Avoiding the Disk Bottleneck in Deduplicated VM Transfer

Jacob Slack, Samer Al-Kiswany, Matei Ripeanu
The University of British Columbia
Vancouver, BC, Canada

Email: jslack@alumni.ubc.ca, {samera, matei}@ece.ubc.ca

Abstract. This paper presents an optimization mechanism to increase the performance of cloud services that transfer groups of deduplicated virtual machine (VM) images. This is necessary as the naive data transfer approach for groups of deduplicated VM images is extremely inefficient as it generates highly random disk access pattern. The optimization mechanism presented significantly improves the VM image transfer performance through intelligent buffering, block-level transfer scheduling, and data indexing to increase disk access sequentiality.

1. Introduction
Data deduplication is a commonly adopted optimization in the cloud infrastructure for optimizing the storage [1] and the transfer [2] of virtual machines (VM). Data deduplication efficiently identifies and eliminates similarity across VM image files leading to a reduction in the amount of data that needs to be stored and/or transferred. This reduction in data size is significant: recent studies show that similarity across virtual machines can be as high as 96% [3].

Moving a group of VM images within, or across, data centers is a frequent operation to support application migration, new application deployment, as well as backup and maintenance operations. While deduplication reduces the overall size of a group of VM images it complicates their efficient transfer and re-incarnation at the destination site. The reason is mainly because a single block of deduplicated data can be part of multiple VM images and this introduces coupling between logically independent VM images.

The current naive approach for transferring a group of VMs over the network to a destination site works as follows: the destination site retrieves each block of the deduplicated VM images from the source site and writes it to all the VM images the block is part of at the destination. This approach is embarrassingly inefficient. The reason is that each data block (of typical size of 4-8KB) is often written to multiple VM images generating multiple, seemingly random, disk accesses. This disk access pattern significantly reduces the disk performance, due to the high disk rotation and seek overhead, leading to embarrassingly low performance for the complete VM transfer process. Experiments show that the performance degradation of this approach is significant leading to transfer rates with throughput in the order of tens of KBps [2] (two orders lower than sequential throughput of commodity disks, and one order of magnitude lower than typical WAN throughput).

This paper presents a mechanism to reduce the randomness in the disk access pattern when transferring groups of deduplicated VM images. This mechanism uses intelligent data buffering and optimized block ordering to increase disk access sequentiality. Analytical analysis of the proposed mechanism shows that the mechanism can bring tangible performance gains compared to the current approaches.

The rest of this paper details the deduplication techniques commonly used in VM image transfer and storage (section 2), surveys work related to the proposed mechanism (section 3), presents the design of the proposed mechanism (section 4), proves through analytical analysis the significant performance gains the proposed mechanism can bring (section 5), and presents experimental evaluation to estimate the performance gains the proposed mechanism can bring and to evaluate the proposed approach effectiveness in eliminating the disk access bottleneck (section 6). We conclude by discussing related issues (section 0).
2. Background
Data deduplication is a commonly used technique for detecting and eliminating similarities across VM images [1, 2, 4]. Deduplication divides the processed files into blocks, computes blocks’ identifiers using a collision-resistant hash function such as SHA1 or MD5, and compares these identifiers to detect similar blocks (as equal hashes indicate similar block content).

Two common solutions are adopted for dividing the file into blocks: fixed size blocks, and detecting block boundaries based on file content (a.k.a. variable size blocks). The first solution uses fixed-size blocks: i.e., it divides the file into equal size blocks (e.g. 4KB). The second approach identifies block boundaries (i.e., markers for block start and end) based on file’s content. For instance, the Low-Bandwidth File System (LBFS) [5] and JumboStore [6] both detect block boundaries by passing all successive 48 byte ‘windows’ of the file through a hash function and declaring a block boundary if the last 20 bits of the hash value are all zeros. The advantage of this approach is that, unlike fixed-blocks, the ability to detect block similarities is preserved even in the presence of data insertion and deletion.

The result of deduplication is a set of unique data blocks and a block map for each VM image. Each block map lists the blocks (and their order) that form the image. It is important to note that (in case of high similarity across files) most of the blocks will be part of more than one VM image.

3. Related work
VM Transfer. A number of VM migration tools use deduplication to reduce the amount of data transferred between the source and destination. Sapuntzakis et al. [4] present a system for virtual machine live migration that includes the VM images. The approach reduces the amount of data transferred using data deduplication between the transferred image and images at the destination. Hirofuchi et al. [7] and Bradford et al. [8] present an optimized VM migration system for migrating live (i.e. running) VM images. Finally, VMFlock [2] presents a deduplication based VM migration system optimized for migration of a group of VM images over WAN.

While the tools surveyed above adopt deduplication for VM migration, they focus on optimizing the system for live migration, without optimizing the disk migration. Our own experience demonstrates that the bottleneck for this use case is in the disk throughput when re-incarnating the VMs at the destination site (surprisingly, the network bandwidth, even in across WAN migrations, or computing hash functions for deduplication, are not bottlenecks).

The mechanism this paper proposes will enable these tools to achieve higher migration throughput.

4. Mechanism Design
The goal of the proposed mechanism is to increase the sequentiality of disk accesses during the VM transfer operation. While, the mechanism can be executed at the source or the destination the rest of this section assumes that the mechanism is running at the destination.

Further the proposed mechanism can be used with fixed size blocks as well as content based block boundaries. To simplify the analysis we analyze the mechanism when used with fixed size blocks.

The rest of this section details the mechanism system design.
A) System Design
The system is composed of two main components (Figure 1): The assembler and the block transfer module.

- **The Transfer Module**: Is the module responsible for implementing the block transfer protocol between source and destination. The module receives the transfer request from the assembler specifying which blocks to transfer, retrieves the blocks from the source, and returns the transferred blocks to the assembler.

- **The Assembler**: Is the central component responsible for assembling the VM images. The main engine in the assembler analyzes the VM blocks to select the next blocks to transfer, manages the block buffer and writes the blocks to the disk. The next subsection details the assembler main engine algorithm. The assembler uses two data structures during the assembly process:
  a. **The Block Buffer**: is an in-memory buffer for data blocks. The Block Buffer is implemented as a hash table accessed using the block identifiers. Each block in the buffer has a count of number of times it shows up across all VM images that are transferred. Every time a block is written to a VM image the counter is decremented. Once the counter reaches zero the block is removed from the buffer. The buffer can hold a maximum of $M$ data blocks. Further, the buffer has a threshold $T < M$ that indicates when the buffer is almost full.
  b. **The Images Block maps**: The system maintains a block map for every image. This shows the order of the blocks in the image. Additionally, each block map is divided into regions composed of $b$ consecutive data blocks. The block map maintains the following information per block: size, identifier (i.e., the block’s hash value), number of block occurrences in all images, and status (at-source, buffered in the block buffer, or on-disk at destination).

B) Block maps construction.
System employing deduplication maintain a minimal image block map (contains block hashes and block order) for each image. The Assembler analyzes the minimal block map provided by the source to build a map relating blocks to their list of locations. The analysis includes counting the number of occurrences of each block across VM images using an auxiliary hash table.

Assembler Algorithm
The assembler main engine follows the following steps:

1. The assembler retrieves the images minimal block maps from the source. These minimal block maps are produced when the VM images are deduplicated at the source. The assembler analyzes the minimal block maps and creates the extended block maps described in the previous subsection.
2. The assembler retrieves and writes sequentially to the disk the first \( k \) regions of the first VM image. To maintain sequentiality, any data block in these first \( k \) regions that is also part of another VM image is buffered in the block buffer.

3. When the number of data blocks in the block buffer exceeds the threshold \( T \), the mechanism tries to create more space by writing the buffered blocks to disk and at the same time preserve sequentiality. The assembler analyzes the block maps and selects the region that has the largest number of buffered blocks and starts retrieving and writing blocks for that region. While this decision may free blocks from the data buffer it may also add new blocks as the mechanism retrieves the blocks missing in the region.

Note that a block maybe part of more than one VM image, consequently, the block is removed from the buffer only when it is written to all the images it is part of.

4. If step 3 succeeds in reducing the number of buffered blocks below \( T \), the mechanism will continue to retrieve and write sequentially to the current location in the VM image.

5. If step 3 fails to reduce the number of buffered blocks and the buffer becomes full, the assembler selects the region with highest number of blocks in the buffer and writes the blocks from the buffer to the region without bringing any new blocks. While this will not completely transfer the region it will effectively reduce the number of buffered blocks. The mechanism will process as many regions as needed until the number of buffered blocks is less that \( T \).

5. **Performance Analysis**

This section presents an analytical analysis of the expected improvements the proposed mechanism can bring. Table 1 presents the variables used in the proof.

| Variable | Description |
|----------|-------------|
| \( F \)  | The total number of VM images to be transferred. |
| \( S \)  | The size of a VM image (we assume all images have the same size). The total size of VM images to be migrated is \( F \cdot S \). |
| \( b \)  | The number of data blocks per region. |
| \( r \)  | The number of regions the data buffer can hold. i.e. the size of the buffer in regions. The number of blocks the buffer can hold is \( b \cdot r \). |
| \( c \)  | The number of blocks that always (in every region of every file) appear consecutive. E.g. \( c = 2 \) means in every region there are two blocks that appear in the same consecutive order in all the other files. |
| \( s(d) \) | The time taken to seek to a new location that is \( d \) bytes away on the disk. This time equals: disk head seek time + rotation latency. In our analysis we assume that the head seek time is constant. Further we note that rotation latency is directly dependent on the distance the head needs to rotate, \( d \). |

As the disk seeks overhead is the main contributor to the performance degradation we evaluate the number of seeks required to assemble a set of \( F \) images.

To make the analytical analysis tractable and provide an insight on the factors that affect performance, we assume the worst case scenario in which the similarity rate is 100% meaning all the VM images contain the same data blocks (but possibly in a different order), hence every block needs to be written to \( F \) images at the destination.
Our mechanism’s performance is highly impacted by the degree of similarity of regions across files. This is modeled by the parameter $c$ in our model.

**Number of seeks in the current approaches:**
As each data block is part of the $F$ images under transfer the naïve approach will suffer from $F$ disk seeks for each data block, resulting in total of $(bF)$ disk seeks per region regardless of the block pattern; i.e., regardless of the parameter $c$. Consequently the total seek overhead is: $(bF.s(d_a))$ as $d_a$ denotes the average distance traveled per seek using the current approaches.

**Number of seeks generated using the proposed mechanism:**
To simplify the analysis of the proposed approach we analyze a simplified version of our mechanism. This provides an upper bound for the number of seeks the original mechanism will require (and thus a lower bound for the performance of our mechanism).

**The Simplified Migration Algorithm:**
The simplified migration algorithm follows the following steps:

1. Select a region in the file and transfer the blocks in the region sequentially. As all the blocks in the region are part of other $F$ files all the blocks in the region need to be retained in the buffer.
2. When the data buffer is full (i.e. has buffered $r$ regions) the algorithm writes all the buffered blocks to the $(F-1)$ files they are part of, without bringing any new blocks.

**Analysis of the Simplified Migration Algorithm:**
- During the first step (point 1) in the simplified algorithm the algorithm will have one disk seek to start writing to a region and then write all the region’s blocks sequentially. This leads to one seek per region (or per $b$ blocks).
- After writing $r$ regions the buffer is full. The algorithm will write every block to the $(F-1)$ files it is part of. Since $c$ of the blocks of every region are consecutive in every file the algorithm will acquire $(F-1)$ seeks for the consecutive $c$ blocks and $(F-1)$ for every non-consecutive block. In total the algorithm will acquire $(F-1) + (b-c)(F-1) = (b-c+1)(F-1)$ seeks per region.

In total, the algorithm imposes $(1 + (b-c+1)(F-1)).s(d_a)$ seek time to transfer one region. $d_a$ denotes the average distance traveled per seek using the simplified algorithm.

Compared to the naïve approach, the relative performance of the simplified approach in transferring one region is:

$$\frac{(1+(b-c+1)(F-1)).s(d_a)}{bF.s(d_a)}$$

This analysis shows that the performance gain is impacted by: the size of the region, the level of region similarity, and the seek distance.

**Constant Seek Overhead Analysis**
To understand the impact of the region size $b$ and region similarity $c$ on the relative performance we assume that the seek overhead is equal between the two approaches; that is $d_a = d_c$ (we will analyze the effect of the seek distance in the following paragraphs). The relative performance becomes:

$$\frac{1+(b-c+1)(F-1)}{bF}$$

This formula shows that in the worst case (with $c = 1$, i.e. no two blocks appear in the same region more than once across files), for every seek in the naïve approach the simplified algorithm performs $(F-1)/F$ seeks. In other terms the simplified approach reduces the number of seeks by $1/F$. This pattern represents the worst scenario for the proposed mechanism as it is less likely to buffer more than one
data block in the same region. On the other end of the spectrum, in the best case \( c = b \), i.e. the blocks in a region in a file appear exactly in the same order across files, for every seek in the simplified algorithm the naïve approach performs \( b \) seeks; achieving \( b \) times performance improvement; this is mainly because the mechanism can efficiently use the buffer to achieve high sequentiality.

**Seeks-Distance Impact Analysis**

The seek overhead is directly dependent on the distance the disk needs to rotate. The following analyzes the seek distance of the two approaches:

*The average seek distance in the current approaches:*

In the current approaches every block needs to be written to every file. In the best case the current approaches may impose a seek distance of 0 (in case the current block is the last block in the current image and the first block in the next image) and a seek distance of \( 2.S \) in the worst case. The seek distance follows a triangular probability distribution with parameters \( a = 0 \), \( b = 2.S \) and \( c = S \) with a mean value of \( S \). Consequently, on average the current approaches impose an average seek overhead of \( s(S) \).

*The average seek distance in the simplified algorithm:*

The simplified approach writes the blocks (total of \( b.r \) blocks) in all regions in order (i.e. the blocks are sorted based on their offset). Consequently, assuming a uniform random distribution of blocks across files and offsets the mechanism imposes on average \( s(S.F/b.r) \) seek overhead.

This analysis shows that the proposed mechanism achieves \( F/b.r \) times shorter seek distance. We note that this is a significant improvement since \( r.b >> F \). The experimental evaluation (section 6) shows that the impact of this improvement on transfer rate is significant.

**Conclusion**

The proposed approach achieves significant performance gains compared to the naïve approach. This is because it does not only reduce the number of seeks required but it also significantly reduces the seek overhead. For instance for migrating 5 VM images with \( b \) set to a reasonable value of 100 4KB blocks and assuming equal seek overhead) the mechanism can achieve up to 100x, and at least 20%, better performance compared to the naïve approach.

Finally, given the nature of VM images, often a group of blocks appear always in the same region across files (i.e., there is high region similarity), since these blocks often belong to the same file in the VM image (same OS kernel, library, or application files). This leads us to believe that the performance gains are considerably higher than the minimum of \( 1/F \).

### 6. Experimental Evaluation

This section presents an evaluation of the impact the seek overhead has on the write operation throughput of large VM images and evaluates the effectiveness of the proposed approach in reducing the seek overhead.

We run our evaluation on two machines each with an Intel Xeon E5345 4-core, 2.33-GHz CPU, 4-GB RAM, and a 300GB 7200rpm SATA disks. Each experiment is repeated 10 times and we report average and standard deviation (as error bars).

#### 6.1. Seek Overhead Evaluation

The evaluation writes 10GB files to disk using different block sizes. The evaluation compares three write patterns: sequential, random, and a ‘regular’ write pattern. For the random pattern the block write order is selected randomly from the possible \( n! \) block’ orders – where \( n \) is the total number of blocks. Using probability analysis, the expected seek distance is \( 1/3 \) of the file size (in our evaluation, it is equal to disk distance for storing 3.3GB of data). The regular pattern divides the file into 10,000 segments and writes one block to each segment in a round robin fashion. The goal of this pattern is to measure the impact of seek distance on the disk write throughput. Like the random pattern, this pattern
will require a seek for every block write, unlike the random pattern the seek distance is 1/10,000 of the file size (in our evaluation is equal to disk distance for storing 1MB of data).

We note that random pattern represents the pattern generated by the current VM transfer approaches, while the regular pattern represents pattern generated by our mechanism in the worst case ($c = 1$ in our analysis). Finally, the sequential pattern represents the highest possible performance in our system and approximates the performance of our mechanism in the best case scenario.

We note that our evaluation evaluates the complete disk system including disk and OS caches.

Figure 2 shows the write throughput of the three patterns while varying the block size. The results lead to the following observations:

- Random writes with small block sizes (typical in VM deduplication) achieve embarrassingly low performance (110 KBps with 1KB blocks), up to three orders of magnitudes slower than the sequential write pattern.
- Regular write pattern always performs better than random writes (up to two orders of magnitude better in small blocks) due to the much shorter seek distance.

**Summary.** Our evaluation shows the low disk performance the current VM transfer mechanism achieves. Further, our evaluation shows that not only increasing disk access sequentiality improves performance, but also minimizes seek distance.

![Figure 2. Write throughput of the three patterns. Lines show average throughput of 10 runs, while error bars show standard deviation.](image)

6.2. **Mechanism Evaluation**

To evaluate our approach effectiveness in eliminating the disk random access problem we have built a VM migration tool that implements the approach we propose as well as the naïve migration approach (as a comparison baseline). The prototype uses a single node as a source and a single node as a destination. The destination buffer size was set to 234MB (60,000 4KB blocks).

The evaluation uses two workloads. The main difference between these workloads is in the similarity, at the operating system level. Similarity can vary from using the same operating system installation yet with different applications on top, to using different versions of the same operating system, to using completely different operating systems. Note that all images we use are sparse images; that is, it stores only the blocks of the image disk that are used, not the unused blocks. Consequently, images are often significantly smaller in size than the actual disk size they represent.

The workloads are:

- **Application.** This is a group of three VM images part of OpenCart eCommerce application. The application contains: an image running the OpenCart and an Apache server, one running a MySql
server, and an NFS backend storage node. The three images use the same OS distribution: Debian 6.0.4. The total size of all images is 4.4 GB.

- **Different OS.** This is a group of three VM images using different OS distribution. The images contain: Debian server, Fedora server and Ubuntu server. The total size of all images is 3.5GB. This group of VM images represents the worst case workload for deduplication based migration mechanisms as it contains lowest level of similarity.

To characterize the migration disk access pattern we instrumented the tool to report the offset of the write operation. Analyzing the write offsets enables measuring the ratio of sequential write operations.

Our evaluation shows that, when migrating the application images, the naïve approach achieves 15% sequential access, while the proposed approach achieves 99% sequential access. Similarity, for the different OS workload, the naïve approach achieves 9% sequential access, while the proposed approach still achieves 99%. This result highlights the effectiveness of the proposed approach in eliminating the disk bottleneck when migrating a group of VM images.

7. **Discussion**

This section focuses on a number of related questions:

1.) *Will this mechanism help if the storage systems at both ends are deduplicated?*

The proposed mechanism optimizes the storage and transfer of a group of VM images in all cases that require assembling the VM images into separate files. This is required not only when transferring a group of VM images from a deduplicated VM repository to a non-deduplicated one, but also when transferring group of VM images across deduplicated VM repositories that do not use the same deduplication technique and configuration, as the incompatibility in deduplication mechanism or configuration can only be addressed by assembling the VM images and deduplicating them again following the destination deduplication mechanism and configuration.

2.) *Is this mechanism tied to any particular VM transfer system?*

The proposed mechanism focuses on intelligent buffering and reordering of VM transferred data blocks. These techniques are not tied to any particular VM transfer system and can be used to optimize systems as simple as single node transfer service to a complex distributed migration system as VMFlock [2].

8. **References**

[1] D. Reimer, A. Thomas, G. Ammons, T. Mummert, et al. *Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl.* in ACM Virtual Execution Environments (VEE). 2008.

[2] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. *VMFlock: virtual machine co-migration for the cloud.* in International Symposium on High Performance Distributed Computing (HPDC). 2011.

[3] A. Liguori and E. V. Hensbergen. *Experiences with content addressable storage and virtual disks.* in Workshop on I/O Virtualization (WIOV). 2008.

[4] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, et al. *Optimizing the migration of virtual computers.* in Symposium on Operating Systems Design and Implementation (OSDI). 2002.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres. *A Low-bandwidth Network File System.* in Symposium on Operating Systems Principles (SOSP). 2001. Banff, Canada.

[6] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, et al. *JumboStore: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service.* in USENIX Conference on File and Storage Technologies, FAST. 2007.

[7] T. Hirofuchi, H. Ogawa, H. Nakada, S. Itoh, et al. *A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services on Clouds.* in International Symposium on Cluster Computing and the Grid (CCGrid). 2009.
[8] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioberg. *Live wide-area migration of virtual machines including local persistent state.* in *International conference on Virtual Execution Environments (VEE)*. 2007.