# Towards Efficient Live Migration of I/O Intensive Workloads: A Transparent Storage Transfer Proposal

Bogdan Nicolae, Franck Cappello

Joint INRIA-UIUC Laboratory on PetaScale Computing

# Towards Efficient Live Migration of I/O Intensive Workloads: A Transparent Storage Transfer Proposal

Bogdan Nicolae, Franck Cappello

Technical Report TR-JLPC-11-11

# Towards Efficient Live Migration of I/O Intensive Workloads: A Transparent Storage Transfer Proposal

Bogdan Nicolae[1], Franck Cappello [1,2]
bogdan.nicolae@inria.fr, fci@lri.fr

[1] INRIA Saclay, France
[2] University of Illinois at Urbana-Champaign, USA

**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 I/O intensive workloads 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 completely hypervisor-transparent 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, transparency ensures 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, 5x less bandwidth consumption and 62% less performance degradation over state-of-art.

## 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 [1, 2]. 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* [3] 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 [4]. 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 pro-actively moved to safer locations [5]. 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 [6]: VMs need a "scratch space", i.e. a place where to store temporary data generated during their runtime. For this purpose, cloud providers typically install local disks on the physical machine that are dedicated for this purpose. 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 the destination host while the VMs keep running and may be altering the disk state.

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 that involve the scratch space. 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 [7], 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 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. For this purpose, we rely on a series of building blocks covered by our previous work: BlobSeer, a distributed storage repository specifically designed for high throughput under concurrency [8, 9], as well as an associated FUSE-based mirroring module that exposes VM disk images as regular files to the hypervisor [10]. (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. Thus, transferring it to the destination can be considered negligible. 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, in practice a VM is not working in isolation: it may host a server or be part of a distributed application, etc. Thus, offline migration causes an unacceptably long downtime during which the VM is not accessible from the outside, potentially leading to service loss, violation of service level agreement and failures.

To alleviate this issue, *live migration* [3] was introduced, enabling 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.

The key challenge of live migration is to keep a *consistent view* of memory and storage at all times for the VM, while converging 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 have been extensively studied for memory, 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, 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.

With a growing tendency for VMs to exhibit I/O intensive workloads, live migration needs to efficiently tolerate such scenarios. 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 indicators. Our goal is to optimize the storage transfer according to these indicators:

**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 [11, 12], dedicated virtual disk storage services based on shadowing and cloning [10], fork-consistent replication systems based on log-structuring [13], de-duplication of memory pages [14]. Many times, 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 [15].

Extensive live migration research was done for memory-to-memory transfer. The *pre-copy* strategy [3, 16] 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 are used to reduce the overhead incurred by the background transfer of memory pages, such as delta compression [17].

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, several approaches were proposed. Checkpoint/Restart and Log/Replay was successfully adopted by [18] to significantly reduce downtime and network bandwidth consumption over pre-copy. A *post-copy* strategy was proposed in [19]. Unlike pre-copy, it transfers control 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 are also developed in [20]. In this work, the authors propose VM cloning as 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 [21]. 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.

In [22], the authors 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 is proposed in [23]: 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 [22], confirmation is waited from the destination before the write completes on the source. 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.

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* [24] 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

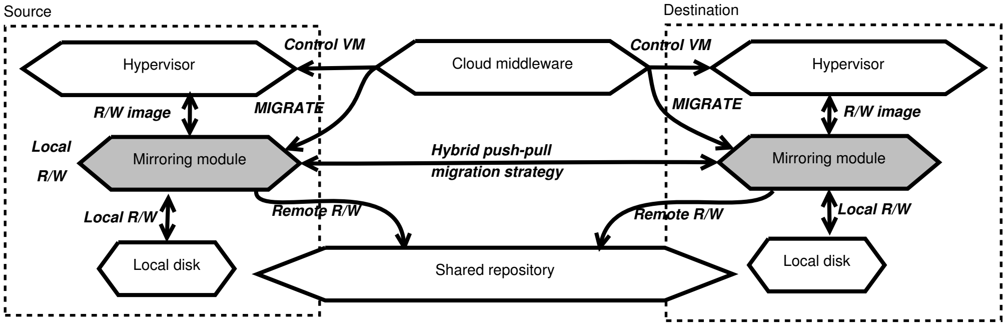


Figure 1: Cloud architecture that integrates our approach via the mirroring module (dark background).

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 *mirroring module*, which is the central actor of our approach and is responsible to implement our live storage migration strategy.

Under normal operation, the mirroring module 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 mirroring module 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 mirroring module 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 mirroring module

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

The cloud middleware can send migration requests to the mirroring module using the MIGRATION_REQUEST primitive (Algorithm 1). Upon receipt of this event, the mirroring module 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 mirroring module 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 and the BACKGROUND_PUSH task is notified (potentially waking it 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.

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

**Algorithm 1** Migration request on the source

```
procedure MIGRATION_REQUEST($Destination$)
    $RemainingSet \leftarrow ModifiedSet$
    for all $c \in VirtualDisk$ do
        $WriteCount[c] \leftarrow 0$
    end for
    start BACKGROUND_PUSH
    $isSource \leftarrow$ true
    invoke MIGRATION_NOTIFICATION on $Destination$
    forward migration request to the hypervisor
    notify BACKGROUND_PUSH
end procedure
procedure BACKGROUND_PUSH
    while true do
        wait for notification
        while $\exists c \in RemainingSet : WriteCount[c] < Threshold$ do
            $buf \leftarrow$ contents of $c$
            push $(c, buf)$ to $Destination$
            $RemainingSet \leftarrow RemainingSet \setminus \{c\}$
        end while
    end while
end procedure
```

**Algorithm 2** Simplified writes of single full chunks

```
function WRITE($c$, $buffer$)
    if $isDestination$ then
        cancel any pull($c$) in progress
        $RemainingSet \leftarrow RemainingSet \setminus \{c\}$
    end if
    contents of $c \leftarrow buffer$
    $ModifiedSet \leftarrow ModifiedSet \cup \{c\}$
    if $isSource$ then
        $WriteCount[c] \leftarrow WriteCount[c] + 1$
        $RemainingSet \leftarrow RemainingSet \cup \{c\}$
        notify BACKGROUND_PUSH
    end if
    return success
end function
```

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

```
procedure MIGRATION_NOTIFICATION
    $isDestination \leftarrow$ true
    accept chunks from $Source$
end procedure
procedure TRANSFER_IO_CONTROL($RS, WC$)
    $RemainingSet \leftarrow RS$
    $WriteCount \leftarrow AC$
    start BACKGROUND_PULL
end procedure
procedure BACKGROUND_PULL
    while $RemainingSet \neq \emptyset$ do
        $c \leftarrow c' \in RemainingSet : WriteCount[c'] = max(WriteCount[RemainingSet])$
        $RemainingSet \leftarrow RemainingSet \setminus \{c\}$
        pull($c$) from $Source$
    end while
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.

**Algorithm 4** Simplified reads of single full chunks

```
function READ($c$)
    if $isDestination$ and $c \in RemainingSet$ then
        if $c$ is being pulled by BACKGROUND_PULL then
            wait until $c$ is available
        else
            suspend BACKGROUND_PULL
            pull($c$) from $Source$
            $RemainingSet \leftarrow RemainingSet \setminus \{c\}$
            resume BACKGROUND_PULL
        end if
    end if
    fetch $c$ from repository if $c$ not available locally
    return contents of $c$
end function
```

Once all remaining chunks have been pulled at the destination, the source is not needed anymore. Both the hypervisor and the mirroring module can be stopped and the source can be shut down (or its resources

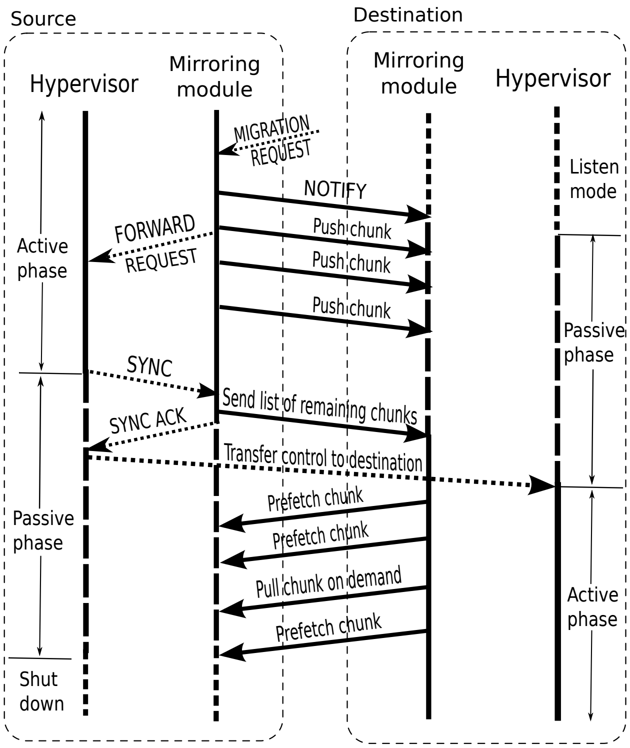


Figure 2: Overview of the live storage transfer as it progresses it time

used for other purposes). At this point, the live migration is complete.

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 mirroring modules. A dotted pattern is used to represent interactions between the mirroring module 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 *mirroring module* on top of *FUSE* (File System in UserspacE) [25]. 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 [10, 26]. The mirroring module 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 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 mirroring module is designed to integrate with *BlobSeer* [8, 9], 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 [27], an experimental testbed for distributed computing that federates nine sites in France. We used 144 nodes of the graphene cluster from the Nancy site, each of which is equipped with a quadcore 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).

The hypervisor running on all compute nodes is QEMU/KVM [21] 0.14.0, 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 three approaches throughout our evaluation:

### 5.2.1 Live storage transfer using our approach

In this setting, the VM instances write all modifications locally and the storage transfer is performed using our approach. We rely on BlobSeer to store base disk image and on the FUSE-based mirroring module (described in Section 4.4) to expose a locally modifiable view of the disk image to the hypervisor. We deploy BlobSeer in the following configuration: a version manager and a provider manager, each on a dedicated node, along with 10 metadata providers, again each on a dedicated node. The rest of 120 nodes are used as compute nodes. On each compute node we deploy a data provider and a mirroring module. Once all processes are deployed, we upload the base image into BlobSeer, using a stripe size of 256 KB (which we found to be large enough to avoid excessive fragmentation overhead, while being small enough to avoid contention under concurrent accesses). At this point, the VM instances can be deployed on the sources using the image locally exposed by the mirroring module. Any live migration will be performed in a two-step procedure: first the storage transfer is initiated on the mirroring module, then QEMU/KVM is instructed to initiate the transfer of memory. 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 pre-copy block migration

We compare our approach to the case when the modifications are locally stored using a qcow2 [11] disk snapshot, whose backing base disk image is shared through a parallel file system. In this case, the storage transfer is performed using QEMU/KVM's incremental block migration feature, which is representative of a pre-copy strategy. For the purpose of this work, we have chosen PVFS [28] as the parallel file system, as it implements high performance data striping and offers similar read performance to BlobSeer under concurrent accesses. PVFS is deployed on all nodes, out of which 120 are reserved as compute nodes. The base image is uploaded to PVFS using the same stripe size as in the case of our approach. Once this step completed, the VM instances are deployed on the sources using the qcow2 files as the underlying disk images. Live migrations are performed by instructing QEMU/KVM to perform incremental block migration in addition to the transfer of memory. For the rest of this paper, we refer to this setting as *pvfs-cow*.

### 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. This corresponds to a traditional solution that avoids storage transfer during live migration altogether in order to better tolerate live migration. However, not taking advantage of local storage can decrease performance and generates additional network traffic overhead. To study this trade-off, we use the same setting as above except for the fact that we store the qcow2 images in the PVFS deployment rather than locally. Live migrations are performed by instructing QEMU/KVM to initiate the transfer the memory only. For the rest of this paper, we refer to this setting as *pvfs-shared*.

These approaches 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 pfvs-cow and pvfs-shared, the live migration ends as soon as the control is transferred to the destination. For our approach, an additional time is required after the control was transferred in order 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. In the case of our approach and pvfs-cow, this means all traffic generated by the live migration, including traffic due to remote read accesses to BlobSeer and PVFS respectively. In the case of pvfs-shared, this is the traffic generated by the live migration of memory and all remote reads/writes to PVFS.

- *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 scientific data-intensive applications.

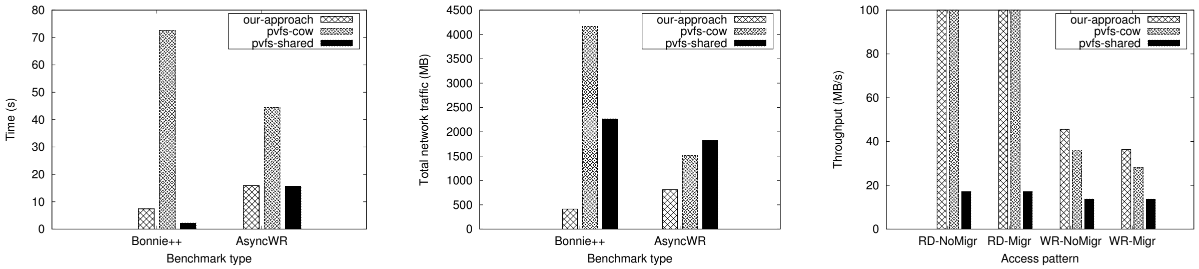
## 5.3 Live migration performance of I/O intensive benchmarks

Our first series of experiments evaluates the performance of live migration for two I/O intensive benchmarks: *Bonnie++* [29] and *AsyncWR*.

*Bonnie++* is a standard benchmarking tool that measures the I/O access performance to any mounted file system. It creates and writes a set of files that fill a large part of the remaining free space of the file system, then reads back the written data, and then overwrites the files with new data, recording throughput in all cases. Other performance factors such as how many files per second can be created and deleted are also recorded. Given the high I/O pressure generated by this workload, it is the ideal candidate to push all three approaches to their limits.

*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 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.

The experiment consists in launching each of the benchmarks inside a VM instance and then performing a live migration after a delay of 10 seconds. This gives the VM instance a small 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 total amount of data written by *Bonnie++*



(a) Migration time for Bonnie++ and AsyncWR

(b) Total network traffic for Bonnie++ and AsyncWR

(c) Performance degradation of Bonnie++: sustained read (RD) and write (WR) throughputs without (NoMigr) and with (Migr) migration

Figure 3: Migration performance of a VM instance (256MB of RAM) that performs I/O intensive workloads

is fixed at 800 MB. The same total amount of data is written by *AsyncWR* in 80 iterations (i.e. 10 MB per iteration). The amount of RAM available to the VM instance is fixed at 256 MB. We have chosen this small amount of RAM in order to limit the effects of write caching inside the VM instance, forcing the guest operating system to flush the written data to the virtual disk. At the same time, a low amount of RAM minimizes the impact of memory transfer.

The total migration time is depicted in Figure 3(a). As can be noticed, for the highly I/O intensive *Bonnie++* workload there is a large difference between the three 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 3x more time, which is more than 10x faster than pvfs-cow. As mentioned in Section 4.1, the reason for this large difference in migration time between our approach and pvfs-cow lies in the fact that the contents of the virtual disk changes faster than it can be pushed by the pre-copy strategy, leading to a scenario where new data written to the disk accumulates on the source and needs to be eventually migrated. This effect is further accentuated by the repeated transfer of modified data, which our approach avoids. Furthermore, our approach transfers control to the destination as early as possible, which enables new data to accumulate directly on the destination, greatly reducing migration time. The same effect is noticeable in the *AsyncWR* workload as well, albeit at lesser extent: in this case our approach achieves a speed-up of 2.8x over pvfs-cow. Thanks to the early transfer of control to the destination, our approach needs to pull only a small amount of data, enabling it to perform almost as well as pvfs-shared.

The high overhead exhibited by the pre-copy strategy is also easily visible in the total network traffic that was generated during the live migration. As can be observed in Figure 3(b), for the *Bonnie++* benchmark the amount of network traffic reaches well over 4 GB. This is 8x more than the amount of network traffic generated by our approach, which is less than 500 MB. Furthermore, it can also be observed that the advantage of pvfs-shared in terms of migration time comes at the expense of network traffic: our approach manages to outperform it by almost 5x. For the *AsyncWR* benchmark, this effect is even more visible: pvfs-shared performs the worst with a total network traffic of 1.5 GB. It is closely followed by pvfs-cow, whose network traffic is considerably less that in the case of *Bonnie++* due to lower I/O pressure. Thanks to its fast transfer of control to the destination, our approach manages to take advantage of local storage at the destination much faster than pvfs-cow, thus consuming around 2x less network traffic than pvfs-cow and even more than 2x when compared to pvfs-shared.

Finally, the impact of live migration on the performance results of *Bonnie++* and *AsyncWR* is illustrated in Figure 3(c) and Table 1 respectively.

For the *Bonnie++* benchmark, a large gap can be observed between pvfs-shared and the other two approaches when comparing the sustained write throughput with and without live migration. At only

Table 1: Performance degradation of AsyncWR under live migration

| Approach | Without migration | With migration |
|---|---|---|
| our approach | 7.7 MB/s | 3.6 MB/s |
| pvfs-cow | 7.7 MB/s | 3.16 MB/s |
| pvfs-shared | 2.8 MB/s | 2.66 MB/s |

15 MB/s, this low throughput is easily explained by the fact that pfvs-shared does not take advantage of local storage, thus experiencing a significant slowdown vs. our approach and pvfs-cow (3x and 2.5x respectively). Our FUSE-based mirroring module achieves a high write throughput that reaches about 45 MB/s without migration and then drops by 20% when performing the live migration. By comparison, pvfs-cow reaches a write throughput of 40 MB/s without migration (slightly lower than our approach because of additional copy-on-write management overhead) that eventually drops by 30% when performing the live migration. The higher drop in performance experienced by pvfs-cow is again explained by the high overhead of its pre-copy strategy. On the other hand, as expected, the write throughput sustained by pvfs-shared during live migration suffers no noticeable drop.

The gap between pvfs-shared and the other two approaches grows even higher when comparing the sustained read throughput. This is explained by the fact that both the source and the destination aggressively cache the contents that is written locally (which is independent of the caching performed inside the VM instance). This is not the case for pvfs-shared, contributing to its throughput of about 17 MB/s. Comparatively, both our approach and pvfs-shared reach well over 600 MB/s, greatly surpassing the maximal value of 100 MB/s used in Figure 3(c) for clarity. For all three approaches, the performance drop of read throughput caused by live migration is negligible.

In the case of *AsyncWR*, the drop in sustained write throughput when performing live migration is significantly higher than in the case of *Bonnie++*. Both our approach and pvfs-cow drop from 7.7 MB/s to 3.6 MB/s (55%) and 2.8 MB/s (64%) respectively. This higher drop in performance is explained by the fact that unlike *Bonnie++*, 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 a lower bandwidth available for storage migration and thus the observed effect. The higher overhead of memory transfer is noticeable in the case of pvfs-shared too, where we recorded a drop from 2.8 MB/s to 2.66 MB/s. However, compared to the other two approaches this drop is significantly lower.

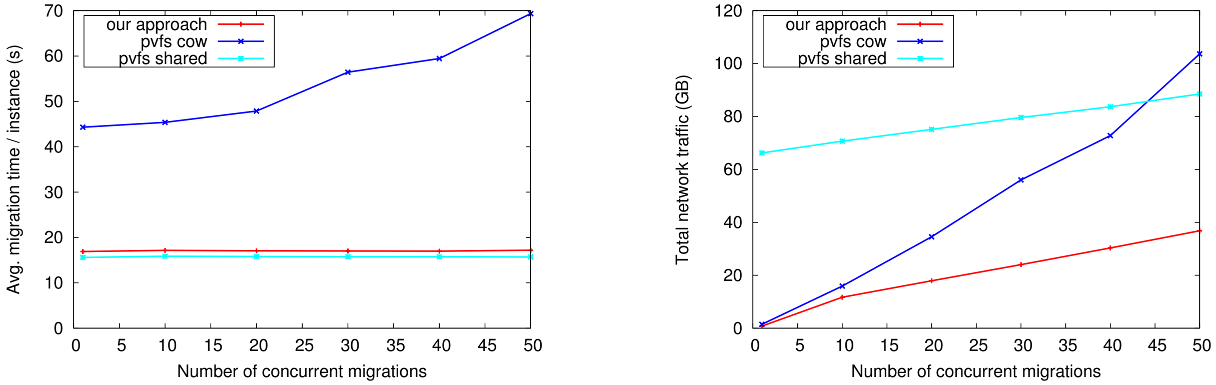
## 5.4 Performance of concurrent live migrations

Our next series of experiments aims to evaluate the performance of all three 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 50 and gradually increase the number of destinations from 1 to 50, in steps of 10. On all sources we launch the *AsyncWR* benchmark, wait until a warm-up period of 10 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 800 MB, while the amount of RAM available to the VM instance is fixed at 256 MB.

As can be observed in Figure 4(a), with increasing number of live migrations, both our approach and pvfs-shared keep a constant average migration time, which is only slightly higher for our approach. On the other hand, pvfs-cow experiences a steady increase in average migration time, which reaches almost 35% for 50 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, pvfs-cow experiences a sharp increase in network traffic, whereas our approach and pvfs-shared experience a much milder trend. Both our approach and pvfs-cow 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



(a) Average migration time

(b) Network traffic generated during the experiment

Figure 4: Performance of AsyncWR when increasing the number of concurrent live migrations from 1 to 50

insufficient to accommodate the instantaneous needs of pvfs-cow, a slowdown in transfer speed occurs when increasing the number of live migrations, which ultimately reflects into increased average migration time.

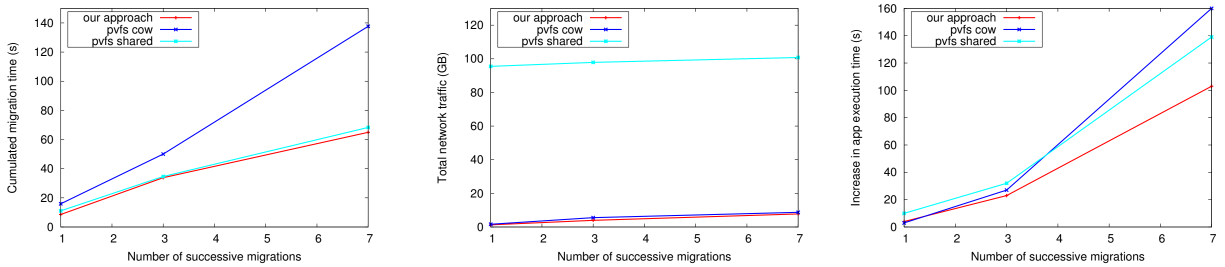
Thanks to earlier transfer of control to the destination, our approach enables new data to be generated directly at the destination, greatly reducing the network traffic generated by storage migration which enables it to avoid reaching the system bandwidth limit. Actually, most of the network traffic generated by our approach is caused by the memory transfer rather than storage migration. This is observable when comparing our approach to pvfs-shared: If we subtract from the curve corresponding to pvfs-shared the network traffic generated by interactions with PVFS (which remains constant at around 62 GB/s), we obtain the traffic generated by the transfers of memory. The resulting curve is very similar to the one corresponding to our own approach, hinting at low storage migration overhead. Note that although very high, the network traffic generated by pvfs-shared is evenly distributed throughout the experiment and thus pvfs-shared remains scalable with respect to migration time (unlike pvfs-cow).

Overall, we conclude that under a concurrent migration scenario, our approach remains highly scalable both with respect to migration time and network traffic. At 50 concurrent live migrations, it is faster than pvfs-cow by 4.6x, and generates 2.5x less network traffic. Furthermore, thanks to local storage it consumes 2x less overall network traffic than pvfs-shared.

### 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,



(a) Cumulative live migration time

(b) Network traffic generated during the experiments (excluding traffic generated by CM1 itself)

(c) Increase in application execution time

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

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 an MPI process of CM1. The memory size of each instance is 1 GB. As input data for CM1, we have chosen a 3D hurricane that is a version of the Bryan and Rotunno simulations [30]. 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 three 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 pvfs-cow, we observe a steady decrease in cumulated migration time of about 2x.

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 our approach / pvfs-cow. Thanks to local storage, both approaches generate more than 90% less network traffic. Since CM1 does not overlap computation with I/O, pvfs-shared and pvfs-cow perform much closer that in our AsyncWR benchmark. Still, our approach outperforms pvfs-cow by an overall 15% less network traffic overhead.

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 live migration. In this context, our approach generates up to 40% less increase in total execution time compared to pvfs-shared. This number grows as high as 62% when compared to pvfs-cow. 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. 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 approaches that require the storage on the source and destination to be synchronized before control can be transferred to the destination, we propose a completely hypervisor-transparent approach that relies on a hybrid active push-prioritized prefetch strategy. This makes our approach highly resilient to rapid changes of disk state which are exhibited by I/O intensive workloads, while keeping a maximum of portability with a wide range of hypervisors.

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 5x less bandwidth and sustained the highest I/O throughput inside the VM instance when compared to other state-of-art approaches, suffering only a 20% drop in I/O performance due to migration. Furthermore, in a scenario of concurrent live migrations, it demonstrated excellent scalability with respect to migration time, while consuming half of the bandwidth required by other approaches. Finally, in a real life scenario that involves HPC stencil applications, we have shown a 2x decrease in cumulative migration time for successive migrations, while consuming up to 90% less bandwidth and causing up to 62% less overall increase in execution time.

Based on these results, we plan to explore the problem of storage transfer for live migration more extensively. In particular, there are two directions we consider. First, we plan to actively involve the cloud repository in the live migration by offloading some chunks modified on the source during live migration there. This could potentially decrease the migration time because of less dependencies on the source after transfer of control. However, if such chunks are modified too often, the source might see a performance drop, resulting in a trade-off that needs to be further analyzed. Second, 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 on the datacenter.

## Acknowledgments

This work was supported in part by the Agence Nationale de la Recherche (ANR) under Contract ANR-10-01-SEGI and the Joint Laboratory for Petascale Computing, an INRIA-UIUC initiative. 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 and RENATER and other contributing partners (see http://www.grid5000.fr/).

## References

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, 2009.

[2] 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*, vol. 53, pp. 50–58, April 2010.

[3] 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 - Volume 2*, Boston, USA, 2005, pp. 273–286.

[4] 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*, Stevenson, USA, 2007, pp. 265–278.

[5] 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*, Seattle, USA, 2007, pp. 23–32.

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

[7] 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*, Beijing, China, 2009, pp. 254–265.

[8] B. Nicolae, "Blobseer: Towards efficient data storage management for large-scale, distributed systems," Ph.D. dissertation, University of Rennes 1, November 2010.

[9] 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.*, vol. 71, pp. 169–184, 2011.

[10] 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*, San José, USA, 2011, pp. 147–158.

[11] M. Gagné, "Cooking with Linux—still searching for the ultimate Linux distro?" *Linux J.*, vol. 2007, no. 161, p. 9, 2007.

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

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

[14] 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*, Newport Beach, USA, 2011, pp. 75–86.

[15] 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*, Seattle, USA, 2011, in press.

[16] 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*, Anaheim, USA, 2005, pp. 1–25.

[17] 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*, Newport Beach, USA, 2011, pp. 111–120.

[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*, Garching, Germany, 2009, pp. 101–110.

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

[20] 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.*, vol. 29, pp. 2:1–2:45, February 2011.

[21] "Qemu/kvm," http://www.linux-kvm.org.

[22] 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*, San Diego, USA, 2007, pp. 169–179.

[23] 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*, Ayia Napa, Cyprus, 2011, pp. 511–518.

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

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

[26] B. Nicolae, F. Cappello, and G. Antoniu, "Optimizing multi-deployment on clouds by means of self-adaptive prefetching," in *Euro-Par '11: 17th International Euro-Par Conference on Parallel Processing*, Bordeaux, France, 2011, pp. 503–513.

[27] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 481–494, November 2006.

[28] 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*, Atlanta, USA, 2000, pp. 317–327.

[29] B. Martin, "Using Bonnie++ for filesystem performance benchmarking," *Linux.com*, vol. Online edition, 2008.

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