Optimizing Live Migration for Virtual Desktop Clouds

Changyeon Jo and Bernhard Egger
School of Computer Science and Engineering
Seoul National University, Seoul, Korea
{changyeon, bernhard}@csap.snu.ac.kr

Abstract—**Live migration of virtual machines (VM) from one physical host to another is a key enabler for virtual desktop clouds (VDC). The prevalent algorithm, pre-copy, suffers from long migration times and a high data transfer volume for non-idle VMs which hinders effective use of live migration in VDC environments. In this paper, we present an optimization to the pre-copy method which is able to cut the total migration time in half. The key idea is to load memory pages duplicated on non-volatile storage directly and in parallel from the attached storage device. To keep the downtime short, outstanding data is fetched by a background process after the VM has been restarted on the target host. The proposed method has been implemented in the Xen hypervisor. A thorough performance analysis of the technique demonstrates that the proposed method significantly improves the performance of live migration: the total migration time is reduced up to 90% for certain benchmarks and by 50% on average at an equal or shorter downtime of the migrated VM with no or only minimal side-effects on co-located VMs.**

I. Introduction

The widening availability of high-performance networks currently drives a shift from running services on privately owned and managed hardware to co-locating these services in cloud data centers [1]. In addition to the comprehensive availability of fast networks, the key technology that enables this shift is *virtualization*. The benefits of virtualization are manifold. Virtualization enables the customers to tailor (and pay for) an environment that fits their exact needs. For enterprises, virtualization enables consolidation of servers to reduce hardware cost, simplifies server management, and empowers the deployment of massive virtual desktop infrastructures (VDI) [2] and virtual desktop clouds (VDC) [3].

An important feature of virtualization is *live migration* [4]. Live migration is attractive to cloud 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. Live migration thus has been and is an actively researched topic [4], [5], [6], [7], [8], [9], [10], [11]. Live migration is only useful if the service provided by the running VM is not interrupted, i.e., if it is transparent to the end-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 comprises 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 typical cloud data center setups the permanent storage is provided through network-attached storage (SAN or 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.

Many efforts have been undertaken to shorten the *downtime* of the VM, i.e., the time during which a VM being migrated is not running. The prevalent technique is *pre-copy* [4] that achieves a relatively short downtime at the expense of a prolonged total migration time and an increased total amount of pages transferred. Pre-copy works especially well if the VM's writable working set (WWS) is small. However, even for moderately write-intensive workloads, pre-copy's performance severely degrades [10] which adversely affects effective utilization of live migration in VDC environments.

In this paper, we present an optimization to pre-copy which is able to reduce the total migration time by 50% at an equal or reduced downtime over a wide range of benchmarks and independent of the workload of co-located VMs. Based on the observation that data duplication between memory pages and disk blocks is common for modern operating systems [12], the proposed method fetches duplicated data directly from the attached storage and only sends memory pages whose content is not available elsewhere directly to the target host. By parallelizing the operation of the two data sources, the total migration time is reduced significantly. The proposed method additionally reduces the bandwidth requirements on the internal network infrastructure because less data needs to be sent from the source to the target host. A similar approach [13] suffered from a greatly increased downtime; our technique eliminates that shortcoming by lazily fetching outstanding memory pages after the VM has been restarted on the target host. The presented method is independent of almost all other optimization techniques such as iterative pre-copying, post-copying, and data compression. Existing techniques can thus be augmented by integrating the approach presented here to further reduce the total migration time.

The remainder of this paper is organized as follows: Section II gives a brief overview of live migration. In Section III, the proposed technique is explained in detail. Section IV discusses the experimental setup and the benchmarks, and Section V evaluates the technique. Section VI discusses related work, and Section VII, finally, concludes the paper.

II. Live Migration of Virtual Machines

Live migrating a VM amounts to moving its entire state from one physical host to another. In a data-center setup, the permanent storage is accessed via NAS and does not need to be moved. The volatile state comprises the state of the VM's VCPU(s) and devices plus the contents of the memory.

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant Number 2012R1A1A1042938). ICT as Seoul National University provided research facilities for this study.

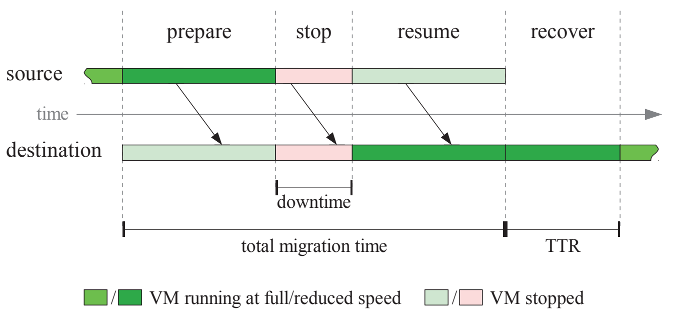


Fig. 1. Phases and metrics in live migration.

The latter constitute by far the largest chunk of data to be transferred and thus offer the highest potential for optimization.

A. Live migration phases

The simplest way of migrating a VM is *stop-and-copy*, i.e., stopping the VM on the source, transferring the entire state to the destination, and then resuming the VM. The long service interruption renders the stop-and-copy approach impractical for VDC environments. It is thus common practice to migrate the state of a VM *while it is running*. The live-migration process can then be modeled with the following four distinct phases as shown in Figure 1:

1) **Preparation**. Live migration is initiated. On the source host, the hypervisor often puts the VM into a managed mode which typically results in reduced performance of the VM. On the destination host, resources are allocated for the VM. The source host may start sending (parts of) the VM's volatile state to the destination host.
2) **Stop**. The VM is stopped both on the source and the destination host and thus not available to the user. The VCPU and device states are preferably transferred during this phase.
3) **Resume**. The VM is restarted on the destination host. If the volatile state of the VM has not yet been transferred in its entirety, the remaining parts are now fetched by the destination host. The VM's performance is still reduced in this phase, and the resources of the VM must still be maintained on the source host.
4) **Recovery**. All volatile state from the source host has been transferred and the resources of the VM can be freed on the source. However, some part of the VM's state may still need to be recovered (typically from the attached NAS).

Not all phases are present in all migration techniques. The stop-and-copy technique, for example, comprises a long stop phase with very short preparation and resume phases. Additionally—and in contrast to the proposed technique—most live migration methods do not have a recovery phase.

B. Live migration metrics

To compare different live migration algorithms, we use the following metrics as indicated in Figure 1:

1) The **downtime** denotes the time interval during which the VM is completely stopped.
2) The **total migration time** covers the time period between initiation of the migration until the resources of the VM can be released on the source host.
3) The **time-to-responsiveness (TTR)** represents the time span after the resume phase has ended until the VM achieves a certain guaranteed minimal utilization (see below).
4) The **amount of transferred data** (not shown in the figure) measures the amount of data received at the destination host from the different sources.

The downtime is undoubtedly the most important metric. It has to be sufficiently small in order for live migration to be applicable since the service of the VM is completely interrupted during this time span. All existing live migration techniques trade a shorter downtime for a prolonged total migration time and an increased amount of transferred data. This is not unproblematic for two reasons: first, the resources of the migrated VM remain allocated on the source host for a longer time. If VMs are migrated away from a host due to, for example, a low amount of available physical memory or because the host needs to perform a quick shutdown, a prolonged total migration time is counterproductive. Secondly, a longer total migration time and an increased amount of transferred data results in higher resource utilization which in turn leads to increased operational cost of a data center.

The *time-to-responsiveness* and the related *minimum mutator utilization (MMU)* are used to pinpoint the point in time after which a VM *appears* to be migrated. The two metrics are typically used to quantify the performance degradation of an application—the mutator—caused by garbage collection. The TTR and MMU metric have recently been applied to VM snapshot restoration [14]. The TTR denotes the time after which a VM is guaranteed to always achieve a certain minimum performance. The $MMU(t, w)$ measures the minimum utilization achieved by the VM at time t during a time window w. The relationship between TTR and MMU is given by

$$TTR(w, u) = min\{t | MMU_{\forall t' \geq t}(t', w) \geq u\}$$

In other words, $TTR(w, u)$ is the earliest time t such that from that point forward the VM always achieves the desired minimum level of utilization u during a time window w. w and u are application-specific parameters: if a VM can tolerate a maximum performance degradation of 20% over a 1 second window, then the $TTR(1.0, 0.8)$ denotes the point in time after which the VM will always achieve at least an 80% utilization in any one-second window and thus appears to be completely migrated – even if some parts of the volatile state have not been restored yet.

III. Efficiently migrating live VMs

The prevalent method to live-migrate VMs in open-source and commercial products [15], [8], [16], [17] is pre-copy [4]. The pre-copy algorithm first sends the entire contents of the VM's volatile storage to the destination host. Then, it iteratively transfers the set of dirtied memory pages until the number of dirtied pages falls below a certain threshold or the pre-defined maximum number of iterations (MAX_ITER) has been reached. This task corresponds to the `prepare` phase in Figure 1. The VM is then stopped and the state of the VCPUs, the devices, and the remaining dirtied memory pages are sent to the destination host (`stop` phase). After the entire state of the VM has been restored, the VM is restarted on the destination host. The `resume` phase merely consists of signalling the source host that the migration has successfully completed. There is no `recover` phase.

The pre-copy algorithm performs very well if the writable working set (WWS) of the migrated VM is reasonably small. If, however, the WWS is large, the number of dirtied pages

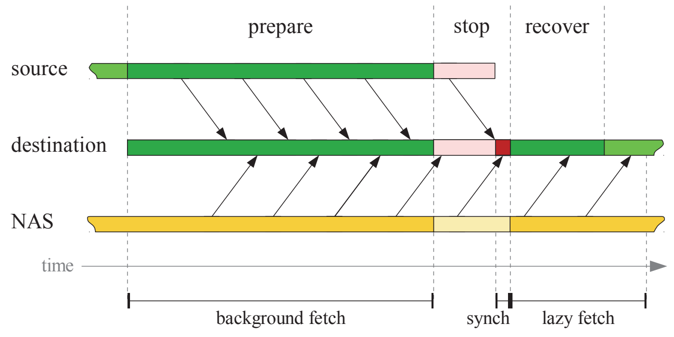


Fig. 2. Fetching duplicated data directly from network-attached storage.

between iterations does not converge and the algorithm will iterate MAX_ITER times. This has two unwanted consequences: first, the total amount of data sent from the source to the destination host may reach a multiple of the VM's volatile storage. Second, the downtime is directly proportional to the amount of dirtied pages send during the `stop` phase. VMs with a large WWS may thus suffer from unacceptabe total migration and downtimes: Song [18] reports a total migration time of 27 minutes with a downtime of over 250 seconds for a VM with 16 GB of memory running a memcached server.

Almost all proposed optimizations to pre-copy use some form of compression to reduce the amount of data sent in each iteration [11], [19], [20]. Other methods exclude free memory pages before migration. The amount of free memory pages can be increased significantly if the page cache of the guest operating system (OS) is dropped first. These methods, however, require some form of cooperation by the guest OS and are thus only applicable to para-virtualized guests or fully-virtualized VMs running a tailored guest OS driver [10].

A. Parallelizing live migration

In contrast to existing methods, the presented technique proposes to optimize live migration by parallelizing the transmission of a VM's volatile state. Based on the observation that modern OSes hide the long-latency accesses to external storage by caching recently accessed blocks in the so-called *page cache*, our technique fetches non-volatile data duplicated in the page cache directly from the attached NAS device. Memory pages whose contents are not duplicated on the NAS are sent directly from the source to the destination host. With page caches occupying as much as 90% of the available memory [12], the amount of data sent directly from the source to the destination can be reduced significantly. Tracking which pages are duplicated on external storage is easy for para-virtualized guests; for fully-virtualized guests, Park [12] has recently demonstrated that transparently tracking duplicated pages is possible without a measurable performance degradation.

In the following, we describe our implementation as a proof-of-concept for fully-virtualized VMs. An implementation for para-virtualized is also possible and simpler because the guest OS can be modified to provide the required information to the hypervisor when needed.

The proposed technique operates as follows:

1) For each VM, the hypervisor transparently maintains an up-to-date mapping of memory pages whose contents are duplicated on the NAS device.
2) In each iteration of the pre-copy algorithm, only dirtied memory pages containing data not duplicated on external storage are sent directly to the destination host. For dirtied pages that are dupliated on the NAS device, only the mapping between the memory page and the corresponding storage block is transmitted.
3) The receiver on the destination host stores data of directly sent memory pages into the (still inactive) VM.
4) A *background fetch queue* on the destination host loads data of duplicated memory pages directly from the NAS device into the VM's memory.
5) After the iterative part of the pre-copy algorithm has completed and the remaining volatile state of the VM has been transmitted, the migration process on the destination host waits for the background fetch queue to finish the current I/O operation and then restarts the VM. Any outstanding memory pages that have not yet been loaded from external storage are mapped *invalid* in the VM's memory space; an access to such a page will trigger a page fault exception.
6) The background fetch queue starts operating as a *lazy fetch queue* while the VM is running: it retrieves yet unloaded data from the NAS device into the VM's memory and enables the corresponding memory pages.
7) Page faults triggered by accesses to yet unloaded page are forwarded to the lazy fetch queue. The lazy fetch queue handles such accesses with priority since the execution of the VM is blocked until the corresponding page has been loaded.

Figure 2 illustrates the idea. The `recover` phase terminates as soon as the target TTR for the given benchmark has been reached; this is, in general, before the lazy fetch queue has loaded all duplicated pages from the attached storage device.

B. Implementation

The proposed technique is implemented as a modification to the Xen hypervisor [21]. The code for transparent I/O tracking has been adopted from Park's work [12]. The background fetch queue, along with several optimizations to achieve better performance, is implemented as outlined by Jo [13].

During the `prepare` phase, the VM's storage on the NAS device is accessed in parallel from both the VM running on the source (read-write access) and the background fetch queue on destination host (read-only access). Depending on what level of write-caching is employed, this may lead to old data being read by the background process. Furthermore, it is possible that RAW (read-after-write) hazards occur if a block in the background fetch queue is overwritten by the running VM. To prevent the former problem, the memory pages to be written to external storage are included in the set of dirtied pages and treated as regular pages (i.e., the data is sent directly from the source to the destination). The RAW hazard can be similarly avoided by including the data of the page to be overwritten in the set of pages sent directly.

Until the iterative algorithm of pre-copy stops, the background fetch queue coalesces up to 4 MB of data per I/O request. In lazy fetch mode, such a large coalescing window leads to long I/O delays which aversely affects the TTR. The queue thus coalesces much less data (up to 64 pages per request) in lazy fetch mode. The switch from background fetch to lazy fetch requires a short synchronization phase with the main migration thread on the destination host. This synchronization

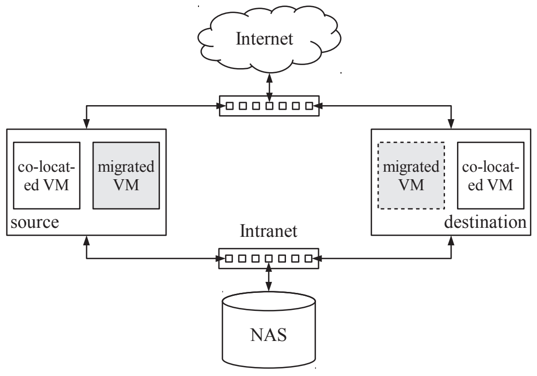


Fig. 3. Experimental setup.

directly affects the downtime of the migration and should thus be as short as possible. In the current implementation, we opt to send all dirtied pages directly from the source to the destination even if some of those pages are known to be duplicated on disk. In addition, the background fetch queue stops processing requests as soon as the last iteration of the pre-copy algorithm has started. Nevertheless, we observe significant synchronization delays when the load on the NAS device is high (this can be caused by I/O-intensive co-located VMs, for example). Implementing a better adaptive control algorithm for the transition from background to lazy fetch queue is part of future work.

Finally, experiments have revealed that page faults caused by instruction fetches to yet unloaded pages have a particularly bad effect on the TTR of a benchmark. In order to minimize such page faults, we thus exclude code pages that are duplicated on external storage and get accessed while the live migration is running. To do so, all known duplicated code pages have the executable flag removed on the source host when live migration is initiated. The page fault handler invoked by instruction fetches to such pages then adds the accessed page to the set of pages to be transmitted directly.

IV. Experimental Setup

All experiments are performed on two identical hosts equipped with an Intel Core i5-2500 processor and 16 GB of main memory. For external storage a QNAP NAS server is used. The network topology comprises two physically separated networks. A gigabit intranet connects the two hosts and the NAS server. Data sent from the source to the destination host as well as data fetched from the NAS by the destination thus flow through the same network. We have measured a peak transfer rate of the original pre-copy algorithm of about 500 MBit/s; i.e., the intranet is only 50% saturated by the migration process (migration data is sent over an SSL-encrypted connection; the encryption/decryption is the bottleneck). The two hosts are connected to the outside world, the Internet, through a physically separate gigabit network. Network performance of co-located VMs, whenever applicable, is measured by a computer connected to the Internet. Figure 3 illustrates the setup.

The hosts run Xen hypervisor 4.1.2 [21] with our modifications. Dom0 runs Ubuntu Server 11.10 (kernel version 3.0.0) [22]. All experiments are performed on HVM guests running Ubuntu Desktop 10.04 LTS with a 2.6.35 kernel. The VMs are configured with two virtual CPUs and 2 GB of RAM. The guests' virtual disk is a file-backed image located on external storage mounted by dom0 using NFS.

TABLE I. Application Scenarios

Application	Description
Desktop	web browsing, editing in an office suite
Movie	playing a movie
Gzip	compressing a large file
Make	compiling the Linux kernel
Copy	backing up data
Postmark	Postmark benchmark

A. 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 [13], [23] and that are representative for a Virtual Desktop Infrastructure (VDI). For all application scenarios the user is connected to the migrated VM by a VNC viewer [24].

Table I lists the application scenarios. `Desktop` and `Movie` represent the standard usage scenario in VDI: a remote user connected to his virtual desktop. In `Desktop` a Firefox [25] web browser is fetching a number of web pages and some scripted editing tasks are performed in the LibreOffice suite [26]. `Movie` plays a movie. Both benchmarks exhibit moderate CPU and I/O activity and are expected to perform well both in unmodified Xen and our solution. `Gzip` and `Make` represent administration tasks. In `Gzip` a 2 GB log file is compressed using gzip. `Make` 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 the proposed method to perform significantly better due to the high activity in the page cache and the rate of dirtied pages. The last group, `Copy` and `Postmark`, represents I/O-intensive benchmarks. In `Copy` a backup process is backing up a large (2 GB) file. `Postmark` runs PostMark [27] (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. The amount of duplication between memory and external storage is roughly 50% for all application scenarios.

B. Network and Co-location Configuration

To evaluate the effect of the network speed as well as co-located VMs, the six VMs running one of the aforementioned application scenarios are migrated at two different network configurations and three different co-location setups. For the network configuration, we denote the maximum throughput for the direct link between source and destination host DIR, the throughput for the background fetch queue BGQ, and the throughput for the lazy fetch queue LFQ. A network configuration is then defined by a triple $DIR/BGQ/LFQ$. We run the benchmarks at the configurations $\infty/500/\infty$ and $250/125/\infty$ (unit: MBit/s).

Each of the application scenarios at each network configuration is then run in one of three co-location setups: no co-location, i.e., the migrated VM is the only guest on the hosts, denoted `standalone`, co-location with two PV guests running network throughput benchmarks, termed `netperf`, and finally co-location with two PV guests running PostMark benchmarks, termed `postmark`. Since we are leveraging parallel access to the attached NAS device, the overloaded NAS device of co-location with `postmark` represents the worst-case scenario for the proposed technique.

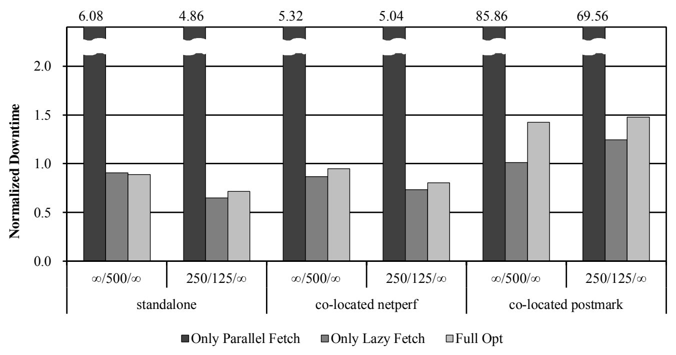


Fig. 4. Normalized downtime for standalone and co-located VMs at two different network configurations.

C. Metrics

For each configuration, we measure the downtime, the total migration time, the TTR, and the total number of transferred pages. In the co-location setups, the I/O throughput of co-located VMs is measured as well. The baseline is the unmodified Xen hypervisor version 4.1.2. We compare the proposed method against a similar approach by Jo [13] which does not support lazy fetch and thus requires the background fetch queue to be processed completely before restarting the VM. This data point is denoted `Only Parallel Fetch`. We expect this case to suffer from an increased downtime. A second data point is obtained by disabling the background fetch queue and load all duplicated pages through the lazy fetch queue. This case, denoted `Only Lazy Fetch`, is expected to provide good downtimes, but suffer from an increased TTR. The proposed technique, finally, as detailed in Section III with a background fetch queue and a lazy fetch queue, is denoted `Full Opt`.

In accordance with Zhang [14] we compute the TTR with a 1 second window and 80% utilization for the interactive benchmarks `Desktop` and `Movie`. For non-interactive benchmarks, we require at least a 50% utilization over a 10 second window, i.e., $TTR(10.0, 0.5)$. The data for each configuration triple $\{$application scenario/network configuration/co-location configuration$\}$ is the average of at least five single runs.

V. Evaluation

A. Downtime

Figure 4 displays the normalized downtime for each of the six network and co-location configurations, averaged over all six application scenarios.The configuration `Only Parallel Fetch` suffers from severely prolonged downtimes caused by the synchronization with the background fetch queue before the VM can be restarted. The effect becomes accentuated if the NAS device is overloaded as shown in the `co-located postmark` setup. Both `Only Lazy Fetch` and `Full Opt` achieve shorter downtimes than unmodified Xen as long as there is no congestion on the NAS. `Only Lazy Fetch` performs slightly better than `Full Opt` because in the current implementation of `Full Opt` because there is a synchronization barrier before the lazy fetch queue is activated.

B. Total migration time including TTR

The goal of the proposed method is to reduce the total time required for migration (`total migration time`) plus the time to achieve responsiveness in the migrated VM (`TTR`). Unmodified Xen and `Only Parallel Fetch` have TTR values equal to 0 since the VM is immediately 100% available as soon as it is restarted on the destination host. A TTR value greater than zero is caused by frequent page faults due to accesses to pages that have not yet been fetched by the lazy fetch queue.

TABLE II. Number of Pages Transferred

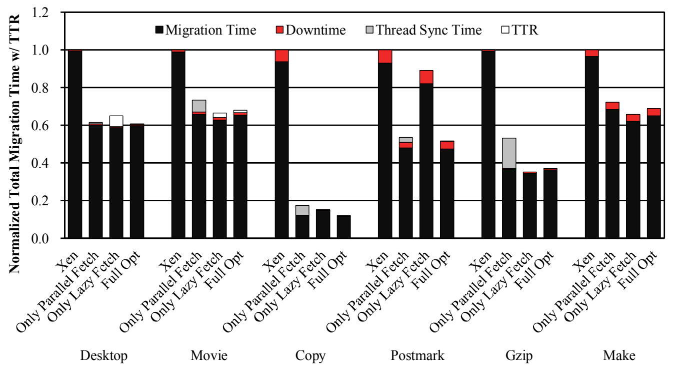
	Only Parallel Fetch	Only Lazy Fetch	Full Opt
Total	74%	80%	74%
From Sender	43%	51%	43%

Figure 5 shows the detailed results for each network and co-location configuration. The bars show (from bottom to top) the `migration time` (excluding the downtime), the `downtime`, the synchronization time with the background queue (denoted `Thread Sync Time`), and the time required to achieve the desired MMU, denoted `TTR`. For standalone and co-located with `netperf`, all three optimizations clearly outperform unmodified Xen. This confirms the potential of parallelizing the data transfer on the receiver side. `Only Parallel Fetch` performs on par with `Only Lazy Fetch` and `Full Opt` if the background fetch queue is able to load all duplicated data from the NAS device before the pre-copy algorithm terminates. If this is not the case, the thread synchronization time clearly affects this method. `Gzip` in the configuration `standalone at` $\infty/500/\infty$ represents such a case. Comparing `Only Lazy Fetch` with `Full Opt` we observe that `Only Lazy Fetch` suffers from increased TTR delays because the lazy fetch queue can only start loading pages after the VM has been restarted. `Desktop` in the configurations (a) to (d) in Figure 5 clearly shows increased TTR values for `Only Lazy Fetch`. The worst-case scenario for all three optimizations is co-location with I/O-intensive VMs. In this case, the NAS becomes the bottleneck which negatively affects the throughput of the background fetch queue and the lazy fetch queue. `Only Lazy Fetch` and `Full Opts` require no or only minimal synchronization and thus do not show a large value for `Thread Sync Time`. Since the latency of I/O requests is severely degraded, both methods suffer from long TTR values. The problem is less severe if the network bandwidth of the direct link is restricted because this results in a prolonged migration time and thus gives the background fetch queue more time to load data from the NAS.

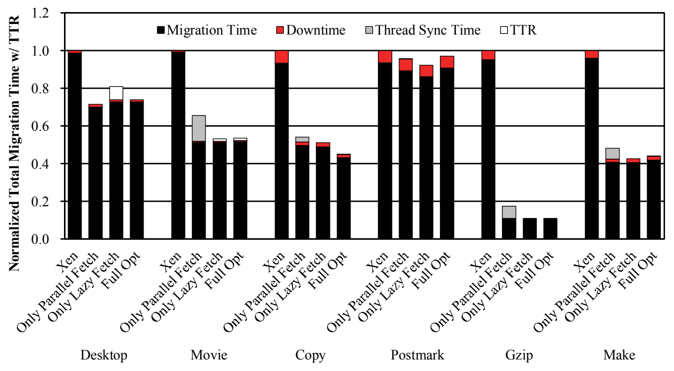
Overall, the proposed technique outperforms unmodified Xen and Jo's method [13] by 50% and 63%, on averagae.

C. Amount of transferred data and TTR

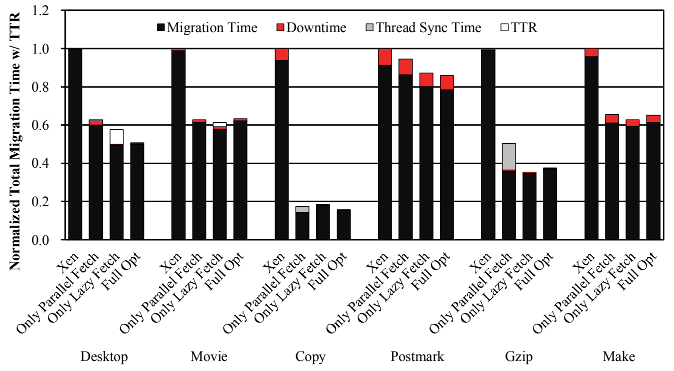
Figure 6 (a) breaks down the total number of transferred pages by data source. Pages loaded from the NAS device are further separated into those loaded by the background fetch queue (`From NAS`) and those loaded by the lazy fetch queue (`Lazy Fetch`). On average, all three optimizations achieve a reduction in the total number of pages transferred to the destination host as shown in Table II. `Only Parallel Fetch` and `Full Opt` perform equally well; the synchronization does not affect the number of loaded pages. `Only Lazy Fetch` for `Postmarks` deserves a closer look. It performs worse than the other two optimizations because the benchmark running inside the VM competes for I/O throughput with the background fetch queue. In the case of `Only Parallel Fetch` and `Full Opt` the parallel fetch leads to reduced performance of `Postmark` inside the VM. The number of dirties pages is thus lower and the pre-copy algorithm converges sooner.



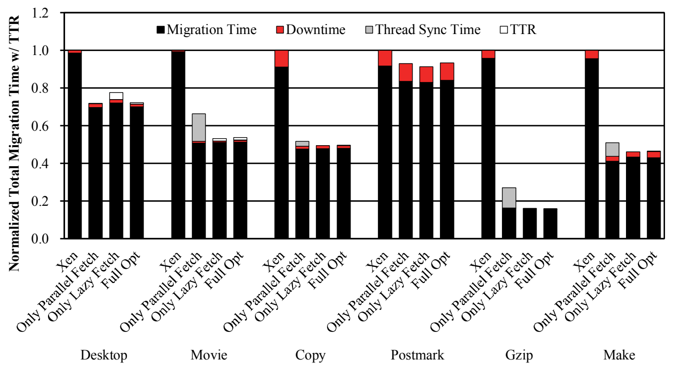
(a) standalone at $\infty/500/\infty$.



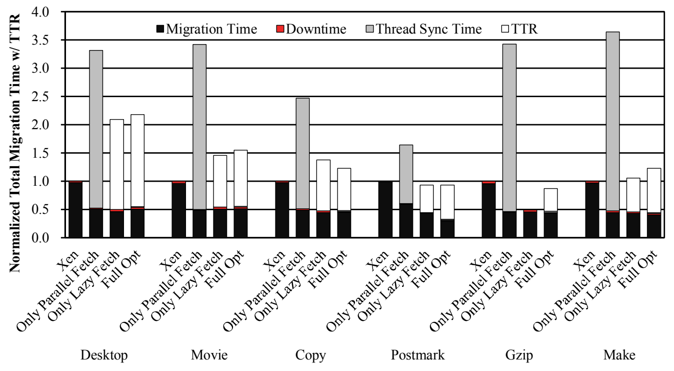
(b) standalone at $250/125/\infty$.



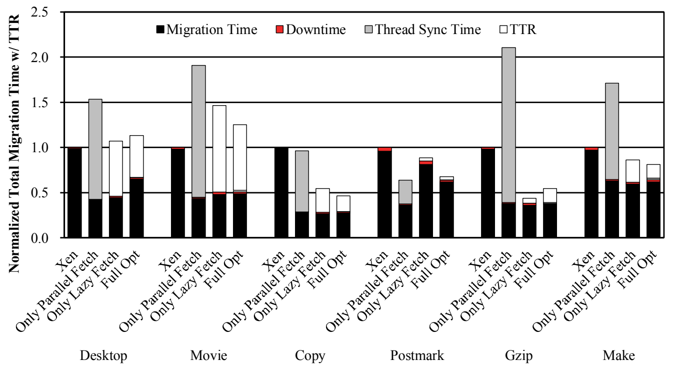
(c) Co-located `netperf` at $\infty/500/\infty$.



(d) Co-located `netperf` at $250/125/\infty$.



(e) Co-located `postmark` at $\infty/500/\infty$.



(f) Co-located `postmark` at $250/125/\infty$.

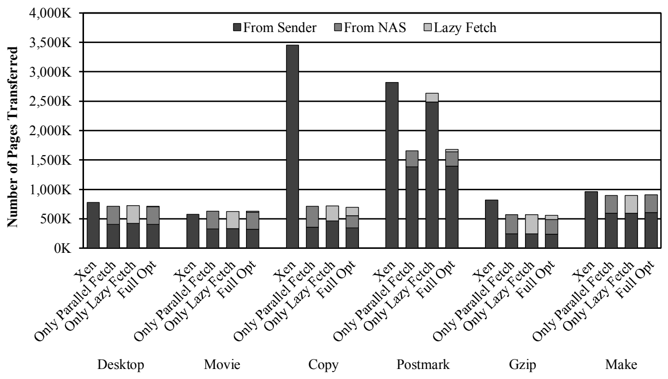
Fig. 5. Normalized total migration time with TTR for standalone and co-located `netperf` and `postmark`.

Figure 6 (b) shows the effect of the background fetch queue on the MMU for a one-second window for `Make` at the configuration standalone, $250/150/\infty$. While the `Only Lazy Fetch` optimization requires over 10 seconds to reach 80% utilization, `Full Opt` achieves this mark after about two seconds.

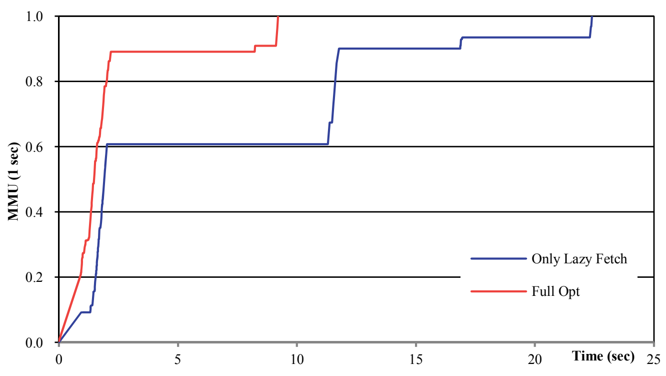
D. Performance of co-located VMs

The side-effects of live-migrating a VM should be minimal for co-located VMs, especially so in VDC environments. Figures 7 and 8 analyze the effect of live migration on the performance of two co-located VMs running `netperf` and `postmark`, respectively. The left-hand graph displays the throughput for unmodified Xen, while the right-hand side displays the traffic graph for the proposed method. While the throughput of the co-located VM on the source host (`*.Send Throughput`) remains largely unaffected by the migration, the throughput of the co-located VM on the destination host is noticeably reduced. Unmodified Xen affects the throughput for the entire duration of the migration (second mark 45 to 280). With the proposed method, the VM co-located on the destination host only achieves about 50% of the nominal throughput in a 10 second window starting at time mark 75. This reduced performance can be attributed to the frequent page faults caused by the VM which has just been restarted on the destination host, i.e., the lazy fetch queue. Note, however, that the entire duration of the migration is reduced to about 60 seconds in total, so that the average throughput of `postmark` is actually higher compared to unmodified Xen.

Figure 8, finally, shows the I/O throughput to the NAS device for the migrated VM and two co-located guests running `postmark`. While unmodified Xen does not affect the throughput in any noticeable way, the background fetch queue and the lazy fetch queue, labeled `Parallel fetch` and `Lazy fetch`, respectively, compete for bandwidth on the NAS. The effect, however, is moderate and only visible in

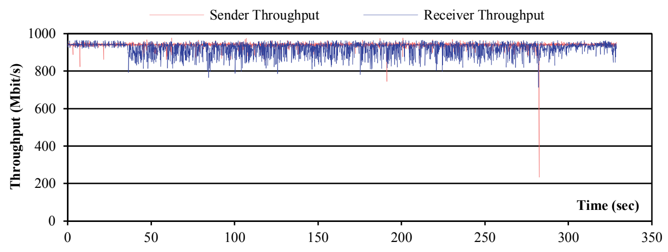


(a) Total number of pages transferred for standalone at $\infty/500/\infty$.

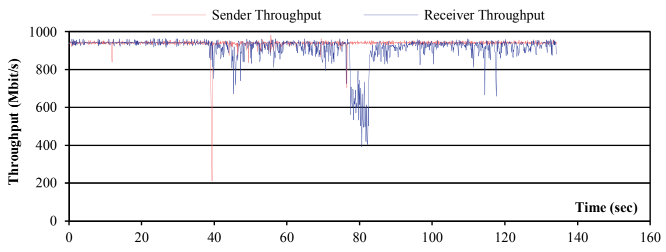


(b) MMU for `make` at $250/125/\infty$.

Fig. 6. Total number of pages transferred and MMU.

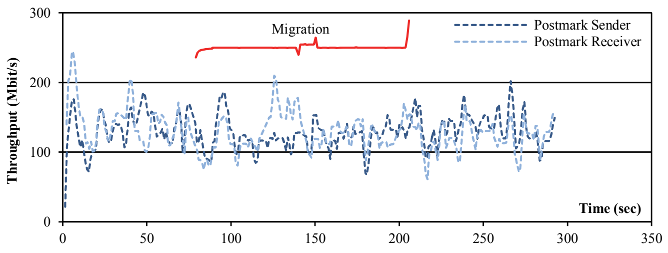


(a) unmodified Xen

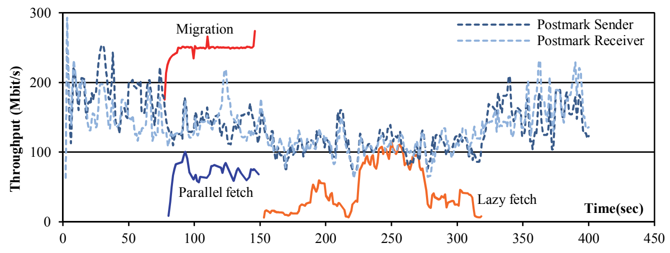


(b) the proposed method

Fig. 7. Network throughput of two co-located VMs while migrating a VM running `copy` at $\infty/500/\infty$.



(a) unmodified Xen



(b) the proposed method

Fig. 8. NAS I/O throughput of two co-located VMs while migrating a VM running `make` at $250/125/\infty$.

the worst-case scenario with three VMs competing for access to the NAS.

VI. Related Work

The predominant approach for live VM migration is pre-copy. The bare-metal hypervisors VMware [16], KVM [8], and Xen [15], plus hosted hypervisors such as VirtualBox [17] employ a pre-copy approach. To reduce the downtime of the VM, the state of the VM is copied in several iterations [4]. While transferring the state of the current iteration, the VM continues to run on the source machine. Pages that are modified during this transfer are marked *dirty* and re-transmitted in the following iteration. 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 [9].

Optimizations of the basic pre-copy approach include live migration based on trace and replay [11], memory compression [19], [20], simultaneous migration of several VMs from one host to another [28], or partial VM migration [29]. Liu *et al.* [11] 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.* [19] 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.* [20] extend this idea by delta-compressing changes in memory pages.

The work most closely related to our technique was presented by Jo *et al.* [13]. The authors also load duplicated memory pages directly from the NAS, however, they do not support a lazy restoration mechanism. This requires them to synchronize with the background fetch queue before the VM can be restarted which severely affects the downtime; especially so when the NAS device is overloaded.

The presented technique is largely independent of related work that are based on data compression or trace/reply and can thus be incorporated into our work to further reduce the total migration time. Also, we have only tackled fully-virtualized guests. Special techniques applicable to partially-virtualized guests such as omitting unused memory pages can also be merged with the presented work.

VII. Conclusion

The goal of the proposed method is to tackle the extended migration time that exist in current VM live migration approaches. The proposed technique parallelizes the transfer of the volatile state of a VM by loading memory pages that contain data duplicated on non-volatile storage directly from the attached storage device. Our evaluation demonstrates that the optimization to pre-copy successfully reduces the total migration time of a VM to about 50% on average at equal downtimes for a wide range of applications, network configurations, and co-located VMs.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[2] K. Miller and M. Pegah, "Virtualization: virtually at the desktop," in *Proceedings of the 35th annual ACM SIGUCCS fall conference*, ser. SIGUCCS'07. New York, NY, USA: ACM, 2007, pp. 255–260.

[3] M. Sridharan, P. Calyam, A. Venkataraman, and A. Berryman, "Defragmentation of resources in virtual desktop clouds for cost-aware utility-optimal allocation," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, dec. 2011, pp. 253 –260.

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

[5] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 25–25.

[6] F. Travostino, "Seamless live migration of virtual machines over the MAN/WAN," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[7] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *Proceedings of the 3rd international conference on Virtual execution environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 169–179.

[8] I. Habib, "Virtualization with KVM," *Linux Journal*, vol. 2008, no. 166, Feb. 2008.

[9] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," in *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, ser. CLUSTER '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 11–20.

[10] M. R. Hines and K. 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*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 51–60.

[11] H. Liu, H. Jin, X. Liao, L. Hu, and C. 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*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 101–110.

[12] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 75–86.

[13] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '13. New York, NY, USA: ACM, 2013, pp. 41–50.

[14] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 87–98.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177.

[16] 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 September 2013.

[17] Oracle, "VirtualBox," https://www.virtualbox.org, 2013, online; accessed September 2013.

[18] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen, "Parallelizing live migration of virtual machines," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '13. New York, NY, USA: ACM, 2013, pp. 85–96.

[19] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-sept. 4 2009, pp. 1 –10.

[20] P. Svärd, B. Hudzia, J. Tordsson, and E. 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*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 111–120.

[21] Citrix Systems, Inc., "Xen Hypervisor," http://www.xen.org/products/xenhyp.html, 2013, online; accessed September 2013.

[22] Canonical Ltd., "Ubuntu," http://www.ubuntu.com, 2013, online; accessed September 2013.

[23] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 161–174.

[24] T. Richardson, "The RFB protocol," http://www.realvnc.com/docs/rfbproto.pdf, 2010, online; accessed September 2013.

[25] The Mozilla Foundation, "Firefox," http://www.mozilla.org, 2013, online; accessed September 2013.

[26] The Document Foundation, "LibreOffice," http://www.libreoffice.org, 2013, online; accessed September 2013.

[27] J. Katcher, "PostMark: A New File System Benchmark," Network Appliance, Tech. Rep. Technical Report TR3022, October 1997.

[28] U. Deshpande, X. Wang, and K. Gopalan, "Live gang migration of virtual machines," in *Proceedings of the 20th international symposium on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 135–146.

[29] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan, "Jettison: efficient idle desktop consolidation with partial vm migration," in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 211–224.