Efficient Live Migration of Virtual Machines Using Shared Storage

Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger
School of Computer Science and Engineering, Seoul National University
{changyeon, erik, jeongseok, bernhard}@csap.snu.ac.kr

Abstract

Live migration of virtual machines (VM) across distinct physical hosts is an important feature of virtualization technology for maintenance, load-balancing and energy reduction, especially so for data centers operators and cluster service providers. Several techniques have been proposed to reduce the downtime of the VM being transferred, often at the expense of the total migration time. In this work, we present a technique to reduce the total time required to migrate a running VM from one host to another while keeping the downtime to a minimum. Based on the observation that modern operating systems use the better part of the physical memory to cache data from secondary storage, our technique tracks the VM's I/O operations to the network-attached storage device and maintains an updated mapping of memory pages that currently reside in identical form on the storage device. During the iterative pre-copy live migration process, instead of transferring those pages from the source to the target host, the memory-to-disk mapping is sent to the target host which then fetches the contents directly from the network-attached storage device. We have implemented our approach into the Xen hypervisor and ran a series of experiments with Linux HVM guests. On average, the presented technique shows a reduction of up over 30% on average of the total transfer time for a series of benchmarks.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—Checkpoint/restart; D.4.2 [*Operating Systems*]: Storage Management—Storage hierarchies; D.4.4 [*Operating Systems*]: Communications Management—Network communication

General Terms Design, Measurement, Performance, Reliability

Keywords Virtualization; Live migration; Storage; Xen

1. Introduction

Over the past few years, the availability of fast networks has lead to a shift from running services on privately owned and managed hardware to co-locating those services in data centers [1]. Along with the wide-spread availability of fast networks, the key technology that enables this shift is virtualization. In a virtualized environment, the software does not run directly on bare-metal hardware anymore but instead on virtualized hardware. The environment (such as the number of CPUs, the amount of RAM, the disk space, and so on) can be tailored to the customer's exact needs. From the perspective of the data center operator, virtualization provides the opportunity to co-locate several VMs on one physical server. This consolidation reduces the cost for hardware, space, and energy.

An important feature of virtualization technology is *live migration* [9]: a running VM is moved from one physical host to another. Live migration is attractive to data center providers because moving a VM across distinct physical hosts can be leveraged for a variety of tasks such as load balancing, maintenance, power management, or fault tolerance. The task of migrating a running VM from one host to another has thus attracted significant attention in recent years [6, 9, 12, 13, 15, 18, 20, 27]. Live migration is only useful if the service provided by the running VM is not interrupted, i.e., if it is transparent to the user. To migrate a running VM across distinct physical hosts, its complete state has to be transferred from the source to the target host. The state of a VM includes the permanent storage (i.e., the disks), volatile storage (the memory), the state of connected devices (such as network interface cards) and the internal state of the virtual CPUs (VCPU). In most setups the permanent storage is provided through network-attached storage (NAS) and does thus not need to be moved. The state of the VCPUs and the virtual devices comprise a few kilobytes of data and can be easily sent to the target host. The main caveat in migrating live VMs with several gigabytes of main memory is thus moving the volatile storage efficiently from one host to the other.

The prevalent approach for live VM migration is *pre-copy* [9]. The contents of the VM's memory are first sent to the target host and then the VM is restarted. To keep the *downtime*, i.e., the time during which the VM is not running, to a minimum, data is sent in several iterations while the VM keeps running on the source host. In each following iteration, only the pages that have been modified since the last round are sent. Another approach is *post-copy* [13]. Here, only the VM's VCPU and device state is sent to the target host and restarted there immediately. Memory pages accessed by the VM are then fetched in parallel and on-demand while the VM is running on the target host.

Both of these approaches minimize the downtime of the VM at the expense of the *total migration time*, i.e., the time from when the migration is started until the VM runs independently on the target and can be destroyed on the source host. Several techniques aim at reducing the total migration time through compression of memory pages [16] or trace-and-replay [18].

In this work, the goal is to reduce the total migration time by minimizing the data sent across the distinct physical hosts. Park *et al.* [22] have observed that in typical setups a considerable amount of data in the memory is duplicated on disk; they report up to 94% of duplication in extreme cases with lots of disk I/O. This duplication is caused by modern operating systems' disk caches

VEE'13, March 16–17, 2013, Houston, Texas, USA.

with which the long latency to physical storage is to be hidden. Park *et al.* have used this observation to minimize the image size of VMs that are to be restarted on the same physical host at a later time. In our work, the same idea is applied to live migration for VMs using shared network-attached storage: instead of transferring the entire (possibly compressed) memory data, we only send the data of memory pages whose content is not available on the shared storage device. In order to restore the complete memory image on the target host, the source host sends a list of shared storage blocks along with the VM's memory locations. The target host then fetches these disk block directly from the attached network storage while the migration continues running. This approach is especially attractive in a setup where the maximum bandwidth between the hosts is limited but the NAS is connected through a high-speed network. Our approach aims to improve the total migration time of VMs within data centers with network-attached shared storage; the proposed method does not optimize live migration across data centers.

The presented method is independent of almost all other optimization techniques such as iterative pre-copying, post-copying, and data compression. Existing techniques can be augmented by integrating the approach presented here to (further) reduce the total migration time.

The contributions of this paper are as follows:

- we propose an efficient technique for live-migrating VMs from one host to another by transferring only unique memory pages directly from the source to the target. Pages that are duplicated on disk located on shared storage are fetched directly by the target host. To the best of our knowledge, this is the first approach that does not send duplicated data directly from the source to the target host.
- we show the feasibility of the proposed technique by providing an implementation of this technique in version 4.1 of the Xen hypervisor [8] for HVM guests.
- we demonstrate the effectiveness of the proposed technique by running a series of benchmarks. We achieve an average improvement in the total migration time of over 30% with up to 60% for certain scenarios at a minimal increase of the downtime. In addition, thanks to the shorter total migration time, the overhead caused by live migration is reduced which leads to an increased performance of the migrated VM.

The remainder of this paper is organized as follows: Section 2 gives an overview of related work on live migration of virtual machines. In Section 3, the technique to efficiently migrate a VM is described in more detail. Section 4 contains the design and implementation of the proposed live migration framework. Section 5 evaluates our technique, and Section 6 concludes the paper.

2. Related Work

Live migration is actively being researched and a number of techniques have been proposed to migrate a running VM from one host to another. The predominant approach for live VM migration is pre-copy. The bare-metal hypervisors VMware [28], KVM [12], and Xen [2], plus hosted hypervisors such as VirtualBox [21] employ a pre-copy approach. To reduce the downtime of the VM, the state of the VM is copied in several iterations [9]. While transferring the state of the last iteration, the VM continues to run on the source machine. Pages that are modified during this transfer are recorded and need to be re-transmitted in the following iterations to ensure consistency. The iterative push phase is followed by a very short stop-and-copy phase during which the remaining modified memory pages as well as the the state of the VCPUs and the devices are transferred to the target host. The pre-copy approach achieves a very short downtime in the best case, but for memory-write-intensive workloads the stop-and-copy phase may increase to several seconds. Remote Direct Memory Access on top of modern high-speed interconnects can significantly reduce memory replication during migration [15].

Post-copy-based techniques take the opposite approach: first, the VM is stopped on the source host and the state of the VCPU and devices is transferred to the target host. The VM is immediately restarted on the target host. Memory pages are fetched on-demand from the source machine as the VM incurs page-faults when accessing them on the target machine. This approach achieves a very short down-time but incurs a rather large performance penalty due to the high number of lengthy page faults on the target machine. Hines *et al.* [13] combine post-copying with dynamic self-ballooning and adaptive pre-paging to reduce both the amount of memory transferred and the number of page faults. Hirofuchi *et al.* [14] employ a post-copy-based approach to quickly relocate VMs when the load of a physical host becomes too high.

Other techniques include live migration based on trace and replay [18], memory compression [16, 24], simultaneous migration of several VMs from one host to another [11], or partial VM migration [5]. Liu *et al.* [18] present a technique that first writes the state of the running VM to the local storage on the source machine. That state is then transferred once to the target machine. The VM continues to run on the source machine and all modifications to memory pages are logged in a trace file which is transferred iteratively to the target host. The target host then replays those modifications in the copy of the running VM. At the end of the iterative phase, a short stop-and-copy phase is used to transfer the final state to the target machine. Jin *et al.* [16] enhance the iterative pre-copy approach by compressing the data that is to be transferred in each round from the source to the target machine. Svärd *et al.* [24] extend this idea by delta-compressing changes in memory pages. Deshpande *et al.* [11] study the problem of simultaneous migration several VMs. Their idea is to detect memory pages of identical contents across different VMs and transfer duplicated pages only once. Our work in this paper is orthogonal to memory compression and simultaneous live-migration. Bila *et al.* [5] propose partial VM migration of idle VMs running on users' desktops to a consolidation server with the goal of reducing overall energy consumption. The authors use a post-copy approach where only the accessed memory pages and disk blocks are transferred to the server. The partially-migrated VMs continue running on the server while the desktops can be put into a low-power mode. As soon as the user continues to work on his desktop, the modified state of the partially migrated VM is transferred back to the desktop. The authors report significantly reduced transfer times to migrate the working set of the idle VM. Other than in our approach, however, the VM on the desktop can not be destroyed since it is never fully transferred.

The work most closely related to our technique was presented by Park *et al.* [22]. The amount of data saved during (iterative) checkpoints is significantly reduced by tracking memory pages residing in identical form on the attached storage device and not including such pages in the memory image. Instead, a mapping of memory pages to disk blocks is stored in the checkpoint image which is used to load the data directly from the storage device instead of the checkpoint image. We use the same technique as Park to transparently intercept I/O requests and maintain an up-to-date mapping of memory pages to disk blocks. Our work differs from Park's approach in that we apply the same idea to live migration: instead of transmitting memory data that exists in identical form on the network-attached storage device, we transfer a list of disk blocks along with the memory locations to the target host. The receiver on the target host then fetches these blocks directly from the attached network drive into the VM's memory.

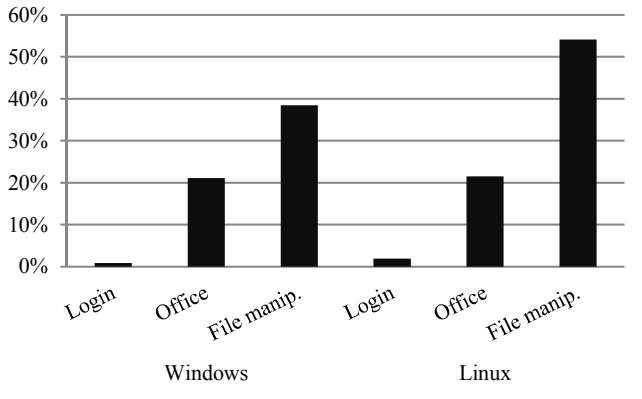

Figure 1. Amount of duplication between memory and external storage.

3. Efficiently migrating live VMs

Migrating a running VM from one physical host to another requires that the entire state of the VM is transferred. The state of a VM comprises the VCPUs, the configuration of the drivers, the VM's memory and the permanent storage. In data center setups, the permanent storage is typically a network storage device - an iSCSI partition or a drive mounted via NFS. Consequently, the contents of permanent storage do not need to be moved to the target host. With memory sizes of several gigabytes even for virtual machines, transferring the memory state to the target host thus becomes the bottleneck of live migration.

3.1 Motivation

Modern operating systems cache data from permanent storage in unused volatile memory to hide the long access latency. The longer the system is running, the bigger an amount of otherwise unused memory is dedicated to this cache. In data centers, the physical hosts are typically attached over a very fast network to a network storage device as well as to the external world. During live migration, a lot of data needs to be sent from one physical host to another. In order not to affect the quality of service of the whole data center, the maximum bandwidth with which data is sent between distinct physical hosts is usually limited. This can easily lead to migration times of tens of minutes for VMs that have several gigabytes of main memory. Our goal is thus to detect duplicated data and fetch this data directly from the attached storage device. This considerably reduces the amount of data sent between the two hosts involved in live migration and has the potential to significantly shorten the total migration time.

Duplication of data between memory and external storage. The operating system and the running applications often occupy only a small fraction of the total available memory, leaving most of the memory unused. Modern operating systems use this unused memory to cache recently accessed blocks of the attached storage device. The data of this cache is thus duplicated: one copy resides on the permanent storage device, another copy exists in the memory of the VM. In addition to cached data, application data such as code pages or read-only data pages also exist in external storage as well as in the memory. It is not uncommon that the amount of duplication between the permanent storage and the memory reaches more than 50%, and this trend is likely to continue with ever-increasing memory sizes. Park *et al.* [22] have measured the amount of duplication between disk and memory for Linux and Windows HVM guests running in Xen. Even for relatively small memories (1 GB), they have observed a duplication ratio of 93% for the Linux HVM guest after heavy I/O. Figure 1 shows the results of our experiments with Linux and Windows HVM guests running on a VM with 4 GB of RAM. The first data point represents the amount of duplication after booting the system up and logging in. The second data point was taken after performing some editing in the LibreOffice application suite [25]. The third data point, finally, was taken after copying data from a USB stick to the local storage device. Both HVM guests, Windows and Linux, show that over time, more and more data is cached in memory. For the last data point in Figure 1, in the Windows HVM 40% or 1.5 GB of the total memory is duplicated on external storage. For Linux, the amount of duplication is even higher with 55% or 2.2 GB of data. These results demonstrate that there is a lot of potential for reducing the amount of data that needs to be sent from the source to the target host during live migration.

Transparent live migration. To make live migration as transparent as possible, the utmost concern is the *downtime*, that is, the time between the moment the VM is stopped on the source host and the moment when the VM is restarted on the target host. To achieve a short migration downtime, several techniques have been developed [9, 13, 18, 24] that have very short downtimes at the expense of the total amount of transferred data and/or the total migration time. The prevalent method is to send data in iterations to the target host while the VM keeps running on the source host [9]. Memory pages that get modified on the source host need to be retransmitted in one of the following iterations. With pre-copying approaches the total amount of data can thus be significantly bigger than the size of the VM's memory. Other techniques, such as post-copy [13] immediately restart the VM on the target host and fetch the accessed memory pages on-demand. Such approaches have a very short downtime and transmit each memory page exactly once; however, the VM on the target host can experience a significant performance degradation due to the frequent high-latency page fetch operations.

Network topology in data centers. A high-speed connection to the outside world as well as the network-attached storage device are indispensable for a data center setup. The interconnection network between the physical hosts, on the other hand, does not need to be as fast; furthermore, the available bandwidth is preferably used to provide access to the VMs from the outside world. To minimize the effects of live migration on the quality-of-service in a data center environment, the maximum bandwidth available during live migration is often limited. Unavoidably, a rate-limited connection increases the total migration time.

Based on the three observations above, we propose the following method to efficiently migrate a running VM: at runtime, we track all I/O operations to permanent storage and maintain an updated list of memory pages that are currently duplicated on the storage device. When migrating, instead of transferring those pages over the rate-limited connection from the source to the target host, the target machine reads them directly from the attached storage device. This technique is orthogonal to iterative pre-copying or post-copy approaches, and can also be combined with techniques that compress the data to be transferred. In the following section, the proposed method is described in more detail.

3.2 Design

The key idea of the proposed method is to fetch memory pages that also exist on the attached storage device directly from that device instead of transferring them directly from the source to the target host. Figure 2 illustrates the idea: memory pages that have been recently read from or written to external storage and are thus duplicated are shown in green. Such pages include pages from code and read-only data sections of running applications and cached disk

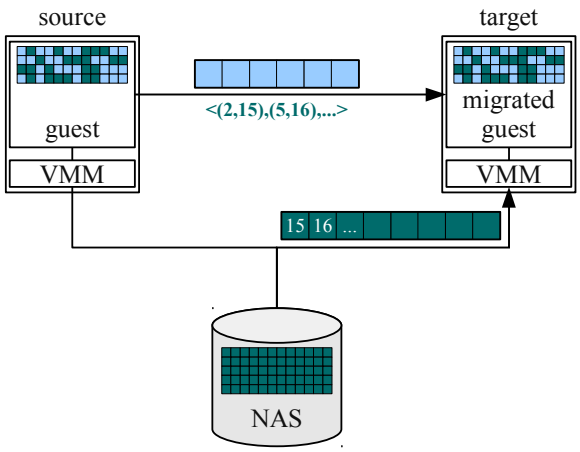

Figure 2. Duplicated memory pages are loaded directly from external storage.

blocks. Shown in blue are memory pages which contain data that is not duplicated. These pages include, for example, the stack.

Detecting duplication between memory pages and disk blocks immediately before transmission is infeasible due to the large computational overhead. One would have to keep a hash value for every disk block, then compute and compare the hash value of a memory page with those of all disk blocks. Instead, we transparently intercept all I/O requests issued by VMs and maintain an up-to-date list of memory pages to disk blocks.

Transparent I/O interception is possible because the VMM has to arbitrate accesses to shared devices such as NICs and attached storage devices in order to ensure correct operation and proper isolation between the different VMs. To detect changes to memory pages that contain disk block data, we re-map such pages *read-only*. Whenever a guest tries to modify such a page by writing to it, a memory page fault is raised. This page fault is intercepted by the VMM which then removes the page-to-block mapping, restores the write access permission of the page and restarts the operation. Transparently intercepting I/O operations and especially the extra page faults cause some overhead; however, as shown in Section 4.1, due to the long-latency nature of I/O operations and the fact that for each tracked memory page at most one extra page fault can occur, this overhead is not noticeable.

When a VM running on a source host is about to be migrated to a target host, the contents of the VM's memory are sent to the target host. We consider iterative pre-copying here, however, since the presented method is largely independent of existing techniques it can easily be integrated into those techniques as well. In an iterative pre-copy approach, the contents of the running VM's memory are sent to the target host over several iterations[9]. Since the VM continues to run while the data is being transferred, memory pages that have already been transferred can get modified. The VMM thus tracks changes to already transferred pages by marking the corresponding page table entry (PTE) in the memory management unit (MMU) *read-only*, much in the same way the I/O tracking process is operating. In the following iteration, only modified memory pages need to be sent to the target host. This iterative process stops if (a) only very few memory pages have been modified since the last iteration or (b) if a maximum number of iterations is reached. Which termination condition holds depends on what tasks the running VM is executing: tasks that generate only few modifications to memory pages (i.e., *dirty* pages at a low rate) tend to terminate the iterative process early, while VMs that dirty memory pages at a high rate will eventually hit the iteration threshold. Once the iterative process has stopped, the VM is stopped on the source host, the remaining dirtied memory pages are sent to the target host along with the state of the VM's devices and VCPUs, and then restarted on the target host.

The proposed technique seamlessly integrates into the iterative pre-copy approach: instead of sending all dirtied memory pages to the target host, only pages whose contents are not duplicated on external storage are transferred. Dirtied pages whose contents are known - because they have been loaded from or written to external storage and have remained unmodified since - are assembled into a list containing pairs of the form `(PFN, disk block)` where `PFN` denotes the memory page in the guest and `disk block` contains the index of the disk block(s) containing the data.

The tracking of dirtied pages is identical to that in the unmodified iterative pre-copy approach, and happens in addition to the transparent interception of I/O. The list of duplicated pages is assembled by joining the bit vector containing the list of dirtied pages with the current mapping of memory pages to disk blocks. It is thus possible that in two subsequent iterations the identical element `(PFN, disk block)` appears in the list of pages to be loaded from external storage. This happens if, for example, block `disk block` is loaded into memory page `PFN` before the first iteration, then subsequently written to and flushed back to external storage between the first and the second iteration. The list of duplicated pages contains the same element `(PFN, disk block)`, but the dirty bit for page `PFN` will trigger inclusion of the page in the current round.

On the target host, a receiver process fetches the list of duplicated pages and the contents of the dirty pages. Dirty pages are simply copied 1:1 into the memory space of the VM being migrated. The list of duplicated pages is sent to the *NAS fetch queue*, a background process that processes the items on the fetch queue by loading the contents from disk and copying them into the appropriate memory page. Since the NAS fetch queue operates in parallel to the migration process, it is possible that a data from disk arrives after a newer version of the same page has already been received in one of the successive iterations. To detect such cases, the target host maintains a version number for each memory page. For each memory page received in a successive iteration all outstanding requests to the same page in the NAS fetch queue are discarded. Similarly, if a fetch request is received for a page that is still in the NAS fetch queue, the old request is discarded as well. If the NAS fetch queue has just issued a load operation to the NAS device while the target memory page is being overwritten with newer data, then the data fetched from disk is discarded upon arrival at the host.

After the data in the final round has been sent to the target host, the VM cannot be restarted until all entries on the NAS fetch queue have been processed. This synchronization has the potential to adversely affect the downtime of live migration. A feedback mechanism ensures that this does not happen: the NAS fetch queue continuously measures the I/O bandwidth of the NAS device and reports it back to the source host along with the number of outstanding requests. The source host can estimate the time required to process all outstanding plus new requests and decide to send some duplicated pages over the direct link instead of including them in the list of duplicated pages. In the last round, all dirtied pages, duplicated or not, are send directly over the direct link.

Figure 3 illustrates the operation of the proposed technique with an example. When the live migration is started on the source host, initially all memory blocks are marked dirty. The list of duplicated pages shows that memory pages 2, 5, and 8 contain the contents of disk blocks 15, 16, and 3, respectively. During the first round, the source host thus first sends `<(2,15), (5,16), (8,3)>` to the target host, followed by the contents of memory pages 1, 3, 4, 6, and 7. The target host adds the list of duplicated pages to the (initially empty) NAS fetch queue and copies the memory pages

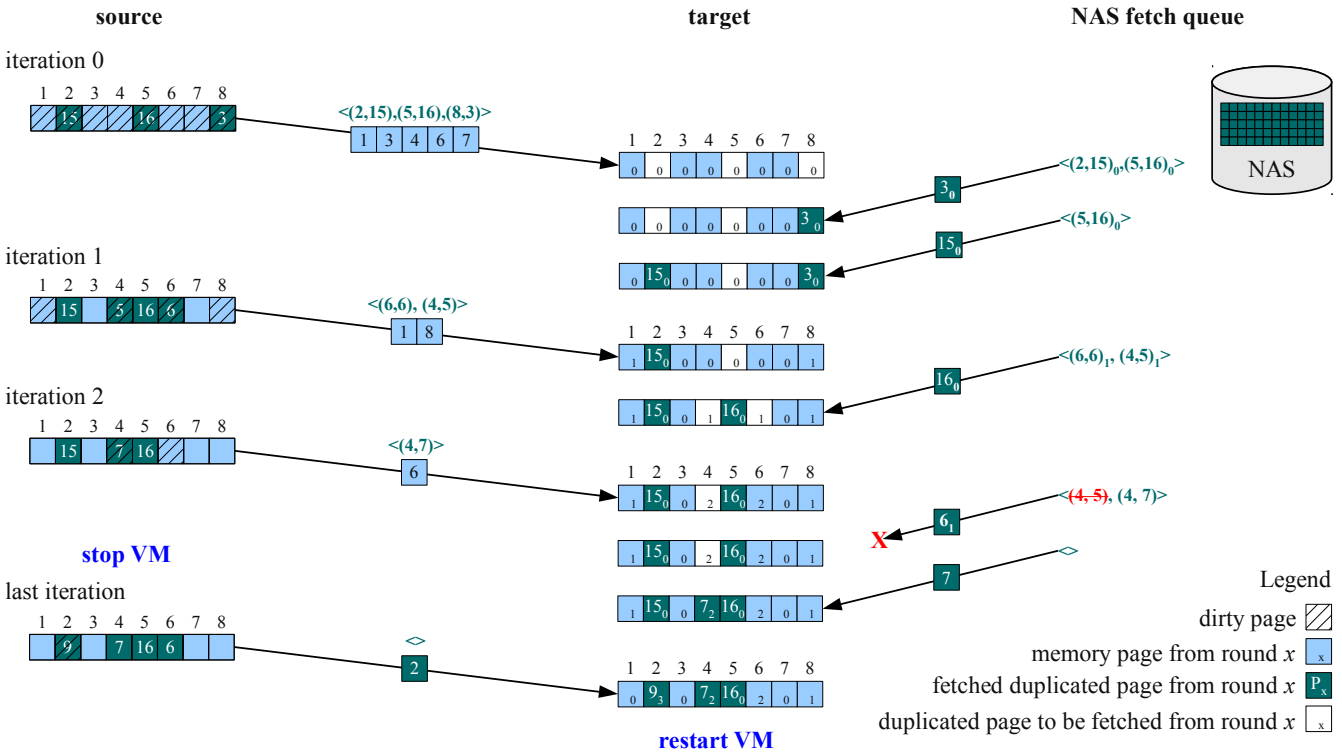

Figure 3. Memory pages whose contents are known to be duplicated on disk are fetched directly from the NAS device.

into the memory space of the VM being migrated. For each page and fetch request is recorded as well. The NAS fetch queue then proceeds to fetch disk blocks 3 and 15 from the NAS device while the VM continues to run on the source host. Each processed fetch request is removed from the queue. In iteration 1, the source host again sends the list of duplicated pages `<(6,6), (4,5)>` and the dirtied memory pages (1 and 8) to the target host. The list of duplicate pages is appended to the NAS fetch queue, and page data is directly written into the VM's memory. Between iteration 1 and 2, memory page 4 has been replaced by disk block 7 and is resent to the target host. The NAS fetch queue thus removes the old request for memory page 4, `(4,5)` from the queue and replaces it with the new one, `(4, 7)`. Additionally, while the NAS fetch queue is processing the entry `(6, 6)`, a new version for memory page 6 has also been received in iteration 2. Before writing the block data to memory page 6, the NAS fetch queue compares the versions of the data currently residing in page 6 (=2) and the version of the fetched block (=1). Since the memory page already contains a newer version, the data fetched from the NAS is discarded. In the final iteration, only one memory page has been modified. Its contents are duplicated on disk, however, since this is the last round the source sends the contents of the memory over the direct link to the target and does not include it in the list of duplicated pages. The target host waits until the NAS fetch queue has processed all outstanding entries and then restarts the VM.

4. Implementation

We have implemented the proposed technique in the open-source virtual machine monitor (VMM), Xen 4.1 [8]. This is section discusses the details and issues of our implementation. We focus on fully-virtualized guests (HVM guests) in our discussion; however, the implementation for para-virtualized guests is analogous.

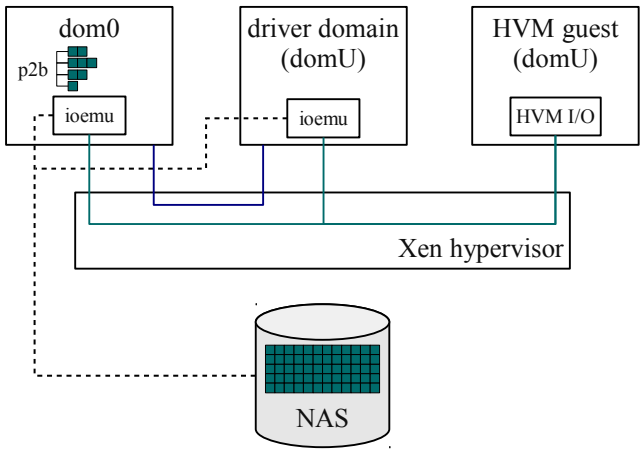

Figure 4. I/O operations in the Xen VMM

4.1 Transparent I/O Interception

In the Xen VMM, all I/O operations to secondary storage are under control of the VMM. The Xen hypervisor itself does not contain any device drivers; instead it delegates the task of dealing with a variety of hardware devices to the privileged `dom0` or, more recently, unprivileged Xen driver (or *stub*) domains. In both cases, I/O requests of the HVM guest running in an unprivileged user domain (`domU`) go through `ioemu`, a modified version of the `qemu` processor emulator [3] (Figure 4).

Mapping page frames to disk blocks. We associate the VM's memory contents with physical storage blocks by transparently tracking all read and write requests from the HVM guest to external

storage. The collected information is stored in a list of duplicated pages, the page-to-block map `p2b`. For each running VM, a separate `p2b` map is maintained. The map is indexed by the VM's memory page frame number (PFN), since the number of memory page frames is typically much smaller than the number of disk blocks. The data stored in the `p2b` map is the 8-byte storage block number.

Maintaining consistency. In order to maintain a consistent `p2b` map, the hypervisor needs to track memory writes to memory pages currently included in the `p2b` map. For HVM guests, the hypervisor either maintains shadow page tables [19] or hardware-assisted paging (HAP) provided by newer hardware [4]. We only support HAP at this moment; however, Park *et al.* [22] have shown that tracking modifications using shadow page tables is also possible. Whenever an entry is added to the `p2b` map, the corresponding memory page is marked *read-only* using the MMU's page tables. When the guest issues a store operation to such a page, the subsequent page fault is caught by the Xen hypervisor. If the affected memory page is currently included in the `p2b` map, the handler removes the entry, re-maps the page *read-write* and restarts the store operation.

Overhead. Both the space and runtime overhead of the `p2b` map are small: implemented as a hash map, the `p2b` map contains one entry per memory page that is currently duplicated on external storage. In terms of space requirements the worst case is when every memory block is duplicated on external storage. In that case, the number of entries is equal to the VM's memory size divided by the size of a memory page (typically 4KB). With 8 bytes per entry this translates to 2 MB of storage per one GB of virtual memory, a space overhead of 0.2%. The `p2b` map needs to be updated on each I/O operation to external storage. Such I/O operations have a long latency so that the `p2b` update operations are completely hidden. To track (memory) write operations to memory pages that are known duplicates of disk blocks, the Xen VMM maps such pages as *read-only*. A subsequent write operation thus incurs the overhead of an extra page fault during which the page table entry of the affected page is re-mapped *read-write* and the corresponding entry is removed from the `p2b` hash map. Since an entry is removed from the map on a write operation, each entry on the list can incur at most one additional page fault. To add an entry, a costly I/O operation is required; even in the worst-case where every page read from external storage is invalidated by performing one write operation to it, the time overhead caused by the extra page fault is hidden by the long-latency I/O operation.

4.2 Iterative Pre-Copy

When the live migration is started, all memory pages are initially marked dirty. In each iteration, the memory pages that are sent to the target host are marked clean. Similar to I/O tracking, Xen intercepts modifications to memory pages by mapping such pages *read-only* with the help of the MMU.

Integration of the proposed technique into the iterative pre-copy algorithm is easy: in each iteration, the pre-copy algorithm assembles the list of memory pages to be sent to the target host in this round. We do not modify this process; however, before the data is sent over the network, our implementation first inspects the list of assembled pages. Pages that are known to be duplicates of blocks residing on external storage are removed from the batch and added to the list of duplicated pages. This list is then sent first, followed by the data of memory pages that are not duplicated on external storage.

4.3 Assembling the Memory Image

On the target host, more changes are necessary. First, the receiver is modified to receive a list of duplicated pages before the actual page data. In each iteration, this list if given to a separate background process, the *NAS fetch queue*. This process maintains a queue of blocks that need to be fetched from external storage and loaded into memory. Consecutive blocks are coalesced whenever possible to improve the efficiency of the read requests to external storage. When adding a new batch of entries to the queue, older entries that refer to the same memory page are deleted from the queue. Additionally, all entries referring to memory pages whose contents are transferred in the current round are also removed from the queue.

The data fetched from external storage is not directly loaded into the VM's memory. Instead, it is first loaded in to a buffer. Before copying the buffer to the corresponding page in the VM's memory, the NAS fetch queue checks whether the page contains more recent data from a subsequent iteration. Such situations occur when a page that was known to be duplicated on external storage is added to the NAS fetch queue in iteration i, modified on the source host and then sent over the direct link by one of the following iterations j, where $j > i$. Page 6 in Figure 3 exhibits this situation: in iteration 1, the page is sent as part of the list of duplicated pages and added to the NAS fetch queue (element `(6,6)`). Between iteration 1 and 2, page 6 is modified on the source host and its content are sent and directly copied into the VM's memory in iteration 2. When the (now obsolete) contents of page 6 are finally fetched from external storage, the NAS fetch queue recognizes this situation by comparing the timestamp of both items and discards the old data.

Guaranteeing consistency. Since the proposed method accesses the same external storage device from distinct physical hosts data consistency is an issue. During migration, only the source host issues write requests to the external storage device while the target host only performs read operations. We can guarantee consistency by ensuring the following conditions hold:

- no *write-caching* occurs on any level (except for the guest VM itself) to external storage.
- the target host can issue *uncached reads* to external storage.

The first condition does not prevent the guest VM from performing write-caching; however, once the VM flushes the data to external storage there must be no more write-caching, for example in the source host or the NAS device itself. Similarly, the target host needs to read the most recent data. The attached storage device must provide either direct-I/O that bypasses all caches or not perform any read-caching on any level below the target host.

In order to prevent write-after-read (WAR) hazards - due to different network latencies the target host may read a block before the write command originating from the source host has completed - we ensure that entries on the `p2b` map are only considered committed when the write transaction to the external storage device has completed. Uncommitted entries are treated like ordinary memory pages, i.e., the contents are sent directly to the target host.

5. Evaluation

This section presents the performance characteristics of the proposed technique. We measure the total migration time, the total amount of transmitted data, and the downtime for a variety of benchmarks on a number of network setups. The results show that our approach has the potential to significantly improve the total migration time of live VM migration at an equal downtime.

5.1 Experimental Setup

All experiments are performed on two identical hosts equipped with an Intel Core i5-2500 processor running at 3.3 GHz and 16 GB of main memory. For external storage a QNAP NAS server is used. The two hosts and the NAS server are connected via Gigabit Ethernet. The hosts run the Xen hypervisor 4.1.2 [8] with our modifications. Dom0 runs Ubuntu Server 11.10 (kernel version 3.0.0) [7].

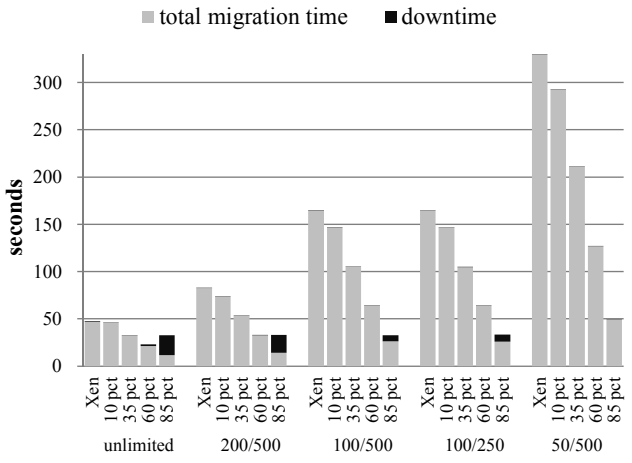

Figure 5. Performance in relation to duplication.

The experiments are performed on an HVM guest running Ubuntu Desktop 10.04 LTS with a 2.6.35 kernel. The VM is configured with two virtual CPUs and 2 GB of RAM. The guest's virtual disk is a file-backed image located on external storage mounted by dom0 using NFS.

5.2 Performance in Relation to Duplication

The proposed technique fetches memory pages whose contents are duplicated directly from external storage; the performance thus directly relates to the amount of duplication. In this first series of benchmarks, an idle VM is migrated. To trigger different ratios of duplication between memory and disk, the VM first copies a varying amount of data from disk to `/dev/null`. Figure 5 illustrates results. `Unmodified` represents the unmodified Xen. In `10 pct`, the VM is booted up and then immediately migrated and thus represents a case with minimal duplication. `35 pct`, `60 pct`, and `85 pct` show the results for duplication ratio of 10, 35, 60, and 85 percent, respectively. To illustrate the effect of the network bandwidth, we run each benchmark using Xen's rate-limiting algorithm with five different configurations: `unlimited`, `200/500`, `100/500`, `100/250`, and `50/500`. In `unlimited` the network bandwidth is not restricted; for the other configurations, the first number denotes the starting rate, and the second number the maximum network bandwidth available for migration in megabits per second.

The results show that the proposed technique successfully reduces the total migration time for all network configurations and amount of duplication. This is expected since if there is no duplication at all, then the proposed solution will perform exactly like unmodified Xen. As the amount of duplication increases, the benefits of the proposed solution become apparent. In the `100/500` configuration with about 50% of duplication (the bars labeled `512MB`), the total migration time including the downtime is reduced from 165 to 64 seconds.

These benchmarks also reveal the main weakness of the proposed technique: the downtime of the VM can increase significantly if the VM is idle and almost all available memory is duplicated on disk. For `unlimited` and `1024MB`, the worst case, we observe an increase of the downtime from 0.55 seconds to 20.8 seconds, almost a 40-fold increase. The reason for this extreme slowdown is as follows: with the proposed solution, live migration finishes within very few iterations. Most of the memory is duplicated on disk, hence only very few memory pages are sent directly to the destination host; the majority of the pages is given to the background fetch process on the destination host which then starts to load these pages into the VM's memory. The adaptive iterative pre-copy algorithm notices that the amount of sent pages is sufficiently low to stop the VM on the source host, however, on the destination host the background process has not yet finished loading the pages from external storage. Since the current implementation waits for the background process to complete before resuming the VM, this delay can lead to a significant degradation in the downtime. For future work, we propose two strategies to eliminate this problem: (1) provide feedback about the progress of the background fetch process to the iterative pre-copy algorithm. The pre-copy algorithm can then, for example, send duplicated pages through the direct link if there is a long back-log of pages to be loaded on the target host. (2) resume the VM before the background fetch process has completed. This will require marking yet unloaded pages as invalid and interception of pagefaults caused by accesses to such pages.

Table 1. Application Scenarios

Application	Description
RDesk I	web browsing, editing in an office suite
RDesk II	playing a movie
Admin I	compressing a large file
Admin II	compiling the Linux kernel
File I/O I	backing up data
File I/O II	Postmark benchmark

5.3 Application Scenarios

Lacking a standard benchmark suite for live migration of virtual machines, we have selected several general application scenarios similar to what has been used in related work [10, 22] and that are representative for a Virtual Desktop Infrastructure (VDI) deployed in data centers. For all application scenarios the user is connected to the migrated VM by a VNC viewer using the RFB protocol [23].

Table 1 lists the application scenarios. `RDesk I and II` represent the standard usage scenario in VDI: a remote user connected to his virtual desktop. In `RDesk I` a Firefox [26] web browser is fetching a number of web pages and some scripted editing tasks are performed in the LibreOffice suite [25]. `RDesk II` plays a movie. These benchmarks exhibit moderate CPU and I/O activity and are expected to perform well both in unmodified Xen and our solution. `Admin I and II` represent administration tasks. In `Admin I` a 2 GB log file is compressed using gzip. `Admin II` compiles the Linux kernel. Both benchmarks exhibit a high CPU load, a large number of dirtied memory pages, as well as a high number of I/O requests (both read and write). We expect our solution to perform significantly better due to the high activity in the page cache and the rate of dirtied pages. The last group, `File I/O I and II` represents I/O-intensive benchmarks. In `File I/O I` a backup process backing up a large (2 GB) file. `File I/O II` runs PostMark [17] (500 files from 100 to 500 KB, 80'000 transactions, 4 KB read/write buffers). These benchmarks are expected to migrate quickly but exhibit performance degradation in the VM due to the simultaneous access to the shared storage caused by the background fetch process.

For each scenario, we measure the total migration time, the downtime of the VM, and the amount of data transferred in each round of the iterative pre-copy algorithm. For the `Administration` and `File I/O` scenario, we also measure the time to completion of the benchmarks running inside the migrated VM (the `Remote Desktop` scenarios have no well-defined completion point). The baseline is the unmodified Xen hypervisor 4.1.2.

All benchmarks are migrated using Xen's rate-limiting algorithm in three different configurations: `100/250`, `200/500`, and `unlimited`. As above, the first number represents the starting and the second number the maximal network bandwidth in megabits

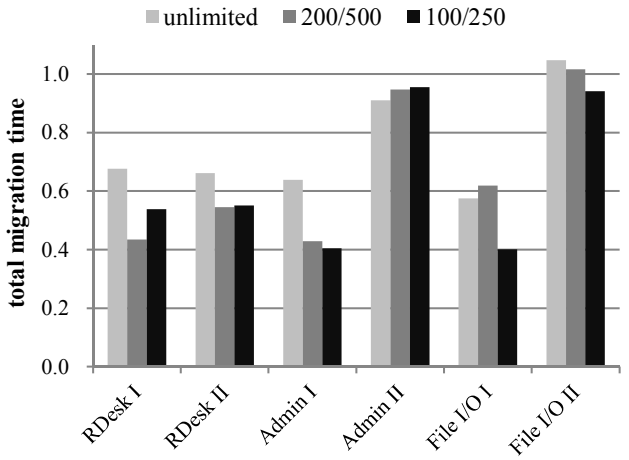

Figure 6. Normalized total migration time.

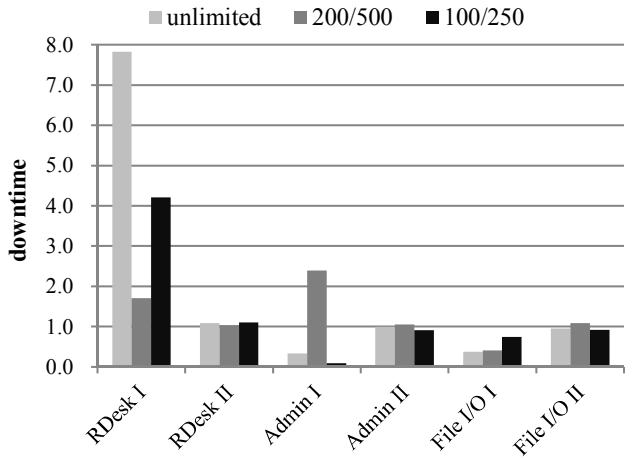

Figure 7. Normalized downtime.

per second, `unlimited` is allowed to use all of the available bandwidth. The rate-limiting algorithm is used as implemented in Xen. We expect that the proposed technique performs better in configurations with stricter rate-limiting, especially for benchmarks with lots of duplication, because the connection to the NAS device is always allowed to run at full speed.

5.4 Performance of Efficient Live Migration

Table 2 compares the performance of unmodified Xen with the proposed solution for the six application scenarios for three different network configurations. The first column lists the application scenario. Column two, denoted `dup`, shows the percentage of memory that is duplicated on disk when the live migration is started. The following columns show the performance for the three different network configurations, `unlimited`, `200/500`, and `100/250` for both `unmodified` Xen and the proposed solution (denoted `optimized`). For each network configuration, the total migration time (`mig`), the downtime (`down`), and the time to complete the benchmark inside the migrated VM (`bm`) is shown for unmodified Xen and the proposed solution. All time values are given in *seconds*.

Total Migration Time. Figure 6 shows the total migration time for the six application scenarios listed in Table 1. All results are normalized to unmodified Xen. Except for `File I/O II`, the PostMark benchmark, the proposed solution can improve the total migration time significantly. We observe that the proposed solution performs better if adaptive rate limiting (`200/500` and `100/250`) is used. This is not surprising; in the `unlimited` scenario, the source host can saturate the network connection to the destination host; parallel fetches from external storage thus have a smaller effect than in a constrained environment. In the case of `File I/O II` and sufficient memory bandwidth, the VM still running on the source host, the migration process, and the background fetch process are all competing for network resources which leads to a 5 and 2% increase of the total migration time for `unlimited` and `200/500`. On average, the proposed technique reduces the total migration time by 25, 34, and 37% for the different network configurations compared to unmodified Xen.

Downtime. Figure 7 displays the downtime of the application scenarios normalized to unmodified Xen. In most cases, the total migration time is similar to or even below unmodified Xen. `RDesk I` suffers from a severely increased downtime (up to 8-fold in the `unlimited` case). This scenario represents the worst-case for the proposed technique: an idle VM with lots of duplicated data. The synchronization with the background fetch process before the VM

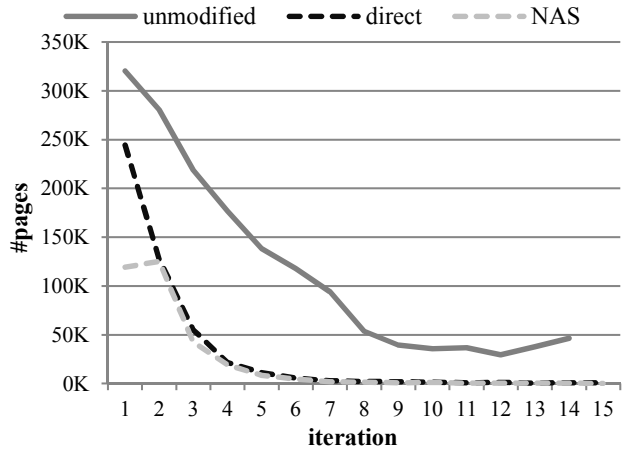

Figure 8. Pages per iteration for `File I/O I`.

is restarted on the target host is responsible for this increase; we have outlined two solutions to remedy this situation in Section 5.2. In many other cases, the downtime is actually *reduced*. This at first seemingly illogical result can be explained as follows: thanks to the reduced total migration time, the benchmarks running inside the VM are making less progress and therefore dirty fewer memory pages. In addition, the iterative pre-copy algorithm terminates the migration early because in each iteration less pages are sent directly to the destination host. This situation is illustrated in Figure 8 for the `File I/O I` scenario. The last iteration, during which the VM is stopped, comprises much fewer pages than in unmodified Xen. This leads to a reduced transfer time and in turn a shorter downtime of the VM.

Benchmark Performance The last important measure is performance degradation of the VM caused by live migration. Unmodified Xen itself causes a slight performance degradation due to the tracking of dirtied memory pages between iterations. A only source of performance degradation in the proposed solution is the additional load put on the NAS storage device. For I/O-intensive benchmarks that read or write a lot of data to the remote disk, the I/O requests generated by the background fetch process may hurt the running VM. Figure 9 shows the results for the application sce-

Table 2. Comparison of unmodified Xen with the proposed solution for the different application scenarios.

application	dup	unlimited						200/500						100/250					
		unmodified			optimized			unmodified			optimized			unmodified			optimized		
		mig	down	bm	mig	down	bm	mig	down	bm	mig	down	bm	mig	down	bm	mig	down	bm
RDesk I	45	61.0	1.02	-	41.2	7.95	-	94.6	0.59	-	51.0	2.50	-	175.8	0.57	-	76.3	0.97	-
RDesk II	46	53.2	1.04	-	35.2	1.13	-	83.3	1.05	-	45.9	1.16	-	165.4	1.01	-	90.2	1.05	-
Admin I	18	82.5	6.96	224	52.6	2.33	229	171.4	3.13	276	72.6	0.33	266	224.1	1.12	296	96.5	2.43	276
Admin II	10	62.4	5.37	215	56.8	5.31	237	88.8	7.13	217	84.8	6.47	234	155.7	8.57	229	147.5	9.06	242
File I/O I	24	92.6	19.49	222	53.2	7.31	226	176.6	4.52	266	69.3	1.02	223	127.7	22.81	249	79.1	9.41	205
File I/O II	12	51.8	8.49	240	54.3	8.09	262	85.0	7.35	252	79.5	7.75	270	147.1	9.04	274	149.5	9.81	290

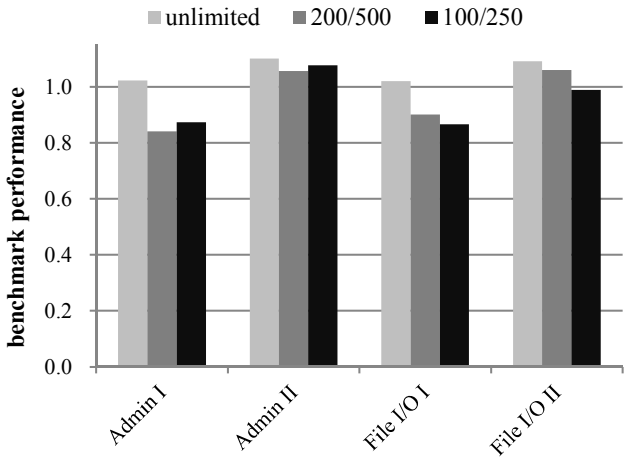

Figure 9. Normalized benchmark performance.

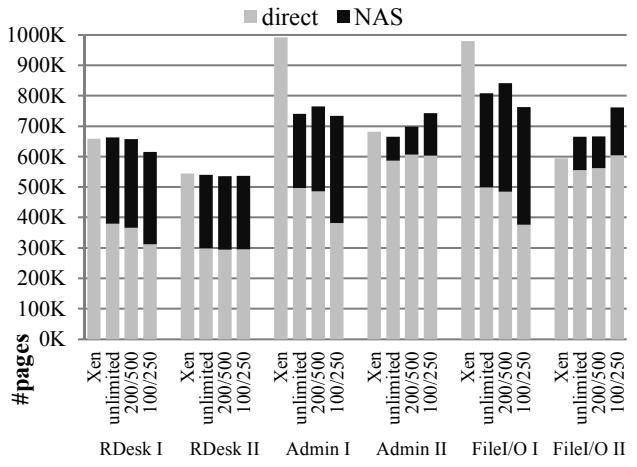

Figure 10. Number of pages transferred.

narios that run I/O-intensive benchmark with well-defined start and end points. We observe the expected behavior for `unlimited` in all benchmarks, and in the case of `Admin II` and `File I/O II` for rate-limited migrations. However, in the majority of cases the benchmark performance actually *improves*. The reason for this result is again the reduced total migration time: during migration, I/O-intensive benchmarks see a performance hit caused by the high network I/O activity of the migration and the overhead of tracking dirtied pages. Reducing the total migration time thus shortens the period during which the benchmark suffers from reduced performance which has a positive effect on the total benchmark time. On average, the benchmark time increased by 6% for `unmodified`, and is reduced by 4, resp. 5% in the case of `200/500` and `100/250`.

Pages transferred. Figure 10 shows the amount of pages transferred during the entire live migration. For each application scenario, unmodified Xen as well as the three network configurations are shown. In most cases, the total number of pages received by the target host is similar to unmodified Xen. `RDesk I and II` show the expected ratio of direct vs. pages fetched from NAS as listed in column two of Table 2. `Admin II` and `File I/O II`, on the other hand, show in increase in the total number of pages transferred. This is caused by two factors: first, the amount of duplication is low (10 and 12%, respectively) and both scenarios dirty many pages and generate a lot of I/O requests. Consequently, these two scenarios also perform worst when it comes to the total migration time (see Figure 6).

5.5 Limitations and Future Work

Overall, the proposed technique shows very promising results. The total migration time can be significantly reduced in almost all usage scenarios. The main weakness are idle VMs with a large amount of duplication which suffer from a significantly increased downtime. One might argue that an increased downtime does not matter that much if the VM is idle; nevertheless, we are currently working on eliminating this problem by the approaches outlined in Section 5.2.

The proposed technique cannot be applied to live migration between hosts that do not share storage. Furthermore, there are a number of implementation issues that may prevent the use of this technique: first, we assume that the attached storage is mounted in `dom0` and accessed as a file-backed disk in the VM. If the VM itself mounts the network-attached storage, then the current method of transparent I/O interception does not work anymore. A similar problem occurs with the newest PVH drivers where I/O requests cannot be intercepted transparently anymore. A possible solution to integrate the proposed technique in HVM guests using PVH drivers or VMs that mount the remote storage directly is providing a driver which communicates with `xentools` during live migration to inform the tools about the contents of the page cache. This is, in fact, the way we are implementing the proposed technique for PV guests. We are currently also working on a solution for PVH drivers.

Last but not least, the NAS fetch queue is required to synchronize with the receiver process before the VM can be restarted on the target machine. We plan to implement a post-copy approach in which the NAS fetch queue does not need to be empty before the VM can be restarted. Instead, all outstanding memory pages are marked *disabled*. The NAS fetch queue continues to fetch the pages in the background. If the VM traps on a yet unavailable page, that page is then fetched immediately via the page fault handler before the VM is restarted.

6. Conclusion

We have presented technique for efficient live migration of virtual machines. The key idea is that it is not necessary to send data of memory pages that are also duplicated on the attached storage

device over the (often slow) network link between the distinct physical hosts; instead that data can be directly fetched from the NAS device.

To detect duplication between memory pages and storage blocks, we transparently track all I/O operation to the attached storage and maintain an up-to-date map of duplicated pages. When migrating a VM across distinct physical hosts, instead of sending the data of all dirty memory pages to the target host, only the data of ordinary (i.e., not duplicated) memory pages is sent. For pages that also exist on the attached storage device, the mapping memory pages to disk blocks is sent to the target host from where the data is fetched by a background process. We show that consistency can be guaranteed by keeping a version number for each transferred page. We have implemented and evaluated the proposed technique in the Xen hypervisor 4.1. For a number of benchmarks run on a Linux HVM guest we achieve an average reduction of the total migration time of over 30%; for certain benchmarks we observe a reduction of up to 60%.

While the implemented technique is effective in many cases, it still leaves room for improvement. For future work, we plan to implement a lazy-fetch algorithm on the target host. The VM can then be restarted even sooner and before all blocks have been fetched from the attached storage device. Minor improvements for the current implementation such as forwarding a list of skipped pages to the target host as well as support for para-virtualized guests are underway.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[3] Fabrice Bellard. QEMU. `http://www.qemu.org`, 2013. Online; accessed February 2013.

[4] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 26–35, New York, NY, USA, 2008. ACM.

[5] Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen, and Mahadev Satyanarayanan. Jettison: efficient idle desktop consolidation with partial vm migration. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 211–224, New York, NY, USA, 2012. ACM.

[6] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 169–179, New York, NY, USA, 2007. ACM.

[7] Canonical Ltd. Ubuntu. `http://www.ubuntu.com`, 2013. Online; accessed February 2013.

[8] Citrix Systems, Inc. Xen Hypervisor. `http://www.xen.org/products/xenhyp.html`, 2012. Online; accessed February 2013.

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

[10] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[11] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 135–146, New York, NY, USA, 2011. ACM.

[12] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008(166), February 2008.

[13] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[14] Takahiro Hirofuchi, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. Reactive consolidation of virtual machines enabled by postcopy live migration. In *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, VTDC '11, pages 11–18, New York, NY, USA, 2011. ACM.

[15] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 11–20, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[17] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report Technical Report TR3022, Network Appliance, October 1997.

[18] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 101–110, New York, NY, USA, 2009. ACM.

[19] Raymond A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2(1):91–104, March 1977.

[20] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[21] Oracle. VirtualBox. `https://www.virtualbox.org`, 2012. Online; accessed February 2013.

[22] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 75–86, New York, NY, USA, 2011. ACM.

[23] Tristan Richardson. The RFB protocol. `http://www.realvnc.com/docs/rfbproto.pdf`, 2010. Online; accessed February 2013.

[24] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[25] The Document Foundation. LibreOffice. `http://www.libreoffice.org`, 2013. Online; accessed February 2013.

[26] The Mozilla Foundation. Firefox. `http://www.mozilla.org`, 2013. Online; accessed February 2013.

[27] Franco Travostino. Seamless live migration of virtual machines over the MAN/WAN. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[28] VMware. VMware VMotion: Live migration of virtual machines without service interruption. `http://www.vmware.com/files/pdf/VMware-VMotion-DS-EN.pdf`, 2009. Online; accessed February 2013.