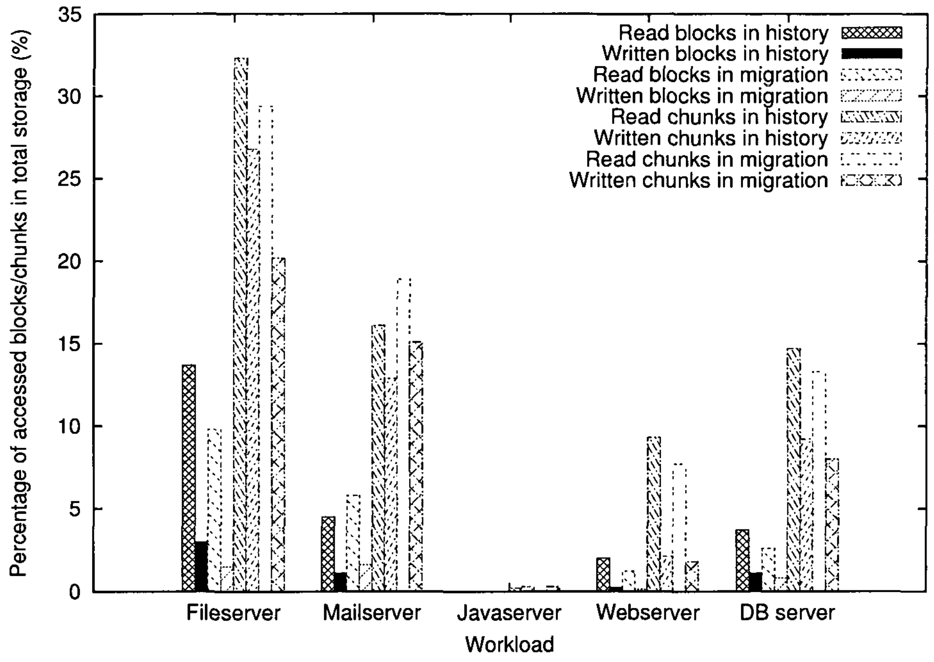


Figure 4.2 : The percentage of storage accessed. The block size is 512 B and the chunk size is 1 MB.

to access new blocks that have not been written before. Again, take the file server as an example. Only 32% of the blocks that are written in the migration were written in history. Therefore, simply counting the write accesses in history is not enough to predict the write accesses in migration.

Note that the temporal locality improves dramatically when 1MB chunk is used as the basic unit of counting accesses. We will explain this finding next.

4.3 Spatial Locality Characteristics

Although many written blocks during migration were not written in history, we find that most of them are located near the written blocks in history. That is, strong spatial locality exists for write accesses.

Again, take the file server as an example. For the 68% of the blocks that are freshly written in migration but not in history, we compute the distance between each of these

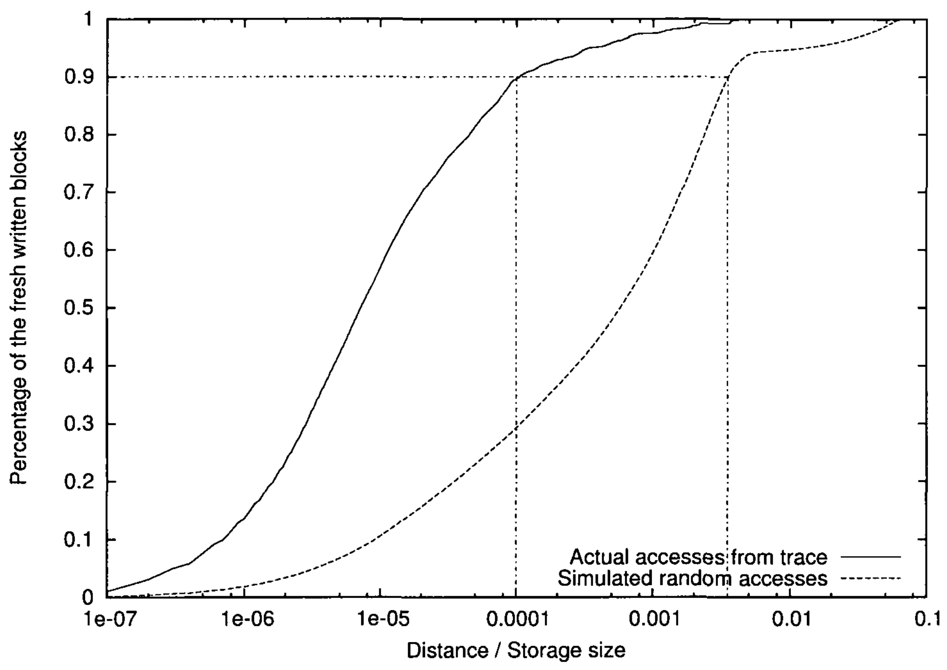


Figure 4.3 : File server spatial locality. The fresh written blocks in migration are located very close to the written blocks in history, suggesting strong spatial locality.

blocks and its closest neighbor block that was written in history. The distance is defined as $(block\_id\_difference * blocksize)$. Figure 4.3 plots, for the file server, the cumulative percentage of the fresh written blocks versus the closest neighbor distance normalized by the storage size (8GB). For all the fresh written blocks, their closest neighbors can be found within a distance of 0.0045*8GB=36.8MB. For 90% of the cases, the closest neighbor can be found within a short distance of 0.0001*8GB=839KB. For comparison, we also plot the results for simulated random write accesses, and as can be seen, the spatial locality found in the real trace is far stronger. The 90th percentile is 0.0035, which is 35 times farther than the 90th percentile of the real trace. Taken together, in the file server example, $32\% + 68\% * 90\% = 93.2\%$ of the written blocks in the migration are found within a range of 839KB of the written blocks in history.

This spatial locality explains why, across all workloads, the temporal locality of write accesses increases dramatically in Figure 4.1 when we consider 1MB chunk instead of

512B block as the basic unit of counting accesses. Also, as can be seen, the temporal locality of read accesses also increases.

The caveat is that as the chunk size increases, the percentage of covered accessed blocks in migration will no doubt increase, but each chunk will also cover more unaccessed blocks. In the extreme case, the whole virtual disk becomes a single chunk. Therefore, to provide useful read and write access prediction, a balanced chunk size is necessary and will depend on the workload. We will return to this chunk size selection issue in Chapter 5.

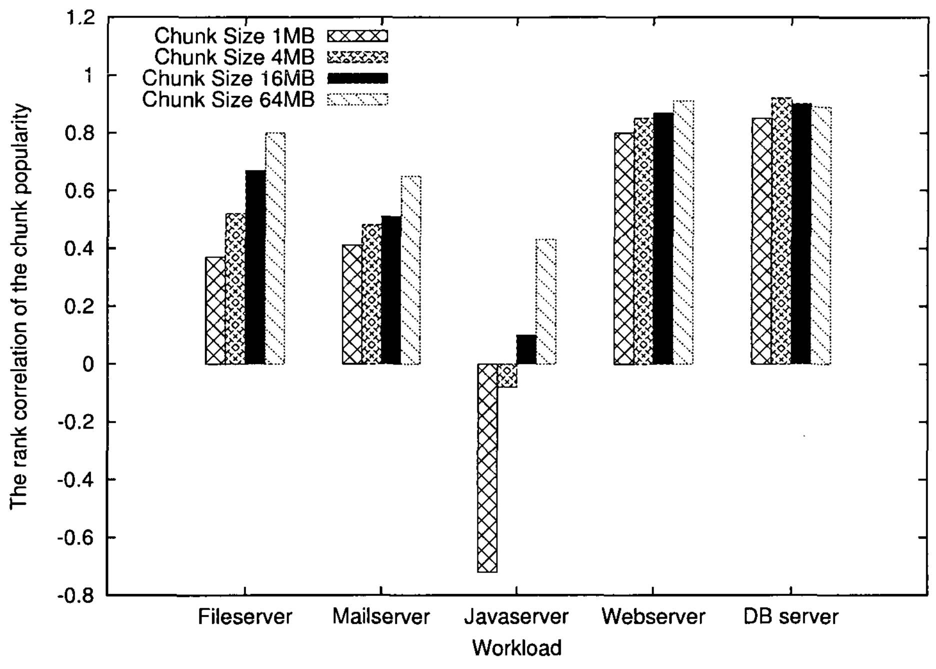
4.4 Popularity Characteristics

Another useful property we find is that if a particular chunk is popular in history, it is likely to be popular in migration. To illustrate this, we count the read/write access frequency for each chunk in history and in migration and rank the chunks based on the read/write frequencies. Then, we compute the rank correlation between the ranking in history and the ranking in migration. Figure 4.4 shows that a positive correlation exists for most cases except for the Java server read accesses at 1MB and 4MB chunk sizes. This is because the Java server has extremely few read accesses and little read locality. As the chunk size increases, the rank correlation increases. This increase is expected since if the chunk size is set to the size of the whole storage, the rank correlation will become 1 by definition. Again, a balanced chunk size is required to exploit this popularity characteristic effectively.

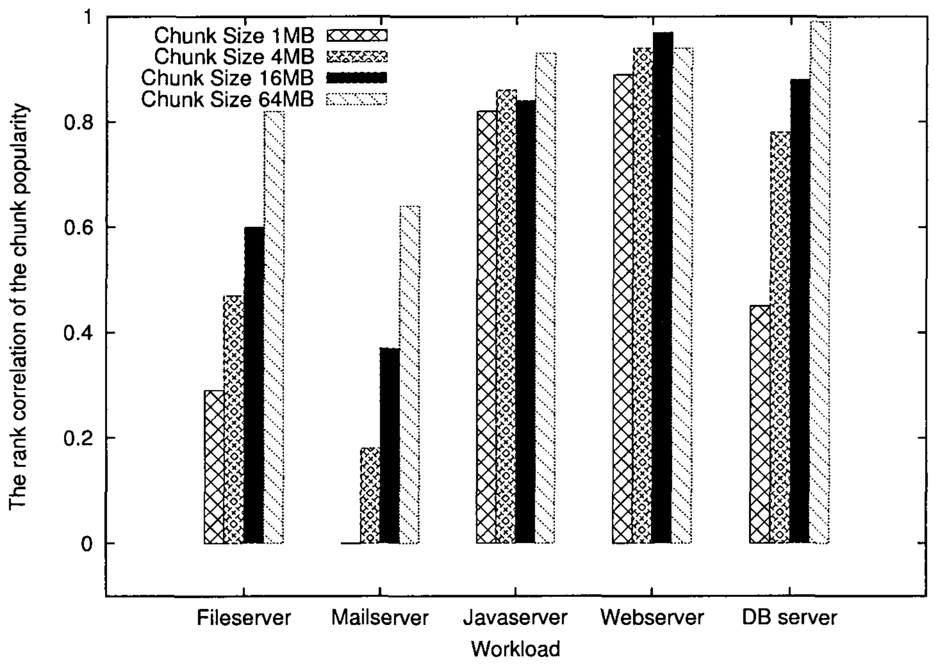
4.5 Effects of File System on Workload Characteristics

When we explored the workload characteristics, we treated the virtual disk as a sequence of blocks, no matter what kind of file system the virtual machines run on. The methodology is not specific for certain types of file system. However, we know that some aspects of the file systems, such as block allocation mechanism and caching mechanism, may affect the characteristics.

Temporal locality and popularity are not affected by the block allocation mechanism.



(a) Read Access



(b) Write Access

Figure 4.4 : The rank correlation of the chunk popularity in history vs. in migration.

The frequently accessed blocks and their access frequency are mostly decided by the applications' behavior. For example, disk blocks that store the source code files and the invoked libraries of the applications are generally the frequently read blocks. No matter what the block allocation mechanism is, the timing and the number of the accesses to these blocks remain the same.

Spatial locality is related to the block allocation mechanism in the file system. Spatial locality exists in most file systems for two reasons. First, the applications often tend to access a particular region of a file or tend to access a file sequentially. Second, file systems tend to allocate contiguous blocks to a file. These mechanisms help improve I/O performance. ReiserFS and NTFS, which are the file systems used in the VMmark, leverage different approaches to achieve the goal of local grouping. ReiserFS assigns each file or directory a unique key. The files or directories whose key values lie closely together are assigned block numbers that are also close together [25]. NTFS manages block storage in clusters. Each cluster is a group of consecutive sectors. When NTFS wants to create a new file, it will look into its Master File Table(MFT) for free clusters and run a best-fit algorithm to allocate contiguous blocks to the file to minimize the file system fragmentation [26]. Besides ReiserFS and NTFS, many popular file systems have the similar mechanisms to achieve local grouping. For example, the EXT2 and EXT3 [27] file systems are divided into a number of fixed size block groups. Each block group manages a fixed set of inodes and data blocks and contains a copy of the superblock. All the related metadata blocks and real data blocks are allocated close to each other. In addition, most file systems include journaling as an add-on feature to help recovery from a system failure [25] [26] [27]. File systems usually have a separate pool of disk blocks used for journaling. Journal blocks are organized as a circular buffer to log the changes for data blocks. Write accesses on journal blocks are always sequential. Therefore, spatial locality exhibited by journal blocks is different from data blocks. Fortunately, it does not greatly affect the workload characteristics, because the journal blocks represent only a small portion of the total blocks in most file systems. In the extreme case, the journal is the only structure on a disk in some special file systems, such as Sprite LFS [28] which is a log-structured file system. It is easy to log and predict the future access pattern for this kind of file systems. The log-structured file system is not in common commercial use. We may include the sequential access pattern in the future work.

File system caching mechanism may influence the temporal locality, spatial locality and the popularity. Caching can significantly help improve performance. For example, it can reduce the number of disk read operations when the same block is accessed by multiple times. If the users aggressively utilize caching, such as caching both real data and metadata, fewer operations will be observed at the disk level and the number of operations recorded in the traces will be reduced. However, since the cache size is limited, cache misses are still very common. Caching will not change the relative locality and popularity characteristics.

In summary, the workload characteristics that we explore in this chapter exist in most file systems. However, we also understand that different file systems may slightly alter the traces we observed. Leveraging the file system information and customizing the algorithm to adapt to the different file systems could be the future work.

4.6 Effects of Virtual Disk Format on Workload Characteristics

There are two format options for a VM's virtual disk: flat and sparse. Flat format virtual disk file owns a pre-allocated storage space that is equal to the virtual disk size. Sparse format virtual disk file is designed to save physical disk space [29]. The saving is achieved by describing a large empty region of the virtual disk using metadata instead of allocating the actual disk space for empty region. The number of contiguous empty blocks must be larger than a threshold which is described by the metadata. Figure 4.5 shows a simple example of the physical disk layout of the flat format and sparse format.

No matter what kind of disk format is used, the workload characteristics are similar. The guest OS is unaware of the underlying disk format. Its logical view of the virtual disk is like a physical disk. When a disk I/O operation is issued from the guest OS, the hypervisor will translate it into an access to an image file. The access offset on the disk is different

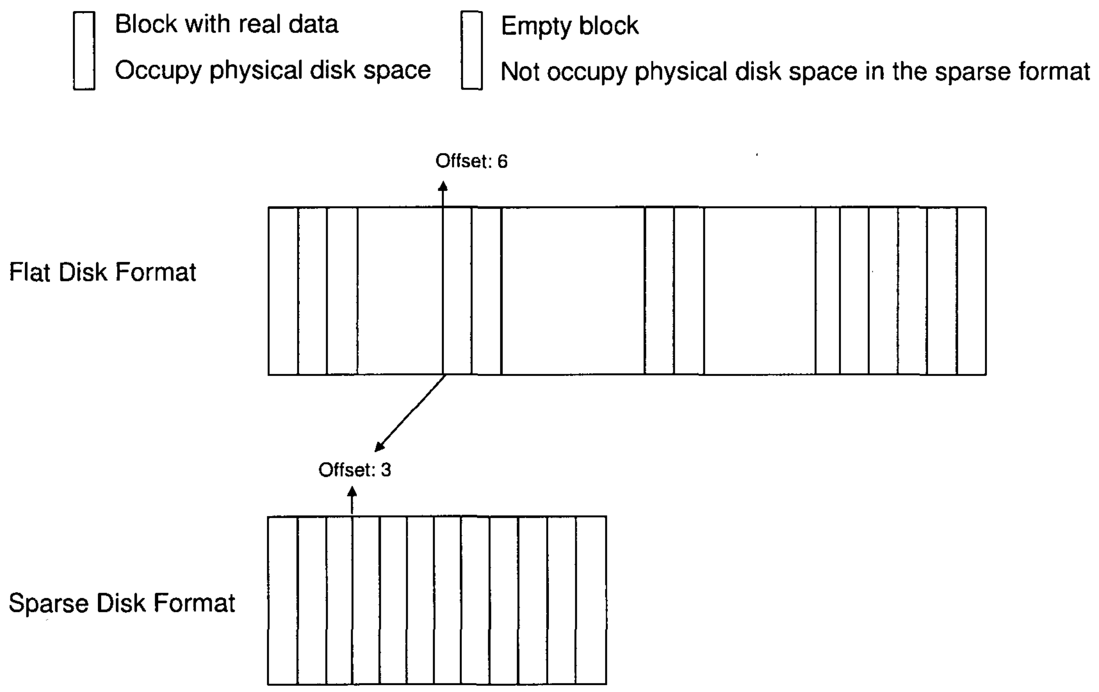


Figure 4.5 : A simple example of the flat disk format and the sparse disk format.

for the two formats because of their different physical disk layout. However, the difference does not affect the workload characteristics. The VM virtual disk in our experiments is a VMDK flat format file. Our observation on locality and popularity has been made in an environment where the disk is representative in the flat format. We believe that even if we use a sparse format virtual disk file, we do not imagine a major change for the workload characteristics. First, for the sparse disk format, the temporal locality and popularity are not affected, because they are more related to applications' behavior. Second, the allocation policy for the sparse disk format also attempts to minimize the file system fragmentation. When the file system need more space, it allocates an extent on the virtual disk. The extent is designed to best fit the new files or directories that belong together. Related files are allocated in the nearby regions. Therefore, the workload characteristics are independent on virtual disk format.

Chapter 5

Scheduling Algorithm

The main idea of the algorithm is to exploit locality to compute a more optimized storage migration schedule. We intercept and record a short history of the recent disk I/O operations of the VM, then use this history to predict the temporal locality, spatial locality, and popularity characteristics of the I/O workload during migration. Based on these predictions, we compute a storage transfer schedule that reduces the amount of extra migration traffic in the pre-copy model, the number of remote reads in the post-copy model, and reduces both extra migration traffic and remote reads in the pre+post-copy model. The net result is that storage I/O performance during migration is greatly improved. Figure 5.1 shows the three models with scheduling algorithm. We will describe the details in the following sections.

5.1 History of I/O Accesses

To collect history, we record the most recent N I/O operations in a FIFO queue. We will show that the performance improvement is significant even with a small N in Chapter 6. Therefore the memory overhead for maintaining this history is very small. Different migration models are sensitive to different types of I/O accesses as discussed in Chapter 2. This is related to the cause of the performance degradation. The extra migration traffic in the pre-copy model is caused by the write operations during the migration, while the remote reads in the post-copy model are caused by the read operations before certain blocks have been migrated. Therefore, when the pre-copy model is used, we collect only a history of write operations; when the post-copy model is used, we collect only a history of read operations; and when the pre+post-copy model is used, both read and write operations are collected. For each operation, a four-tuple, $< flag, offset, length, time >$, is recorded,

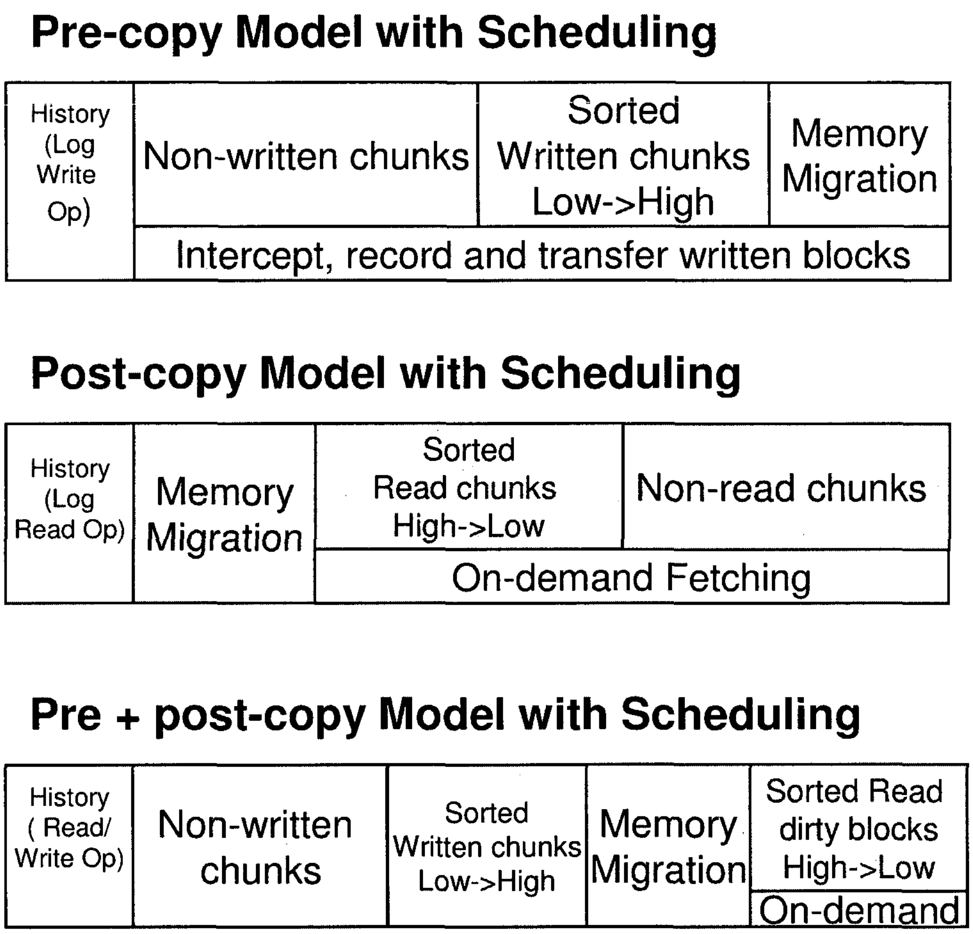

Pre-copy Model with Scheduling
History (Log Write Op)
Non-written chunks
Sorted Written chunks Low->High
Memory Migration
Intercept, record and transfer written blocks
Post-copy Model with Scheduling
History (Log Read Op)
Memory Migration
Sorted Read chunks High->Low
Non-read chunks
On-demand Fetching
Pre + post-copy Model with Scheduling
History (Read/ Write Op)
Non-written chunks
Sorted Written chunks Low->High
Memory Migration
Sorted Read dirty blocks High->Low
On-demand

Figure 5.1 : Models of live storage migration with scheduling

where $flag$ indicates whether this is a read or write operation, $offset$ indicates the block number being accessed, $length$ indicates the size of the operation, and $time$ indicates the time the operation is performed. The recording of each operation therefore only requires a few memory accesses which adds negligible processing overhead to the I/O operation. These actions are summarized in pseudo-code as follows. The time and space complexities are $O(1)$.

INPUT OF ALGORITHM: N and $model\_flag$

OUTPUT OF ALGORITHM: A queue of access operations: $Q_{history}$

```
$Q_{history} = \{ \};$
WHILE TRUE
  receive an OP $=< flag, offset, length, time >;$
  IF $((model\_flag == PRE\_COPY) \&\& (OP.flag == WRITE)$
    $\|(model\_flag == POST\_COPY) \&\& (OP.flag == READ)$
    $\|(model\_flag == PRE + POST\_COPY))$
    IF $(Q_{history}.length == N)$
      $Q_{history}.dequeue();$
      $Q_{history}.enqueue(OP);$
    ELSE $Q_{history}.enqueue(OP);$
    END-IF
  END-IF
  receive migration starting signal
    break the WHILE loop;
END-WHILE
RETURN $Q_{history};$
```

5.2 Scheduling Based on Access Frequency of Chunks

In this section, we discuss how we use I/O access history to compute a storage transfer schedule. Figure 5.2 and 5.3 illustrate how the migration and the I/O access sequence interact to cause the extra migration traffic and remote reads for the pre-copy and post-copy models. The pre+post-copy model combines these two scenarios but the problems are similar. Without scheduling, the migration controller will simply transfer the blocks of the virtual disk sequentially from the beginning to the end. In the example, there are only 10 blocks for migration and several I/O accesses denoted as either the write or read sequence. With no scheduling, under pre-copy, the total extra traffic is 31 blocks as all the blocks

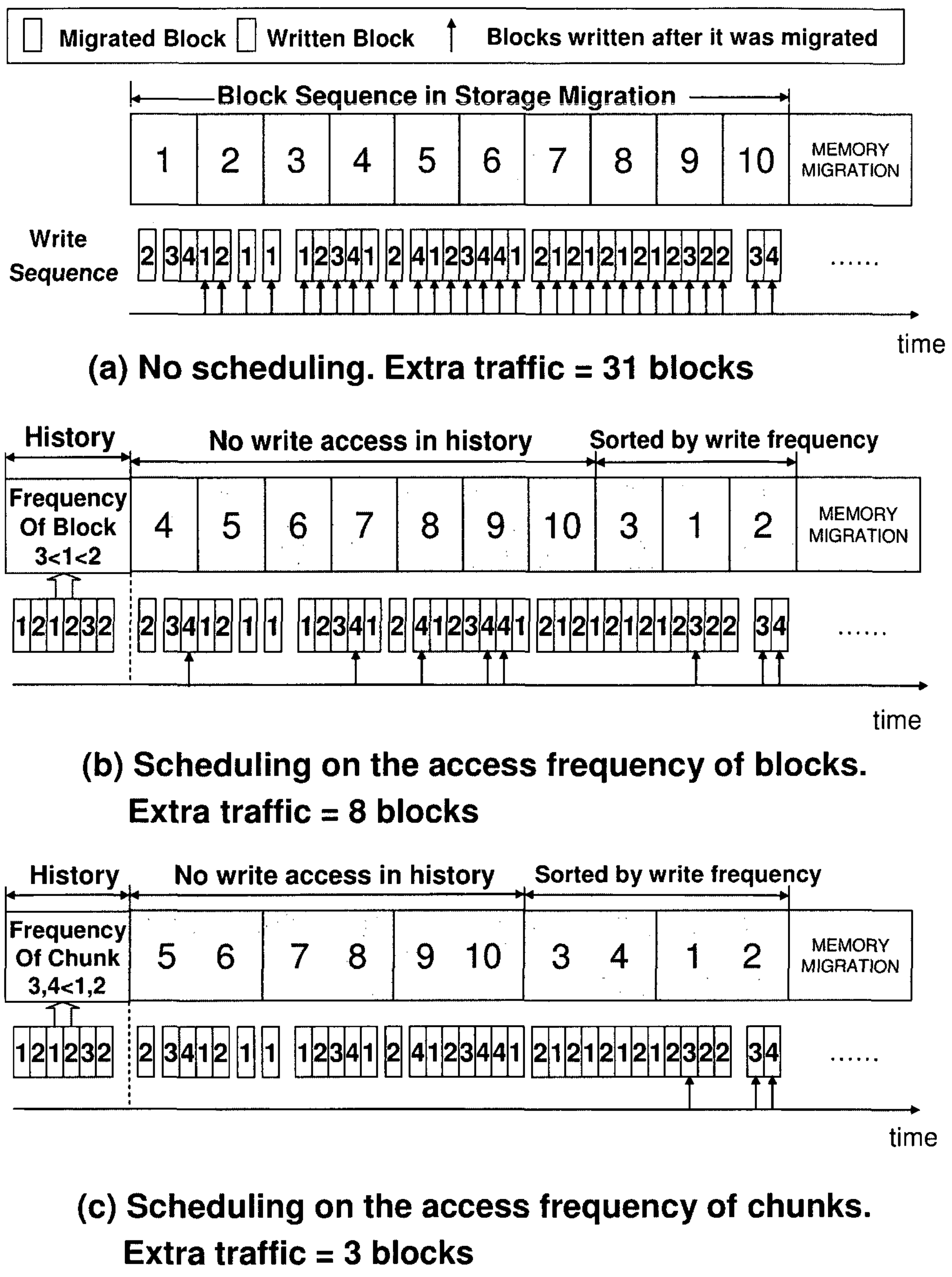

Migrated Block
Written Block
Blocks written after it was migrated
Block Sequence in Storage Migration
MEMORY MIGRATION
Write Sequence
time
(a) No scheduling. Extra traffic = 31 blocks
History
No write access in history
Sorted by write frequency
Frequency Of Block 3<1<2
(b) Scheduling on the access frequency of blocks. Extra traffic = 8 blocks
Frequency Of Chunk 3,4<1,2
(c) Scheduling on the access frequency of chunks. Extra traffic = 3 blocks

Figure 5.2 : A simple example of the scheduling algorithm applied to the pre-copy model.

that were written to during migration had to be resent. If we had an oracle that knew in advance the exact I/O sequence during the migration, then we could have waited to transmit

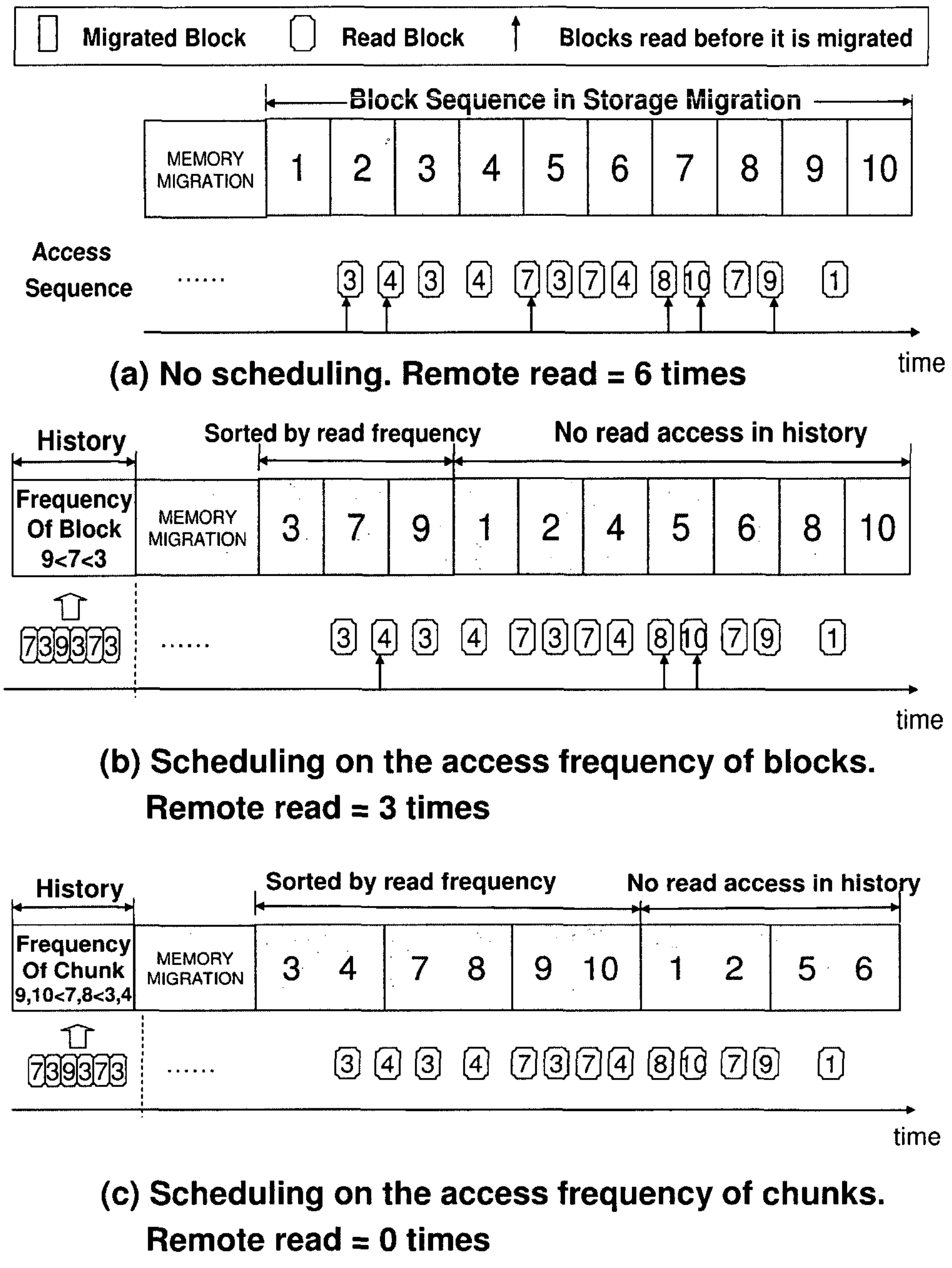

Migrated Block
Read Block
Blocks read before it is migrated
Block Sequence in Storage Migration
MEMORY MIGRATION
1 2 3 4 5 6 7 8 9 10
Access Sequence
3 4 3 4 7 3 7 4 8 10 7 9 1
time
(a) No scheduling. Remote read = 6 times
History
Sorted by read frequency
No read access in history
Frequency Of Block 9<7<3
3 7 9 1 2 4 5 6 8 10
7 3 9 3 7 3
time
(b) Scheduling on the access frequency of blocks. Remote read = 3 times
Frequency Of Chunk 9,10<7,8<3,4
3 4 7 8 9 10 1 2 5 6
(c) Scheduling on the access frequency of chunks. Remote read = 0 times

Figure 5.3 : A simple example of the scheduling algorithm applied to the post-copy model.

the blocks that were written to during migration after the write operations were completed resulting in no extra traffic. Similarly, under post-copy, there are 6 remote read operations where the block needed to be read before it was transferred to the destination. Again, with an oracle, we could schedule those blocks to be transferred prior to when the read operation would have been issued to improve performance.

In reality we do not have an oracle. Our scheduling algorithm exploits the temporal locality and popularity characteristics and uses the information in the history to perform predictions. That is, the block with a higher write frequency in $Q_{history}$ (i.e., more likely to be written to again) should be migrated later in the pre-copy model, and the block with a higher read frequency (i.e., more likely to be read again) should be migrated earlier in the post-copy model. In the illustrative example in Figures 5.2 and 5.3, when we schedule the blocks according to their access frequencies, the extra traffic and remote reads can be reduced from 31 to 8 and from 6 to 3.

In the example, block 4 in the pre-copy model and the blocks $\{4,8,10\}$ in the post-copy model are not found in the history, but they are accessed a lot during the migration due to spatial locality. The scheduling algorithm exploits spatial locality by scheduling the migration based on chunks. Each chunk is a cluster of n contiguous blocks. We call the number n the chunk size. The chunk size in the simple example is 2 blocks. We note that different workloads may have different effective chunk sizes and present a chunk size selection algorithm later in Section 5.3.

The access frequency of a chunk is defined as the sum of the access frequencies of the blocks in that chunk. The scheduling algorithm for the pre-copy model migrates the chunks that have not been written to in history first as those chunks are unlikely to be written to during migration and then followed by the written chunks. The written chunks are further sorted by their access frequencies to exploit the popularity characteristics. For the post-copy model, the read chunks are migrated in decreasing order of chunk read access frequencies, and then followed by the non-read chunks. The scheduling ensures that chunks that have been read frequently in history are sent to the destination first as they are more likely to be accessed. In the example, by performing chunk scheduling, the extra traffic and remote reads are further reduced to 3 and 0.

The scheduling algorithm is summarized in pseudo-code as follows. The time complexity is $O(n \cdot log(n))$, the space complexity is $O(n)$, where n is the number of blocks in the disk.

DATA STRUCTURE IN ALGORITHM:

-L_{bw}: A block write access list of $< block_{id}, time >$

-L_{br}: A block read access list of $< block_{id}, time >$

-L_{cwfreq}: A chunk write frequency list of $< chunk_{id}, frequency >$

-L_{crfreq}: A chunk read frequency list of $< chunk_{id}, frequency >$

-L_{sorted_wchunk}: A list of $chunk_{id}$ sorted by write frequency

-L_{sorted_rchunk}: A list of $chunk_{id}$ sorted by read frequency

-$L_{nwchunk}$: A list of $chunk_{id}$ which are not written in history

-$L_{nrchunk}$: A list of $chunk_{id}$ which are not read in history

INPUT OF ALGORITHM: $Q_{history}$, $model\_flag$ and $\alpha \in$[0, 1]

OUTPUT OF ALGORITHM: migration schedule $S_{migration}$

```
$S_{migration} = \{ \}$;
IF $((model\_flag == PRE\_COPY)$
  $||(model\_flag == PRE + POST\_COPY))$
    $L_{bw}$ = Convert $\forall OP \in Q_{history}$ whose $flag == WRITE$
      into $< block_{id}, time >$;
    $chunksize$=ChunkSizeEstimation($L_{bw}$,$\alpha$);
    Divide the storage into chunks;
    $S_{all} = \{All\ chunks\}$;
    FOR EACH $chunk_i \in S_{all}$
      $frequency_i$=$\sum frequency_{block_k}$
        where $block_k \in chunk_i$ and $block_k \in L_{bw}$;
        $frequency_{block_k}$ =# of times $block_k$ appearing in $L_{bw}$;
    END FOR
```

```
        $L_{cwfreq} = \{(chunk_i, frequency_i) | frequency_i > 0\}$;
        $L_{sorted\_wchunk}$ =Sort $L_{cwfreq}$ by $frequency$ low→high and
          chunks with the same frequency are sorted by id low→high;
        $L_{nwchunk} = S_{all} - L_{sorted\_wchunk}$ with id low → high;
        $S_{migration}$=$\{L_{nwchunk}, L_{sorted\_wchunk}\}$;
    ELSE IF($model\_flag == POST\_COPY$)
        $L_{br}$ = Convert $\forall OP \in Q_{history}$ whose $flag == READ$
          into $< block_{id}, time >$;
        $chunksize$=ChunkSizeEstimation($L_{br}$,$\alpha$);
        Divide the storage into chunks;
        $S_{all} = \{All\ chunks\}$;
        FOR EACH $chunk_i \in S_{all}$
          $frequency_i$=$\sum frequency_{block_k}$
            where $block_k \in chunk_i$ and $block_k \in L_{br}$;
            $frequency_{block_k}$ = # of times $block_k$ appearing in $L_{br}$;
        END FOR
        $L_{crfreq} = \{(chunk_i, frequency_i) | frequency_i > 0\}$;
        $L_{sorted\_rchunk}$ =Sort $L_{crfreq}$ by $frequency$ high→low and
          chunks with the same frequency are sorted by id low→high;
        $L_{nrchunk} = S_{all} - L_{sorted\_rchunk}$ with id low → high;
        $S_{migration}$=$\{L_{sorted\_rchunk}, L_{nrchunk}\}$;
    END IF
    RETURN $S_{migration}$;
```

Note that α is an input value for the chunk size estimation algorithm and will be explained later. The pre+post-copy model is a special case which has two migration stages. The above algorithm works for its first stage. The second stage begins when the VM memory migration has finished. In this second stage, the remaining dirty blocks are scheduled.

The algorithm works as follows. The time complexity is $O(n \cdot log(n))$, the space complexity is $O(n)$, where n is the number of dirty blocks.

```
DATA STRUCTURE IN ALGORITHM:
-$L_{dirtyblock}$: A dirty block list of $block_{id}$
-$L_{br}$: A block read access list of $< block_{id}, time >$
-$L_{dbrfreq}$: A dirty block read frequency list of $< block_{id}, frequency >$
-$L_{sorted\_block}$: A dirty block list of $block_{id}$ sorted by read frequency

INPUT OF ALGORITHM: $Q_{history}$, $L_{dirtyblock}$
OUTPUT OF ALGORITHM: migration schedule $S_{migration}$


$S_{migration} = \{ \}$;
$L_{br}$ = Convert $\forall OP \in Q_{history}$ whose $flag == READ$
   into $< block_{id}, time >$;
FOR EACH $block_i \in L_{dirtyblock}$
  find $block_i$ in $L_{br}$
  IF found
    $frequency_i$= # of times $block_i$ appearing in $L_{br}$;
  ELSE $frequency_i = 0$;
  END IF
END FOR
$L_{dbrfreq} = \{(block_i, frequency_i) | block_i \in L_{dirtyblock}\}$;
$L_{sorted\_dblock}$ =Sort $L_{dbrfreq}$ by $frequency$ high$\rightarrow$low and
   blocks with the same frequency are sorted by id low$\rightarrow$high;
$S_{migration}$=$\{L_{sorted\_dblock}\}$;
RETURN $S_{migration}$;
```

5.3 Chunk Size Selection

The chunk size used in the scheduling algorithm needs to be judiciously selected. It needs to be sufficiently large to cover the likely future accesses near the previously accessed blocks, but not so large as to cover many irrelevant blocks that will not be accessed. To balance these factors, for a neighborhood size n, we define a metric called $Balanced\_coverage = Access\_coverage + (1 - Storage\_coverage)$. Consider splitting the access history into two parts based on some reference point. Then, $Access\_coverage$ is the percentage of the accessed blocks (either read or write) in the second part that are within the neighborhood size n around the accessed blocks in the first part. $Storage\_coverage$ is simply the percentage of the overall storage within the neighborhood size n around the accessed blocks in the first part. The neighborhood size that maximizes $Balanced\_coverage$ is then chosen as the chunk size by our algorithm.

Figure 5.4 shows the $Balanced\_coverage$ metric for different neighborhood sizes for different server workloads. As can be seen, the best neighborhood size will depend on the workload itself.

In the scheduling algorithm, we divide the access list L_H in the history into two parts, S_{H1} consists of the accesses in the first α fraction of the history period, where α is a configurable parameter, and S_{H2} consists of the remaining accesses. If all of the blocks accessed in the second part are also accessed in the first part, the optimal neighborhood size becomes zero. Therefore, we set the lower bound of the chunk size to the block size. The algorithm also bounds the maximum selected chunk size. In the evaluation, we set this bound to 1GB.

The algorithm pseudo-code is shown below. The time complexity of this algorithm is $O(n \cdot log(n))$ and the space complexity is $O(n)$, where n is the number of blocks in the disk.

```
DATA STRUCTURE IN ALGORITHM:
-$L_H$: the access list from the history.
-$\alpha$: the fraction of simulated history.
```

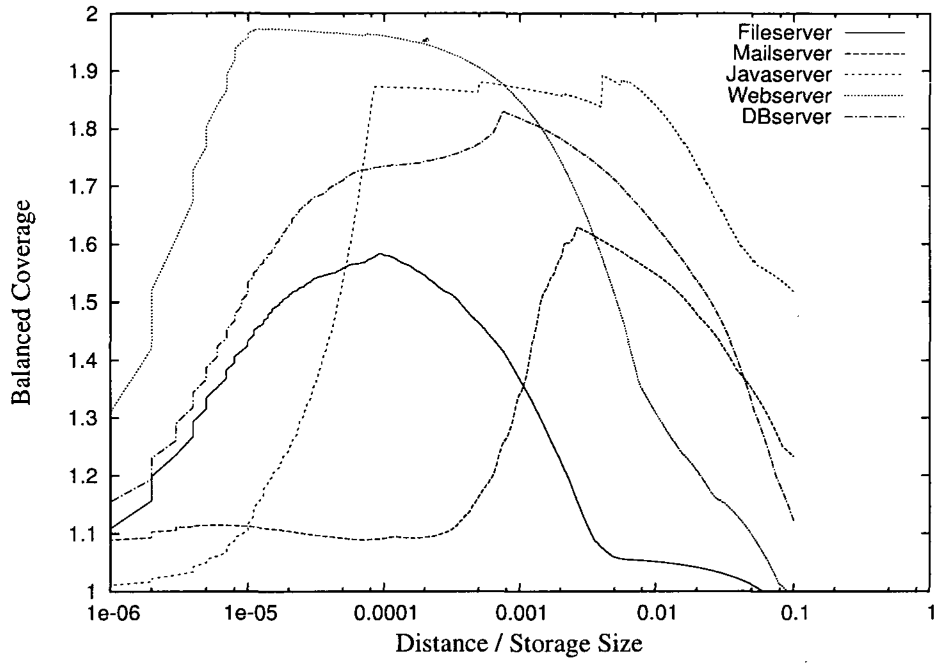


Figure 5.4 : A peak in balanced coverage determines the appropriate chunk size for a given workload.

-*total_block*: the number of total blocks in the storage.

-*upper_bound*: the maximum allowed chunk size, e.g. 1GB.

-*lower_bound*: the minimum allowed chunk size, e.g. 512B.

-S_{H1}: A set of blocks accessed in the first part.

-S_{H2}: A set of blocks accessed in the second part.

-*distance*: The storage size between the locations of two blocks.

-ND: Normalized distance computed by $distance/storage\_size$.

-$S_{NormDistance}$: A set of normalized distances for blocks that are in S_{H2} but not accessed in S_{H1}.

-$S_{NormDistanceCDF}$: A set of pair $< ND, \% >$. The percentage is the cumulative distribution of ND in the set $S_{NormDistance}$.

-ES_{H1}: A set of blocks obtained by expanding every block in S_{H1} by covering its neighborhood range.

-$BalanceCoverage_{max}$: the maximum value of $BalanceCoverage$

-ND_{BCmax}: the neighborhood size (a normalized distance) that maximizes $BalanceCoverage$

```
INPUT OF ALGORITHM: $L_H$, $\alpha$, $total\_block$,
        $lower\_bound$, $upper\_bound$, $block\_size$
OUTPUT OF ALGORITHM: $chunk\_size$

$max\_time$= the duration of $L_H$;
FOR EACH $< block_{id}, time > \in L_H$
  IF $time < max\_time * \alpha$
    Add $block_{id}$ into $S_{H1}$;
  ELSE ADD $block_{id}$ into $S_{H2}$;
  END IF
END FOR
FOR EACH $block_{id} \in S_{H2}$
  IF $block_{id} \notin S_{H1}$
    $NormDistance = \frac{\{min\ (|block_{id}-m|) \forall m \in S_{H1}\}}{total\_block}$;
    Add $NormDistance$ into $S_{NormDistance}$;
  END IF
END FOR
$S_{NormDistanceCDF}$ = compute the cumulative distribution
 function of $S_{NormDistance}$;
$BalancedCoverage_{max} = 0$;
$ND_{BCmax} = 0$;
$ND_{min}$= the minimal $ND$ in $S_{NormDistanceCDF}$;
$ND_{max}$= the maximal $ND$ in $S_{NormDistanceCDF}$;
$ND_{step} = \frac{ND_{max}-ND_{min}}{1000}$;
```

```
FOR $ND = ND_{min}; ND \leq ND_{max}; ND+ = ND_{step}$
  $distance = ND * total\_block;$
  $ES_{H1} = \{ \}$
  FOR EACH $m \in S_{H1}$
    add $block_{id}$ from $(m - distance)$ to $(m + distance)$ to $ES_{H1};$
  END FOR
  $storage\_coverage = \frac{\#\ of\ blocks\ in\ ES_{H1}}{total\_block};$
  $access\_coverage$ =the percentage of $ND$ in $S_{NormDistanceCDF};$
  $balanced\_coverage = access\_coverage + (1 - storage\_coverage);$
  IF $balanced\_coverage > BalancedCoverage_{max}$
    $BalancedCoverage_{max} = balanced\_coverage;$
    $ND_{BCmax} = ND;$
  END IF
END FOR
$chunk\ size = ND_{BCmax} * total\_block * block\_size;$
IF $(chunk\_size == 0)$
  $chunk\_size = lower\_bound;$
ELSE IF $chunk\_size > upper\ bound$
  $chunk\_size = upper\_bound;$
END IF
RETURN $chunk\_size;$
```

5.4 Potential Robustness Improvements

The scheduling algorithm relies on the precondition that the access history can help predict the future accesses during migration, and our analysis has shown this to be the case for a wide range of workloads. However, an actual implementation might want to include certain safeguards to ensure that even in the rare case that the access characteristics are turned upside down during the migration, any negative impact is contained. First, a test can be performed on the history itself, to see if the first half of the history does provide good prediction for the second half. Second, during the migration, newly issued I/O operations can be tested against the expected access patterns to find out whether they are consistent. If either one of these tests fails, a simple solution is to revert to the basic non-scheduling migration approach.

Chapter 6

Evaluation

To estimate the performance of all three different storage migration models with and without scheduling, we perform trace-based simulations. Although we cannot simulate all nuances of a fully implemented system, our estimates are sufficiently accurate, and the observed benefits are significant enough to provide reliable guidance to system designers.

6.1 Simulation Methodology

We assume the network has a fixed bandwidth and a fixed delay. We assume there is no network congestion and no packet loss. Thus, once the migration of a piece of data is started at the source, the data arrives at the destination after $\frac{data\_size}{bandwidth} + delay$ seconds. In our experiments, we simulate a delay of 50ms and use different values of fixed bandwidths for different experiments.

For the following discussion, it may be helpful to refer to Figure 5.1. Each experiment is run 10 times using different random migration start times t chosen from $[3000, 5000]$ seconds. When the simulation begins at time t, we assume the scheduling algorithm has already produced a queue of block IDs ordered according to the computed chunk schedule to be migrated across the network in the specified order. Let us call this the primary queue. The schedule is computed based on using a portion of the trace prior to time t as history. The default history size is 50,000 operations. We configure the parameter α in the chunk size selection algorithm with different values and find that it is not sensitive. α is 0.7 in all the following experiments. In addition, there is an auxiliary queue which serves different purposes for different migration models. As we simulate the storage and memory migrations, we also playback the I/O accesses in the trace starting at time t, simulating the continuing execution of the virtual machine in parallel. We assume each I/O access is independent. In other words, one delayed operation does not affect the issuance of the subsequent operations in the trace.

We do not simulate disk access performance characteristics such as seek time or read and write bandwidth. The reason is that, under the concurrent disk operations simulated from the trace, the block migration process, remote read requests, and operations issued by other virtual machines sharing the same physical disk, it is impossible to simulate the effects that disk characteristics will have in a convincing manner. Thus, disk read and write operations are treated to be instantaneous in all scenarios. However, under our scheduling approach, blocks might be migrated in an arbitrary order. To be conservative, we do add a performance penalty to our scheduling approach. Specifically, the start of the migration of a primary queue block is delayed by 10ms if the previous migrated block did not immediately precede this block.

In the pre-copy model, dirty blocks that need to be retransmitted are enqueued to the auxiliary queue. The primary queue and the auxiliary queue receive service in round robin. Thus, when both queues are backlogged, each queue gets an equal share of the network bandwidth. When a queue is serviced, the transfer of the head of queue block is simulated. When the primary queue is empty, the memory migration begins, which simply completes in $\frac{memory\_size}{bandwidth} + delay$ seconds.

In the post-copy model, the memory migration is simulated first and starts at time t. When it is completed, storage migration begins according to the order in the primary queue. Subsequently, when a read operation for a block that has not yet arrived at the destination is played back from the trace, the desired block ID is enqueued to the auxiliary queue after a network delay (unless the transfer of that block has already started), simulating the remote read request. The auxiliary queue is serviced with strict priority over the primary queue. When a block is migrated through the auxiliary queue, the corresponding block in the primary queue is removed. Note that when a block is written to at the destination, we assume the source is not notified, so the corresponding block in the primary queue remains.

In the pre+post-copy model, in the pre-copy phase, the storage is migrated according to the primary queue; the auxiliary queue is not used in this phase. At the end of the memory migration, the dirty blocks' migration schedule is computed and stored in the primary queue. Subsequently, the simulation in the post-copy phase proceeds identically to the post-copy model.

Finally, when scheduling is not used, the simulation methodology is still the same, except that the blocks are ordered sequentially in the primary queue.

6.2 Performance Metrics

We use the following performance metrics for evaluation.

- **Extra traffic in number of blocks:** In the pre-copy and the pre+post-copy models, this is the number of retransmitted blocks. A large amount of extra traffic in the pre-copy model could lead to write operation throttling, which will dramatically degrade VM performance.

- **Number of postponed operations:** In the pre-copy model, if no write throttling is performed, then by the end of the memory migration, there could still be dirty blocks left in the auxiliary queue. A perfect throttling mechanism must therefore work in a way such that the issuance of the write operations corresponding to these dirty blocks are postponed until after the memory migration is finished. We call these operations the postponed operations. Any postponed operation obviously will negatively impact the VM performance. Note that this metric is very conservative as it assumes the throttling is performed perfectly, and the read operations are unaffected.

- **Postponed time:** For each postponed operation, we compute a metric called postponed time. It is the difference between the time at which memory migration finishes (which is the first opportunity for the postponed operation to be issued) and the original issue time of the operation in the trace (i.e. the natural issue time had there been no throttling). Note that this metric is very conservative as it assumes all the postponed operations can be issued instantaneously after memory migration is finished.

- **Number of remote reads:** In the post-copy and pre+post-copy models, a remote read will be delayed by at least one network round trip delay. Therefore, a large number of remote reads is detrimental to VM performance. We measure the number of remote read blocks.

6.3 Benefits Under Pre-Copy

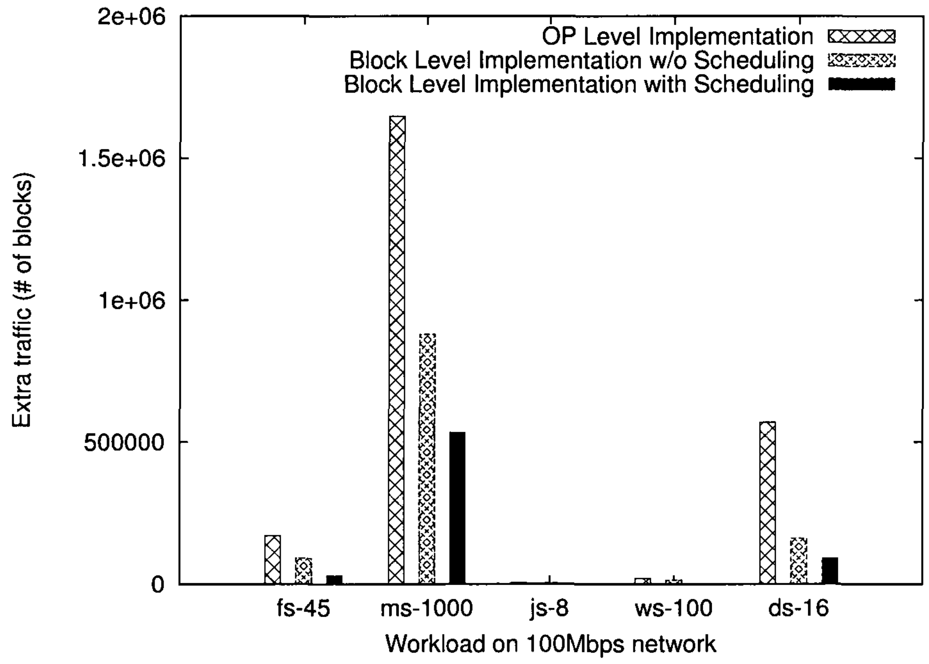
A pre-copy-based system was proposed in [11]. Its implementation is at the I/O operation level. Specifically, it records the write operations during migration. These recorded write operations are then transmitted to the destination as they are and replayed at the destination. Note that each recorded operation can write to multiple underlying disk blocks. In other words, it does not try to eliminate unnecessary transmissions at the block granularity. A block level implementation could be more efficient. For example, a block may be written to in two consecutive write operations. Then the second operation overwrites the first. Thus, only the data in the second operation for the block needs to be transmitted. We compare results for the operation level implementation, block level implementation, and block level implementation with scheduling.

6.3.1 Reduction in Extra Traffic

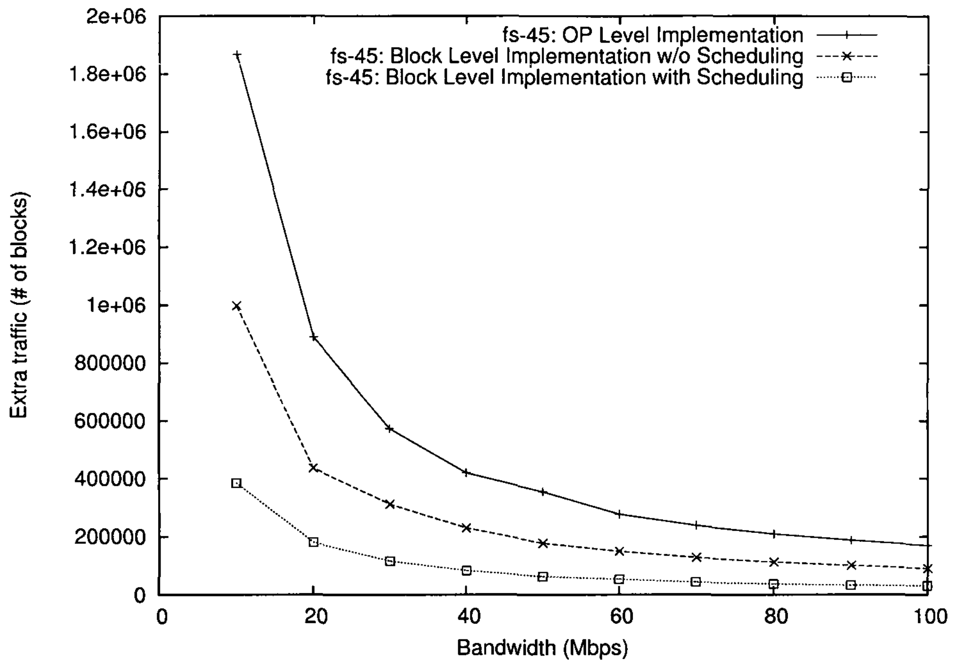
Figure 6.1(a) shows that compared to the existing operation-level implementation, the scheduling algorithm can reduce the extra traffic in the file server, mail server, Java server, web server and database server by 82%, 68%, 88%, 91% and 84% respectively. Compared to the block level implementation without scheduling, the improvement is 66%, 39%, 82%, 88% and 43% respectively. We can see that the scheduling algorithm is very effective at reducing traffic for both operation-level and block-level implementations of pre-copy.

When the network bandwidth decreases, we expect the extra traffic to increase. However, Figure 6.1(b) shows that, for the file server fs-45 workload, with the scheduling algorithm, the rate at which extra traffic increases is much lower. At a network bandwidth of 10 Mbps, the extra traffic is reduced by 79% compared to the operation level implementation, and 61% compared to the block level implementation. The results for the other workloads are similar. Compared to the operation level implementation, the extra traffic in the mail server, Java server, web server and database server is reduced by 69%, 83%, 92%, and 89% respectively when the network bandwidth is 10 Mbps. Compared to the block level implementation without scheduling, the improvement is 34%, 73%, 88% and 41% respectively.

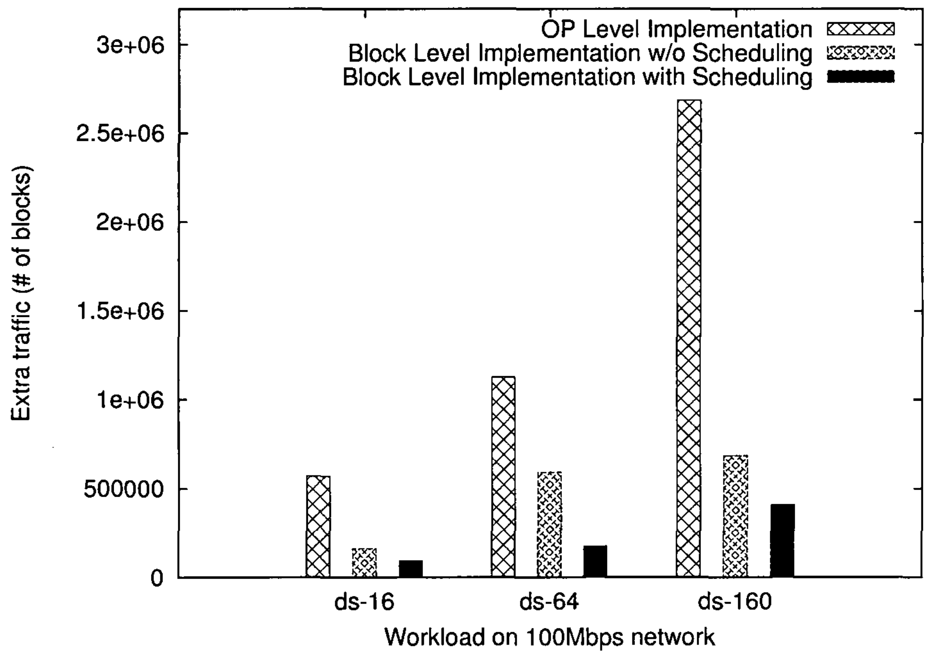
When the number of clients increases, I/O rates increase, and the extra traffic is also expected to increase. Figure 6.1(c) shows that the operation level implementation incurs over 2 million blocks of extra traffic under the database server ds-160 workload. The



(a) Different Workload



(b) Different Bandwidth



(c) Different Number of Clients

Figure 6.1 : The improvement in extra traffic under the pre-copy model.

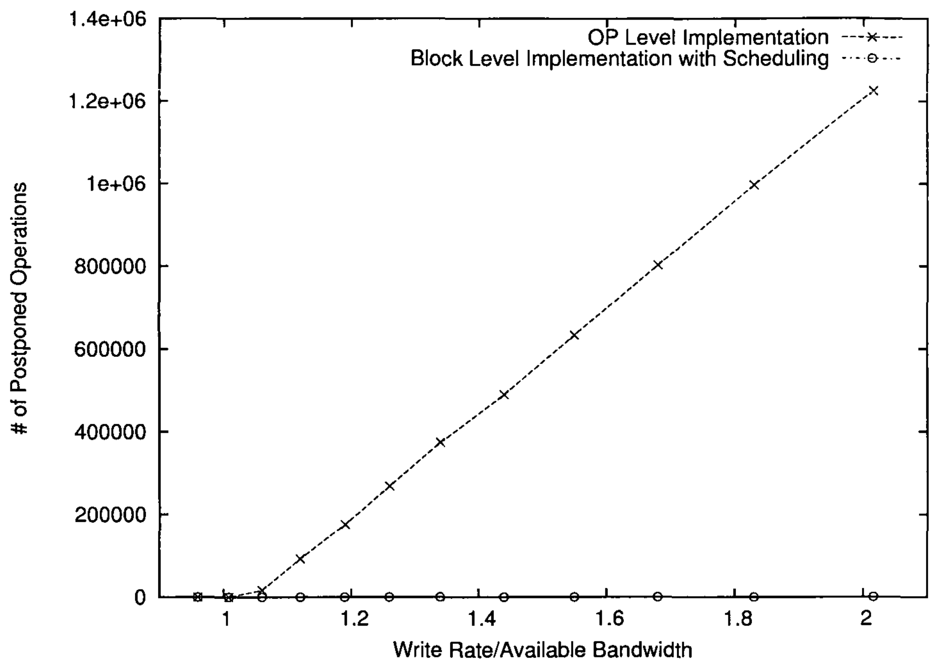


Figure 6.2 : The improvement of number of postponed operations under the pre-copy model (ds-160 workload). The postponed operations are reduced from millions to less than 800 when scheduling is used.

scheduling algorithm is able to reduce 83% of this extra traffic. For the file server, when the number of clients increases to 70, the extra traffic is reduced by 73% compared to the operation level implementation and by 53% compared to the block level implementation.

6.3.2 Reduction in Postponed Operations

When a workload is write-intensive relative to the available bandwidth for block retransmission (which is 50% of the network bandwidth in our simulation), write throttling becomes necessary in the pre-copy model, resulting in postponed operations. Take the database server ds-160 workload for example. The write data rate is 10.08 Mbps on average.

As Figure 6.2 shows when the available bandwidth drops to 5 Mbps, without scheduling, more than 1 million operations are postponed in the operation level implementation.

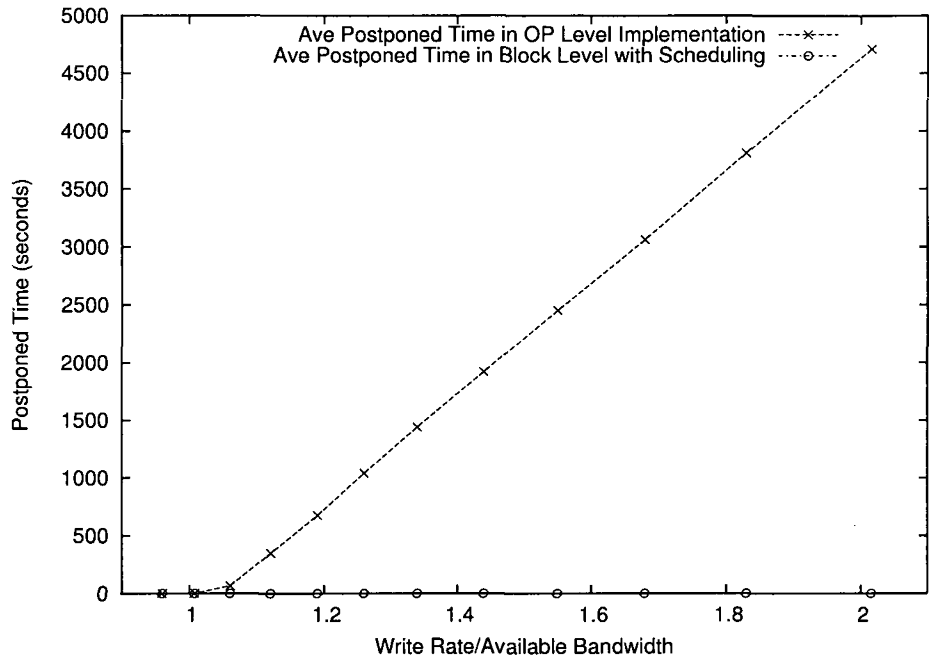


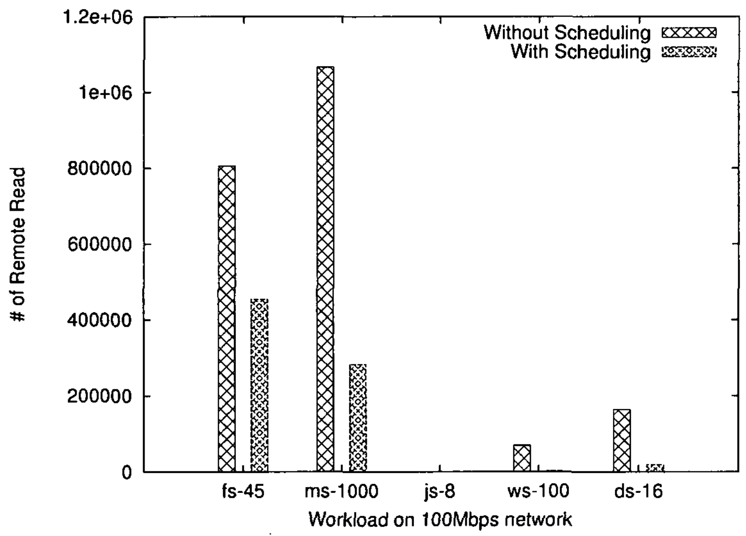
Figure 6.3 : The improvement of postponed time under the pre-copy model (ds-160 workload). The average postponed time are reduced from thousands of seconds to less than 0.5 second when scheduling is used.

In contrast, with scheduling, only 800 operations are postponed. Furthermore, from Figure 6.3, we can see that the average postponed time is reduced from thousands of seconds to less than 0.5 second. Note that under the ds-160 workload, the basic block level implementation incurs low enough extra traffic (though still significantly higher than with scheduling) that the number of postponed operations is also very low. The average postponed time is almost the same as the scheduling algorithm when the available bandwidth is 5Mbps. However, when the available bandwidth decreases, the average postponed time is expected to increase. With the scheduling algorithm, the rate at which the average postponed time increases is lower. At an available bandwidth of 1 Mbps, the average postponed time is reduced by 10 seconds compared to the block level implementation without scheduling.

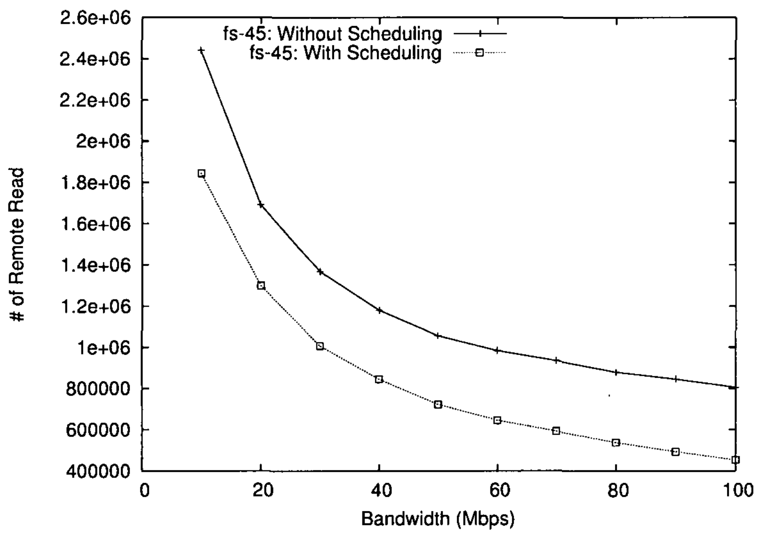
6.4 Benefits Under Post-Copy

Figure 6.4 shows the benefits of scheduling in terms of the number of remote reads under the various server types, bandwidths, and workload intensities. The reductions in the number of remote reads are 44%, 74%, 96% and 89% in the file server, mail server, web server and database server respectively when the network bandwidth is 100 Mbps. The Java server performs very few read operations, so there is no remote read. When the network bandwidth is low, the file server (fs-45) suffers from more remote reads because the migration time is longer. At 10 Mbps, 0.6 million (or 24%) remote reads are eliminated by scheduling in the file server. For the mail server, web server and database server, their remote reads are reduced by 41%, 92% and 86% respectively when the network bandwidth is 10Mbps.

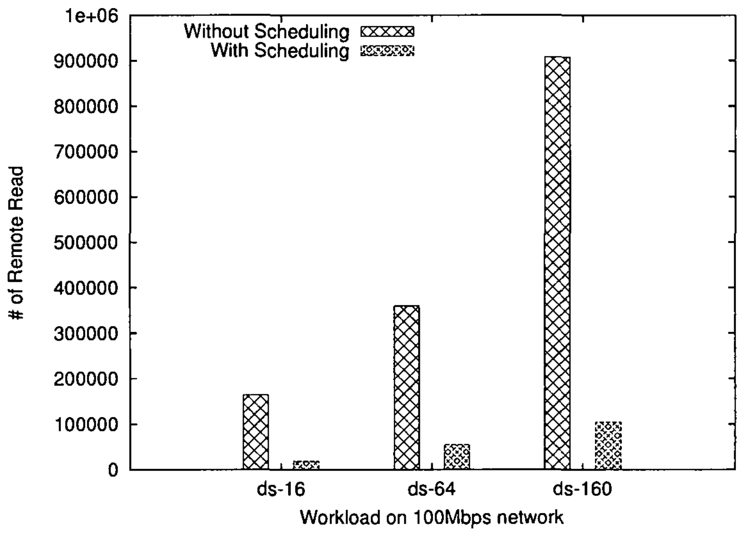
When the number of clients increases, the read rate becomes more intensive. For example, the ds-160 workload results in 0.9 million remote reads when scheduling is not used. With scheduling, remote read is reduced by 85%-89% under the ds-16, ds-64, and ds-160



(a) Different Workload



(b) Different Bandwidth



(c) Different Number of Clients

Figure 6.4 : The improvement of remote reads under the post-copy model.

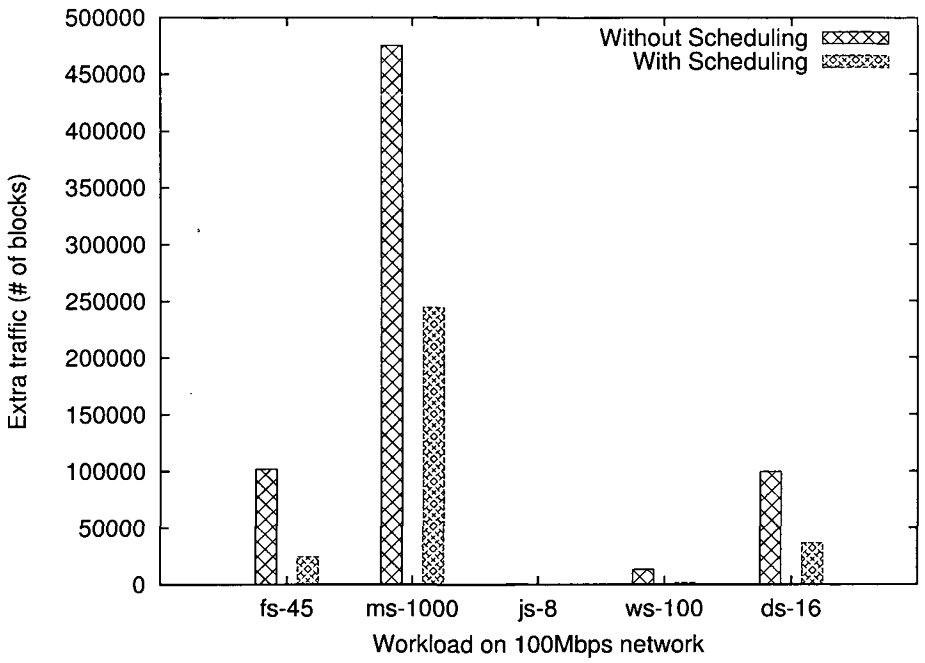


Figure 6.5 : The improvement of extra traffic under the pre+post-copy model.

workloads. When the number of the clients in the file server is increased to 70, there is over 1 million remote reads. With scheduling, it can be reduced by 41%.

6.5 Benefits Under Pre+Post-Copy

In the pre+post-copy model, the extra traffic consists of only the final dirty blocks at the end of memory migration. As Figure 6.5 shows, the scheduling algorithm reduces the extra traffic in the five workloads by 76%, 50%, 58%, 87% and 64% respectively.

In the pre+post-copy model, remote reads exist only during the retransmission of the dirty blocks. Since the amount of dirty data is much smaller than the virtual disk size, the problem is not as serious as in the post-copy model. Figure 6.6 shows that the Java server, web server and database server have no remote read because their amount of dirty data is small. But the file server and mail server suffer from remote reads, and applying scheduling can reduce them by 97% and 88% respectively.

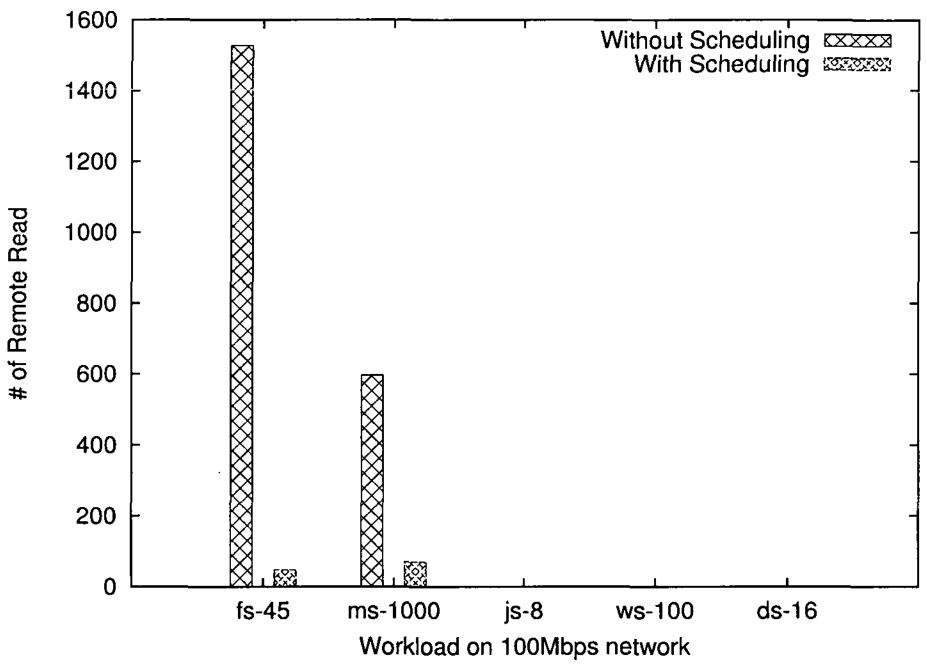


Figure 6.6 : The improvement of remote reads under the pre+post-copy model.

	fs-45	ms-1000	js-8	ws-100	ds-16
Worst chunk size performance gain	49%	43%	49%	74%	54%
Optimal chunk size performance gain	77%	70%	64%	90%	66%
Algorithm selected chunk size performance gain	76%	50%	58%	87%	64%

Table 6.1 : Comparison between selected chunk size and measured optimal chunk size (extra traffic under pre+post-copy).

6.6 Optimality of Chunk Size

In order to understand how optimal is the chunk size selected by the algorithm, we conduct experiments with various manually selected chunk sizes, ranging from 512KB to 1GB in factor of 2 increments, to measure the performance gain achieved at these different chunk sizes. The chunk size that results in the biggest performance gain is considered the measured optimal chunk size. The one with the least gain is considered the measured worst chunk size. Table 6.1 compares the selected chunk size against the optimal and worst chunk sizes in terms of extra traffic under the pre+post-copy model. As can be seen, the gain achieved by the selected chunk size is greater than the measured worst chunk size across the 5 workloads. Most of them are very close to the measured optimal chunk size except the mail server. There are two reasons that explain why the selected chunk size of the mail server is not as good as the chunk size of the other workloads. First, the default history period configured in the algorithm is not long enough and that may affect the performance. For the following discussion, it is helpful to refer to the Figure 5.4 in the Section 5.3. It shows the relationship between the $Balanced\_coverage$ and the neighborhood size. In order to explore the characteristics, we use a long enough history period which is 3000 seconds in that experiment. However, the history period of 3000 seconds requires a huge space in the memory to store all the operations during that period. It is not acceptable in the real use, so a default history size of 50,000 operations is used instead. For the mail server, the history buffer holds the operations issued over only 500 seconds. It shows the trade off between the memory space used by the history buffer and the performance achieved by the algorithm. The second reason is related to the spatial locality characteristics of the mail server. For other servers, the blocks which are closer to the accessed blocks during the history have a higher possibility to be accessed. For the mail server, the future accessed blocks tend to be much farther away from the accessed blocks during the history. As the Figure 5.4 shows, when the neighborhood size increases, the $Balanced\_coverage$ does not increase until the neighborhood size reaches $0.0004 * Storage\_Size$. Then it increases sharply from $0.0004 * Storage\_Size$ to $0.00275 * Storage\_Size$. The selected chunk size in the algorithm is around $0.0004 * Storage\_Size$ due to the short history. That is why the performance is not close to the measured optimal chunk size.

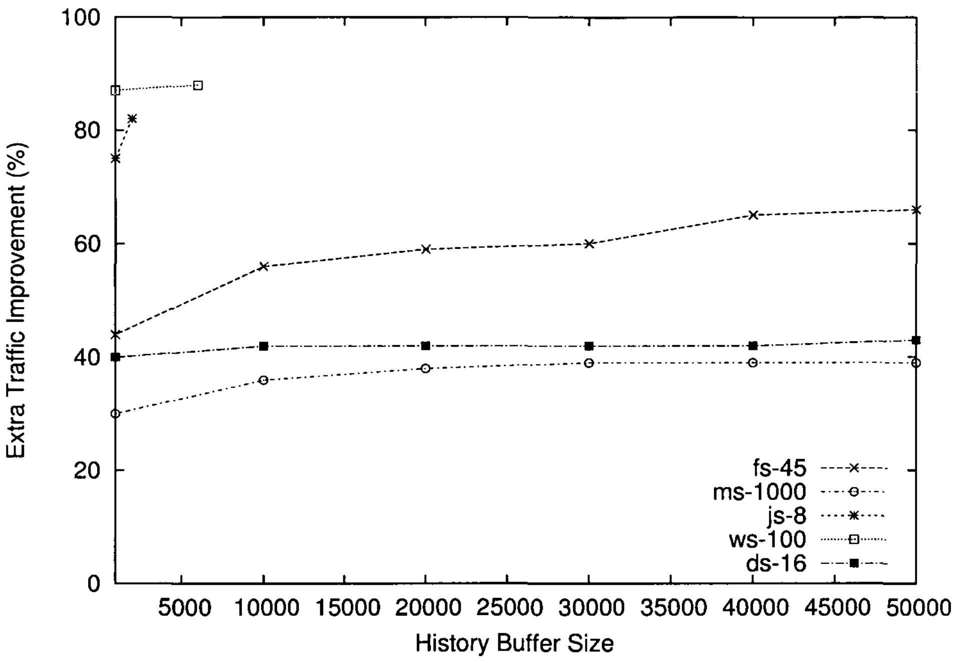


Figure 6.7 : Sensitivity of algorithm to history size. Using a longer history results in bigger gains, but even a short history is already useful.

6.7 Sensitivity to History Size

We conduct experiments varying the history size to see how it affects the resulting performance improvements over the block level implementation without scheduling. Figure 6.7 shows that when the history size is reduced to 1000 operations, the improvement is reduced compared to using a longer history. However, even such a short history can provide significant performance benefits. The Java server and web server perform fewer write operations than the other servers. From the trace starting time to the migration starting time, the Java server issues roughly 1000 write operations and the web server issues roughly 6000 write operations. Therefore, their history buffer is not full when the buffer size is large.

Chapter 7

Summary and Future Work

Migrating virtual machines between clouds is an emerging requirement to support open clouds and to enable better service availability. While there are several existing solutions for wide-area migration, they all share one common goal which is to minimize disruption on the running services undergoing migration. Although existing solutions each have their strengths for certain types of I/O workloads, we show that they also have weaknesses that end up significantly degrading performance. In this thesis, we demonstrate that their weaknesses can be mitigated by taking a workload-aware approach to storage migration. We collect traces of I/O workloads of five representative applications and establish insights on the temporal locality, spatial locality and access popularity that widely exists. Based on these insights, we design a scheduling algorithm that exploits individual virtual machine's workload to compute an efficient schedule for transferring storage at the appropriate granularity in terms of *chunks* rather than blocks. In order to evaluate our scheduling algorithm, we use a trace-driven framework. Under a wide range of I/O workloads and network conditions, we show that workload-aware scheduling can effectively reduce the amount of extra traffic and I/O throttling for the pre-copy model of storage migration. In addition, for the post-copy model, we can also significantly reduce the number of remote reads to improve the performance. Our scheduling algorithm can be incorporated into the existing work to enable them to work well under challenging environments with higher I/O intensity, more client requests, or lower available bandwidth. Our work has applicability for migration across clouds as well as across virtualized data centers which are also increasingly popular.

Up to now, the most widely used open source virtualization platforms are Xen [30] and KVM [31]. The VM live migration operation on Xen still requires shared storage [32].

In other words, it does not support storage migration. KVM added the storage migration feature in January of 2010 [33]. It uses the pre-copy model without scheduling and has the problems we discussed in Section 2.2. In the future, we will apply our scheduling algorithm in existing open source platforms to minimize the disruption of virtual machine I/O performance when performing live migration.

There are also some other potential directions for VM live migration research. For example, we are interested in understanding what is the most efficient way to migrate hundreds or even thousands of virtual machines. Simply migrating the virtual machines one by one may not be a good solution. First, the total amount of data that need to be migrated is huge. Some of them may be redundant and unnecessary to be migrated. Second, virtual machines that will be migrated may cooperate with each other to provide services. When a portion of them has been migrated to the remote destination, they may suffer a long latency when communicating across WAN with each other. Third, migrating a cluster of virtual machines may affect the resource allocation of the source and destination clouds. For example, the bandwidth in the cloud may be occupied by the migration for a long time and it may affect other services that coexist in the same cloud. All of these challenges should be taken into account when we schedule the large scale migration in the future.

Bibliography

[1] A. W. S. Blog, "Animoto - scaling through viral growth." http://aws.typepad.com/aws/2008/04/animoto—scali.html, Apr. 2008.

[2] D. Gottfrid, "The New York Times Archives + Amazon Web Services = Times-Machine." http://open.blogs.nytimes.com/ 2008/05/21/the-new-york-times-archives-amazon-web- services-timesmachine/, May 2008.

[3] M. Armbrust, A. Fox, R. Griffith, and et. al, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[4] O. C. Manifesto, "Open Cloud Manifesto." http://www.opencloudmanifesto.org/, Jan. 2010.

[5] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *USENIX'05: Proceedings of the 2005 Usenix Annual Technical Conference*, (Berkeley, CA, USA), USENIX Association, 2005.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, (Berkeley, CA, USA), pp. 273–286, USENIX Association, 2005.

[7] I. Redbooks, *IBM Powervm Live Partition Mobility IBM International Technical Support Organization.* Vervante, 2009.

[8] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *NSDI*, 2007.

[9] H. Jin, L. Deng, S. Wu, and X. Shi, "Live virtual machine migration integrating memory compression with precopy," in *IEEE International Conference on Cluster Computing*, 2009.

[10] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2009.

[11] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioberg, "Live wide-area migration of virtual machines including local persistent state," in *ACM/Usenix VEE*, June 2007.

[12] T. Wood, P. Shenoy, A. Gerber, K. Ramakrishnan, and J. V. der Merwe, "The Case for Enterprise-Ready Virtual Private Clouds," in *Proc. of HotCloud Workshop*, 2009.

[13] K. Ramakrishnan, P. Shenoy, and J. Van der Merwe, "Live data center migration across wans: A robust cooperative context aware approach," in *ACM SIGCOMM Workshop on Internet Network Management (INM)*, (Kyoto, Japan), aug 2007.

[14] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Y. Wang, "Seamless live migration of virtual machines over the man/wan," *Future Gener. Comput. Syst.*, vol. 22, no. 8, pp. 901–907, 2006.

[15] T. Hirofuchi, H. Nakada, H. Ogawa, S. Itoh, and S. Sekiguchi, "A live storage migration mechanism over wan and its performance evaluation," in *VIDC'09: Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing*, (Barcelona, Spain), ACM, 2009.

[16] T. Hirofuchi, H. Ogawa, H. Nakada, S. Itoh, and S. Sekiguchi, "A live storage migration mechanism over wan for relocatable virtual machine services on clouds," in

CCGRID'09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, (Shanghai, China), IEEE Computer Society, 2009.

[17] 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 *IEEE International Conference on Cluster Computing*, 2008.

[18] J. Hamilton, "The Cost of Latency." http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx, Oct. 2009.

[19] VMWare, "VMmark Virtualization Benchmarks." http://www.vmware.com/products/vmmark/, Jan. 2010.

[20] M. Blaze, "NFS tracing by passive network monitoring," in *Proceedings of the USENIX Winter 1992 Technical Conference*, pp. 333–343, 1992.

[21] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, "A quantitative analysis of cache policies for scalable network file systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, no. 1, pp. 150–160, 1994.

[22] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD file system," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 5, p. 24, 1985.

[23] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," *ACM SIGOPS Operating Systems Review*, vol. 25, no. 5, p. 212, 1991.

[24] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, p. 4, USENIX Association, 2000.

[25] F. Buchholz, "The structure of the Reiser file system." http://homes.cerias.purdue.edu/ florian/reiser/reiserfs.php, Jan. 2006.

[26] Microsoft, "How NTFS Works." http://technet.microsoft.com/en-us/library/cc781134(WS.10).aspx, 2003.

[27] S. D.Pate, *UNIX Filesystems: Evolution, Design and Implementation.* WILEY, 2003.

[28] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[29] VMWare, "Virtual Disk Format 1.1." http://www.vmware.com/app/vmdk/?src=vmdk, Nov. 2007.

[30] Xen. http://www.xen.org.

[31] KVM, "Kernal Based Virtual Machine." http://www.linux-kvm.org.

[32] Xen, "Xen Users' Manual." http://bits.xensource.com/Xen/docs/user.pdf, 2008.

[33] KVM, "qemu-kvm-0.12 adds block migration feature." http://www.linux-kvm.com/content/qemu-kvm-012-adds-block-migration-feature, Jan. 2010.