A Hybrid Local Storage Transfer Scheme for Live Migration of I/O Intensive Workloads

Bogdan Nicolae, Franck Cappello

A Hybrid Local Storage Transfer Scheme for Live Migration of I/O Intensive Workloads

Bogdan Nicolae
Joint Laboratory for Petascale Computing
INRIA, France
bogdan.nicolae@inria.fr

Franck Cappello
Joint Laboratory for Petascale Computing
INRIA, France
University of Illinois at Urbana-Champaign, USA
fci@lri.fr

ABSTRACT

Live migration of virtual machines (VMs) is key feature of virtualization that is extensively leveraged in IaaS cloud environments: it is the basic building block of several important features, such as load balancing, pro-active fault tolerance, power management, online maintenance, etc. While most live migration efforts concentrate on how to transfer the memory from source to destination during the migration process, comparatively little attention has been devoted to the transfer of storage. This problem is gaining increasing importance: due to performance reasons, virtual machines that run large-scale, data-intensive applications tend to rely on local storage, which poses a difficult challenge on live migration: it needs to handle storage transfer in addition to memory transfer. This paper proposes a memory-migration independent approach that addresses this challenge. It relies on a hybrid active push / prioritized prefetch strategy, which makes it highly resilient to rapid changes of disk state exhibited by I/O intensive workloads. At the same time, it is minimally intrusive in order to ensure a maximum of portability with a wide range of hypervisors. Large scale experiments that involve multiple simultaneous migrations of both synthetic benchmarks and a real scientific application show improvements of up to 10x faster migration time, 10x less bandwidth consumption and 8x less performance degradation over state-of-art.

Categories and Subject Descriptors

D.4.2 [**OPERATING SYSTEMS**]: Storage Management

General Terms

Design, Performance, Experimentation

Keywords

live migration; block migration; local storage transfer; I/O intensive workloads; IaaS cloud computing; data-intensive applications

HPDC'12, June 18–22, 2012, Delft, The Netherlands.

1. INTRODUCTION

Over the last few years, a large shift was recorded from privately own and managed hardware to Infrastructure-as-a-Service (IaaS) cloud computing [7]. Using IaaS, users can lease storage space and computation time from large datacenters in order to run their applications, paying only for the consumed resources.

Virtualization is the core technology behind IaaS clouds. Computational resources are presented to the user in form of virtual machines (VMs), which are fully customizable by the user. This equivalent to owning dedicated hardware, but without any long term cost and commitment. Thanks to virtualization, IaaS cloud providers can isolate and consolidate the workloads across their datacenter, thus being able to serve multiple users simultaneously in a secure way.

Live migration [11] is a key feature of virtualization. It gives the cloud provider the flexibility to freely move the VMs of the clients around the datacenter in a completely transparent fashion, which for the VMs is almost unnoticeable (i.e. they typically experience an interruption in the order of dozens of milliseconds or less). This ability can be leveraged for a variety of management tasks, such as:

Load balancing. the VMs can be rearranged across the physical machines of the datacenter in order to evenly distribute the workload and avoid imbalances caused by frequent deployment and termination of VMs.

Online maintenance. when physical machines need to be serviced (e.g. upgraded, repaired or replaced), VMs can be moved to other physical machines while the maintenance is in progress, without the need to shutdown or terminate any VM.

Power management. if the overall workload can be served by less physical machines, VMs can be consolidated from the hosts that are lightly loaded to hosts that are more heavily loaded [21]. Once the migration is complete, the hosts initially running the VMs can be shutdown, enabling the cloud provider to save on energy spending.

Proactive fault tolerance. if a physical machine is suspected of failing in the near future, its VMs can be proactively moved to safer locations [20]. This has the potential to reduce the failure rate experienced by the user, thus enabling the provider to improve the conditions stipulated in the service level agreement. Even if the machine will not

completely fail, migration may still prevent VMs from running with degraded performance.

A particularly difficult challenge arises in the context of live migration when the VMs make use of local storage. This scenario is frequently encountered in practice [5]: VMs need a "scratch space", i.e. a place where to store temporary data generated during their runtime. Such a feature can significantly speed up large-scale data-intensive applications, as it eliminates the need to rely on a much slower parallel file system or cloud repository. To this end, cloud providers typically install local disks on the physical machines that are running the user workloads and enable the VMs to access it. Since one of the goals of live migration is to relinquish the source as fast as possible, no residual dependencies on the source host should remain after migration. Thus, the complete disk state needs to be transferred to destination while it is actively changed by the VM.

In this paper we propose a live storage transfer mechanism that complements existing live migration approaches in order to address the challenge mentioned above. Our approach is specifically optimized to withstand rapid changes to the disk state during live migration, a scenario that is frequently caused by VMs executing I/O intensive workloads. We aim to minimize the migration time and network traffic overhead that live migrations generate under these circumstances, while at the same time minimizing the I/O performance degradation perceived by the VMs. This is an important issue: as noted in [31], the impact of live migration on a heavy loaded VM cannot be neglected, especially when service level agreements need to be met.

Our contributions can be summarized as follows:

- We present a series of design principles that facilitate efficient transfer of local storage during live migration. Unlike conventional approaches, our proposal is designed to efficiently tolerate I/O intensive workloads inside the VM while the live migration is in progress. (Section 4.1)
- We show how to materialize these design principles in practice through a series of algorithmic descriptions, that are applied to build a completely transparent implementation with respect to the hypervisor. (Sections 4.2, 4.3 and 4.4)
- We evaluate our approach in a series of experiments, each conducted on hundreds of nodes provisioned on the Grid'5000 testbed, using both synthetic benchmarks and real-life applications. These experiments demonstrate significant improvement in migration time and network traffic over state-of-art approaches, while reducing at the same time the negative impacts of live migration on performance inside the VMs. (Section 5)

2. THE PROBLEM OF STORAGE TRANSFER DURING LIVE MIGRATION

In order to migrate a VM, its state must be transferred from the host node to the destination node where it will continue running. This state consists of three main components: *memory*, *state of devices* (e.g. CPU, network interface) and *storage*. The state of devices typically comprises a minimal amount of information (hardware buffers, processor execution state, etc.) and is usually considered negligible. However, the size of memory and storage however can explode to huge sizes and can take extended periods of time to transfer.

One solution to this problem is to simply pay for the cost of interrupting the application while transferring the memory and storage, which is known as *offline migration*. However, distributed data-intensive applications do not work in isolation: there are dependencies between distributed processes running in different VM instances. Thus, offline migration causes an unacceptably long downtime during which the VM is not not able to communicate with the outside, which leads to accumulation of jitter that degrades the performance of the whole application.

To alleviate this issue, *live migration* [11] is a potential solution: it enables a VM to continue almost uninterrupted (i.e. with a downtime in the order of dozens of milliseconds) while experiencing minimal negative effects due to migration. In order to be possible, live migration needs to solve a difficult problem: keeping a *consistent view* of memory and storage at all times for the VM, while performing background transfers that converge to a state where the memory and storage is fully available on the destination and the source is not needed anymore. Techniques to do so efficiently in the context of HPC applications have been studied before for memory [16], but how to achieve this for storage remains an open issue.

Initially, the problem of keeping a consistent view of storage between the source and destination was simply avoided by using a parallel file system rather than local disks. Thus, the source and destination are always fully synchronized and no transfer of storage is necessary. However, under normal operation, this approach can have several disadvantages: (1) it consumes system bandwidth and storage space on shared disks for temporary I/O that is not intended to be shared; (2) it limits the sustained throughput that the VM can achieve for I/O; (3) it raises scalability issues, especially considering the growing sizes of datacenters.

Given these disadvantages, such a solution is not feasible to adopt just for the purpose of supporting live migrations. It is important to enable VMs to use local storage as scratch space, while still providing efficient support for live migration. As a consequence, in order to obtain a consistent view of storage that does not indefinitely depend on the source, storage must be transferred from the source to the destination.

Apparently, transferring local storage is highly similar to the problem of transferring memory: one potential solution is simply to consider local storage as an extension of memory. However, such an approach does not take into account the differences between the I/O workload and memory workload, potentially performing sub-optimally. Furthermore, unlike memory, VM storage does not always need to be fully transferred to the destination, as a large part of it is never touched during the lifetime of the VM and can obtained in a different fashion, for example directly from the cloud repository.

In this context, the storage transfer strategy plays a key role. To quantify the efficiency of such a strategy, we rely on a series of performance metrics. Our goal is to optimize the storage transfer according to these metrics:

Migration time.

is the total time elapsed between the moment when the live migration was initiated on the source and the moment

when all resources needed by the VM instance are fully available at the destination. This parameter is important because it indicates the total amount of time during which the source is busy and cannot be reallocated to a different task or shut down. Even if migration time for a single VM instance is typically in the order of seconds and minutes, when considering the economy of scale, multiple migrations add up to huge amounts of time during which resources are wasted. Thus, a low migration time is highly desirable.

Network traffic.
is the amount of network traffic that can be traced back to live migration. This includes memory (whose transfer cannot be avoided), any direct transfer of storage from source to destination, as well as any traffic generated as a result of synchronizing the source with the destination through shared storage. Network traffic is expensive: it steals away bandwidth from VM instances, effectively diminishing the overall potential of the datacenter. Thus, it must be lowered as much as possible.

Impact on application performance.
is the extent to which live migrations cause a performance degradation in the application that runs inside the VM instances. This is the effect of consuming resources (bandwidth, CPU time, etc.) during live migrations that could otherwise be leveraged by the VM instances themselves to finish faster. Obviously, it is desirable to limit the overhead of migration as much as possible, in order to minimize any potential negative effects on the application.

3. RELATED WORK

If downtime is not an issue, *offline migration* is a solution that potentially consumes the least amount of resources. This is a three-stage procedure: freeze the VM instance at the source, take a snapshot of its memory and storage, then restore the VM state at the destination based on the snapshot. Several techniques to take a snapshot of VM instances have been proposed, such as: dedicated copy-on-write image formats [12, 30], dedicated virtual disk storage services based on shadowing and cloning [26], fork-consistent replication systems based on log-structuring [13], de-duplication of memory pages [28]. Many times, for HPC applications it is cheaper to save the state of the application inside the virtual disk of the VM instance and then reboot the VM instance on the destination, rather than save the memory inside the snapshot [27]. Approaches such as *VMFlock* [4] are specifically optimized to migrate a whole set of VMs simultaneously in an offline fashion between clouds, by taking advantage of similarities between their corresponding images and de-duplicating them in a distributed fashion.

Extensive live migration research was done for memory-to-memory transfer. The *pre-copy* strategy [11, 22] is by far the most widely adopted approach implemented in production hypervisors. It works by copying the bulk of memory to the destination in background, while the VM instance in running on the source. If any transmitted memory pages are modified in the mean time, they are re-sent to the target subsequently, based on the assumption that eventually the memory on the source and destination converge up to a point when it is cheap to synchronize them and transfer control to the destination. Several techniques can be used to reduce the overhead incurred by the background transfers, such as online compression [29, 24].

However, pre-copy has its limitations: if memory is modified faster than it is copied in the background to the destination, this solution never converges. To address this limitation, Ibrahim et al. [16] propose an optimized pre-copy strategy that dynamically adapts to the memory change rate in order to guarantee convergence. Other approaches, such as Checkpoint/Restart and Log/Replay were successfully adopted by Liu et al. [18] to significantly reduce downtime and network bandwidth consumption over pre-copy. A *post-copy* strategy was proposed by Hines et al. [15]. Unlike pre-copy, control is transferred to the destination from the beginning, while relying on the source to fetch the needed content in the background up to the point when the source is not needed anymore. This approach copies each memory page only once, thus guaranteeing convergence regardless of how often the memory is modified.

Although not directly related to live migration, live memory transfer techniques were also proposed by Lagar-Cavilla et al. [17] in from of *VM cloning*, an abstraction for VM replication that works similar to the fork system call. This is similar to live migration in that the destination must receive a consistent view of the source's memory, however, the goal is to enable the source to continue execution on a different path rather than shut it down as quickly as possible.

The problem of storage transfer was traditionally avoided in favor of shared storage that is fully synchronized both at source and destination. However, several attempts break from this tradition.

A widely used approach in production is incremental block migration, as available in QEMU/KVM [3]. In this case, copy-on-write snapshots of a base disk image, shared using a parallel system, are created on the local disks of the nodes that run the VMs. Live migration is then performed by transferring the memory together with the copy-on-write snapshots, both using pre-copy. Thus, this approach inherits the drawbacks of pre-copy for I/O: under heavy I/O pressure the disk content may be changed faster than it can be copied to the destination, which introduces an infinite dependence on the source.

Bradford et al. [8] propose a two-phase transfer: in the first phase, the whole disk image is transferred in the background to the destination. Then, in the second phase the live migration of memory is started, while all new I/O operations performed by the source are also sent in parallel to the destination as incremental differences. Once control is transferred to the destination, first the hypervisor waits for all incremental differences to be successfully applied, then resumes the VM instance. However, waiting for the I/O to finish can increase downtime and reduce application performance. Furthermore, since a full disk image can grow in the order of many GB, the first phase can take a very long time to complete, negatively impacting the total migration time.

A similar approach, called *mirroring*, is proposed by Haselhorst et al. [14]: the first phase transfers the disk content in the background to the destination, while in the second phase all writes are trapped and issued in parallel to the destination. However, unlike Bradford et al., writes complete on the source only after they also complete on the destination. Under I/O intensive workloads, this can lead to increased latency and decreased throughput for writes that happen before control is transferred to the destination. Furthermore,

it can happen that some workloads repeatedly overwrite the same location. In this case, mirroring unnecessarily slows down migration and consumes bandwidth. To alleviate this, Mashtizadeh et al. [19] complemented mirroring with hot block avoidance. However, according to the authors, this introduced a high complexity and did not work for certain workloads.

Our own effort tries to overcome these limitations while achieving the goals presented in Section 2.

4. OUR APPROACH

To address the issues mentioned in Section 2, in this section we propose a completely transparent live storage transfer scheme that complements the live migration of memory. We introduce a series of design principles that are at the foundation of our approach (Section 4.1), then show how to integrate them in an IaaS cloud architecture (Section 4.2) and finally introduce a series of algorithmic descriptions (Section 4.3) that we detail how to implement in practice (Section 4.4).

4.1 Design principles

Transfer only the modified contents of the VM disk image to the destination. Conceptually, the disk space of the VMs is divided into two parts: (1) a basic part that holds the operating system files together with user applications and data; (2) a writable part that holds all temporary data written during the lifetime of the VM. The basic part is called the base disk image. It is configured by the user, stored persistently on the cloud's repository and then used as a template to deploy multiple VM instances.

As this part is never altered, it must not necessarily be transferred from the source to the destination: it can be obtained directly from the cloud repository. Thus, we propose to transfer only the actually written data from the source to the destination, while any data that is required from the basic part is directly accessed from the cloud repository where the base disk image is stored.

To reduce latency and improve read throughput on the destination, we transparently prefetch the hot contents of the base disk image according to hints obtained from the source. Note that under concurrency, this can incur a heavy load on the repository. To avoid any potential bottleneck introduced by read contention, we assume a distributed repository is present that can evenly distribute a read workload under concurrency. Under these circumstances, we can store the disk image in a striped fashion: it is split into small chunks that are distributed among the storage elements of the repository.

Transparency with respect to the hypervisor. The I/O workload of the VM can be very different from its memory workload. At one extreme, the application running inside the VM can change the memory pages very frequently but rarely generate I/O to local storage. At the other extreme, the application may generate heavy I/O to local storage but barely touch memory (e.g. it may need to flush in-memory data to disk). For this reason, the best way to transfer memory can be different from the best way to transfer storage.

To deal with this issue, we propose to separate the storage transfer from memory transfer and handle it independently from the hypervisor. Doing so has two important advantages. First, it enables a high flexibility in choosing what strategy to apply for the memory transfer and how to fine tune it depending on the memory workload. Second, it offers high portability, as the storage transfer can be used in tandem with a wide selection of hypervisors without any modification.

Note that this separation implies that our approach is not directly involved in the process of transferring control from source to destination. This is the responsibility of the hypervisor. How to detect this moment and best leverage it to our advantage is detailed in Section 4.4.

Hybrid active push-prioritized prefetch strategy. Under an I/O intensive workload, the VM rapidly changes the disk state, which under live migration becomes a difficult challenge for the storage transfer strategy. Under such circumstances, attempting to synchronize the storage on the source and destination before transferring control to the destination introduces two issues:

- The same disk content may change repeatedly. In this case, content is unnecessarily copied at the destination, eventually being overwritten before the destination receives control. Thus, migration time is increased and network traffic is generated unnecessarily.
- Disk content may change faster than it can be copied to the destination. This has devastating consequences, as live migration will never finish and control will never be transferred to the destination. Thus all network traffic and negative impact on the application is in vain, not to mention keeping the destination busy.

To avoid these issues, we propose a hybrid strategy described below.

As long as the hypervisor did not transfer control to the destination, the source actively pushes all local disk content to the destination, While the VM is still running at the source, we monitor how many times each chunk was written. If a chunk was written more times than a predefined $Threshold$, we mark this chunk as *dirty* and avoid pushing it to the destination. Doing so enables us deal with the first issue: each chunk is transferred no more than $Threshold$ times to the destination.

Once the hypervisor transfers control to the destination, we send the destination the list of remaining chunks that it needs from the source in order to achieve a consistent view of local storage. At this point, our main concern is to eliminate the dependency on the source as fast as possible. In order to do so, we prefetch the chunks in decreasing order of access frequency. This ensures that *dirty* chunks, which are likely to be accessed in the future, arrive first on the destination. If the destination needs a chunk from the source before it was prefetched, we suspend the prefetching and serve the read request with priority.

Doing so enables us to deal with the second issue, since storage does not delay in any way the transfer of control to the destination. No matter how fast disk changes, once control arrives at the destination, the source is playing a passive role and does not generate any new disk content, thus leaving only a finite amount of data to be pulled from the source.

4.2 Architecture

The simplified architecture of an IaaS cloud that integrates our approach is depicted in Figure 1. The typical elements found in the cloud are illustrated with a light background, while the elements introduced by our approach are illustrated by a darker background.

The *shared repository* is a service that survives failures and is responsible to store the base disk images that are used as a template by the compute nodes. It can either use a dedicated set of resources or simply aggregate a part of each of the local disks of the compute nodes into a common pool. For example, *Amazon S3* [6] or a parallel file system can serve this role. The cloud client has direct access to the repository and is allowed to upload and download the base disk images.

Using the *cloud middleware*, which is the frontend of the user to the cloud, an arbitrary number of VM instances can be deployed starting from the same base disk image. Typically these VM instances form a virtual distributed environment where they communicate among each other. The cloud middleware is also responsible to coordinate all VM instances of all users in order to meet its service level agreements, while minimizing operational costs. In particular, it implements the VM scheduling strategies that leverage live migration in order to perform load-balancing, power saving, pro-active fault tolerance, etc.

Each compute node runs a *hypervisor* that is responsible for running the VM instances. All reads and writes issued by the hypervisor to the underlying virtual disk are trapped by the *migration manager*, which is the central actor of our approach and is responsible to implement our live storage migration strategy.

Under normal operation, the migration manager presents the disk image to the hypervisor as a regular file that is accessible from the local disk. Whenever the hypervisor writes to the image file, the migration manager generates new chunks that are stored locally. Whenever the hypervisor reads a region of the image that has never been touched before, the chunks that cover that region are fetched from the repository and copied locally. Thus, future accesses to a region that has been either read or written before are served from the local disk directly. Using this strategy, the I/O pressure put on the repository is minimal, as contents is fetched on-demand only. At the same time, the migration manager is listening for migration requests and implements the design principles presented in Section 4.1. The next section is dedicated to detail this aspect.

4.3 Zoom on the migration manager

The migration manager is designed to listen for two types of events: *migration requests* and *migration notifications*.

The cloud middleware can send migration requests to the migration manager using the MIGRATION_REQUEST primitive (Algorithm 1). Upon receipt of this event, the migration manager assumes the role of *migration source* (by setting the *isSource* flag). At this point, all chunks that were locally modified (part of $ModifiedSet$) are queued up into the *RemainingSet* for active pushing to the destination in the background. Furthermore, it starts keeping track of how many times each chunk is modified during the migration process. This information is stored in $WriteCount$, initially 0 for all chunks. Once this initialization step completed, it sends a migration notification to the migration manager running on *Destination*, which assumes the role of *migration destination* and starts accepting chunks that are pushed from the source. At the same time, the source forwards the migration request to the hypervisor, which independently starts the migration of memory from the source to the destination. As soon as the migration has started, the BACKGROUND_PUSH task is launched, which starts pushing all chunks whose access count is less than $Threshold$ to the source.

If a chunk c is modified before control is transferred to the destination, its write count is increased. This results in the BACKGROUND_PUSH task being notified (which potentially wakes up if not already busy with pushing other chunks). A simplified WRITE primitive that achieves this (but does not handle writes to partial chunks or writes spanning multiple chunks) is listed in Algorithm 2.

Algorithm 1 Migration request on the source

```
1: procedure MIGRATION_REQUEST(Destination)
2:     RemainingSet ← ModifiedSet
3:     for all c ∈ VirtualDisk do
4:         WriteCount[c] ← 0
5:     end for
6:     start BACKGROUND_PUSH
7:     isSource ← true
8:     invoke MIGRATION_NOTIFICATION on Destination
9:     forward migration request to the hypervisor
10:    notify BACKGROUND_PUSH
11: end procedure
12: procedure BACKGROUND_PUSH
13:    while true do
14:        wait for notification
15:        while ∃c ∈ RemainingSet : WriteCount[c] <
   Threshold do
16:            buf ← contents of c
17:            push (c, buf) to Destination
18:            RemainingSet ← RemainingSet \ {c}
19:        end while
20:    end while
21: end procedure
```

Algorithm 2 Simplified writes of single full chunks

```
1: function WRITE(c, buffer)
2:     if isDestination then
3:         cancel any pull(c) in progress
4:         RemainingSet ← RemainingSet \ {c}
5:     end if
6:     contents of c ← buffer
7:     ModifiedSet ← ModifiedSet ∪ {c}
8:     if isSource then
9:         WriteCount[c] ← WriteCount[c] + 1
10:        RemainingSet ← RemainingSet ∪ {c}
11:        notify BACKGROUND_PUSH
12:    end if
13:    return success
14: end function
```

Once the hypervisor is ready to transfer control to the destination and invokes SYNC on the disk image exposed by the migration manager, the BACKGROUND_PUSH task is stopped and the source enters in a passive phase where it

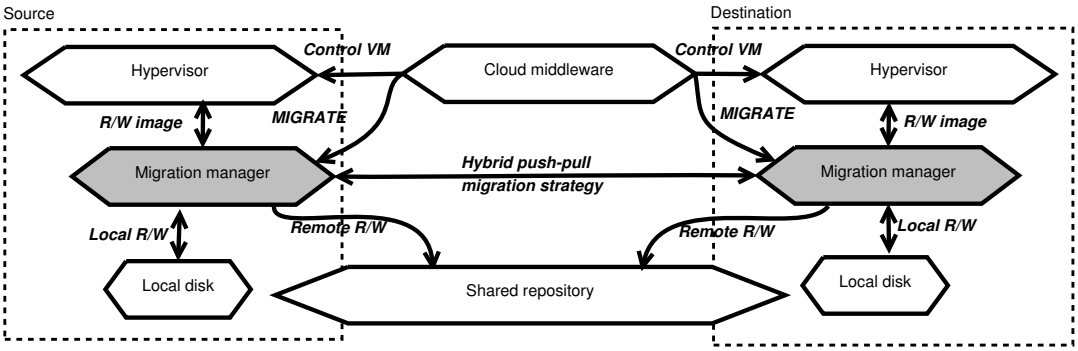


Figure 1: Cloud architecture that integrates our approach via the migration manager (dark background).

listens for pull requests coming from the destination. In order to signal that it is ready for this, the source invokes TRANSFER_IO_CONTROL on the destination, as shown in Algorithm 3. The TRANSFER_IO_CONTROL primitive receives as parameters the remaining set of chunks that need to be pulled from the source, together with their write counts. It then starts the BACKGROUND_PULL task, whose role is to prefetch all remaining chunks from the source. Priority is given to the chunks with the highest write count, under the assumption that frequently modified chunks will also be modified in the future.

Algorithm 3 Migration notification and transfer of control on destination

```
1: procedure MIGRATION_NOTIFICATION
2:     isDestination ← true
3:     accept chunks from Source
4: end procedure
5: procedure TRANSFER_IO_CONTROL(RS, WC)
6:     RemainingSet ← RS
7:     WriteCount ← AC
8:     start BACKGROUND_PULL
9: end procedure
10: procedure BACKGROUND_PULL
11:     while RemainingSet ≠ ∅ do
12:         c ← c′ ∈ RemainingSet : WriteCount[c′] = max(WriteCount[RemainingSet])
13:         RemainingSet ← RemainingSet \ {c}
14:         pull(c) from Source
15:     end while
16: end procedure
```

Note that chunks may be needed earlier than they are scheduled to be pulled by BACKGROUND_PULL. To accommodate this case, the READ primitive needs to be adjusted accordingly. A simplified form that handles only reads of single full chunks is listed in Algorithm 4. There are two possible scenarios: (1) the chunk c that is needed is already being pulled - in this case it is enough to wait for completion; (2) c is scheduled for prefetching but the pull has not started yet - in this case BACKGROUND_PULL is suspended and resumed at a later time in order to allow READ to pull c. On the other hand, if a chunk c that is part of the *RemainingSet* is modified by WRITE, the old content must not be pulled from the source anymore and any pending pull of chunk c must be aborted.

Once all remaining chunks have been pulled at the destination, the source is not needed anymore. Both the hypervisor and the migration manager can be stopped and the source can be shut down (or its resources used for other purposes). At this point, the live migration is complete.

Algorithm 4 Simplified reads of single full chunks

```
1: function READ(c)
2:     if isDestination and c ∈ RemainingSet then
3:         if c is being pulled by BACKGROUND_PULL then
4:             wait until c is available
5:         else
6:             suspend BACKGROUND_PULL
7:             pull(c) from Source
8:             RemainingSet ← RemainingSet \ {c}
9:             resume BACKGROUND_PULL
10:        end if
11:    end if
12:    fetch c from repository if c not available locally
13:    return contents of c
14: end function
```

A graphical illustration of the interactions performed in parallel by the algorithms presented above, from the initial migration request on the source to the moment when the live migration is complete, is depicted in Figure 2. Solid arrows are used to represent interactions between the migration managers. A dotted pattern is used to represent interactions between the migration manager and the hypervisor, as well as the interactions between the hypervisors themselves. Note that the transfer of memory is not explicitly represented, as our approach is completely transparent with respect to the hypervisor and its migration strategy.

4.4 Implementation

We implemented the *migration manager* on top of *FUSE* (File System in UserspacE) [1]. Its basic functionality (i.e. to intercept the reads and writes of the hypervisor with the purpose of caching the hot contents of the base disk image locally, while storing all modifications locally as well) is based on our previous work presented in [26]. The migration manager exposes the local view of the base disk image as file inside the mount point, accessible to the hypervisor through the standard POSIX access interface.

To keep a maximum of portability with respect to the hypervisor, we exploit the fact that the hypervisor calls the sync system call right before transferring control to the destination. Thus, our implementation of the sync system

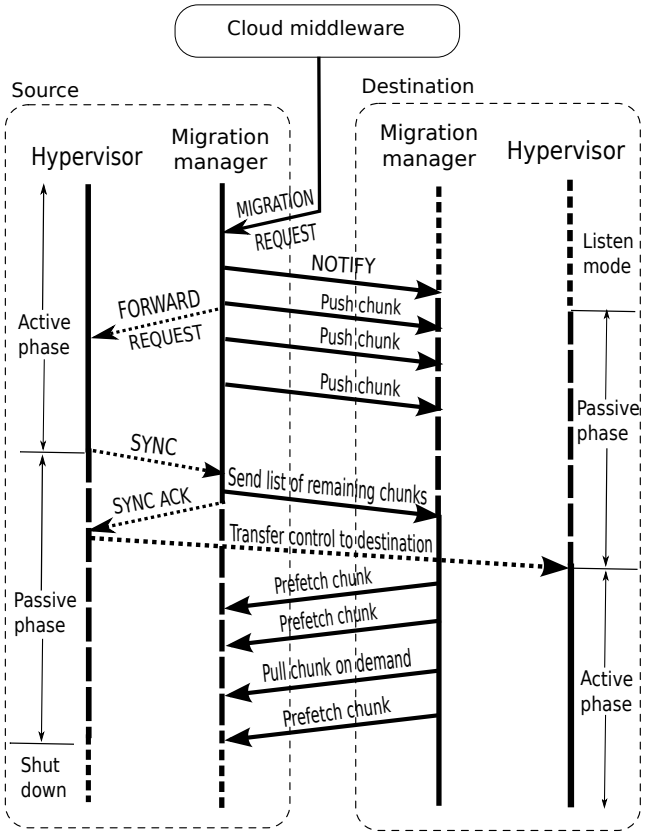


Figure 2: Overview of the live storage transfer as it progresses in time.

call invokes TRANSFER_IO_CONTROL on the destination, ensuring that the destination is ready to intercept reads and writes before the hypervisor transfers control to the VM instance itself. Furthermore, we strive to remain fully POSIX-compliant despite the need to support migration requests. For this reason, we implemented the MIGRATION_REQUEST primitive as an *ioctl*.

Finally, the migration manager is designed to integrate with *BlobSeer* [23, 25], which acts as the repository that holds the base VM disk images. BlobSeer enables *scalable aggregation of storage space* from a large number of participating nodes, while featuring transparent data striping and replication. This enables it to reach high aggregated throughputs under concurrency while remaining highly resilient under faults.

5. EVALUATION

5.1 Experimental setup

The experiments were performed on Grid'5000, an experimental testbed for distributed computing that federates nine sites in France. We used 100 nodes of the graphene cluster from the Nancy site, each of which is equipped with a quad-core Intel Xeon X3440 x86_64 CPU with hardware support for virtualization, local disk storage of 278 GB (access speed ≃55 MB/s using SATA II ahci driver) and 16 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of ≃0.1 ms).

We use QEMU/KVM [3] 1.0 as the hypervisor. It is running on all compute nodes is , while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 4 GB raw disk image file based on the same Debian Sid distribution was used as the guest environment. We rely on the standard live migration implemented in QEMU (pre-copy) in order to transfer the memory. In order to minimize the overhead of migration, we set the maximum migration speed to match the maximum bandwidth of the network interface (i.e. 1G).

5.2 Methodology

The experiments we perform involve a set of VM instances, each of which is running on a different compute node. We refer to the nodes where the VM instances are initially running as *sources*. The rest of the nodes act as *destinations* and are prepared to receive live migrations at any time.

We compare five approaches throughout our evaluation:

5.2.1 Live storage transfer using our approach

In this setting we rely on BlobSeer to store base disk image and on the FUSE-based migration manager (described in Section 4.4) to expose a locally modifiable view of the disk image to the hypervisor. BlobSeer is deployed on all compute nodes and stores the initial base disk image (4 GB) in a distributed fashion, using a stripe size of 256 KB (which from our previous experience is large enough to avoid excessive fragmentation overhead, yet small enough to avoid contention under concurrent read accesses). Any live migration request is treated according to the strategy described in Section 4.3. Both the transfer of storage and memory proceed concurrently and independently. For the rest of this paper, we refer to this setting as our−approach.

5.2.2 Live storage transfer using other techniques

We compare our approach to three other techniques: (1) a pure pre-copy approach (denoted precopy), in which case we assume the local modifications are stored in a qcow2 [12] disk snapshot and the storage transfer is performed using QEMU/KVM's standard incremental block migration; (2) an improved precopy approach (denoted mirror), based on our FUSE implementation, that performs all writes synchronously both at the source and destination with the intent to reproduce the approach presented in [14]; and finally (3) a pure post-copy approach (denoted postcopy), that is a based on our approach and simply remains passive during the push phase, deferring any transfer until after the moment when control is transferred to the destination.

5.2.3 Synchronization through a parallel file system

We include in our evaluation a third setting where the modifications to the base disk image are not stored locally but are synchronized on the source and destination through a parallel file system (denoted pvfs−shared). This corresponds to the traditional solution that avoids storage transfer during live migration altogether. For the purpose of this work, we have chosen PVFS [10] as the parallel file system, as it is a popular POSIX-compliant high performance storage solution that hypervisors can use for storage synchronization. In this setting, the base disk image is stored in a PVFS deployment that spans all compute nodes, while the local

modifications are shared between the source and the destination through a qcow2 disk snapshot that is stored in the same PVFS deployment.

Table 1: Summary of compared approaches

Approach	Local storage transfer strategy
our−approach	As presented in Section 4.3
mirror	Sync writes both at src and dest
postcopy	Pull from src after transfer of control
precopy	Push to dest before transfer of control
pvfs−shared	Does not apply (All writes go to PVFS)

These approaches are summarized in Table 1 and are compared based on the performance metrics defined in Section 2:

- *Migration time*: is the time elapsed between the moment when the migration has been initiated and the source has been relinquished. For precopy, mirror and pvfs−shared, the live migration ends as soon as the control is transferred to the destination. For our−approach and postcopy, migration time also includes the time spent by the destination to pull all remaining local modifications from the source.
- *Network traffic*: is the total network traffic generated during the experiments by the VM instances due to I/O to their virtual disks and live migration. Except pvfs−shared, this traffic includes all memory and storage transfers. In the case of pvfs−shared, it includes the memory transfers and all I/O generated during the lifetime of the VMs (which is redirected to PVFS), regardless whether inside or outside of migration.
- *Impact on application performance*: is the performance degradation perceived by the application during live migration when compared to the case when no migration is performed. For the purpose of this work, we are interested in the impact on the sustained I/O throughput in various benchmarking scenarios, as well as the impact on total runtime for data-intensive HPC applications.

5.3 Live migration performance of I/O intensive benchmarks

Our first series of experiments evaluates the performance of live migration for two I/O intensive benchmarks: *IOR* [2] and *AsyncWR*.

IOR is a popular HPC I/O benchmarking tool. It measures read and write throughput using various access interfaces (POSIX, HDF5, MPI-IO). For the purpose of this work, we focus on the POSIX access interface, as it is the choice of many HPC applications that typically write output data and checkpointing data through it. Our benchmark consists in performing 10 iterations of IOR using a single process running inside the VM that writes and then reads a 1 GB large file in blocks of 256 KB. Under no live migration, the maximal achieved read and write performance is 1 GB/s and 266 MB/s respectively.

AsyncWR is a benchmarking tool that we developed to simulate the behavior of data-intensive applications that mix computations with intensive I/O. It runs a fixed number of iterations, each of which performs a computational task that keeps the CPU busy (increments a counter) while generating random data into a memory buffer. This memory buffer is copied at the beginning of next iteration into an alternate memory buffer and written asynchronously to the file system. Using this workload, we aim to study the impact of storage migration in a scenario where a moderate constant I/O pressure is generated inside the VM instances while the CPU is busy. We fixed the number of iterations to 180 using a data size of 1 MB/iteration, which corresponds to a constant I/O pressure of about 6 MB/s when no live migration occurs.

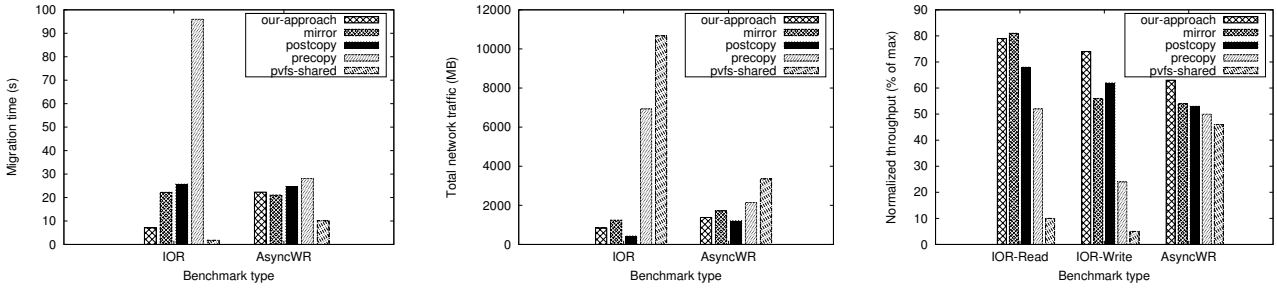
The experiment consists in launching each of the benchmarks inside a VM instance and then performing a live migration after a delay of 100 seconds. This gives the VM instance a warm-up period that avoids instant migrations due to lack of accumulated changes, while at the same time forcing the live migration to withstand the full I/O pressure from the beginning. The amount of RAM available to the VM instance is fixed at 4 GB.

The total migration time is depicted in Figure 3(a). As can be noticed, for the highly I/O intensive *IOR* workload there is a large difference between the five approaches. Since the pvfs−shared approach needs to transfer memory only, it is the fastest of all three. Comparatively, our−approach manages to perform a complete storage transfer during live migration in about 4x more time, which itself is more than 3x faster than postcopy. This underlines the importance of the push phase in overlapping the memory transfer with the storage transfer, which ultimately reduces overall migration time. Pure precopy seems to transfer a lot of storage more than once, which explains its more than 10x slower migration time when compared to our−approach. Although this problem is alleviated by mirror, it slows down writes, making it about 2.8x slower than our−approach. However, for *AsyncWR* (i.e. when the I/O workload is moderate), mirroring can lead to a faster migration than our−approach. Still, our approach is faster than pure postcopy and precopy. As expected, pvfs−shared is the fastest in the *AsyncWR* case as well.

When comparing network traffic (Figure 3(b)), a clear trend is visible: our−approach and postcopy have a clear advantage over the other approaches, with postcopy slightly better due to the fact that it needs to transfer only a minimal amount of data. Pure precopy generates a lot of network traffic due to accumulation of writes on the source, which can be alleviated by mirroring. Finally, the worst of all approaches is as expected pvfs−shared: it generates network traffic for every I/O request, regardless whether it is during live migration or not. Compared to pvfs−shared, our approach conserves bandwidth by more than an order of magnitude in the *IOR* case.

Finally, the impact of live migration on the performance results of *IOR* and *AsyncWR* is illustrated in Figure 3(c). We focus on the achieved throughputs for the IOR benchmark, both for reading (*IOR-Read*) and writing (*IOR-Write*), as well as the write throughput achieved for *AsyncWR*. Due to large differences between the absolute values of the three throughput types, we have chosen to normalize the average values obtained during the experiment with respect to the maximal achieved values when no live migration is performed. These maximal values are: 1 GB/s for *IOR-Read*, 266 MB/s for *IOR-Write* and 6 MB/s for *AsyncWR*.

For the IOR benchmark, a large gap can be observed between pvfs−shared and the other approaches. Here it becomes clearly visible that under I/O intensive scenarios, synchro-



(a) Migration time (lower is better) (b) Total network traffic (lower is better) (c) Normalized average throughput compared to fastest no migration scenario (higher is better)

Figure 3: Migration performance of a VM instance (4 GB of RAM) that performs I/O intensive workloads

nizing through the parallel file system in order to avoid storage transfer during live migration has a high price: the total read throughput is less than 10% of the maximum achievable throughput, while the write throughput is less than 5%. Our approach achieves 80% of maximal read throughput, which is marginally surpassed only by mirror, due to the fact that no data resides on the source anymore and thus it does not trigger on-demand pulls after the control was transferred to the destination. This effect is also visible when comparing our approach to postcopy: in this case, our approach achieves 10% more read bandwidth due to the fact that the push phase diminishes the amount of data that needs to be pulled on-demand after the transfer of control. The repeated transfers of the same data triggered by precopy are also visible in the sustained throughput: it reaches only 50% of the maximal read throughput and 25% of the maximal write throughput. Our approach is the clear winner when comparing the sustained write throughputs: it performs by almost 15% better than postcopy. This also demonstrates the trade-off necessary to achieve a slightly higher read throughput using mirror: the achieved write throughput is by 20% smaller than our approach.

In the case of *AsyncWR*, an overall drop in sustained write throughput can be observed when comparing to the *IOR* case. The only exception to this is pvfs−shared, which significantly catches up with the other approaches, but still offers the lowest throughput. This is explained by the fact that unlike *IOR*, which is a purely I/O oriented workload, *AsyncWR* performs memory-intensive operations on the data before it is written, which increases the overhead of the memory transfer, ultimately leading to contention for bandwidth between memory and storage transfer, thus the observed effect. Nevertheless, our approach achieves at least 10% more bandwidth when compared to the other approaches, whose performance is quite close one to another.

5.4 Performance of concurrent live migrations

Our next series of experiments aims to evaluate the performance of all five approaches in a highly concurrent scenario where multiple live migrations are initiated simultaneously. To this end, we use the *AsyncWR* benchmark presented in Section 5.3.

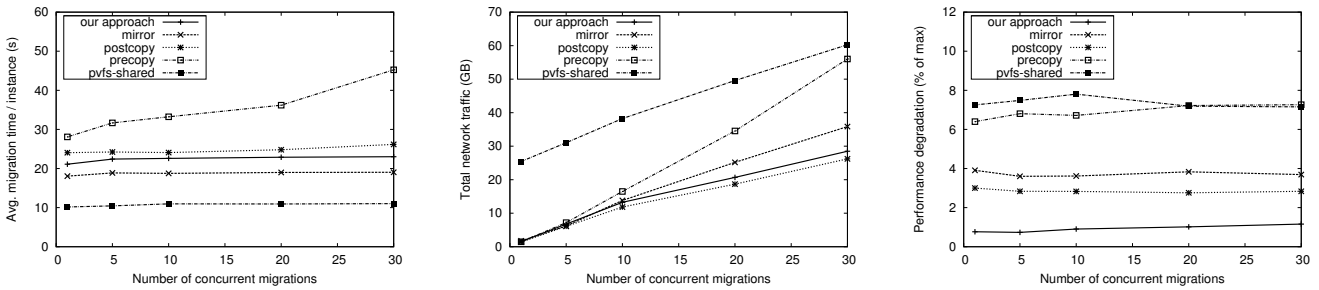
The experimental setup is as follows: we fix the number of sources to 30 and gradually increase the number of destinations from 1 to 30, in steps of 10. On all sources we launch the *AsyncWR* benchmark, wait until a warm-up period of 100 seconds has elapsed, and then simultaneously initiate the live migrations to the destinations. We keep the same configuration as in the previous section: the total amount of data is fixed at 1800 MB, while the amount of RAM available to the VM instance is fixed at 4 GB.

As can be observed in Figure 4(a), with increasing number of live migrations, all approaches except precopy keep an almost constant average migration time. On the other hand, precopy experiences a steady increase in average migration time, which reaches over 50% for 30 migrations when compared to 1 migration.

In order to explain this finding, it needs to be correlated to the total network traffic, depicted in Figure 4(b). As can be noticed, precopy experiences a sharp increase in network traffic, whereas the rest of approaches experience a much milder trend. Except for pvfs−shared, all approaches generate network traffic exclusively because of the live migrations. Thus, the depicted network traffic is concentrated over very short periods of time. Since the total system bandwidth (approx. 8 GB/s provided by a Cisco Catalyst switch) is insufficient to accommodate the instantaneous needs of precopy, a slowdown in transfer speed occurs when increasing the number of live migrations, which ultimately reflects into increased average migration time.

Thanks to earlier transfer of control to the destination, our approach and postcopy enables new data to be generated directly at the destination, greatly reducing the network traffic induced by the storage migration, which enables it to avoid reaching the system bandwidth limit. Although counter-intuitive, reaching this limit is also avoided by mirror. This happens because mirroring actually slows down writes, which in itself slows down the application and thus reduces the rate at which memory is changed. Nevertheless, the benefits of mirroring are smaller when compared to our approach and postcopy. Note that although very high, the network traffic generated by pvfs−shared is not generated over short periods of time, which enables it to remain scalable with respect to average migration time.

The impact on the computation is depicted in Figure 4(c). We measure the overall amount of computation lost due to live migrations as a percent of the maximum computational potential achieved in a migration-free scenario. Since the *AsyncWR* computation is simply meant to keep the CPU busy (i.e. increment a counter) we define the computational



(a) Average migration time (lower is better) (b) Total network traffic (lower is better) (c) Performance degradation (lower is better)

Figure 4: AsyncWR when increasing the number of concurrent live migrations from 1 to 30

potential as simply the aggregate end-value of the counters from all processes of all VM instances. As can be observed, our approach manages to reach a degradation of slightly more than %1. This is up to 8x better than pvfs−shared and precopy and 3x-4x better than postcopy and mirror.

Overall, we conclude that under a concurrent migration scenario, our approach remains highly scalable both with respect to migration time and network traffic, while having a minimal on performance.

5.5 Impact on real life applications

Our next series of experiments illustrates the behavior of our proposal in real life. For this purpose we have chosen *CM1*, a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. This application is used to study small-scale processes that occur in the atmosphere of the Earth, such as hurricanes.

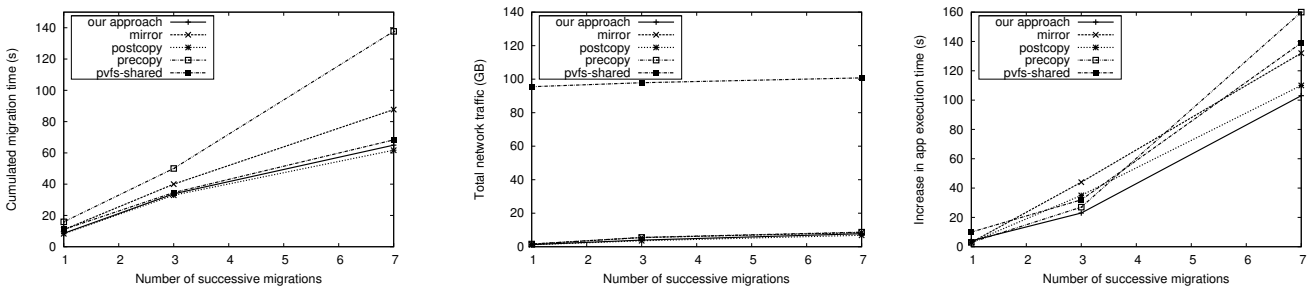
CM1 is representative of a large class of HPC stencil applications that model a phenomenon in time which can be described by a spatial domain that holds a fixed set of parameters in each point. The problem is solved iteratively in a distributed fashion by splitting the spatial domain into subdomains, each of which is managed by a dedicated MPI process. At each iteration, the MPI processes calculate the values for all points of their subdomain, then exchange the values at the border of their subdomains with each other, which is a highly network intensive process. After a certain number of iterations was successfully completed, each MPI process dumps the values of the subdomain it is responsible for into a file on the local storage, which generates a moderately intensive I/O write pressure. These files are then asynchronously collected and processed in order to visualize the evolution of the phenomenon in time. For the purpose of this work, we omitted the visualization part.

The experiment consists in deploying a fixed number of 64 sources, each of which hosts a VM instance that runs a process of CM1. The memory size of each instance is 4 GB, while the number of allocated cores is 2. As input data for CM1, we have chosen a 3D hurricane that is a version of the Bryan and Rotunno simulations [9]. We split the spatial domain into 8x8 subdomains, each of which has a size of 200x200. The output frequency is set at 30 seconds of simulated time, which for this configuration roughly translates to 40 seconds of computation time, during which approx. 200 MB of data per process are generated. While CM1 is running, we perform an increasing number of live migrations, starting from 1 to 7. The migrations are initiated successively at an interval of 60 seconds in the following pattern: source 1 is migrated to a target node after 60 seconds, source 2 is migrated to a target node after 120 seconds, etc. This simulates a highly dynamic datacenter where live migrations happen frequently.

The *cumulated* migration time, i.e. the sum of the migration time from all sources is depicted in Figure 5(a). As expected, all five approaches exhibit a linear trend in growth as the number of successive migrations increases. Interestingly enough, our approach outperforms pvfs−shared by a small margin, despite transferring local storage in addition to memory. This effect can be traced back to the lower I/O throughput sustained by pvfs−shared, which ultimately impacts the memory access pattern in a way that generates more memory transfer overhead than our own approach. Compared to precopy, we observe a steady decrease in cumulated migration time of about 2x. The same trend is visible when comparing to mirror: we observe a decrease of up to 33%. This indicates that mirroring significantly improves precopy in practice, but is still far from postcopy, which is close to our own approach.

The network traffic incurred by live migrations is shown in Figure 5(b). Since CM1 generates network traffic during normal operation, we subtracted this amount from the total observed network traffic in order to obtain the network traffic that can be traced back to live migration. As can be noticed, a huge gap exists between pvfs−shared and the rest of approaches. Thanks to local storage, this translates to more than 90% less network traffic. Since CM1 does not overlap computation with I/O, precopy performs much closer to the other approaches than in our AsyncWR benchmark. Still, our approach outperforms precopy and mirror by an overall 10-15% less network traffic overhead. However, it is outperformed by postcopy in its turn by 12%.

Finally, the impact on application performance is shown in Figure 5(c). As can be observed, live migration introduces a considerable increase in execution time that even surpasses the cumulated migration time, despite the fact that the application was not interrupted. This shows how sensitive HPC workloads are to performance degradation (one single slow VM can drag all other VMs down), underling the importance of minimizing the negative impact of



(a) Cumulative live migration time (lower is better)

(b) Network traffic, excluding CM1 communication (lower is better)

(c) Increase in application execution time (lower is better)

Figure 5: Performance of CM1 when performing an increasing number of live migrations separated by a one minute interval

live migration. In this context, our approach generates up to 10% less increase in total execution time when compared to postcopy. This number grows as high as 40% when compared to pvfs−shared and mirror, which is further augmented up to 62% when compared to precopy.

When further increasing the number of live migrations, an even higher gap can be expected.

6. CONCLUSIONS

Live migration is a key feature of virtualization. It enables a large variety of management tasks (such as load balancing, offline maintenance, power management and pro-active fault tolerance) that are critical in the maintenance of large IaaS cloud datacenters that run scientific data-intensive applications. In such datacenters, virtual machines often take advantage of locally available storage space in order to efficiently handle I/O intensive workloads. However, this poses a difficult challenge for live migration.

In this paper, we have presented a storage transfer proposal for live migration that is highly efficient under such circumstances. Unlike other state-of-art live migration approaches, we propose a memory-independent approach that relies on a hybrid active push-prioritized prefetch strategy. This enables us to separate the memory transfer strategy from the storage transfer strategy, enabling high migration flexibility for I/O intensive workloads regardless of memory access pattern.

We demonstrated the benefits of our approach through experiments that involve hundreds of nodes, using both benchmarks and real applications. When pushed to the extreme, such as the live migration of I/O benchmarks, our approach finished the migration up to 10x faster, consumed up to 10x less bandwidth and sustained the highest I/O write throughput inside the VM instance when compared to other state-of-art approaches, while showing up to 8x less negative impact on application performance.

Overall, we argue that synchronization through a parallel file system as an alternative to local storage is not scalable and expensive both in terms of performance and resources. Furthermore, we believe that the wide adoption of I/O pre-copy in practice as a consequence of its perceived higher safety (i.e. tolerates the failure of the destination during migration) does not justify the performance penalty and extra consumed resources in the context of large-scale scientific applications (which already implement fault-tolerant protocols). Finally, we argue that pure I/O post-copy is not enough on its own and can be further augmented to better adapt to the migration process, as illustrated by our approach.

Based on these results, we plan to explore the problem of storage transfer for live migration more extensively. In particular, we did not find acceptable implementations of alternate memory transfer techniques in practice (e.g. post-copy), but plan to experiment how our approach behaves in such a context on the first occasion. This might reveal interesting directions with respect to optimizing the storage transfer. Furthermore, we plan to study techniques such as de-duplication to further reduce the migration cost and potentially improve performance despite extra computational overhead. Finally, we plan to monitor I/O patterns with the purpose of predicting the best moment to initiate a live migration. Such information could be leveraged by the cloud middleware to better orchestrate live migrations within the datacenter.

Acknowledgments

We thank Eugen Feller for his remarks on the usefulness of live migration in the context of power management. This work was supported in part by the ANR MAPREDUCE, ANR-JST FP3C, ANR RESCUE projects and the Joint Laboratory for Petascale Computing (INRIA, UIUC, NCSA). The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS, RENATER and other contributing partners (see http://www.grid5000.fr/).

7. REFERENCES

[1] File System in UserspacE (FUSE). http://fuse.sourceforge.net.

[2] IOR. http://sourceforge.net/projects/ior-sio/.

[3] QEMU/KVM. http://www.linux-kvm.org.

[4] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. Vmflock: virtual machine co-migration for the cloud. In *HPDC '11: Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pages 159–170, San Jose, USA, 2011.

[5] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[6] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[7] 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. *Commun. ACM*, 53:50–58, April 2010.

[8] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 169–179, San Diego, USA, 2007.

[9] G. H. Bryan and R. Rotunno. The maximum intensity of tropical cyclones in axisymmetric numerical model simulations. *Journal of the American Meteorological Society*, 137:1770–1789, 2009.

[10] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, USA, 2000.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, pages 273–286, Boston, USA, 2005.

[12] M. Gagné. Cooking with Linux—still searching for the ultimate Linux distro? *Linux J.*, 2007(161):9, 2007.

[13] J. G. Hansen and E. Jul. Scalable virtual machine storage using local disks. *SIGOPS Oper. Syst. Rev.*, 44:71–79, December 2010.

[14] K. Haselhorst, M. Schmidt, R. Schwarzkopf, N. Fallenbeck, and B. Freisleben. Efficient storage synchronization for live migration in cloud infrastructures. In *PDP '11: Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 511–518, Ayia Napa, Cyprus, 2011.

[15] M. R. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43:14–26, July 2009.

[16] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman. Optimized pre-copy live migration for memory intensive applications. In *SC '11: 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 40:1–40:11, Seattle, USA, 2011.

[17] H. A. Lagar-Cavilla, J. A. Whitney, R. Bryant, P. Patchin, M. Brudno, E. de Lara, S. M. Rumble, M. Satyanarayanan, and A. Scannell. Snowflock: Virtual machine cloning as a first-class cloud primitive. *ACM Trans. Comput. Syst.*, 29:2:1–2:45, February 2011.

[18] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC '09: Proceedings the 18th ACM international symposium on High Performance Distributed Computing*, pages 101–110, Garching, Germany, 2009.

[19] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai. The design and evolution of live storage migration in VMware ESX. In *USENIX ATC '11: Proceedings of the 2011 USENIX Annual Technical Conference*, pages 1–14, Portland, USA, 2011.

[20] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*, pages 23–32, Seattle, USA, 2007.

[21] R. Nathuji and K. Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. In *SOSP '07: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 265–278, Stevenson, USA, 2007.

[22] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the 2005 USENIX Annual Technical Conference*, pages 1–25, Anaheim, USA, 2005.

[23] B. Nicolae. *BlobSeer: Towards Efficient Data Storage Management for Large-Scale, Distributed Systems*. PhD thesis, University of Rennes 1, November 2010.

[24] B. Nicolae. On the benefits of transparent compression for cost-effective cloud data storage. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 3:167–184, 2011.

[25] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. Blobseer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, 71:169–184, 2011.

[26] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu. Going back and forth: Efficient multideployment and multisnapshotting on clouds. In *HPDC '11: 20th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 147–158, San José, USA, 2011.

[27] B. Nicolae and F. Cappello. BlobCR: Efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots. In *SC '11: 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 34:1–34:12, Seattle, USA, 2011.

[28] E. Park, B. Egger, and J. Lee. Fast and space-efficient virtual machine checkpointing. In *VEE '11: Proceedings of the 7th International Conference on Virtual Execution Environments*, pages 75–86, Newport Beach, USA, 2011.

[29] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *VEE '11: Proceedings of the 7th International Conference on Virtual Execution Environments*, pages 111–120, Newport Beach, USA, 2011.

[30] C. Tang. Fvd: a high-performance virtual machine image format for cloud. In *ATEC '11: Proc. of the 2011 USENIX Annual Technical Conference*, pages 1–18, Portland, USA, 2011.

[31] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing*, pages 254–265, Beijing, China, 2009.