VMFlock: VM Co-Migration for the Cloud

Samer Al-Kiswany[1], Dinesh Subhraveti[2], Prasenjit Sarkar[2], Matei Ripeanu[1]

[1] University of British Columbia
Vancouver, BC, Canada
{samera, matei}@ece.ubc.ca

[2] IBM Almaden Research Center
San Jose, CA, USA
dineshs@us.ibm.com,
psarkar@almaden.ibm.com

ABSTRACT

This paper presents VMFlockMS, a migration service optimized for cross-datacenter transfer and instantiation of groups of virtual machine (VM) images that comprise an application-level solution (e.g., a three-tier web application). We dub these groups of related VM images *VMFlocks*. VMFlockMS employs two main techniques: first, data deduplication within the VMFlock to be migrated and between the VMFlock and the data already present at the destination datacenter, and, second, accelerated instantiation of the application at the target datacenter after transferring only a partial set of data blocks and prioritization of the remaining data based on previously observed access patterns originating from the running VMs. VMFlockMS is designed to be deployed as a set of virtual appliances which make efficient use of the available cloud resources to locally access and deduplicate the images and data in a distributed fashion with minimal requirements imposed on the cloud API to access the VM image repository.

VMFlockMS provides an incrementally scalable and high-performance migration service. Our evaluation shows that VMFlockMS can reduce the data volumes to be transferred over the network to as low as 3% of the original VMFlock size, enables the complete transfer of the VM images belonging to a VMFlock over transcontinental link up to 3.5x faster than alternative approaches, and enables booting these VM images with as little as 5% of the compressed VMFlock data available at the destination.

Categories and Subject Descriptors

C.2.4 (**Computer-Communication Networks**): Distributed Systems – *distributed applications, cloud computing.* E.4 (**Coding and Information Theory**): *data compaction and compression.*

General Terms

Design, Algorithms, Performance, Evaluation.

Keywords

Distributed deduplication, VM migration, Cloud computing.

1. INTRODUCTION

The ability to boot a virtual machine (VM) image on any available physical node in a datacenter, or even across datacenters, is a key enabler for the many benefits promised by cloud computing such as resource consolidation, elastic scaling, and computation migration.

This project starts from the observation that today, while VM images are not tied to any specific physical node, they are still locked within the boundaries of a datacenter.

Flexible deployment of VMs across datacenters enables essential load-management services that support a number of important scenarios. For instance, if a datacenter is overloaded, requests to launch new VM instances can be served by a different datacenter of the same provider. This functionality enables reducing datacenter provisioning costs as peak load can be shared across datacenters. Similarly, to accommodate scheduled maintenance operations, load can be redirected to new VMs instantiated at another datacenter. Finally, flexible VM deployment across datacenters enables arbitration for energy costs, that is, providers can deploy VMs at the datacenter that benefits from the lowest energy price for a certain time interval, as suggested by Qureshi et al. [1] who evaluate the benefits of this technique.

Additionally, flexible deployment of VMs across datacenters is a key enabler for federating clouds. Tools that support users to transfer and instantiate their VMs on any cloud that is part of a federation will eliminate one of the important adoption barriers of the cloud technology: the users' concern that they will be 'locked in' by a specific cloud provider. Section 2.3 further motivates the need to support cross-datacenter VM deployment.

Migration of VMs across clouds and datacenters is challenging for four main reasons: First, migrating VMs across datacenters involves transferring large volumes of data. On the one side, VM images are large (typically 1-30GB in size); on the other side, applications deployed on the cloud often involve multiple VMs with different images [2] (e.g., a three-tier web application [3], a business analytics solution [4], or a virtual cluster [5]). Consequently, application migration often involves transferring multiple large VM images. Second, these large data transfers are often done over wide area networks with limited bandwidth. A naive application migration approach may easily clog the network links between the datacenters, leading to unacceptable performance degradation. Third, a VM image transfer service has to operate within the limits imposed by the existing APIs of clouds' VM image repositories (e.g., IBM's Mirage [6]). Not only these APIs are still not standardized but, more importantly, they expose limited information on the data stored in the VM repository and are not designed to support efficient cross-datacenter VM image transfer. At the same time, a migration mechanism that requires changes to a cloud's VM image repository is not acceptable by the cloud providers for strategic as well as for technical reasons. Finally, the problem is more complex when the transferred VM images are to be instantiated (i.e., booted) at the destination site, in which case the image transfer needs to be completed in a reasonable time.

Despite the aforementioned challenges, three characteristics of the VM images can be exploited to build an efficient migration mechanism. First, the volumes of data transferred over the network can be reduced as the VM images often have high similarity. This is especially true for the VM images belonging to the same application (which we dub virtual machine 'flocks', or simply *VMFlocks*). The reason is that the VM images belonging to the same VMFlock are often based on the same OS distribution, yet with different installed applications, leading to similarity ratios across images as high as 96% [7]. Second, the transferred

data volumes can be further reduced as the VM images to be transferred often have high similarity with some of the VM images already existing at the destination datacenter. Finally, the VM instantiation and application startup time can be reduced by intelligent prioritization of the data blocks that are transferred. The reason is that while the VM images may include millions of data blocks, only a fraction of these blocks are needed to boot the VM and start the application.

This project proposes *VMFlockMS,* a migration system optimized for virtual machine flocks. VMFlockMS exploits the aforementioned characteristics to reduce the migration time and to reduce the data volume transferred across the network. First, VMFlockMS, exploits the similarity among the VM images within the same VMFlock and the similarity among VM images across datacenters to reduce the amount of data transferred. Second, VMFlockMS accelerates application startup by prioritizing the data transfers and by booting the VMs at the destination datacenter immediately after the blocks needed for VM boot and application startup have been transferred.

Notably, the main component of the migration service (i.e., the VM image transfer service) harnesses these opportunities without requiring access to the internal information of the cloud's VM image repository. The VM image transfer service can be deployed as a user-level 'appliance' at any datacenter that provides access to VM images through a basic interface able to read/write VM images and does not require any infrastructural changes at the datacenter. This characteristic reduces the adoption barrier for the migration service, and makes it a good building block to support aggregating resources across cloud providers and cloud federation. Additionally, this design facilitates separation of concerns between the VM repository and the VM transfer service. We note that, to accelerate application startup (by booting VMs before the entire image transfer completes) however, VMFlockMS requires additional support from the cloud infrastructure, as detailed in §3.

The contribution of this work is threefold:

- First, this paper presents a distributed high-throughput data deduplication algorithm (§3.3). The algorithm uses multiple nodes at the source and destination datacenter to identify similarities among local and remote VM images. The algorithm is completely distributed and incrementally scalable. Additionally, the algorithm evenly balances the deduplication effort among the participating nodes, naturally supports the use of multiple streams to accelerate the data transfers, and is optimized to minimize the memory footprint at each node.
- Second, this paper presents the design (§3.2) and implementation of a migration system for flocks of virtual machines (VMFlockMS), which, to the best of our knowledge, is the first to address the challenges of migrating applications that are deployed over multiple VMs. Not only VMFlockMS uses the specialized data deduplication algorithm mentioned above to reduce the volume of data transferred over the network (§3.3), but it exploits the fixed order in which data is used at boot and application startup to accelerate application migration. To this end, VMFlockMS includes additional components: VMProfiler (§3.4) which records the order in which VM image data is used at startup, a prioritization algorithm for data transfers, as well as VMLaunchPad (§3.5) a solution to start booting at the destination site when key data is available, yet before the VM image transfer is complete.
- Third, this paper presents a detailed evaluation of the VMFlockMS prototype (§4). The evaluation compares the performance of the VMFlockMS to current alternatives under migration scenarios for real applications. We evaluate the reduction in data volumes transferred over the network, the compression overheads, the feasibility of and the potential gains brought by early VM image boot, and we perform end-to-end VMFlock migration experiments between two IBM Research cloud datacenters. The performance of the proposed algorithm varies with the offered workload. However, for a setup that is arguably close to the scenarios that will be seen in practice, the proposed mechanism achieves up to 10x better compression rate, compresses the data up to 2x faster, and enables migrating a complete application up to 3.5x faster than alternative techniques.

We argue that VM image transfer and instantiation are sufficient to enable flexible deployment of VMs across datacenters as migrating *still* VM images (as opposed to *live* migration of running VMs [8]) is sufficient for supporting most of the aforementioned use cases. Further, the techniques developed in this work can be applied to support live migration of running VM images; either *directly*, through taking a snapshot of the running VM and including the snapshot state within the VM image as Qemu/KVM does (in such a case, our system can be tuned to prioritize the snapshot data for a quick VM resume),or by *integrating* the proposed distributed migration mechanism in a live migration service [9, 10]. For the rest of this paper we will use VM migration to mean migration of VM still images across datacenters.

Finally, while we present and evaluate VMFlockMS in the context of VM migration, the ideas we put forward and even our implementation can be used to optimize other computational pipelines that combine data transfers and computation by exploiting the same characteristics our system exploits. On the one side, data transfers volumes can be reduced not only through compression of individual items or batches of items, but more importantly, by detecting and harnessing the partial similarities between the data to be transferred and the data already at the destination. On the other side, intelligent transfer protocols that harness data usage profiles to prioritize data transfers will enable overlapping data transfer and data consumption thus accelerating long computational pipelines. Section 5 discusses existing and potential applications in others areas than VM migration (e.g., a scientific computing scenario supported by a GridFTP service).

2. BACKGROUND AND RELATED WORK

Our work draws from two research fields: virtual machine migration (§2.1) and data deduplication (§2.2). This section briefly presents the ways VMFlockMS builds on this past work and presents a number of cloud scenarios that are enabled by efficient cross-datacenter VM image transfer and fast VM instantiation.

2.1 Virtual Machine Migration

A number of projects have proposed mechanisms for migrating individual VM images over local- and wide-area networks. To the best of our knowledge, no previous work addressed the issue of optimizing migration for *VMFlocks,* multiple VMs that are

coupled together in an application.

Clark et al. [8] build a live VM migration tool capable of migrating live VMs between LAN-connected nodes. The mechanism assumes that the source and destination nodes share a network-accessed storage system that maintains the VM's persistent image, and only proposes a solution for migrating the in-memory state of the live VM. Similarly, Lagar-Cavilla et al. [11] propose a mechanism for cloning live VMs in LAN-connected platforms. The proposed mechanism assumes as well a shared copy-on-write storage system storing the VM images.

Sapuntzakis et al. [12] present a system for virtual machine live migration that includes disk images. The approach proposes using copy-on-write images to derive users' custom images from a root image. The approach reduces the amount of data transferred using two optimizations: exploiting similarities between the transferred image and the root image and the possible similarities with images already present at the destinations site. Hirofuchi et al. [9] present a mechanism that complements the live migration of VM memory with transferring the VM disk image over long-haul networks. After completing the migration of a live VM state stored in memory and starting the VM at the destination site, the approach proposed by Hirofuchi et al. transfers the VM disk image to the destination, giving priority to the blocks accessed by the VM after migration. In the same vein, Bradford et al. [10] propose transferring the VM disk image at the same time as transferring the live VM's in-memory state. The VM keeps running at the source machine while the transfer of the VM memory and disk image takes place. Similar to past approaches (e.g., Clark et al. [8]) changes to migrated disk blocks are forwarded to the destination as deltas. When most of the memory and disk are migrated, the VM machine is stopped at the source and the rest of its memory and disk are migrated to the destination. Finally, the VM is started on the destination machine.

The aforementioned projects either do not address the migration of VMs' disk images, or focus on migrating a single VM image across wide area networks. As our evaluation will demonstrate, migrating multiple VM images using these approaches results in data transfer redundancies and high overheads.

VMFlockMS complements these approaches: it exploits the similarity among the VM images within a VMFlock and between the VMFlock and the destination VM image repository to significantly reduce the amount of data transferred. To detect and take advantage of these similarities, VMFlockMS proposes a distributed deduplication algorithm so that multiple end-nodes can be allocated to match the size of the VMFlock to transfer. Finally, VMFlockMS is designed with ease of deployment as a main focus. Unlike the aforementioned projects which require infrastructure changes, the main image transfer component of VMFlockMS can be deployed as a virtual appliance on multiple VM instances and it does not require changes to the VM image repository provided by the cloud infrastructure.

2.2 Data Deduplication

A number of storage systems [13, 14, 15] use data deduplication to reduce their storage footprint or to reduce the volume of data transferred across the network. These storage systems often adopt a 'content addressable storage' (CAS) approach which names the data-blocks based on their content. In this context, hashing is used as a naming technique: data-block names are simply the hash value of the block's data. To store a new file, the system divides the file into blocks, hashes them, and compares the blocks' hashes with the hashes of the already stored blocks. This way the system identifies new data blocks (which are stored) and uses references to already stored blocks thus reducing the storage costs.

Defining the data block boundaries: Blocks of fixed size vs. detecting block boundaries based on content. Once a content addressable storage approach is selected, a second issue to decide on is the technique to define block boundaries. The fixed block size approach divides each file into equal size blocks and names blocks by their hash value. Venti [16] and Foundation [14] are two storage systems focused on archival workloads that adopt this approach for deduplication. Alternatively, block boundaries can be defined based on the data itself, by using Rabin fingerprints over a sliding window [17]. While this approach results in higher compression rates (as it is stable against file insertions/deletions), it also incurs higher computational overheads. LBFS [13], a storage system optimized for backup operations over low bandwidth links, and JumboStore [18],, a storage system optimized for multimedia content, adopt this approach. Stdchk [15] project explores quantitatively the tradeoffs between these two approaches.

VMFlockMS directly draws from these data-deduplication techniques to reduce the volume of data transferred across the network. We have implemented and experimented with both deduplication approaches – fixed blocks and detecting block boundaries based on content. While the later approach offered a slightly better compression rate, it adds additional computation overheads and implies higher code complexity. Thus for this work we use fixed size blocks.

Improving data deduplication throughput. Two recent projects propose parallel solutions to improve deduplication throughput. The DataDomain system [19] and Debar [20] are custom-built hardware/software stacks optimized for high throughput deduplication. Both systems use specialized or dedicated hardware resources, and are optimized for long-term retention of large amounts of data. Consequently, most of their optimizations relate to avoiding accessing the metadata information which is often stored on the disk.

VMFlockMS differs from the aforementioned approaches in two ways. First, unlike DataDomain and Debar, this project, targets a cloud deployment rather than a deployment over custom resources. Consequently, the VMFlockMS system is designed to be incrementally scalable such that it will benefit form any number of nodes allocated and to efficiently operate in spite of the scarcer resources available per allocated node. Second, VMFlockMS targets a different workload, namely migrating VM images instead of processing data for long-term archival. The main implication is that while archival services are generally designed for offline operation, VM migration is often in the critical path thus requiring higher performance.

2.3 Motivating Scenarios

This section presents a set of services that are enabled by providing efficient, cross-datacenter VM image transfer and fast VM instantiation. The goal is twofold: to further motivate and to inform the system requirements for the VMFlockMS migration service (discussed in §3.1).

We assume that users can personalize VM images and store them in the VM image repository at one datacenter. Users can launch a

VM at the same datacenter where the VM image repository is hosted, yet they need an external service to transfer images across repositories. We also assume that VM repositories, regardless of whether they belong to the same cloud provider or not, do not share data.

Cross Datacenter Load Management. An efficient VM image transfer and instantiation service enables flexible load management across datacenters. Migrating VMs between datacenters to accommodate scheduled maintenance operations, or redirecting new VM instantiation requests from a highly loaded datacenter to another are only two of the possible use cases. The main requirement is the ability to reduce image transfer overheads and to bootstrap the VM images as quickly as possible at the destination site (ideally, within a few minutes).

Our migration scheme complements live migration by expanding the scope of migration and by addressing the state of persistent storage. While live VM migration is a useful primitive for fine grain load balancing within a data center, it is not suitable for migrating VFlocks across the cloud. In particular, the typically employed technique of iteratively transferring dirty memory blocks may not converge in bounded time across a high latency wide area link. We expect that VMFlockMS can be used in conjunction with live migration, where live migration is used for a more dynamic control over load placement and VMFlockMS is used by a scheduler at a higher layer.

Replication. Similarly, an efficient VM migration service can be used as the main building block for multi-site VM image and virtual disk replication. For instance, virtual machine record-replay techniques [21, 22] often require an initially synchronized persistent storage state, from which they make identical incremental updates to keep the replicas synchronized. While, for this case, bootstrapping the VMFlock at the destination datacenter is not a requirement, efficient usage of the limited network bandwidth is still necessary.

Cloud Federation. Efficient VM migration is a basic service required for federation clouds regardless of whether this is achieved as a third-party integration service or through pair-wise inter-cloud agreements. Cloud federation is envisioned as a public, open, inter-cloud infrastructure that enables the users to use the resource of multiple clouds. This is envisioned as 'the next step in cloud computing technology' [23, 24] and is the driver for active standardization efforts ongoing nowadays [25, 26, 27].

3. SYSTEM DESIGN

This section discusses the VMFlockMS design requirements (§3.1), briefly presents the overall system architecture (§3.2), and presents in detail, the design of each of the system's main components: the VM image transfer service (VMFlockMA in §3.3), the service to profile VM image usage to infer the data access patterns at VM boot and application startup (VMProfiler in §3.4), and the service to launch VM images in spite of partially completed data transfers (VMLaunchPad in §3.5).

3.1 Design Requirements

This section presents the requirements of a VM migration system:

- *Data compression.* The migration system should reduce the volume of data to be transferred across the network to reduce the migration time and the load on network connections.
- *Low overheads.* To be attractive, the migration system should lead to low overheads. Three types of overheads are crucial: first, computational overheads resulting from the data deduplication technique to compress/decompress data; second, storage system overheads resulting from additional accesses to the storage system to fingerprint VM images that will not be transferred, to store and access VM boot profiles, and to store and access VM images' metadata; and, finally, the memory footprint.
- *Boot time.* The migration system should be able to boot the migrated VM images as soon as possible. This is especially important for cross-datacenter load management.
- *Scalability over multiple axes*. On the one side, the migration tool should scale with the offered load. That is, the migration tool should scale to support migration of tens to hundreds of virtual machines, each potentially hundreds of GBs in size. On the other side, the migration tool should make efficient use of additional compute resources, that is, make efficient use of possibly multiple nodes at the source and destination sites allocated to the VMFlockMS system.
- *Easy to deploy*. The migration tool should be easy to deploy by cloud providers as well as by cloud end-users. To this end, a popular approach for encapsulating complex services [28] is designing them as 'virtual appliances'. In this approach, a virtual machine is setup with necessary software stack, and configured to provide the specific service (e.g., a database appliance, or an Apache webserver appliance). Consequently, the migration service should not assume any custom hardware and should be able to run using the sometimes modest resources allocated by the cloud to individual VM machine instances (for example, the system should be able to operate with a memory footprint on each node limited by the capabilities of the target cloud infrastructure).
- *Easy to adopt*. The migration tool should be easy to adopt by current cloud infrastructures. Consequently, the migration tool can not assume access to cloud infrastructure internal state (e.g., it can not assume access to block-level information of the VM image repository). An advantage of delivering the migration service as an appliance is that it allows it to be transparent to the details of the cloud infrastructure. Given that cloud providers limit access to their underlying system and resources, an appliance provides an available means to package our "logic" and directly deploy it where the data is located.

3.2 System Architecture

Figure 1 shows the VMFlockMS system architecture. The system is composed of three main components: the VM Profiler (VMProfiler), the VM Migration Appliance (VMFlockMA) present at the source and destination nodes, and the VM Launch Pad (VMLaunchPad).

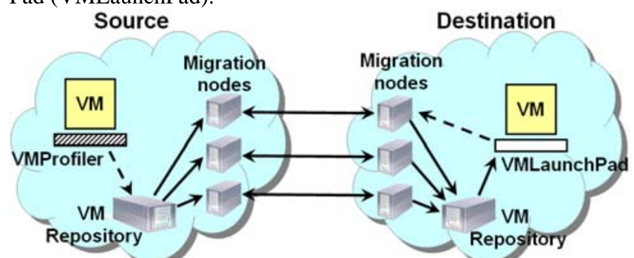

Figure 1. VMFlock migration system architecture.

The VMProfiler (§3.4) profiles a VM image to identify which data blocks are accessed at boot time (and possibly at application startup time). This profiling information is collected at VM boot and application startup time and stored as part of the VM metadata in the repository. We assume that this information is available before the migration of the virtual machine.

The VMFlockMA migration appliance (§3.3) implements the distributed data deduplication and transfer algorithm. The nodes collaborate to divide the VMFlock's VM images into data blocks, deduplicate the blocks (i.e., identify the similarities among images), migrate unique blocks from the source datacenter to the destination while prioritizing the blocks the image's profile indicates as needed at startup time, and, finally, reconstruct full images at the destination.

Finally, the VMLaunchPad (§3.5) enables launching a VM image at the destination even if the transfer of a VM image is not complete (i.e., not all the image's data blocks are present at the destination). The VMLaunchPad intercepts the I/O requests of the running VM image and, if the requested data is not yet migrated from the source, the VMLaunchPad will ask the migration nodes to migrate the needed data block at a higher priority. Once the needed data block is migrated, the VMLaunchPad services the I/O request and the running VM resumes its execution.

Note that the three systems above are independent. The migration appliance can transfer VMFlocks and they can be booted at the destination site when transfer finishes even without having a VMLaunchPad available. However, having the profile information enables prioritizing data transfers such that VMFlocks can be booted earlier if a VMLaunchPad is available at the destination site. We note that while the main component of our system, VMFlockMA, the image compression and transfer service, can be deployed as a virtual appliance, the two support services, the VM Profiler and the VM Launch Pad, need support from the cloud infrastructure.

The following three subsections present the detailed design of each of these three components.

3.3 VMFlockMA - A Migration Appliance for Flocks of Virtual Machines

VMFlockMA reduces the amount of data transferred between datacenters by exploiting the data similarity that exists at multiple levels: first, within a single VM image; second, between the VM images that belong to the same VMFlock; and, finally, between the VM images of the migrated VMFlock and the images that already exist at the destination site. While traditional data compression techniques [29] might be able to harness the first two levels of similarity mentioned above (i.e., by compressing single images or bundles of images) these techniques can not exploit similarities across datacenters. Additionally, these compression techniques are unable to exploit the available parallelism in the workload and the wealth of cloud resources potentially available to increase the compression and data transfer performance.

Designing an efficient deduplication system that can be deployed as a virtual appliance for VM migration is a challenging task for two main reasons: First, *data size*, the amount of data that need to be deduplicated is large (tens to hundreds of GBs). Second, depending on the data compression technique used and on its configuration, *metadata size*, the size of the deduplication metadata (block hash value, position in a file) can be large (GBs of metadata). To tackle these challenges, we harness the elasticity of the cloud and aim for a migration appliance that is able to control the load allocated to the individual node by integrating multiple resources, proportional to the size of the VMFlock to be transferred.

In summary, VMFlockMA adopts a distributed approach to deduplication and employs the compute power, memory space, and I/O bandwidth of multiple virtual nodes allocated to the appliance to provide an efficient migration service. Additionally, VMFlockMA operates parallel data transfer streams, and, if the necessary information is available, it prioritizes the data transfers to make the blocks necessary for VM boot and application startup available early at the destination site.

VMFlockMA operates as follows:

Pre-processing at the source datacenter: The VMFlockMA nodes at the source datacenter (source nodes for short) read the VMFlock images from the VM repository, chunk the images into fixed size blocks, and hash the blocks.

To identify the similarities across the blocks processed at different source nodes, these nodes need to exchange blocks' metadata. This step is complicated by the fact that, to fulfill scalability requirements, none of the source nodes can be assumed to have enough memory space to hold all the metadata for all the migrated blocks. Thus, to exchange the blocks' metadata while limiting the memory footprint of each individual node, the source nodes build a distributed metadata index similar to a one-level distributed hash table (or consistent hashing) [30]: depending on its ID, each node is responsible for holding metadata for the blocks with hashes in a specified range. The source nodes send out the metadata for the blocks not part of their hash range. This data structure is then used to identify the data blocks that are identical across the VM images that belong to the same VMFlock.

Pre-processing at the destination datacenter: VMFlockMA at the destination site selects a set of VM images from images found in the destination VM repository. The selection is made using the metadata of the images to be transferred (e.g., operating system version, list of applications installed) to find images with potentially higher similarity to the migrating VMFlock. VMFlockMA then chunks the selected images and hashes each data block and prepare a similar distributed metadata index at the source datacenter.

Data transfer: The block metadata obtained is used to identify the blocks at the source that are already present at the destination. Each source node collaborates with the destination node responsible for the same hash-range to transfer these data blocks, and each node pair attempts to further reduce the transfer volume transferring only the blocks not already present at the destination.

The block transfer is prioritized based on the information found in the profile, such that the blocks needed at startup time are sent first.

Post-processing at the destination datacenter: The destination nodes reassemble images: once a block is retrieved from the destination, or from a locally stored image, the block is written to all new VM images it is part of.

The rest of this section presents in more detail the algorithm at source (§3.3.1) and destination nodes (§3.3.2), and highlights the main advantages of the proposed approach (§3.3.3).

3.3.1 Source Node Algorithm

The source nodes collaborate on deduplicating the images belonging to the VMFlock to be transferred. All the source nodes follow the following steps to migrate a set of virtual machines:

- Each source node reads, and chunks, one or a few VM images. Each block is hashed using SHA1 hashing.
- The source nodes participate in building a distributed data structure to hold blocks' metadata (i.e., block's hash value, and a list of locations --image name, offset-- where the block appears). Depending on their node IDs, each node is responsible for holding the metadata for blocks that hash in a specific range.
- After all source nodes finish their chunking and hashing step, the source nodes exchange the blocks metadata according to a solution similar to that used for consistent hashing [30]..At the end of this step each node will only hold the metadata for blocks in its hash space.
- Each source node is paired with a destination node to transfer the blocks for which it stores metadata.
- Each source node sends the blocks metadata it has to the pair node at the destination, identifies the blocks that are not already present at the destination, and transfers them.

3.3.2 Destination Node Algorithm

VMFlockMA uses an equal number of source and destination nodes. The destination nodes perform two tasks:

Building the local block metadata repository. The destination nodes locate a set of local images which, based on available image metadata (e.g., operating system version) are likely to be similar with those belonging to the migrating VMFlock. The destination nodes will then chunk and hash these local images and build a distributed data structure to store block metadata using the same algorithm as the source nodes and described in the previous section. This step runs concurrently with the deduplication step performed by the source nodes. This metadata will be used to identify which of the blocks to be migrated already exist at the destination.

Block Migration. Each destination node works with the source node responsible for the same range of hash values to transfer of the blocks it is responsible for. The destination node runs the following steps:

- The destination node receives the blocks' metadata from the corresponding source node.
- For each block, the destination node checks if the block is present in the local images. If it is not, then the block will be requested from the source node (Transfer from the source is batched for higher throughput; additionally each block is further compressed using the *zlib* [31] compression library).
- After receiving a block (or the information that a block already exists at destination), the data is written to all images the block is part of. This information is found in the block's metadata.

3.3.3 Algorithm Analysis

The VMFlock migration algorithm has the following set of important characteristics:

- *Decentralized*. The algorithm does not have any central component that can from a performance bottleneck.
- *Parallel*. The algorithm is highly parallel: the source and destination nodes chunk and hash VM images in parallel, further the data is transferred in parallel streams between the source and destination nodes.
- *Load Balanced*. The algorithm uses a hash function to assign blocks to migration nodes. Past work [30] has explored the load balancing properties of this solution.
- *Incrementally Scalable*. The algorithm can integrate any number of nodes and handle workloads of migrating tens to hundreds VM images using tens of source and destination nodes. This is mainly due to its load balanced and distributed nature.
- *Low Memory Footprint*. Each migration node is responsible of handling the metadata for, at most, one or a few VM images. The metadata generated for a single VM image is well below the memory space available to a virtual appliance node. Further, the block metadata is evenly distributed among the migration nodes.
- *Easy to Adopt*. The algorithm is able to exploit the similarity across VM images within the same VMFlock and across datacenters without requiring access to the VM repository's internal data block information. VMFlockMA only requires some from of access to read/write VM images.

We note that one possible drawback of this algorithm is that, at least conceptually, it performs two passes when reading the data at the source site: in the first pass all data is sequentially read from the disk to hash the blocks, and in the second pass blocks are read from the disk and sent them across the network. However, if there is high similarity between source and destination (and our experiments demonstrate that, for the scenarios we target, similarity can be as high as 96%) then only for a small portion of the data, the part that is dissimilar between source and destination, duplicated reads are performed. A second possible drawback is the lack of failure tolerance: if one source fails the entire system needs to be restarted. However, since the system is meant to be run as an appliance for relatively short time intervals we do not consider this a major issue.

3.4 The VM Profiler

The main objective of the VMProfiler and VMLaunchPad (detailed in §3.5) is to enable faster booting of the migrated VM images. It is important to note that due to the non-sequential access pattern at VM boot time, traditional read-ahead/pre-fetching techniques fail to reduce boot time. Hence a profile that captures the boot-time access pattern in detail is required.

The VMProfiler extracts this profile: it identifies the VM image regions necessary to boot the VM image and to launch applications in order to prioritize the data transfers. The profiles are stored as part of the VM metadata.

Note that the blocks within the VM image may be rearranged either by the VM image repository due to an offline maintenance operation or by the guest file system itself as a part of a defragmentation. As a result, some of the block offsets captured in the profile may point to incorrect data. To guarantee correctness, when data is transferred, it is checked against a checksum for the

region. The checksum is matched against the checksum of the region computed at the time of capturing the profile, and, if discrepancies are detected, the profile is marked as invalid and deleted. Each time a profile is used, it is updated -- invalid entries are pruned and new entries are added based on new accesses.

We have implemented the VMProfiler as a user-level file system based on FUSE [32] that profiles the VM disk access pattern at startup time. The VMProfiler intercepts and records the running VM access pattern and builds the image profile. To profile a VM image, the profiling script mounts the VMProfiler, starts the VM image, and stops the VM image after a configurable timeout.

3.5 The VM Launch Pad

VM images can be booted at the destination datacenter before the full VM image transfer is complete, only the data blocks indicated by the profile as needed at boot time and application startup are required. The VMLaunchPad fulfils exactly this function. This enables parallelizing data transfer and application startup, thus leading to lower *apparent* VMFlock migration time.

To evaluate the feasibility and the potential benefits of such a mechanism, we have implemented VMLaunchPad as a user-level file system based on FUSE [32].

We note that the boot and application startup processes might be slightly different at different datacenters. Thus, even when all blocks indicated by the profile have been fetched, the VM might access blocks that have not been migrated yet. This would cause the boot process to fail if no further measures are taken. To handle this case VMLaunchPad maintains a *file block-map* indicating which data blocks are available locally for the given image. The VMLaunchPad intercepts the VM disk access operations and checks if the VM is trying to access blocks that have not been fetched yet. If this is the case, the VMLaunchPad will inform the VMFlockMA to prioritize the transfer of these blocks. At the same the VMLaunchPad will block the I/O operation until the needed data is available locally.

We note that, while blocking the I/O request of a running machine for a long time may cause the VM device driver to timeout and cause I/O errors, we have not experienced such error in our deployments even when migrating VMFlocks over a network with more than 40ms latency.

File block-map construction. VMLaunchPad requires access to block-level metadata for a VM image. More specifically, since data accesses are offset based, VMLaunchPad needs to be able to identify the block that is accessed at a certain offset. However, VMFlockMA transfers data in a different format: The metadata managed by VMFlockMA is block oriented, that is, for each block the metadata contains a list of images and locations where the block appears. The VMFlockMA avoids storing a blockmap per image to reduce the memory footprint of the appliance.

In the current implementation, to construct an image's block-map, the VMLaunchPad contacts all the VMFlockMA nodes at the destination asking for all block metadata related to a particular VM image. Armed with this information, the VMLaunchPad can reconstruct the image's blockmap.

3.6 Implementation Details

VMFlockMS is implemented in 19,000 lines of C++ code (12,000 lines for VMFlockMA, and about 6,000 for the VMProfile, and VMLaunchPad). We have used the tool internally to migrate VM images among the datacenters hosting IBM's internal research cloud and the University of British Columbia in Vancouver BC, Canada for the last two months.

The implementation uses FUSE [32] kernel module for developing the VMProfiler and VMLaunchPad. Zlib compression library [31] for compressing/uncompressing the data, PolarSSL [33] for the SHA1 implementation.

4. EVALUATION

We evaluate the VMFlockMS performance and compare it with the performance of currently proposed mechanisms for VM migration. The rest of this section presents the experimental setup used (§4.1), an evaluation of VMFlockMA ability to reduce the data volumes transferred over the network and VMFlockMA overheads (§4.2), an evaluation of the gains resulted from early booting of VM images (i.e., before the whole image transfer completes §4.3), and an evaluation of the VMFlockMS end-to-end data transfer performance (§4.4)

4.1 Experimental Setup

This section presents the workload (i.e., groups of VM images) used for experimentation (§4.1.1), the experimental platform (§4.1.2), and the set of alternative approaches we compare VMFlockMS performance against (§4.1.3). All results present the average and standard deviation of 5 runs of each experiment.

4.1.1 VM Images

The evaluation uses three workloads representative of popular cloud applications. The main difference between these workloads is in the similarity, at the operating system level, among the VM images belonging to the same VMFlock. 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 based on the QCOW2 (QEMU copy-on-write) VM image format [34] a compact representation of VM images. QCOW2 is optimized for sparse images; that is, it stores only the blocks of the image disk that are used, not the unused blocks. Consequently, QCOW2 images are often significantly smaller in size than the actually disk size they represent. All VM images in the workloads we use are about 30GB in RAW format and QCOW2 reduces them roughly ten fold.

The workloads are:

- *Application* (labeled as *app* in all plots). This is a group of three VM images part of Spree eCommerce application [35] The application contains: an image running the Spree 0.9.4 application itself, one running a MySql server, and an NFS backend storage node. The three images use the same OS distribution: Fedora 13. The total size of all images is 7.9 GB. *This is the closest to the target workload we estimate for VMFlockMS* and is used for all experiments where we choose to use a single workload to reduce the exploration space.
- *Same OS* (*same-os*). This is a group of four VM images using the same base operating system distribution but that are not part of the same application. The images are based on Fedora 13 distribution with the different installation options offered by Fedora: 'desktop', 'development', 'plain', and 'server'. The total size of all images is 10.6GB. It is worth noting that, while

these images are based on the same OS distribution, there can be significant differences among them, since each installation option is optimized for a different use.

- Different OS (*diff-os*). This is a group of four VM images using different OS distribution. The images contain: Fedora 13 desktop, OpenSUSE desktop, Ubuntu Desktop, Ubuntu server. The total size of all images is 10.6GB. This group of VM images represents the worst case workload for deduplication based migration mechanisms.

4.1.2 Evaluation Testbed

We evaluate the VMFlockMS performance by transferring a set of VM images between IBM research cloud datacenters located at IBM Almaden Research Center, CA and IBM T. J. Watson Research Center, NY. The network latency between the two sites is 41ms and the throughput reported by iperf [36] is 16Mbps (single TCP stream).

At each datacenter we have three nodes available. At IBM Almaden each node has four dual-core Intel Xeon @ 2GHz processors with 8GB memory and a SAS disk. At IBM T. J. Watson each node has eight quad-core Intel Xeon @ 3.1GHz, with 8GB memory and a SAS disk.

One node at the source and destination is used as the VM repository which is accessed through an NFS server.

4.1.3 Alternative Migration Tools

We compare the VMFlockMS performance with the performance of the migration alternatives available nowadays. These are:

- *GZip-All*. This approach puts all the images in a single 'tar-ball', then compresses it using GZip. The compressed archive is migrated to the destination and uncompressed.

  This approach does not require any serious implementation effort, but it has two major drawbacks. First, it is difficult to exploit the compute power of multiple cores on the same node or to use multiple nodes. Second, this approach does not exploit the similarity that may exist between the migrated VMFlock's images and the images that already exist at the destination.

- *Gzip-Separate*. This approach improves on the previous *Gzip-All* approach by distributing the workload over a number of nodes. Each node will compress a single image and migrate it to a peer node at the destination. The peer node at the destination will uncompress the image.

  While this approach parallelizes the compression and the transfer steps, it does not exploit the similarity that may exist between a VM image to be migrated and other VM images in the same VMFlock or images already existent at the destination.

- *Dedup-Separate*. This approach re-implements the approach proposed by Hirofuchi et al. [9, 37] for VM migration, that is, each VM image is deduplicated and transferred separately. This approach detects similarities within the VM image and between images across datacenters (yet it can not detect the similarities within the same VMFlock). For fairness of comparison, we extend the implementation of this approach with compressing all blocks sent over the network.

4.2 Compression Rate Evaluation

This section has two goals: first, to estimate compression gains achievable through each of the three types of data similarity VMFlockMS exploits (intra-image, across images in the same VMFlock, and across datacenters), and to compare VMFlockMS performance and overheads with those of the alternative compression techniques described in §4.1.3.

Figure 2 presents the achieved compression rate when the destination does not have any files, while Figure 3 presents the compression rate when the destination has a single VM image: a 'Fedora-desktop' image.

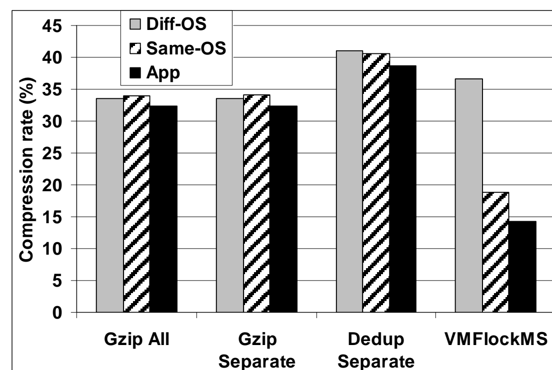

Figure 2. Compression rate evaluation when the destination site does not store any VM images. The compression rate is calculated as the ratio of the volume of data transferred over the network to the original data size. (The lower the ratio the better). Gzip uses the default compression level which is biased towards better compression rather than speed. The deduplication algorithm uses 1KB blocks.

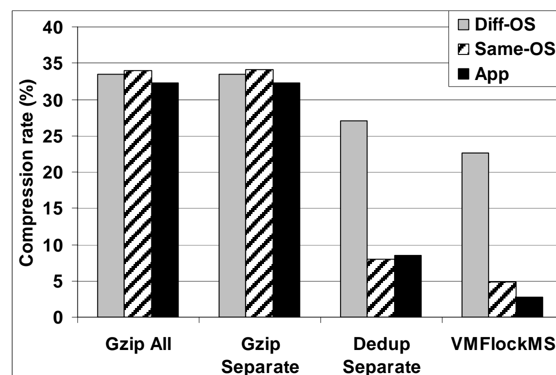

Figure 3. Compression rate evaluation when the destination site has one image similar to one image in the migrating VMFlock.

The compression rate is calculated as the ratio of the volume of data transferred over the network (including metadata-related overheads) to the original data size. Consequently, the lower this ratio, the better the compression rate. The results presented in Figure 2 and Figure 3 lead to the following observations:

- All approaches achieve a 40% or better compression rate.
- As expected, the VMFlockMS achieves the highest compression rate with the *app* workload. This is the result of the high similarity (the same OS version) between the VM

images part of the same application. We expect that this will be the most common scenario in practice.

- When the destination site does not have any images that may have some similarity with the images transferred (Figure 2) VMFlockMS generally achieves a better compression rate than alternatives. When images in the VMFlock are based on the same OS distribution, VMFlockMS achieves up to 2.2x better compression rate than the best GZip based approach, and up to 2.7x better than the *Dedup-separate* approach. However, when there is limited similarity between images in the same VMFlock, that is for the *diff-os* workload, VMFlockMS leads to a slightly worse compression rate than GZip based approaches.
- When the destination site stores a VM image that is similar to (some of) the VM images in the migrating VMFlock (Figure 3), deduplication based approaches achieve much better compression rate (4x-12x better) than the best GZip based approach for the *app* and *same-os* workloads and slightly better for the *diff-os* workload. Additionally, depending on the workload VMFlockMS can compress up to 2.8x better that the *dedup-separate* approach; due to its ability to exploit intra-flock similarity. .
- Unexpectedly, *GZip-All* does not achieve a better compression rate than *GZip-Separate* although the data is compressible. We attribute this shortcoming to its design: *gzip* is unable to detect similar data blocks that are tens of MBs apart.
- *Dedup-separate* achieves the worst performance when the destination does not contain any images. This is mainly due to two reasons. First, the mechanism does not exploit similarities across the VM images in the migrating VMFlock. Second, the mechanism chunks the VM images into small blocks (1KB each) and the associated block metadata that is transferred over the network is relatively large.

4.2.1 Compression Overheads

In addition to the compression rate, an important metric to consider is the compression effort. To give rough approximation, Figure 4 compares the time taken to read the data from the disk, compress it using each of the alternatives explored, and write the result back to disk data using the different tools (the three boxes of each bar in Figure 4 present this three-way split).

To include the overheads resulted by detecting similarity across datacenters, Figure 4 presents the compression time for two alternatives: when the destination does not have any files, and when the destination has provided the hash-index for a single 'Fedora-desktop' VM image (thus, for each technique, Figure 4 presents a pair of bars).

Since our goal is to compare the overall compression effort, the experiments use a single-thread implementation of the tools. We note that each of these tools can be parallelized to reduce the compression time (VMFlockMS demonstrates this for data deduplication), however, our point here is to roughly estimate the compression overheads.

The results lead to the following observations:

- The deduplication-based approaches (VMFlock and *dedup-separate*) have the lowest compression effort. They are significantly faster than GZip-based approaches. This is testimonial to the lower computational overheads imposed.
- The deduplication based approaches (VMFlock and *Dedup-seperate*) work at significantly higher throughput than compression based approaches (over 250MB/sec when excluding disk overheads). This makes the reading the data form the VM repository the main throughput bottleneck in our experimental setup. (For comparison *gzip-all* achieves 19.8MB/sec compression throughput).

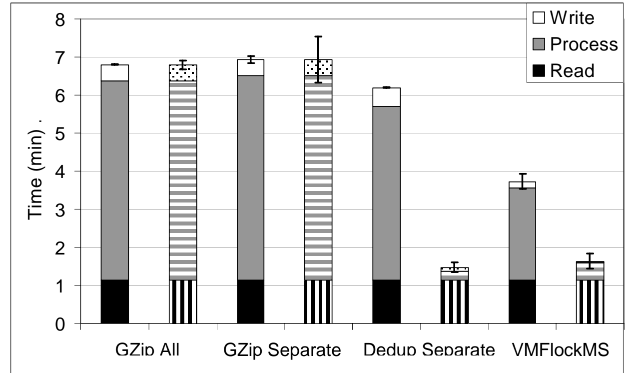

Figure 4. Compression time evaluation with the *app* workload. Each pair of bars represents one compression/deduplication technique with and without a similar VM image as the destination. (e.g., the first bar presents the runtime for *gzip-all* technique with no data at the destination). Each bar has three boxes: The bottom box is the time to read the data form disk, the middle box is the time to compress/deduplicate the data, and the top box is the time to write the data back to disk.

4.2.2 Block Size Effect on Compression Rate

The block size is an important parameter for configuring the deduplication algorithm and controlling the associated tradeoffs. While larger block sizes produce smaller metadata (and thus reduce the tool's memory footprint), they often lead to worse compression rates [15]. Figure 5, shows the compression rate for the a*pp* workload while varying the block size. A block size of 1KB leads to the best compression rate in both cases, that is, regardless of whether a similar VM image is present at the destination or not. A block size of 4KB is an attractive alternative: it preserves most of the compression gains, yet it is 4x better in terms of memory footprint.

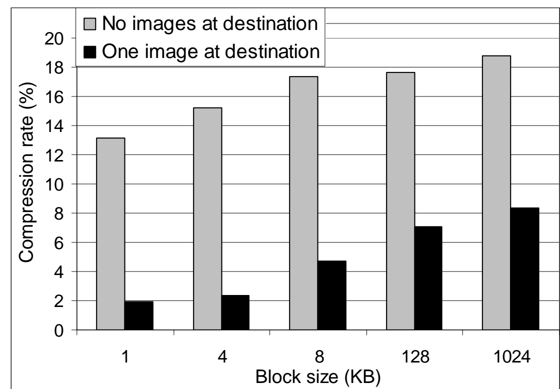

Figure 5. Block size effect on the VMFlockMS compression rate for *app* workload.

4.2.3 Summary

These experiments demonstrate that VMFlockMS is able to efficiently detect the multiple types of similarity present in the data and to significantly reduce the data transfer volumes required for the transfer of VMFlocks across datacenters. Additionally, the evaluation confirms that the overheads of the deduplication scheme used are lower those of competing approaches. Finally, the experiments inform the choice of block size to control the tradeoff between the compression rate and the memory overheads and ultimately the data granularity VMFlockMS operates with.

4.3 Evaluating the Opportunity to Support Early VM Boot

To evaluate the potential gains brought by the VMProfiler/VMLaunchPad pair we estimate the data volumes needed to completely boot a VM image and launch an application. To this end we have used the data produced by the VMProfiler.

- To boot the VMs belonging to the *app* workload, only around 13.5MB of the data (representing 0.6% of the uncompressed VM image size, and roughly 5% of the compressed VMFlock size) are needed. The boot process takes 41s once the data is available locally.

  We expect, however, that if a VM image similar to the one transferred by VMFlockMS is already available at the destination, the data volume actually transferred will be much lower than 13.5MB (as we expect the similarity rate to be high for kernel data) and the main overhead to be related to deduplication and transferring block metadata indexes.

- To start the Spree e-commerce application, 1.5MB of additional data is consumed. The process takes an additional 12s.

- We measured the overhead introduced by VMLaunchPad by running synthetic IO intensive application with and without VMLaunchPad. The overhead introduced by VMLaunchpad is about 3%.

While these values will vary across various operating systems, applications, and hardware platforms they support our claim that there are sizeable benefits to be reaped by starting the VM boot process early and overlapping the data transfers and the application startup processes.

4.4 End-to-end Data Transfer Evaluation

To evaluate the efficiency of the entire system we evaluate the transfer time for an entire VMFlock between two distant datacenters: IBM Almaden and IBM T.J.Watson. The transfer time is measured as the time from the launch of the VMFlock transfer command until all the VMFlock data is transferred and all VM images are completely reconstructed at the destination site.

The experiment is configured as follows: We use the *app* workload and assume that the destination has one VM image, a 'Fedora-desktop' image. Deduplication is configured to use 1KB blocks.

Figure 6 compares the performance of VMFlockMS to the other alternatives.

The results are surprising: despite the fact that VMFlockMS (as well as *dedup-separate,* the other deduplication-based technique) offers significantly better compression rates than gzip-based solutions, end-to-end they perform significantly worse (more than 2x slower) in terms of migration time using our testing platform.

There are two reasons behind this degraded performance: first, an intensely random disk workload for the deduplication-based techniques (as opposed to a sequential workload for the gzip-based ones) and, second, our poorly provisioned experimental testbed (the VM image repository is hosted on a single node using a single SAS commodity disk and accessed through NFS). While gzip-based approaches generate a sequential write workload, the deduplication based approaches generate an intense random read/write pattern: all destination nodes operate the deduplication in parallel, thus their reads/writes to the VM image repository are interleaved. Additionally, even the accesses from a single destination node are random as (at least with the current implementation) the blocks are processed as they arrive from the network (in what is essentially random order), and each new block might need to be written to multiple VM images. Similarly a block identified as already existing at the destination implies a read then possible multiple writes. Additionally, our chosen block size (1KB) stresses the disk to the maximum. With this workload, the roughly 100MBps sequential read/write throughput of our VM repository is degraded to a few MBps.

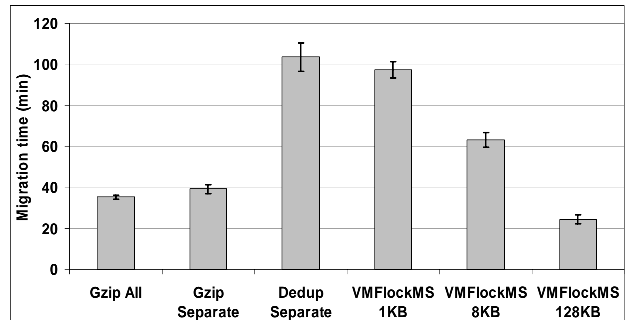

Figure 6. Transfer time for an entire VMFlock. The workload used is *app; t*he destination has one image similar to one image in the migrating VMFlock. We evaluate VMFlockMS with 1, 8, and 128KB block sizes.

While various optimizations to increase the sequentiality of the disk accesses are possible we have not explored them in this work.

We explore instead two avenues to get around the above limitations: First, we explore the performance with larger blocks: 8KB, and 128KB blocks lead to a worse compression rate (4.5% and 7.5%, respectively, vs. 2.07% see Figure 5) yet they reduce the number of disk seeks. Indeed we observe significantly better performance; with block sizes of 128KB VMFlock achieves 1.5x better migration time than GZip based approaches.

Second, we explore the impact of having a faster storage system, more realistic for the environment where the VM image repository will be deployed to support a cloud. To this end we have developed a storage system emulator based on the FUSE [32] file system. The emulator receives the VMFlockMS write operations and emulates writing them to a storage system by imposing a fixed *seek* delay and serving the read/write requests at certain *throughput.* To validate the emulator we have emulated the NFS based repository in our testbed and repeated the VMFlock transfer experiments: the transfer times obtained this way roughly match the actual results (presented in Figure 6). We configure the emulator to match a mid-level storage system with 4 SAS disks @

10,000 RPMs and configure the emulator to use 0.20ms average seek time and 100MBps throughput. We also configure the deduplicaton mechanisms to use 4KB block sizes.

We repeat the same experiments using the emulator to host the VM image repository. Figure 7 presents the migration time. For this system, VMFlockMS completes migrating the VMFlock in around 10min, 3.5x faster than GZip based approaches, and 1.3x faster than dedup-separate.

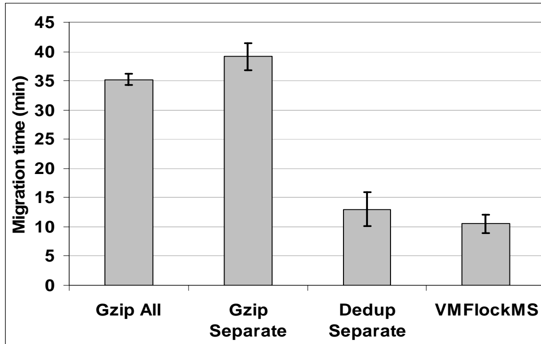

Figure 7. Migration time evaluation using a VM image repository emulator. The workload used is *app;* the destination has one image similar to one image in the migrating VMFlock.

Summary. These experiments demonstrate that, even over our poorly provisioned testbed, a well configured VMFlockMS is able to achieve 1.5x shorter migration time than compression based approaches. Further, emulating a more realistic VM repository infrastructure indicates that VMFlockMS can achieve up to 4x shorter migration time. Finally, optimizations to increase the sequentiallity of the VMFlockMS workload will likely bring further performance improvements.

5. DISCUSSION

This section focuses on a number of interrelated questions:

1.) Can VMFlockMS migrate VMs across different cloud providers given the incompatible cloud APIs? Can it transfer data not stored in VM images?

VMFlockMS can migrate VM images and virtual disks across the cloud. Further, VMFlockMS can be extended to transfer other virtual assets, e.g. Amazon S3 buckets [38], among datacenters as long as a read/write or even a put/get API is provided. We estimate that inter-cloud compatibility issues will be limited: VMFlockMS will need to be specialized to interface with the specific API provided by each cloud to access the VM image repository or its data store.

2.) Can the techniques introduced by VMFlockMS be used in other scenarios?

While this paper presents and evaluates VMFlockMS in the context of VM migration, VMFlockMS can be used as a general tool for data migration across datacenters as it can exploit multiple types of similarity existing in the dataset to reduce the transferred data volume.

One of the scenarios where our approach can be directly applied is: scientific computing. Scientific data sets, in fields as diverse as bioinformatics and high energy physics, are large (TBytes) and are often transferred between collaborating datacenters for further analysis. GridFTP [39] is currently a de facto standard for data transfer between datacenters and uses multiple nodes at the source and destination to accelerate data transfers. VMFlockMS techniques could directly be imported into GridFTP to reduce the volume of data volume transferred. Further, our infrastructure to prioritize data transfers according to application data-usage profiles can also be adapted to prioritize data transfers and enable earlier application launch in order to accelerate transfer / compute pipelines. Monti et al. [40] demonstrate the benefits of a similar approach, dubbed prefix computation, limited however to sequential data accesses.

3.) Can hash collisions corrupt the VM images?

Deduplicating data using hash functions has been a topic of debate [41, 42] since a hash collision can lead to incorrectly inferring that two blocks are similar, and thus lead to silent data corruptions.

While we do not intend to settle this debate for the general case, we note that, VMFlockMS uses SHA1 with a hash length of 160 bits. With this length the probability of a hash collision is significantly lower (10^{-48}) than other sources of silent errors (e.g., disk errors [43], or an alpha particle hitting the bits storing the data in memory [44]). Moreover, it is easy to further reduce this probability if this issue becomes a concern either by using longer hashes (e.g. SHA256, and SHA512) or by adding a mechanism to verify the integrity of complete VM images after retrieving all data blocks.

4.) What if instead of preserving the isolation between the VMFlock migration service and the VM image repository they are co-designed?

Nowadays, cloud infrastructures do not provide an easy path for VM migration. As clouds mature, we envision that VM migration across data centers will be adopted by cloud providers and will be integrated as a cloud service (as it is the case today with intra-datacenter live migration).

Integrating the VM migration service with the cloud infrastructure opens new design avenues to co-design the migration service and the VM image repository. Multiple opportunities can be harnessed: First, as we argue in this paper, the cloud infrastructure can facilitate early launch of migrated VMs through services similar to the VMProfile/VMLaunchPad we prototype. Second, the VM image repository can be built as a content addressable storage (CAS) (e.g., similar to Mirage [6]) and allow the migration service to access its block-level information. This will eliminate most of the deduplication-based overheads. Finally, the cloud infrastructure may provide an SSD based scratch space for on the fly VM image creation. This will significantly increase the migration performance since SSDs better handle the random access workload generated by reconstructing VMFlocks.

6. SUMMARY

This paper presents VMFlockMS, a migration service optimized for cross-datacenter transfer and instantiation of groups of virtual machine (VM) images. We dub these groups of related VM images *VMFlocks*. The optimizations we propose include: data deduplication within the VMFlock to be migrated and between the VMFlock and the data already present at the destination datacenter, as well as prioritization of data transfers and system support to accelerate VM and application startup. VMFlockMS is designed to be deployed as virtual appliance in the cloud: it can make efficient use of the available could resources to accelerate the deduplication and data transfer processes and has minimal

requirements on the cloud API to access the VM image repository.

VMFlockMS provides an incrementally scalable and high-performance migration service. Our evaluation shows that VMFlockMS can achieve compression rates as low as 3% of the original VMFlock size, enables the complete transfer of the VM images belonging to a VMFlock over transcontinental link up to 3.5x faster than alternative approaches, and enables booting these VM images as soon as little as 5% of the compressed VMFlock data is available at the destination.

7. ACKNOWLEDGMENTS

We thank Mark Seaman for his contribution at the early stage in the project and his support for running the experiments.

8. REFERENCES

[1] A. Qureshi, R. Weber, H. Balakrishnan, et al. *Cutting the electric bill for internet-scale systems*. *SIGCOMM* 2009.

[2] T. C. Chieu, A. Mohindra, A. Karve, and A. Segal. *Solution-based Deployment of Complex Application Services on a Cloud*. in *IEEE International Conference on Service Operations and Logistics and Informatics (SOLI)*. 2010.

[3] *WebSphere*. [cited 2010; http://www.ibm.com/software/websphere/.

[4] T. Chieu, S. Kapoor, A. Mohindra, and A. Shaikh. *Cross Enterprise Improvements Delivered via a Cloud Platform: A Game Changer for the Consumer Product and Retail Industry*. in *IEEE International Conference on Services Computing (SCC)*. 2010.

[5] K. Keahey, M. Tsugawa, A. Matsunaga, and J. A. B. Fortes, *Sky Computing*. IEEE Internet Computing, 2009. **13**(5).

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

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

[8] C. Clark, K. Fraser, Steven Hand, et al. *Live Migration of Virtual Machines*. *NSDI* 2005.

[9] T. Hirofuchi, H. Ogawa, H. Nakada, 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.

[10] 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.

[11] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, et al. *SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing*. in *European Conference on Computer Systems (Eurosys)*. 2009.

[12] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, et al. *Optimizing the migration of virtual computers*. *OSDI*. 2002.

[13] A. Muthitacharoen, B. Chen, and D. Mazieres. *A Low-bandwidth Network File System*. *SOSP*. 2001.

[14] S. C. Rhea, R. Cox, and A. Pesterev. *Fast, Inexpensive Content-Addressed Storage in Foundation*. in *USENIX Annual Technical Conference*. 2008.

[15] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, and A. Gharaibeh. *stdchk: A Checkpoint Storage System for Desktop Grid Computing*. in *International Conference on Distributed Computing Systems (ICDCS '08)*. 2008. Beijing, China.

[16] S. Quinlan and S. Dorward. *Venti: A New Approach to Archival Data Storage*. *FAST* 2002.

[17] M. O. Rabin. *Fingerprinting by random polynomials*. in *Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University*. 1981.

[18] K. Eshghi, M. Lillibridge, L. Wilcock, et al. *JumboStore: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service*. *FAST* 2007.

[19] B. Zhu, K. Li, and H. Patterson. *Avoiding the disk bottleneck in the data domain deduplication file system*. *FAST* 2008.

[20] T. Yang, H. Jiang, D. Feng, Z. Niu, et al. *DEBAR: A Scalable High-Performance De-duplication Storage System for Backup and Archiving*. in *International Parallel & Distributed Processing Symposium (IPDPS)*. 2010.

[21] P. Bergheaud, D. Subhraveti, and M. Vertes. *Fault Tolerance in Multiprocessor Systems Via Application Cloning*. in *Int. Conf. on Distributed Computing Systems (ICDCS)*. 2007.

[22] L. Lu, P. Sarkar, D. Subhraveti, S. Sarkar, et al. *CARP: Handling silent data errors and site failures in an integrated program and storage replication mechanism*. in *Int. Conf. on Distributed Computing Systems (ICDCS)*. 2009.

[23] *Cisco Systems Inc. Cisco Cloud Computing - Data Center Strategy, Architecture, and Solutions*. in *White Paper*. 2009.

[24] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. *How to Enhance Cloud Architectures to Enable Cross-Federation*. in *IEEE Conference on Cloud Computing (CLOUD)*. 2010.

[25] *Cloud Computing Interoperability Forum (CCIF)*. [cited 2010; http://groups.google.com/group/cloudforum.

[26] *Open Cloud Consortium (OCC)*. [cited 2010; http://opencloudconsortium.org/.

[27] *Cloud Standards Coordination*. [cited 2010; http://cloud-standards.org/.

[28] *VMware Virtual Appliances Marketplace*. [cited 2011; http://www.vmware.com/appliances/.

[29] *GZip compression tool*. [cited 2011; http://www.gzip.org/.

[30] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, et al. *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web*. in *ACM Symposium on Theory of Computing (STOC)*. 1997.

[31] J.-l. Gailly and M. Adler. *zlib Compression Library*. [cited 2011; http://www.zlib.net/.

[32] *FUSE, Filesystem in Userspace*. [cited 2010; http://fuse.sourceforge.net/.

[33] *PolarSSL*. [cited 2011; http://polarssl.org/.

[34] *QEMU copy-on-write disk format (QCOW2)*. [cited 2010; http://www.linux-kvm.org/page/Qcow2.

[35] *Spree: Open Source E-Commerce Application*. [cited 2011; http://spreecommerce.com/.

[36] *Iperf website*. [cited 2008; http://dast.nlanr.net/Projects/Iperf/.

[37] T. Hirofuchi, H. Nakada, H. Ogawa, S. Itoh, et al. *A live storage migration mechanism over WAN and its performance evaluation*. in *international workshop on Virtualization technologies in distributed computing (VTDC)*. 2009.

[38] *Amazon Simple Storage Service* [cited 2010; http://aws.amazon.com/s3/.

[39] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, et al. *The Globus Striped GridFTP Framework and Server*. in *SuperComputing*. 2005.

[40] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. *Reconciling Scratch Space Consumption, Exposure, and Volatility to Achieve Timely Staging of Job Input Data*. in *IEEE Int. Parallel & Distributed Processing Sym. (IPDPS)*. 2010.

[41] V. Henson. *An Analysis of Compare-by-hash*. in *Workshop on Hot Topics in Operating Systems (HotOS)*. 2003.

[42] J. Black. *Compare-by-Hash: A Reasoned Analysis*. in *USENIX Annual Technical Conference*. 2006.

[43] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. *An analysis of latent sector errors in disk drives*. *SIGMETRICS* 2007.

[44] T. May and M. Woods, *Alpha-particle-induced soft errors in dynamic memories.* IEEE Transactions on Electron Devices, 1979 **26**(1).