Workload-Aware Live Storage Migration for Clouds [*]

Jie Zheng T. S. Eugene Ng
Rice University

Kunwadee Sripanidkulchai
NECTEC, Thailand

Abstract

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 paper 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 fully implemented system on KVM and a trace-driven framework, we show that our algorithm provides large performance benefits across a wide range of popular virtual machine workloads.

Categories and Subject Descriptors D.4.0 [*Operating Systems*]: General

General Terms Algorithms, Design, Experimentation, Performance

Keywords Live Storage Migration, Virtual Machine, Workload-aware, Scheduling, Cloud Computing

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. While many cloud users' early successes have been realized using a single cloud provider [4, 8], using multiple clouds to deliver services and having the flexibility to move freely among different providers is an emerging requirement [1]. 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 [16]. A key barrier to cloud adoption identified in the manifesto is data and application portability. Users who implement their applications using one cloud provider ought to have the capability and flexibility to migrate their applications back in-house or to other cloud providers in order 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, virtual machines (VMs) could be migrated from one cloud to another cloud to leverage better resource availability, to avoid hardware or network maintenance down-times, or to avoid power limitations in the source cloud. 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 paper we look at the enabling mechanisms without which migrations would remain a costly effort.

Live migration provides the capability to move VMs from one physical location to another while still running without any perceived degradation. Many hypervisors support live migration within the LAN [6, 10, 13, 17, 20, 25]. 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 VM must be migrated (i.e., memory migration), (ii) the storage or virtual disks used by the VM must be migrated, and (iii) existing client connections must be migrated while new client connections are directed to the new location. Memory migration and network connection migration for the wide area have been demonstrated to work well [5, 24]. However, storage migration inherently faces significant performance challenges because of its much larger size compared to memory.

In this paper, we improve the efficiency of migrating storage across the wide area. Our approach differs from the existing work in storage migration that treats storage as one large chunk that needs to be transferred from beginning to end. We introduce *storage migration scheduling* to transfer storage blocks according to a deliberately computed order. We develop a workload-aware storage migration scheduling algorithm that takes advantage of temporal locality, spatial locality, and access popularity – patterns commonly found in a wide range of I/O workloads – at the properly chosen granularity to optimize the data transfer. We show that our scheduling algorithm can be leveraged by previous work in storage migration

[*] This research was sponsored by NSF CAREER Award CNS-0448546, NeTS FIND CNS-0721990, NeTS CNS-1018807, by an IBM Faculty Award, an Alfred P. Sloan Research Fellowship, and by Microsoft Corp. Jie Zheng is additionally supported by an IBM Scholarship. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, IBM Corp., Microsoft Corp., the Alfred P. Sloan Foundation, or the U.S. government.

VEE'11, March 9–11, 2011, Newport Beach, California, USA.

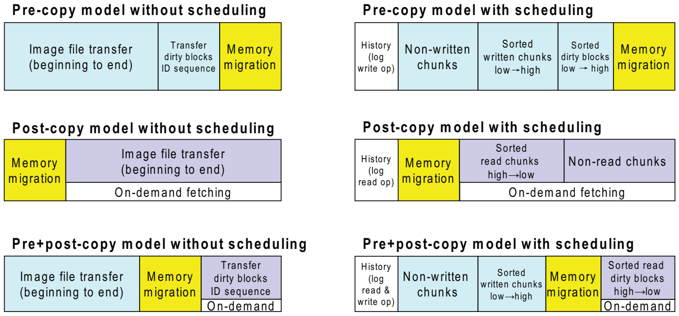

Pre-copy model without scheduling
Image file transfer (beginning to end)
Transfer dirty blocks ID sequence
Memory migration
Pre-copy model with scheduling
History (log write op)
Non-written chunks
Sorted written chunks low→high
Sorted dirty blocks low → high
Memory migration
Post-copy model without scheduling
Memory migration
Image file transfer (beginning to end)
On-demand fetching
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 without scheduling
Image file transfer (beginning to end)
Memory migration
Transfer dirty blocks ID sequence
On-demand
Pre+post-copy model with scheduling
History (log read & write op)
Non-written chunks
Sorted written chunks low→high
Memory migration
Sorted read dirty blocks high→low
On-demand

Figure 1. Models of live storage migration.

to greatly improve storage I/O performance during migration across a wide variety of VM workloads.

In the next section, we provide an overview of the existing storage migration technologies and the challenges that they face. Section 3 quantifies the locality and popularity characteristics we found in VM storage workload traces. Motivated by these characteristics, we present in Section 4 a novel storage migration scheduling algorithm that leverages these characteristics to make storage migration much more efficient. We present the implementation of our scheduling algorithm on KVM in Section 5 and evaluate its performance in Section 6. In Section 7, we further present trace-based simulation results for the scheduling algorithm under two additional migration models not adopted by KVM. Finally, we summarize our findings in Section 8.

2. Background

A VM consists of virtual hardware devices such as CPU, memory and disk. Live migration of a VM within a data center is quite common. It involves the transfer of the memory and CPU state of the VM from one hypervisor to another. However, live migration across the wide area requires not only transferring the memory and CPU state, but also virtual disk storage and network connections associated with a VM. While wide-area memory and network connection migration have matured [5, 19, 22, 24], wide-area storage migration still faces significant performance challenges. The VM's disk is implemented as a (set of) file(s) stored on the physical disk. Because sharing storage across the wide area has unacceptable performance, storage must be migrated to the destination cloud. And because of the larger size storage has compared to memory and the limitations in wide-area network bandwidth, storage migration could negatively impact VM performance if not performed efficiently.

2.1 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 1 depicts the three models on the left-hand side. In the pre-copy model as implemented by KVM [14] (a slightly different variant is also found in [5]), the entire virtual disk file is copied from beginning to end prior to memory migration. During the virtual disk copy, all write operations to the disk are logged. The dirty blocks are retransmitted, and new dirty blocks generated during this time are again logged and retransmitted. This dirty block retransmission process repeats until the number of dirty blocks falls below a threshold, then memory migration begins. During memory migration, dirty blocks are again logged and retransmitted iteratively.

The strength of the pre-copy model is that VM disk read operations at the destination have good performance because blocks are copied over prior to when the VM starts running at the destination. However, the pre-copy model has weaknesses. First, pre-copying may introduce extra traffic. If we had an oracle that told us when disk blocks are updated, we would send only the latest copy of disk blocks. In this case, the total number of bytes transferred over the network would be the minimum possible which is the total size of the virtual disk. Without an oracle, some transmitted blocks will become dirty and require retransmissions, 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 operations so that iterative dirty block retransmission can converge. While throttling is useful, it can degrade application I/O performance.

In the post-copy model [11, 12], 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 remote read. In background copying, the simplest strategy proposed by Hirofuchi et al. [12] is to copy blocks sequentially from the beginning of a virtual disk to the end. They also proposed an advanced background copy strategy which will be discussed later in Section 2.3. 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 remote reads, each block is transferred at most once ensuring that the total amount of data transferred for storage migration is minimized. However, remote reads incur extra wide-area delays, resulting in I/O performance degradation.

In the hybrid pre+post-copy model [15], 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 remote read model is employed for the dirty blocks. While this model still incurs extra traffic and remote read delays, 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 1 summarizes these three models.

2.2 Performance Degradation from Migration

While migration is a powerful capability, any performance degradation caused by wide area migration could be damaging to users that are sensitive to latency. Anecdotally, every 100 ms of latency costs Amazon 1% in sales and an extra 500 ms page generation time dropped 20% of Google's traffic [9]. In our analysis (details in Section 7) of a 10 GB MySQL database server that has 160 clients migrating over a 100 Mbps wide area link using the post-copy model, over 25,000 read operations experience performance degradation during migration due to remote read. This I/O performance degradation can significantly impact application performance on the VM. Reducing this degradation is key to making live migration a practical mechanism for moving applications across clouds.

2.3 Our Solution

Our solution is called *workload-aware storage migration scheduling*. Rather than copying the storage from beginning to end, we deliberately compute a schedule to transfer storage at the appropriate granularity which we call *chunk* and *in the appropriate order* to minimize performance degradation. Our schedule is computed to take advantage of the particular I/O locality characteristics of

Model		Pre-copy [5, 14]	Pre+post-copy [15]	Post-copy [11, 12]
Application Performance Impact	Write Operation Degradation	Yes	No	No
	Read Operation Degradation	No	Medium	Heavy
	Degradation Time	Long	Medium	Long
	I/O Operations Throttled	Yes	No	No
Total Migration Time		>> Best	> Best	Best
Amount of Migrated Data		>> Best	> Best	Best

Table 1. Comparison of VM storage migration methods.

the migrated workload and can be applied to improve any of the three storage migration models as depicted on the right-hand side of Figure 1. To reduce the extra migration traffic and throttling under the pre-copy model, our scheduling algorithm groups storage blocks into chunks and sends the chunks to the destination in an optimized order. Similarly, to reduce the number of remote reads under the post-copy model, scheduling is used to group and order storage blocks sent over during background copying. In the hybrid pre+post-copy model, scheduling is used for both the pre-copy phase and the post-copy phase to reducing extra migration traffic and remote reads.

Hirofuchi et al. [12] proposed an advanced strategy for background copying in the post-copy model by recording and transferring frequently accessed ext2/3 block groups first. While their proposal is similar in spirit to our solution, there are important differences. First, their proposal is dependent on the use of an ext2/3 file system in the VM. In contrast, our solution is general without any file system constraints, because we track I/O accesses at the raw disk block level which is beneath the file system. Second, the access recording and block copying in their proposal are based on a fixed granularity, i.e. an ext2/3 block group. However, our solution is adaptive to workloads. We propose algorithms to leverage the locality characteristics of workloads to decide the appropriate granularity. As our experiments show, using an incorrect granularity leads to a large loss of performance. Third, we show how our solution can be applied to all three storage migration models and experimentally quantify the storage I/O performance improvements.

To our knowledge, past explorations in memory migration [6, 10] leverage memory access patterns to decide which memory pages to transfer first to some extent. However, comparing to our technique, there are large differences.

In a pre-copy model proposed by Clark et al. [6], only during the iterative dirty page copying stage, the copying of pages that have been dirtied both in the previous iteration and the current iteration is postponed. In a post-copy model proposed by Hines and Gopalan [10], when a page that causes a page fault is copied, a set of pages surrounding the faulting page is copied together. This is done regardless of the actual popularity of the set of surrounding pages, and continues until the next page fault occurs.

In contrast, our scheduling technique for storage migration (1) is general for the three storage migration models as discussed above, (2) uses actual access frequencies, (3) computes fine-grained access-frequency-based schedules for migration throughout the disk image copying stage and the iterative dirty block copying stage, (4) automatically computes the appropriate chunk size for copying, and (5) proactively computes schedules based on a global view of the entire disk image and the access history, rather than reacting to each local event. Furthermore, under typical workloads, memory and storage access patterns are different, and the constraints in the memory and storage subsystems are different. Techniques that work well for one may not necessarily work well for the other. Our technique is tailored specifically for storage migration and storage access pattern.

Workload Name	VM Configuration	Server Application	Default # 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 2. VMmark workload summary.

3. Workload Characteristics

To investigate storage migration scheduling, we collect and study a modest set of VMware VMmark virtualization benchmark [23] I/O traces. Our trace analysis explores several I/O characteristics at the time-scale relevant to storage migration to understand whether the history of I/O accesses prior to a migration is useful for optimizing storage migration. It is complementary to existing general studies of other storage workloads [2, 18, 21].

3.1 Trace Collection

VMmark includes servers, listed in Table 2, that are representative of the applications run by VMware users. We collect traces for multiple client workload intensities by varying the number of client threads. A trace is named according to the server type and the number of client threads. For example, “fs-45” refers to a file server I/O trace with 45 client threads. Two machines with a 3GHz Quad-core AMD Phenom II 945 processor and 8GB of DRAM are used. 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. To collect the I/O trace, 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. `tcpdump` is used on the virtual network interface 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 [3, 7] and requires no special OS instrumentation. We experimentally confirmed that the tracing framework introduces negligible application performance degradation compared to placing the virtual disk directly on the hypervisor’s locally attached disk. We trace I/O operations at the disk sector granularity – 512 bytes. We call each 512 byte sector a block. However, this is generally not the file system block size. Each trace entry includes the access time, read or write, the offset in the VMDK file, and the data length. Each trace contains the I/O operations executed by a server over a 12 hour period.

3.2 History and Migration Periods

Let t denote the start time of a migration. Each I/O trace analysis is performed 20 times with different randomly selected migration start times $t \in [3000s, 5000s]$, where $0s$ represents the beginning of the trace. For simplicity, we use a fixed history period of 3000 seconds before t, and a fixed storage migration period of $(2 \times image\_size + memory\_size)$ / $bandwidth$ seconds. The latter

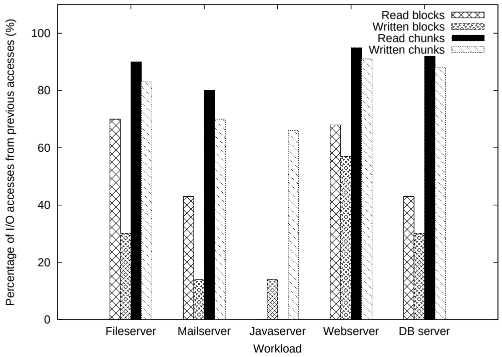


Figure 2. The temporal locality of I/O accesses as measured by the percentage of accesses in the migration that was also previously accessed in the history. The block size is 512B and the chunk size is 1MB. Temporal locality exists in all of the workloads, but is stronger at the chunk level. The Java server has very few read accesses resulting in no measurable locality.

corresponds to a pessimistic case that during the transfer of the disk image, all the blocks were written to by the VM and the entire image needs to be retransmitted. Other reasonable choices for these periods could be used, but the qualitative findings from our analysis are not expected to change. $image\_size$, $memory\_size$, and workload (i.e. # client threads) are as specified in Table 2, and $bandwidth$ is 100 Mbps in the following analysis.

3.3 Temporal Locality Characteristics

Figure 2 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 ≥ 3 in the history. 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 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 will poorly predict the write accesses in migration.

Figure 2 also shows that both read and write temporal locality improves dramatically when 1MB chunk is used as the basic unit of counting accesses. This is explained next.

3.4 Spatial Locality Characteristics

We find strong spatial locality for write accesses. 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 blocks and its closest neighbor block that was written in history. The distance is defined as $(block\_id\_difference * blocksize)$. Figure 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 an equal number of simulated random write accesses, which confirm that the spatial locality found in the real trace is significant. Taken together, in the file server trace, $32\% + 68\% * 90\% = 93.2\%$ of the written blocks in the migration are found within a range of 839KB of the written blocks in history.

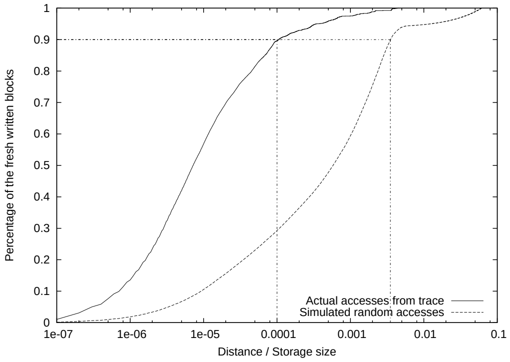


Figure 3. Spatial locality of file server writes and simulated random writes measured by normalized distance to closest block written in history. The 90th percentiles are 0.0001 and 0.0035.

This explains why, across all workloads, the temporal locality of write accesses increases dramatically in Figure 2 when we consider 1MB chunk instead of 512B block as the basic unit of counting accesses. The temporal locality of read accesses also increases. The caveat is that as the chunk size increases, although the percentage of covered accessed blocks in migration will increase, each chunk will also cover more unaccessed blocks. Therefore, to provide useful read and write access prediction, a balanced chunk size is necessary and will depend on the workload (see Section 4).

3.5 Popularity Characteristics

Popular chunks in history are also likely to be popular in migration. We rank chunks by their read/write frequencies and compute the rank correlation between the ranking in history and the ranking in migration. Figure 4 shows that a positive correlation exists for all cases except for the Java server read accesses at 1MB and 4MB chunk sizes.[1] 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. A balanced chunk size is required to exploit this popularity characteristic effectively (see Section 4).

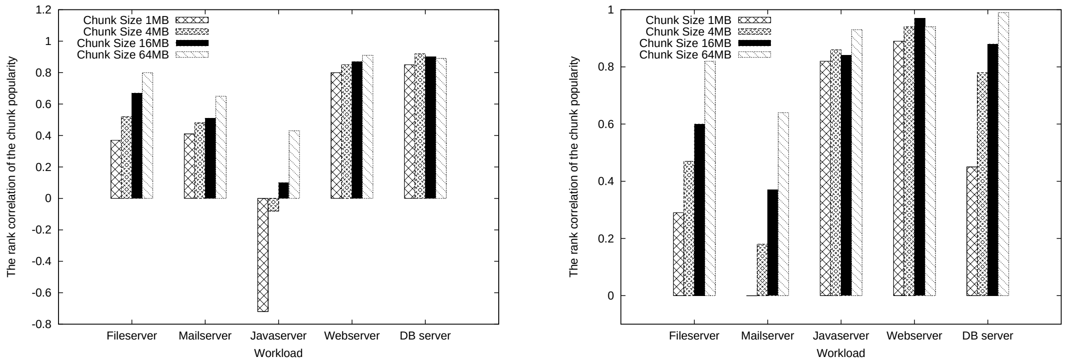
4. Scheduling Algorithm

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 and throttling 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. The algorithm presented below is conceptual. In Section 5, we present several implementation techniques.

4.1 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 Section 6 and 7. Therefore the

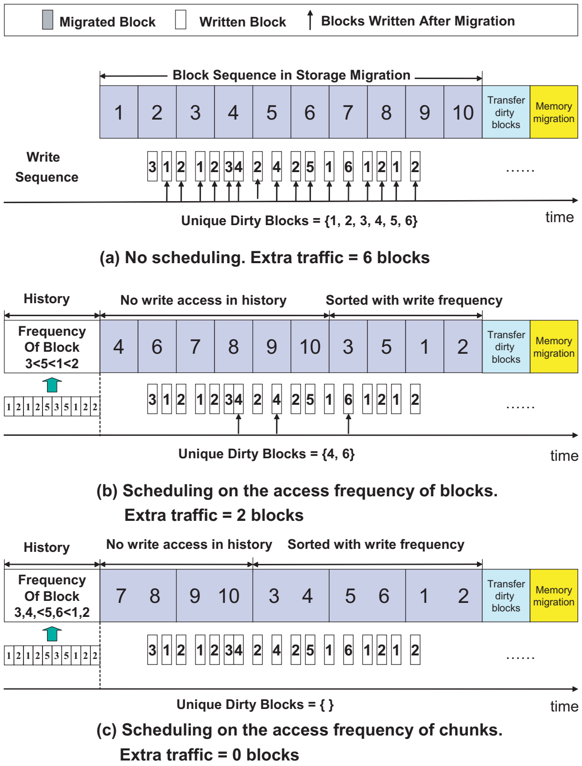
[1] This is because the Java server has extremely few read accesses and little read locality



(a) Read Access

(b) Write Access

Figure 4. The rank correlation of the chunk popularity in history vs. in migration.

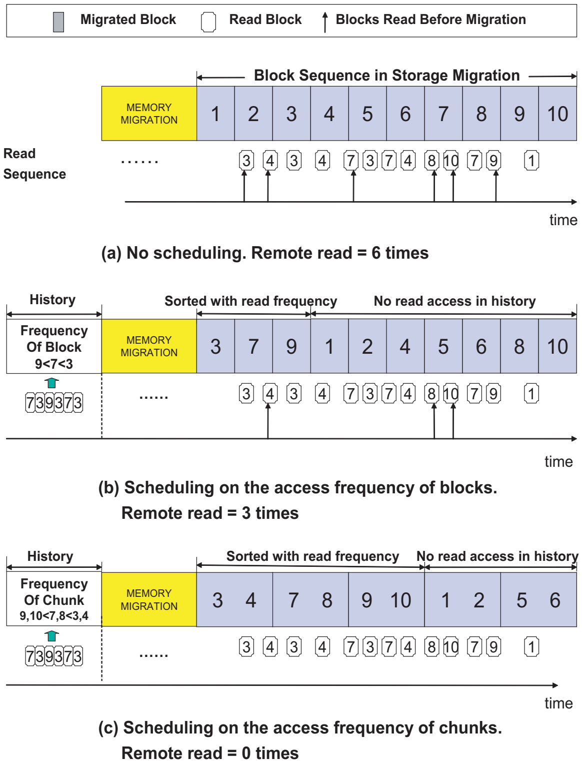

Migrated Block
Written Block
Blocks Written After Migration
Block Sequence in Storage Migration
1 2 3 4 5 6 7 8 9 10
Transfer dirty blocks
Memory migration
Write Sequence
3 1 2 1 2 3 4 2 4 2 5 1 6 1 2 1 2
……
Unique Dirty Blocks = {1, 2, 3, 4, 5, 6}
time
History
No write access in history
Sorted with write frequency
Frequency Of Block 3<5<1<2
4 6 7 8 9 10 3 5 1 2
1 2 1 2 5 3 5 1 2 2
Unique Dirty Blocks = {4, 6}
No write access in history
Sorted with write frequency
Frequency Of Block 3,4,<5,6<1,2
7 8 9 10 3 4 5 6 1 2
Unique Dirty Blocks = { }

(a) No scheduling. Extra traffic = 6 blocks

(b) Scheduling on the access frequency of blocks. Extra traffic = 2 blocks

(c) Scheduling on the access frequency of chunks. Extra traffic = 0 blocks

Figure 5. A simple example of the scheduling algorithm applied to the pre-copy model.



(a) No scheduling. Remote read = 6 times

(b) Scheduling on the access frequency of blocks. Remote read = 3 times

(c) Scheduling on the access frequency of chunks. Remote read = 0 times

Figure 6. A simple example of the scheduling algorithm applied to the post-copy model.

memory overhead for maintaining this history is very small. Different migration models are sensitive to different types of I/O accesses as discussed in Section 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, 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.

4.2 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 and 6 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 respectively. 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 examples, 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, a total of 6 blocks are dirtied after they have been transmitted and have to be resent. Similarly, under post-copy, there are 6 remote read operations where a block is needed before it is transferred to the destination.

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 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 and 6, when we schedule the blocks according to their access frequencies, the extra traffic and remote reads can be reduced from 6 to 2 and from 6 to 3 respectively.

In the example, blocks $\{4{,}6\}$ in the pre-copy model and 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 contiguous blocks. 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 4.3.

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 schedule 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 0.

The scheduling algorithm is summarized in pseudocode as Figure 7 shows. 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.

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 from high to low read frequency. The time complexity is $O(n \cdot log(n))$, the space complexity is $O(n)$, where n is the number of dirty blocks.

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.

```
DATA STRUCTURE IN ALGORITHM:
-op_flag: the flag of operation to indicate it is a read or a write
-$Q_{history}$: the queue of access operations collected from history
-$\alpha$: the fraction of simulated history.
-$L_b(op\_flag)$: A block access list of $< block_{id}, time >$
-$L_{cfreq}(op\_flag)$: A list of $< chunk_{id}, frequency >$
-$L_{schunk}(op\_flag)$: A list of $chunk_{id}$ sorted by frequency
-$L_{nchunk}(op\_flag)$: A list of $chunk_{id}$ not accessed in history

INPUT OF ALGORITHM: $Q_{history}, model\_flag$ and $\alpha$ ∈[0, 1]
OUTPUT OF ALGORITHM: migration schedule $S_{migration}$

$S_{migration} = \{\ \}$;
IF $((model\_flag == PRE\_COPY)$
  $\|(model\_flag == PRE + POST\_COPY))$
    $S_{migration}$=GetSortedMigrationSequence(WRITE);
ELSE $(model\_flag == POST\_COPY)$
    $S_{migration}$=GetSortedMigrationSequence(READ);
RETURN $S_{migration}$;

FUNCTION GetSortedMigrationSequence($op\_flag$)
    $L_b(op\_flag) =$ Convert $\forall OP \in Q_{history}$
      whose $flag == op\_flag$ into $< block_{id}, time >$;
    $chunksize$=ChunkSizeEstimation($L_b(op\_flag)$,$\alpha$);
    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 $block_k \in L_b(op\_flag)$;
      $frequency_{block_k}$ =# of times $block_k$ appearing in $L_b(op\_flag)$;
    END FOR
    $L_{cfreq}(op\_flag) = \{(chunk_i, frequency_i)|frequency_i > 0\}$;
    IF $(op\_flag == WRITE)$
      $L_{schunk}(op\_flag)$ =Sort $L_{cfreq}(op\_flag)$ by $freq$ low→high
    ELSE
      $L_{schunk}(op\_flag)$ =Sort $L_{cfreq}(op\_flag)$ by $freq$ high→low
    (chunks with the same frequency are sorted by id low→high)
    $L_{nchunk}(op\_flag) = S_{all} - L_{schunk}(op\_flag)$ with id low→high;
    IF $op\_flag == WRITE$
      return $\{L_{nchunk}(op\_flag), L_{schunk}(op\_flag)\}$;
    ELSE
      return $\{L_{schunk}(op\_flag), L_{nchunk}(op\_flag)\}$;
```

Figure 7. Scheduling algorithm.

4.3 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 8 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 $L_b(op\_flag)$ 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. We set the lower bound of the chunk size to 512B. The algorithm also bounds the maximum selected chunk size. In the evaluation, we set this bound to 1GB.

The algorithm pseudocode is shown in Figure 9. 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.

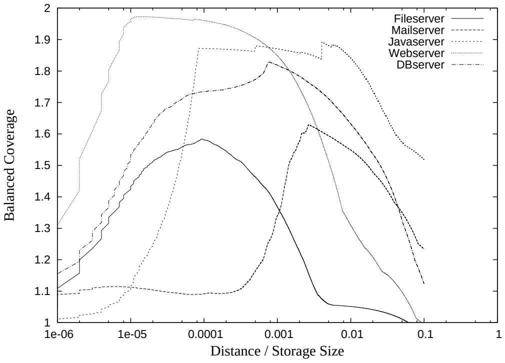


Figure 8. A peak in balanced coverage determines the appropriate chunk size for a given workload.

5. Implementation

We have implemented the scheduling algorithm on kernel-based virtual machine (KVM). KVM consists of a loadable kernel module that provides the core virtualization infrastructure and a processor specific module. It also requires a user-space program, a modified QEMU emulator, to set up the guest VM's address space, handle the VM's I/O requests and manage the VM. KVM employs the pre-copy model as described in Section 2. When a disk I/O request is issued by the guest VM, the KVM kernel module forwards the request to the QEMU block driver. The scheduling algorithm is therefore implemented mainly in the QEMU block driver. In addition, the storage migration code is slightly modified to copy blocks according to the computed schedule rather than sequentially.

Incremental schedule computation

A naive way to implement the scheduling algorithm is to add a history tracker that simply records the write operations. When migration starts, the history is used to compute, on-demand, the migration schedule. This on-demand approach works poorly in practice because the computations could take several minutes even for a history buffer of only 5,000 operations. During this period, other QEMU management functions must wait because they share the same thread. The resulting burst of high CPU utilization also affects the VM's performance. Moreover, storage migration is delayed until the scheduler finishes the computations.

Instead, we have implemented an efficient incremental scheduler. The main idea is to update the scheduled sequence incrementally when write requests are processed. The scheduled sequence is thus always ready and no extra computation is needed when migration starts. The two main problems are (1) how to efficiently sort the chunks based on access frequency and (2) how to efficiently select the optimal chunk size. We first describe our solution to problem (1) assuming a chunk size has been chosen; then we discuss our solution to problem (2).

Incremental and efficient sorting

We create an array and a set of doubly linked lists for storing access frequencies of chunks as shown in Figure 10. The index of the array is the chunk ID. Each element in the array (square) is an object that contains three pointers that point to an element in the frequency list (oval) and the previous and next chunks (square) that have the same frequency. An element in the frequency list (oval) contains the value of the access frequency, a pointer to the head of the doubly linked list containing chunks with that frequency and two pointers to its previous and next neighbors in the frequency list. The frequency list (oval) is sorted by frequency. Initially, the frequency list (oval) only has one element whose frequency is zero and it points to the head of a doubly linked list containing all chunks. When a write request arrives, the frequency of the accessed chunk is increased by 1. The chunk is moved from the old chunk list to the head of the chunk list for the new frequency. A new frequency list element is added if necessary. The time complexity of this update is $O(1)$ and the space complexity is $O(n)$ where n is the number of chunks. The scheduling order is always available by simply traversing the frequency list in increasing order of frequency and the corresponding chunk lists. This same approach is used to maintain the schedule for dirty blocks that need to be retransmitted.

```
DATA STRUCTURE IN ALGORITHM:
-$L_b(op\_flag)$: A block access list of $< block_{id}, time >$
-$\alpha$: the fraction of simulated history.
-$total\_block$: the number of total blocks in the storage.
-$upper, low\_bound$: max & min allowed chunk size, e.g. 512B-1GB.
-$S_{H1}$,$S_{H2}$: the sets of blocks accessed in the first and 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 $S_{H2}$.
-$S_{NormDistanceCDF}$: 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.
-$BalancedCoverage_{max}$: the maximum value of $balanced\_coverage$
-$ND_{BCmax}$: the neighborhood size (a normalized distance) that
  maximizes $balanced\_coverage$

INPUT OF ALGORITHM: $L_b(op\_flag)$ and $\alpha$ $\in$[0, 1]
OUTPUT OF ALGORITHM: $chunk\_size$

FUNCTION ChunkSizeEstimation($L_b(op\_flag)$,$\alpha$)
  $max\_time$= the duration of $L_b(op\_flag)$;
  FOR EACH $< block_{id}, time > \in L_b(op\_flag)$
    IF $time < max\_time * \alpha$
      Add $block_{id}$ into $S_{H1}$;
    ELSE ADD $block_{id}$ into $S_{H2}$;
  END FOR
  FOR EACH $block_{id} \in S_{H2}$
    $NormDistance$=$\frac{\{min\ (|block_{id} - m|) \forall m \in S_{H1}\}}{total\_block}$;
    Add $NormDistance$ into $S_{NormDistance}$;
  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
    $storage\_coverage = \frac{\#\ of\ unique\ blocks\ in\ ES_{H1}}{total\_block}$;
    $access\_coverage$ =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 FOR
  $chunk\ size = ND_{BCmax} * total\_block * block\_size$;
  IF $(chunk\_size == 0)$
    $chunk\_size = lower\_bound$;
  ELSE IF $chunk\_size > upper\ bound$
    $chunk\_size = upper\_bound$;
  RETURN $chunk\_size$;
```

Figure 9. Algorithm for chunk size selection.

Incremental and efficient chunk size selection

To improve efficiency, we estimate the optimal chunk size periodically. The period is called a round and is defined as the duration of every N write operations. N is also the history buffer size. Only the latest two rounds of history are kept in the system, called the previous round and the current round. At the end of each round,

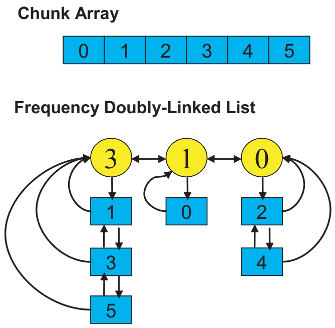


Figure 10. Data structures for maintaining the sorted access frequencies of chunks.

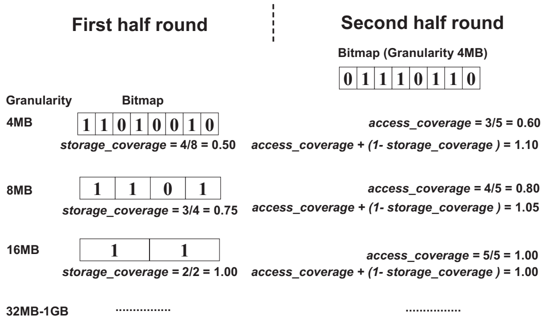

First half round
Second half round
Bitmap (Granularity 4MB)
0 1 1 1 0 1 1 0
Granularity
Bitmap
4MB
1 1 0 1 0 0 1 0
storage_coverage = 4/8 = 0.50
access_coverage = 3/5 = 0.60
access_coverage + (1- storage_coverage) = 1.10
8MB
1 1 0 1
storage_coverage = 3/4 = 0.75
access_coverage = 4/5 = 0.80
access_coverage + (1- storage_coverage) = 1.05
16MB
1 1
storage_coverage = 2/2 = 1.00
access_coverage = 5/5 = 1.00
access_coverage + (1- storage_coverage) = 1.00
32MB-1GB
....
....

Figure 11. Data structures for chunk size selection.

the optimal chunk size is computed and it is used as the chunk size for tracking the access frequency in the next round using the aforementioned data structure. Initially, a default chunk size of 4MB is used.

It is not feasible to do a fine grained search for the optimal chunk size in reality. A trade-off between precision and efficiency must be made. We use a simple example to depict our idea in Figure 11. We pick 8 exponentially increasing candidate chunk sizes: 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, and 1GB. Each round is subdivided into two halves. In the first half, each candidate chunk size is associated with a bitmap for logging which chunk is written to. When a chunk gets the first write access, the bit for that chunk is set to 1 and the storage coverage at that candidate chunk size is updated. Storage coverage is defined as $\frac{\#\ of\ set\ bits}{total\ bits}$. The storage coverage for the candidate chunk size 4MB is 0.5 in the example. In the second half, a new bitmap for the smallest candidate chunk size of 4MB is created to log the new accessed chunks in the second half. Moreover, for each candidate chunk size, we maintain the corresponding access coverage. When a 4MB chunk gets its first write access in the second half, the access bit is set to 1. Access coverage is defined as the percentage of bits set in the second half which are also set in the first half. The access coverage for each candidate chunk size is updated by checking whether the corresponding bit is also set in the previous bitmap in the first half. An update only occurs when a chunk is accessed for the first time. Each update operation is $O(1)$ time. At the end of each round, the storage coverage and access coverage for all candidate chunk sizes are available and we pick the chunk size that maximizes the $balanced\_coverage = access\_coverage + (1 - storage\_coverage)$. In Figure 11, the access coverage for chunk size 4MB is 0.6 and its balanced coverage is 1.1, which is the highest. Thus, 4MB is the selected chunk size for counting access frequencies in the next round.

When migration starts, the frequency list from the previous round is used for deciding the migration sequence and chunk size since the current round is incomplete.

We evaluate the impact of the scheduling algorithm on the latency and throughput of write operations in the file server. The history buffer size is set to 20,000 and we measure 100,000 write operations. With scheduling, the average write latency increases by only 0.97%. The application level write throughput reduces by only 1.2%. Therefore, the incremental scheduler is lightweight enough to be enabled at all time, or if desired, enabled manually only prior to migration.

6. Evaluation on KVM

The experimental platform consists of two machines as the source and destination for migration. Each of them is configured with a 3GHz Quad-core AMD Phenom II X4 945 processor, 8GB of DRAM, runs Ubuntu 9.10 with Linux kernel (with the KVM module) version 2.6.31 and QEMU version 0.12.50. The network is a 1Gbps Ethernet with varying rate-limits to emulate different network speeds.

During the initial transfer of the image file, we configure the migration process to read and copy image file data over the network at a relatively large granularity of 4MB to achieve high disk I/O throughput. Consequently, our scheduling algorithm chunk size is constrained to be a multiple of 4MB. In contrast, we configure QEMU to track dirty blocks at a much finer granularity of 128KB to keep unnecessary data retransmission down. When any bytes of a 128KB block is written to, the entire 128KB block is considered dirty and is retransmitted. We define the disk dirty rate as the number of dirty bytes per second. Setting the granularity to track dirty blocks larger (e.g. the QEMU default granularity of 1MB) will greatly increase the amount of unnecessary retransmissions. The history buffer size is set to 20,000. We run each experiment 3 times with randomly chosen migration start times in $[900s, 1800s]$. The average result across the 3 runs is reported. We present results for the mail server and the file server in VMmark, which have, respectively, a moderate and an intensive I/O workload (average write rate 2.5MB/s and 16.7MB/s). The disk dirty rate during migration could be much higher than the disk write rate because of the dirty block tracking granularity discussed above. Note that the benefit of scheduling is not pronounced when the I/O workload is light.

6.1 Benefits Under Pre-Copy

The following results show the benefits of scheduling under the pre-copy model since this model is employed by KVM. We measure the extra traffic and the time for the migration with and without scheduling. Migration time is defined as the duration from the migration command is received to the time when the VM resumes on the destination machine. Extra traffic is the total size of the retransmitted blocks.

QEMU provides a flow-control mechanism to control the available network bandwidth for migration. We vary the bandwidth from 224Mbps to 128Mbps. Figure 12 shows that when the bandwidth is 224Mbps, with scheduling, the extra traffic is reduced by 25%, and the migration time is reduced by 9 seconds for the mail server. When the network bandwidth decreases, we expect the extra traffic and the migration time to increase. With the scheduling algorithm, the rate at which extra traffic increases is much lower. At a network bandwidth of 128Mbps, the extra traffic is reduced by 41% from 3.16GB to 1.86GB. The migration time is reduced by 7% or 136s.

When the network bandwidth is 128Mbps, we find that the migration of the file server does not converge because its disk dirty rate is too high relative to the network bandwidth, and QEMU has no built-in mechanism to address this problem. There are two pos-

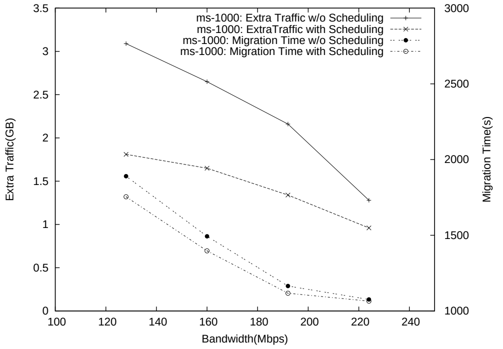


Figure 12. Improvement in extra traffic and migration time for the mail server.

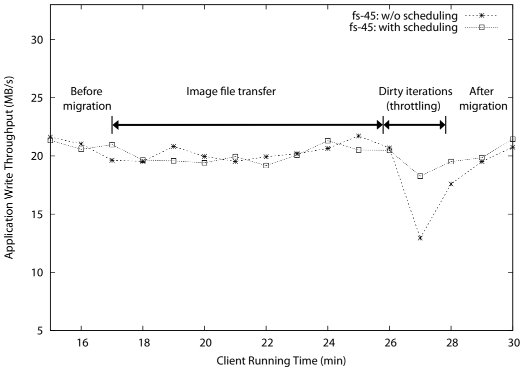


Figure 13. File server performance during migration.

sible solutions: stop the VM [26], or throttle disk writes (mentioned in [5]), with the latter achieving a more graceful performance degradation. However, there is no definitive throttling mechanism described in the literature. Thus, we have implemented our own simple throttling variants called aggressive throttling and soft throttling. During each dirty block retransmission iteration, the throttler limits the dirty rate to at most half of the migration speed. Whenever it receives a write request at time t_0, it checks the dirty block bitmap to compute the number of new dirty blocks generated and estimates the time t it takes to retransmit these new dirty blocks at the limited rate. If the next write operation comes before $t_0 + t$, aggressive throttling defers this write operation until time $t_0 + t$ by adding extra latency. In contrast, soft throttling maintains an average dirty rate from the beginning of the dirty iteration. It adds extra latency to a write operation only when the average dirty rate is larger than the limit. Aggressive throttling negatively impacts more write operations than soft throttling, but it can shorten the migration time and reduce extra traffic. We show results using both throttling variants.

We fix the network bandwidth at 128Mbps and vary the number of file server clients from 30 to 60 to achieve different I/O intensities. First, we evaluate the impact to I/O performance by measuring the number of operations penalized by extra latency due to throttling. Table 3 shows that when soft throttling is applied, scheduling can help to reduce the number of throttled operations from hundreds to a small number (≤ 22). When aggressive throttling is applied, the number of throttled operations is in thousands. The scheduling algorithm can reduce the throttled operations by 25-28% across the three workloads. In order to know how this benefit is translated to application level improvement, Figure 13 shows the average write throughput as reported by the VMmark client before,

		fs-30 no-schel	**fs-30** schel	**fs-45** no-schel	**fs-45** schel	**fs-60** no-schel	**fs-60** schel
Soft-Throt	# of OP with Extra-latency	122	0	205	21	226	22
	Extra Traffic(GB)	1.08	0.82	1.62	1.45	2.01	1.53
	Migration Time(s)	585	575	642	625	694	665
Aggr-Throt	# of OP with Extra-latency	1369	1020	3075	2192	4328	3286
	Extra Traffic(GB)	0.84	0.80	1.51	1.33	1.93	1.38
	Migration Time(s)	582	574	628	613	674	641

Table 3. No scheduling vs. scheduling for file server with different number of clients (network bandwidth = 128Mbps).

during and after migration under the fs-45 workload with aggressive throttling. Migration starts at the 17th minute. Before that, the average write throughput is around 21MB/s. When migration starts, the average write throughput drops by about 0.5-1MB/s because the migration competes with the application for I/O capacity. When the initial image file transfer finishes at around 25.7 minutes, the dirty iteration starts and throttling is enabled. The dirty iteration lasts for 2 minutes and 1 minute 43 seconds for no-scheduling and scheduling respectively. Without scheduling, the average write throughput during that time drops to 12.9MB/s. With scheduling, the write throughput only drops to 18.3MB/s. Under the fs-60 workload, the impact of throttling is more severe and scheduling provides more benefits. Specifically, throughput drops from 25MB/s to 10MB/s without scheduling, but only drops to 17MB/s with scheduling. Under the relatively light fs-30 workload, throughput drops from 14MB/s to 12.2MB/s without scheduling and to 13.5MB/s with scheduling.

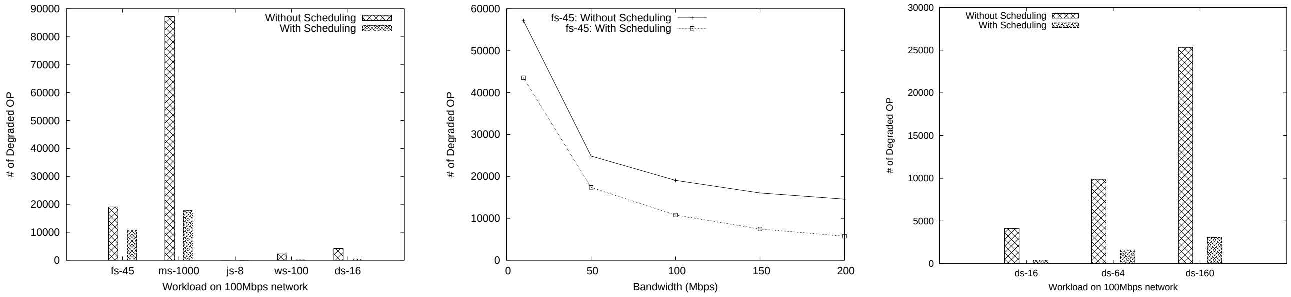
Table 3 also shows that the extra traffic and migration time are reduced across the three workloads with the scheduling algorithm. Take the aggressive throttling scenario as an example, under fs-45, the extra traffic is reduced by 180MB and the migration time is reduced by 15s. Under fs-60, the I/O rate and the extra traffic both increase. The scheduling algorithm is able to reduce the extra traffic by 28% from 1.93GB to 1.38GB and reduce the migration time by 33s. The benefit of scheduling increases with I/O intensity.

We have also experimented with fs-45 under 64Mbps and 32Mbps of network bandwidth with aggressive throttling. At 64Mbps, scheduling reduces throttled operations by 2155. Extra traffic is reduced by 300MB and migration time is reduced by 56s. At 32Mbps, throttled operations is reduced by 3544. Extra traffic and migration time are reduced by 600MB and 223s respectively. The benefit of scheduling increases with decreasing network bandwidth.

7. Trace-Based Simulation

Since KVM only employs the pre-copy model, we perform trace-based simulations to evaluate the benefits of scheduling under the post-copy and pre+post-copy models. Although we cannot simulate all nuances of a fully implemented system, we believe our results can provide useful guidance to system designers.

We use the amount of extra traffic and the number of degraded operations that require remote reads as the performance metrics for evaluation. Extra traffic is used to evaluate the pre+post-copy model. It is the total size of retransmitted blocks. The number of degraded operations is used to evaluate both post-copy and pre+post-copy models. The performance of a read operation is degraded when it needs to remotely request some data from the source machine which incurs at least one network round trip delay. Therefore, a large number of degraded operations is detrimental to VM performance.



(a) Different Workload (b) Different Bandwidth (c) Different Number of Clients

Figure 14. The improvement of degraded operations under the post-copy model.

7.1 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 1. Each experiment is run 3 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. The default α for chunk size computation is 0.7. 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 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.

7.2 Benefits Under Post-Copy

Figure 14 shows the benefits of scheduling in terms of the number of degraded operations under the various server types, bandwidths, and workload intensities. The reductions in the number of degraded operations are 43%, 80%, 95% and 90% in the file server, mail server, web server and database server respectively when the network bandwidth is 100 Mbps. The traffic that is involved in remote reads is also reduced. For example, 172MB and 382MB are saved with the scheduling algorithm for the file server and mail server respectively. If the history size is reduced significantly from 50,000 operations to merely 1,000 operations, the corresponding benefits are reduced to 19%, 54%, 75%, and 83% respectively. Thus, even maintaining a short history could provide substantial benefits. 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 degraded operations because the migration time is longer. At 10Mbps, 13,601 (or 24%) degraded operations are eliminated by scheduling in the file server. For the mail server, web server and database server, their degraded operations are reduced by 45%, 52% and 79% respectively when the network bandwidth is 10Mbps. The traffic involved in remote reads for the file server is reduced from 1.2GB to 0.9GB and for the mail server from 2.6GB to 1.4GB.

When the number of clients increases, the read rate becomes more intensive. For example, the ds-160 workload results in 25,334 degraded operations when scheduling is not used. With scheduling, the degraded operations is reduced by 84%-90% under the ds-16, ds-64, and ds-160 workloads. When the number of the clients in the file server is increased to 70, there are 23,011 degraded operations. With scheduling, it can be reduced by 47%.

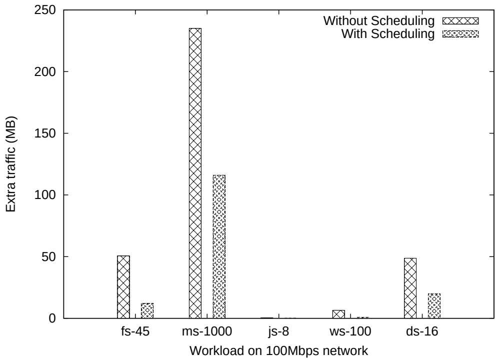


Figure 15. The improvement of extra traffic under the pre+post-copy model.

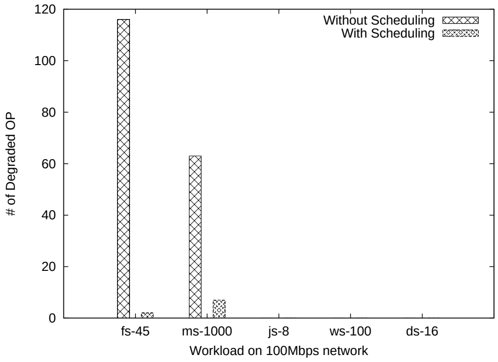


Figure 16. The improvement of degraded operations under the pre+post-copy model.

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

7.3 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 15 shows, the scheduling algorithm reduces the extra traffic in the five workloads by 76%, 51%, 67%, 86% and 59% respectively.

In the pre+post copy model, the degraded operations 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 16 shows that the Java server, web server and database server have no degraded operations because their amount of dirty data is small. But the file server and mail server suffer from degraded operations, and applying scheduling can reduce them by 99% and 88% respectively.

7.4 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 4 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.

8. Summary

Migrating virtual machines between clouds is an emerging requirement to support open clouds and to enable better service availability. We demonstrate that existing migration solutions can have poor I/O performance during migration that could be mitigated by taking a *workload-aware approach* to storage migration. We develop our insight on workload characteristics by collecting I/O traces of five representative applications to validate the extent of temporal locality, spatial locality and access popularity that widely exists. We then design a *scheduling algorithm* that exploits the individual virtual machine's workload to compute an efficient ordering of *chunks* at an appropriate chunk size to schedule for transfer over the network. We demonstrate the improvements introduced by work-load aware scheduling on a fully implemented system for KVM and through a trace-driven simulation 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 and significantly reduce the number of remote reads to improve the performance of post-copy and pre+post-copy model. Furthermore, the overhead introduced by our scheduling algorithm in KVM is low.

Next, we discuss and summarize the benefits of *workload-aware scheduling* under different conditions and workloads using the pre-copy migration model as an example.

Network bandwidth: The benefits of scheduling increases as the amount of available network bandwidth for migration decreases. Given that lower bandwidth results in a longer pre-copy period, the opportunity for any content to become dirty (i.e., requiring a retransmission) is larger. However, with scheduling, content that is likely to be written is migrated towards the end of the pre-copy period thus reducing the amount of time and the opportunity for it to become dirty.

Image size: The benefits of scheduling increases as the image size (i.e., the used blocks in the file system) gets larger even if the active I/O working set remains the same. A larger image size results in a longer pre-copy period, thus the opportunity for any content to become dirty is bigger. However, with scheduling, the working set gets transferred last, reducing the probability that migrated content becomes dirty despite the longer migration time.

I/O rate: As the I/O rate becomes more intense, the benefits of scheduling increases. With higher I/O intensity, the probability that any previously migrated content becomes dirty is higher. Again, by transferring active content towards the end of the pre-copy period, we lower the probability that it would become dirty and has to be retransmitted.

I/O characteristics: As the extent of locality or popularity becomes less pronounced, the benefits of scheduling decreases. For example, when the popularity of accessed blocks is more uniformly distributed, it is more difficult to accurately predict the access pattern. In the extreme case where there is no locality or popularity pattern in the workload, scheduling provides similar performance to non-scheduling.

These characteristics indicate that our workload-aware scheduling algorithm can improve the performance of storage migration under a wide range of realistic environments and workloads.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, and et. al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[2] 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*, 25(5):212, 1991.

[3] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, 1992.

[4] ”Amazon Web Services Blog”. Animoto - Scaling Through Viral Growth. http://aws.typepad.com/aws/2008/04/animoto—scali.html, April 2008.

[5] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schioberg. Live wide-area migration of virtual machines including local persistent state. In *ACM/Usenix VEE*, June 2007.

[6] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[7] M.D. Dahlin, C.J. Mather, R.Y. Wang, T.E. Anderson, and D.A. Patterson. A quantitative analysis of cache policies for scalable network file systems. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):150–160, 1994.

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

[9] James Hamilton. The Cost of Latency. http://perspectives.mvdirona.com/2009/10/31/ TheCostOfLatency.aspx, October 2009.

[10] Michael R. Hines and Kartik 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] Takahiro Hirofuchi, Hidemoto Nakada, Hirotaka Ogawa, Satoshi Itoh, and Satoshi 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, 2009. ACM.

[12] Takahiro Hirofuchi, Hirotaka Ogawa, Hidemoto Nakada, Satoshi Itoh, and Satoshi 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, 2009. IEEE Computer Society.

[13] Hai Jin, Li Deng, Song Wu, and Xuanhua Shi. Live virtual machine migration integrating memory compression with precopy. In *IEEE International Conference on Cluster Computing*, 2009.

[14] KVM. QEMU-KVM code. http://sourceforge.net/projects/kvm/files, January 2010.

[15] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and Incremental Whole-System Migration of Virtual Machines Using Block-Bitmap. In *IEEE International Conference on Cluster Computing*, 2008.

[16] Open Cloud Manifesto. Open Cloud Manifesto. http://www.opencloudmanifesto.org/, January 2010.

[17] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *USENIX'05: Proceedings of the 2005 Usenix Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.

[18] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, and J.G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *ACM SIGOPS Operating Systems Review*, 19(5):24, 1985.

[19] K.K. Ramakrishnan, Prashant Shenoy, and Jacobus 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.

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

[21] 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*, page 4. USENIX Association, 2000.

[22] Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees de Laat, Joe Mambretti, Inder Monga, Bas van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. Seamless live migration of virtual machines over the man/wan. *Future Gener. Comput. Syst.*, 22(8):901–907, 2006.

[23] VMWare. VMmark Virtualization Benchmarks. http://www.vmware.com/products/vmmark/, January 2010.

[24] Timothy Wood, Prashant Shenoy, Alexandre Gerber, K.K. Ramakrishnan, and Jacobus Van der Merwe. The Case for Enterprise-Ready Virtual Private Clouds. In *Proc. of HotCloud Workshop*, 2009.

[25] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[26] XEN. XEN Project. http://www.xen.org, January 2009.