Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds

Bogdan Nicolae, John Bresnahan, Kate Keahey, Gabriel Antoniu

Going Back and Forth: Efficient Multideployment and Multisnapshotting on Clouds

Bogdan Nicolae
INRIA Saclay
France
bogdan.nicolae@inria.fr

John Bresnahan
Argonne National Laboratory
USA
bresnaha@mcs.anl.gov

Kate Keahey
Argonne National Laboratory
USA
keahey@mcs.anl.gov

Gabriel Antoniu
INRIA Rennes
France
gabriel.antoniu@inria.fr

ABSTRACT

Infrastructure as a Service (IaaS) cloud computing has revolutionized the way we think of acquiring resources by introducing a simple change: allowing users to lease computational resources from the cloud provider's datacenter for a short time by deploying virtual machines (VMs) on these resources. This new model raises new challenges in the design and development of IaaS middleware. One of those challenges is the need to deploy a large number (hundreds or even thousands) of VM instances simultaneously. Once the VM instances are deployed, another challenge is to simultaneously take a snapshot of many images and transfer them to persistent storage to support management tasks, such as suspend-resume and migration. With datacenters growing rapidly and configurations becoming heterogeneous, it is important to enable efficient concurrent deployment and snapshotting that are at the same time hypervisor independent and ensure a maximum compatibility with different configurations. This paper addresses these challenges by proposing a virtual file system specifically optimized for virtual machine image storage. It is based on a lazy transfer scheme coupled with object versioning that handles snapshotting transparently in a hypervisor-independent fashion, ensuring high portability for different configurations. Large-scale experiments on hundreds of nodes demonstrate excellent performance results: speedup for concurrent VM deployments ranges from a factor of 2 up to 25, with a reduction in bandwidth utilization of as much as 90%.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Experimentation

HPDC'11, June 8–11, 2011, San Jose, California, USA.

Keywords

large scale, virtual machine images, deployment, snapshotting, lazy propagation, versioning, cloning

1. INTRODUCTION

In recent years, Infrastructure as a Service (IaaS) cloud computing [30] has emerged as a viable alternative to the acquisition and management of physical resources. With IaaS, users can lease storage and computation time from large datacenters. Leasing of computation time is accomplished by allowing users to deploy virtual machines (VMs) on the datacenter's resources. Since the user has complete control over the configuration of the VMs using on-demand deployments [5, 16], IaaS leasing is equivalent to purchasing dedicated hardware but without the long-term commitment and cost. The on-demand nature of IaaS is critical to making such leases attractive, since it enables users to expand or shrink their resources according to their computational needs, by using external resources to complement their local resource base [20].

This emerging model leads to new challenges relating to the design and development of IaaS systems. One of the commonly occurring patterns in the operation of IaaS is the need to deploy a large number of VMs on many nodes of a datacenter at the same time, starting from a set of VM images previously stored in a persistent fashion. For example, this pattern occurs when the user wants to deploy a virtual cluster that executes a distributed application or a set of environments to support a workflow. We refer to this pattern as *multideployment*.

Such a large deployment of many VMs at once can take a long time. This problem is particularly acute for VM images used in scientific computing where image sizes are large (from a few gigabytes up to more than 10 GB). A typical deployment consists of hundreds or even thousands of such images. Conventional deployment techniques [31] broadcast the images to the nodes before starting the VM instances, a process that can take tens of minutes to hours, not counting the time to boot the operating system itself. This can make the response time of the IaaS installation much longer than acceptable and erase the on-demand benefits of cloud computing.

Once the VM instances are running, a similar challenge applies to snapshotting the deployment: many VM images

that were locally modified need to be concurrently transferred to stable storage with the purpose of capturing the VM state for later use (e.g., for checkpointing or off-line migration to another cluster or cloud). We refer to this pattern as *multisnapshotting*. Conventional snapshotting techniques rely on custom VM image file formats [12] to store only incremental differences in a new file that depends on the original VM image as the backing file. When taking frequent snapshots for a large number of VMs, such approaches generate a large number of files and interdependencies among them, which are difficult to manage and which interfere with the ease-of-use rationale behind clouds. Furthermore, with growing datacenter trends and tendencies to federate clouds [17], configurations are becoming more and more heterogeneous. Custom image formats are not standardized and can be used with specific hypervisors only, which limits the ability to easily migrate VMs among different hypervisors.

Therefore, multisnapshotting must be handled in a transparent and portable fashion that hides the interdependencies of incremental differences and exposes standalone VM images, while keeping maximum portability among different hypervisor configurations.

In addition to incurring significant delays and raising manageability issues, these patterns may also generate high network traffic that interferes with the execution of applications on leased resources and generates high utilization costs for the user.

This paper proposes a distributed virtual file system specifically optimized for both the multideployment and multisnapshotting patterns. Since the patterns are complementary, we investigate them in conjunction. Our proposal offers a good balance between performance, storage space, and network traffic consumption, while handling snapshotting transparently and exposing standalone, raw image files (understood by most hypervisors) to the outside.

Our contributions are can be summarized as follows:

- We introduce a series of design principles that optimize multideployment and multisnapshotting patterns and describe how our design can be integrated with IaaS infrastructures (Sections 2 and 3).
- We show how to realize these design principles by building a virtual file system that leverages versioning-based distributed storage services. To illustrate this point, we describe an implementation on top of BlobSeer, a versioning storage service specifically designed for high throughput under concurrency [23, 24].
- We evaluate our approach in a series of experiments, each conducted on hundreds of nodes provisioned on the Grid'5000 testbed, using both synthetic traces and real-life applications.

2. INFRASTRUCTURE AND APPLICATION MODEL

In order to reason about the challenges presented in the previous sections, several important aspects need to be modeled.

2.1 Cloud infrastructure

IaaS platforms are typically built on top of clusters made out of loosely-coupled commodity hardware that minimizes per unit cost and favors low power over maximum speed [5]. Disk storage (cheap hard-drives with capacities in the order of several hundred GB) is attached to each machine, while the machines are interconnected with standard Ethernet links. The machines are configured with proper virtualization technology, in terms of both hardware and software, such that they are able to host the VMs. In order to provide persistent storage, a dedicated repository is deployed either as centralized [3] or as distributed [6] storage service running on dedicated storage nodes. The repository is responsible for storing the VM images persistently in a reliable fashion and provides the means for users to manipulate them: upload, download, delete, and so forth. With the recent explosion in cloud computing demands, there is an acute need for scalable storage [7].

2.2 Application state

The state of the VM deployment is defined at each moment in time by two main components: the state of each of the VM instances and the state of the communication channels between them (opened sockets, in-transit network packets, virtual topology, etc.).

Thus, in the most general case (*Model 1*), saving the application state implies saving both the state of all VM instances and the state of all active communication channels among them. While several methods have been established in the virtualization community to capture the state of a running VM (CPU registers, RAM, state of devices, etc.), the issue of capturing the global state of the communication channels is difficult and still an open problem [19].

In order to avoid this issue, the general case is usually simplified such that the application state is reduced to the sum of states of the VM instances (*Model 2*). Any in-transit network traffic is discarded, under the assumption that a fault-tolerant networking protocol is used that is able to restore communication channels and resend lost information.

Even so, for VM instances that need large amounts of memory, the necessary storage space can explode to huge sizes. For example, saving 2 GB of RAM for 1,000 VMs consumes 2 TB of space, which is unacceptable for a single one-point-in-time deployment checkpoint. Therefore, Model 2 can further be simplified such that the VM state is represented only by the virtual disk attached to it (*Model 3*), which is used to store only minimal information about the state, such as configuration files that describe the environment and temporary files that were generated by the application. This information is then later used to reboot and reinitialize the software stack running inside the VM instance. Such an approach has two important practical benefits: (1) huge reductions in the size of the state, since the contents of RAM, CPU registers, and the like does not need to be saved; and (2) portability, since the VM can be restored on another host without having to worry about restoring the state of hardware devices that are not supported or are incompatible between different hypervisors.

Since Model 3 is the most widely used checkpointing mechanism in practice, we consider the multisnapshotting pattern for Model 3. Note, however, that our approach can be used for Model 2 without modification and is furthermore easy to extend to Model 1 by adding a mechanism to capture and restore the global state of the communication channels.

2.3 Application access pattern

A VM typically does not access the whole initial image. For example, it may never access some applications and utilities that are installed by default with the operating system. In order to model this aspect, it is useful to analyze the life-cycle of a VM instance, which consists of three phases:

- *Boot phase*: involves reading configuration files and launching processes, which translates to random small reads and writes from/to the VM disk image acting as the initial state.
- *Application phase*: translates to either (1) negligible virtual disk access (e.g., CPU-intensive applications that do not require persistent storage or data-intensive applications that rely on dedicated storage services, such as Amazon S3 [6]) or (2) read-your-writes virtual disk access (e.g web server deployment where each web server writes and reads back log files and object caches).
- *Shutdown phase*: generates negligible disk access to the image and is completely missing if the VM instance was terminated prematurely (e.g., because of a hardware failure).

3. OUR APPROACH

We propose a virtual file system aimed at optimizing the multi-deployment and multi-snapshotting patterns based on the observations presented in Section 2.

3.1 Design overview

We rely on four key principles: aggregate the storage space, optimize VM disk access, reduce contention, and optimize multisnapshotting.

3.1.1 Aggregate the storage space locally available on the compute nodes

In most cloud deployments [5, 3, 4], the disks locally attached to the compute nodes are not exploited to their full potential. Most of the time, such disks are used to hold local copies of the images corresponding to the running VMs, as well as to provide temporary storage for them during their execution, which utilizes only a small fraction of the total disk size.

We propose to aggregate the storage space from the compute nodes in a shared common pool that is managed in a distributed fashion, on top of which we build our virtual file system. This approach has two key advantages. First, it has a potential for high scalability, as a growing number of compute nodes automatically leads to a larger VM image repository, which is not the case if the repository is hosted by dedicated machines. Second, it frees a large amount of storage space and overhead related to VM management on dedicated storage nodes, which can improve performance and/or quality-of-service guarantees for specialized storage services that the applications running inside the VMs require and are often offered by the cloud provider (e.g., database engines, distributed hash tables, special purpose file systems, etc.)

An important issue in this context is to be able to leverage the storage space provided by the local disks without interfering with the normal VM execution. For this reason, only a part of the local disk is allocated to the common pool, while the rest is freely usable by the hypervisor and the VMs.

3.1.2 Optimize VM disk access by using on-demand image mirroring

When a new VM needs to be instantiated, the underlying VM image is presented to the hypervisor as a regular file accessible from the local disk. Read and write accesses to the file, however, are trapped and treated in a special fashion. A read that is issued on a fully or partially empty region in the file that has not been accessed before (by either a previous read or write) results in fetching the missing content remotely from the VM repository, mirroring it on the local disk and redirecting the read to the local copy. If the whole region is available locally, no remote read is performed. Writes, on the other hand, are always performed locally.

3.1.3 Reduce contention by striping the image

Each VM image is split into small, equal-sized chunks that are evenly distributed among the local disks participating in the shared pool. When a read accesses a region of the image that is not available locally, the chunks that hold this region are determined and transferred in parallel from the remote disks that are responsible for storing them. Under concurrency, this scheme effectively enables the distribution of the I/O workload, because accesses to different parts of the image are served by different disks.

Even in the worst-case scenario when all VMs read the same chunks in the same order concurrently (for example, during the boot phase), there is a high chance that the accesses get skewed and thus are not issued at exactly the same time. This effect happens for various reasons, such as different hypervisor initialization overhead and interleaving of CPU time with I/O access (which under concurrency leads to a situation where some VMs execute code during the time in which others issue remote reads). For example, when booting 100 VM instances simultaneously, we measured two random instances to have, on average, a skew of about 100 ms between the times they access the boot sector of the initial image. This skew grows higher the longer the VM instances continue with the boot process. What this means is that at some point under concurrency they will access different chunks, which are potentially stored on different storage nodes, and thus contention is reduced.

While splitting the image into chunks reduces contention, the effectiveness of this approach depends on the chunk size and is subject to a trade-off. A chunk that is too large may lead to false sharing; that is, many small concurrent reads on different regions in the image might fall inside the same chunk, which leads to a bottleneck. A chunk that is too small, on the other hand, implies a higher access overhead, both because of higher network overhead, resulting from having to perform small data transfers, and because of higher metadata access overhead, resulting from having to manage more chunks.

To reduce contention even further and provide higher availability and fault tolerance, chunks can be replicated on different local disks. Again, we are faced with a trade-off: a high degree of replication raises availability and provides better fault tolerance; however, it comes at the expense of higher storage space requirements.

3.1.4 Optimize multisnapshotting by means of shadowing and cloning

Saving a full VM image for each VM is not feasible in the

context of multisnapshotting. Since only small parts of the VMs are modified, this would mean massive unnecessary duplication of data, leading not only to an explosion of utilized storage space but also to an unacceptably high snapshotting time and network bandwidth utilization.

For this reason, several custom image file formats were proposed that optimize taking incremental VM image snapshots. For example, Qemu/KVM introduced the QCOW2 [12] format for this purpose, while other work such as [26] proposes the Mirage Image Format (MIF). This approach enables snapshots to share unmodified content, which lowers storage space requirements. However, it presents several drawbacks.

First, a new snapshot is created by storing incremental differences as a separate file, while leaving the original file corresponding to the initial image untouched and using it as a backing file. When taking snapshots of the same image successively, a chain of files that depend on each other is obtained, which raises a lot of issues related to manageability. For example, one must keep track of dependencies between files. Even when such functionality is implemented, the cloud customer has to download a whole set of files from the cloud in order to get a local copy of a single VM snapshot—an operation that makes VM image downloads both costly and complex. Furthermore, in production use, this can lead to significant performance issues: a huge number of files will accumulate over time, thereby introducing a large metadata overhead.

Second, a custom image file format limits the migration capabilities. If the destination host where the VM needs to be migrated runs a different hypervisor that does not understand the custom image file format, migration is not possible.

Therefore, it is highly desirable to satisfy three requirements simultaneously:

- Store only the incremental differences between snapshots.
- Consolidate each snapshot as a standalone entity.
- Present a simple raw image format to the hypervisors to maximize migration portability.

We propose a solution that addresses these three requirements by leveraging two features proposed by versioning systems: *shadowing* and *cloning* [27, 23]. Shadowing means to offer the illusion of creating a new standalone snapshot of the object for each update to it but to physically store only the differences and manipulate metadata in such way that the illusion is upheld. This effectively means that from the user's point of view, each snapshot is a *first-class object* that can be accessed independently. For example, let's assume a small part of a large file needs to be updated. With shadowing, the user sees the effect of the update as a second file that is identical to the original except for the updated part. Cloning means to duplicate an object in such way that it looks like a stand-alone copy that can evolve in a different direction from the original but physically shares all initial content with the original.

Therefore, we propose to deploy a distributed versioning system that efficiently supports shadowing and cloning, while consolidating the storage space of the local disks into a shared common pool. With this approach, snapshotting can be easily performed in the following fashion. The first time a snapshot is built, for each VM instance a new virtual image clone is created from the initial image. Subsequent local modifications are written as incremental differences to the clones and shadowed. In this way all snapshots of all VM instances share unmodified content among one another and still appear to the outside as independent, simple raw image files, which addresses the three requirements mentioned above.

3.2 Applicability in the cloud

The simplified architecture of a 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 that are part of our proposal are highlighted by a darker background. A *distributed versioning storage service* that supports cloning and shadowing is deployed on the compute nodes and consolidates parts of their local disks into a common storage pool. The *cloud client* has direct access to the storage service and is allowed to upload and download images from it. Every uploaded image is automatically striped. Furthermore, the cloud client interacts with the *cloud middleware* through a control API that enables a variety of management tasks, including deploying an image on a set of compute nodes, dynamically adding or removing compute nodes from that set, and snapshotting individual VM instances or the whole set. The cloud middleware in turn coordinates the *compute nodes* to achieve the aforementioned management tasks. Each compute node runs a *hypervisor* that is responsible for running the VMs. The reads and writes of the hypervisor are trapped by the *mirroring module*, which is responsible for on-demand mirroring and snapshotting (as explained in Section 3.1) and relies on both the *local disk* and the distributed versioning storage service to do so.

The cloud middleware interacts directly with both the hypervisor, telling it when to start and stop VMs, and the mirroring module, telling it what image to mirror from the repository, when to create a new image clone (CLONE), and when to persistently store its local modifications (COMMIT). Both CLONE and COMMIT are control primitives that result in the generation of a new, fully independent VM image that is globally accessible through the storage service and can be deployed on other compute nodes or manipulated by the client. A global snapshot of the whole application, which involves taking a snapshot of all VM instances in parallel, is performed in the following fashion. The first time the snapshot is taken, CLONE is broadcast to all mirroring modules, followed by COMMIT. Once a clone is created for each VM instance, subsequent global snapshots are performed by issuing each mirroring module a COMMIT to its corresponding clone.

CLONE and COMMIT can also be exposed by the cloud middleware at the user level through the control API for fine-grained control over snapshotting. This approach enables snapshotting to be leveraged in interesting ways. For example, let's assume a scenario where a complex, distributed application needs to be debugged. Running the application repeatedly and waiting for it to reach the point where the bug happens might be prohibitively expensive. However, CLONE and COMMIT can be used to capture the state of the application right before the bug happens. Since all image snapshots are independent entities, they can be either collectively or independently analyzed and modified in an

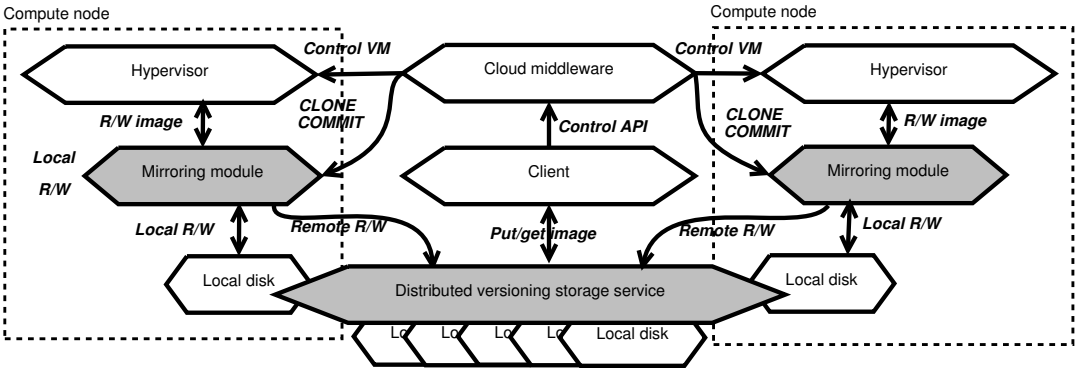


Figure 1: Cloud architecture that integrates our approach (dark background).

attempt to fix the bug. Once this fix is made, the application can safely resume from the point where it left. If the attempt was not successful, the approach can continue iteratively until a fix is found. Such an approach is highly useful in practice at large scale because complex synchronization bugs tend to appear only in large deployments and are usually not triggered during the test phase, which is usually performed at smaller scale.

3.3 Zoom on mirroring

One important aspect of on-demand mirroring is the decision of how much to read from the repository when data is unavailable locally, in such way as to obtain a good access performance.

A straightforward approach is to translate every read issued by the hypervisor in either a local or remote read, depending on whether the requested content is locally available. While this approach works, its performance is questionable. More specifically, many small remote read requests to the same chunk generate significant network traffic overhead (because of the extra networking information encapsulated with each request), as well as low throughput (because of the latencies of the requests that add up).

Moreover, in the case of many scattered small writes, a lot of small fragments need to be accounted for, in order to remember what is available locally for reading and what is not. Fragmentation is costly in this case and incurs a significant management overhead, negatively impacting access performance.

For this reason, we propose two strategies that aim to limit the negative impact of small reads and writes. First, a read operation on a region that is not fully available locally triggers remote reads that fetch the full minimal set of chunks that cover the requested region. While this leads to more network traffic than is strictly required, it improves the performance of correlated reads (i.e., a read on one region that is followed by a read "in the neighborhood") at a minimal cost when using small chunk sizes.

The second strategy we propose limits fragmentation by forcing a single contiguous region to be mirrored locally for each chunk. More specifically, a second write that falls on the same chunk as a previous write such that the gap between them is not available locally will trigger a remote read that will fill the gap. With this approach only the limits of a single contiguous region need to be maintained for each chunk, which places an upper limit on fragmentation overhead: it is directly proportional to the number of chunks in the worst case scenario when a small part of each chunk is written to.

4. IMPLEMENTATION

In Section 3.2 we illustrated how to apply our approach in the cloud by means of two basic building blocks: a *distributed versioning storage service*, which supports cloning and shadowing and is responsible for managing the repository, and a *mirroring module*, which runs on each compute node and is responsible for trapping the I/O accesses of the hypervisor to the image with the purpose of facilitating on-demand mirroring and snapshotting.

In this section we show how to efficiently implement these building blocks in such a way that they achieve the design principles introduced in Section 3.1 on the one hand and are easy to integrate in the cloud on the other hand.

4.1 Dependencies

We have implemented the distributed versioning storage service on top of *BlobSeer* [23, 24, 25]. This choice was motivated by several factors. First, BlobSeer enables *scalable aggregation of storage space* from the participating nodes with minimal overhead in order to store *BLOBs* (Binary Large OBjects). Data striping is performed transparently on BLOBs, which enables direct mapping between BLOBs and virtual machine images and therefore eliminates the need for explicit chunk management. Second, BlobSeer offers out-of-the-box support for shadowing, which significantly simplifies the implementation of the COMMIT primitive. Third, the service was optimized to sustain a high throughput even under heavy access concurrency, which is especially useful in the context of the multideployment and multisnapshotting patterns because it enables efficient parallel access to the image chunks.

The mirroring module was implemented on top of *FUSE* (FileSystem in UserspacE) [2], a choice that presents several advantages. First, it enables exposing the VM image to the hypervisors as a regular raw file that is accessible through the standard POSIX access interface. This ensures maximum compatibility with different hypervisors and enables running the same unmodified mirroring module on all compute nodes, regardless of which hypervisor is installed on them. Second, FUSE takes advantage of the kernel-level virtual file system, which benefits of the cache management implemented in the kernel for extra performance boost. The

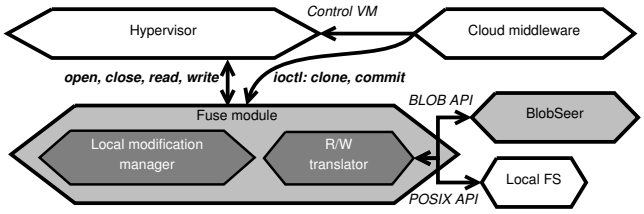


Figure 2: Implementation details: zoom on the FUSE module.

downside of using FUSE is the extra context-switching overhead between the kernel space and the user space when an I/O system call is issued. However, this overhead has a minimal negative impact, as demonstrated in Section 5.4.

4.2 Modus operandi

The FUSE module, presented in Figure 2, exposes each BLOB as a directory and its associated snapshots as files in that directory. It consists of two submodules: the *local modification manager*, responsible for tracking what content is available locally, and the *R/W translator*, responsible for translating each original read and write request, respectively, into local and remote reads and into local writes, according to the strategy presented in Section 3.3.

Whenever a VM image is opened for the first time, an initially empty file of the same size is created on the local disk. This file is then used to mirror the contents of the corresponding BLOB snapshot. For performance reasons, the whole local file is *mmap*ped in the host's main memory for as long as the snapshot is still open. This approach allows local reads and writes to be performed directly as memory access operations, thus avoiding unnecessary copies between memory buffers. Moreover, local writes are optimized this way because they benefit from the built-in asynchronous mmap write strategy implemented in the kernel.

When the snapshot is closed, the mmapped space is unmapped, and the local file is closed. The local modification manager then associates and writes extra metadata to the local file that describes the status of the local modifications (which chunks are available locally, which chunks have been altered, etc.). If the same VM image snapshot is reopened later, this extra metadata is used to reopen the same local file and restore the state of the local modifications.

Since the VM image is accessible from the outside through a POSIX-compliant access interface, we had to implement the CLONE and COMMIT primitives as *ioctl* system calls that are trapped and treated by the FUSE module in a special fashion.

COMMIT relies directly on the shadowing support exposed by BlobSeer in order to write all local modifications into a new BLOB snapshot. Figure 3 shows how this is possible through *distributed segment trees* that are enriched with versioning information [24]. More precisely, each tree node covers a region of the BLOB, with leaves covering chunks directly and nonleaf nodes covering the combined range of their left and right children. Nonleaf nodes may have children that belong to other snapshots, thus enabling sharing not only of unmodified chunks among snapshots of the same BLOB but also of unmodified metadata. In this way, consecutive COMMIT calls to the same VM image generate a totally ordered set of snapshots of the same BLOB, each of which can be directly mapped to a fully independent VM image snapshot-in-time that satisfies the requirements presented in Section 3.1.4.

Although the original BlobSeer implementation did not directly support cloning, we found this functionality to be easy to add: it is enough to add a new tree root corresponding to a different BLOB that has the same children as the segment tree of the original BLOB snapshot (Figure 3(b)). In this way, the implementation of the CLONE primitive generates a minimal overhead, both in space and in time.

For this work, we did not integrate the CLONE and COMMIT primitives in a cloud middleware. Instead, we implemented a simplified service that is responsible for coordinating and issuing these two primitives in a series of experimental scenarios that are described in the next section.

For completeness, however, we show how to integrate these primitives with *Nimbus* [3], a widely used open source cloud toolkit that allows turning clusters into *Infrastructure as a Service* clouds. A Nimbus deployment consists of a *central service*, installed on a dedicated node, and a series of *control agents*, installed on the compute nodes that host the virtual machines. Cloud clients interact by means of web-based messaging protocols with the central service, which in turn processes the client requests and issues control commands to the control agents. These agents have direct access to the resources of the compute node, controling both the hypervisor and the local file system. In order to integrate our approach in this architecture, the control agent needs to be extended such that it can issue the CLONE and COMMIT ioctl calls to the FUSE module. Furthermore, at the level of the central service, the interface exposed to the client needs to be extended to offer additional features, such as global snapshotting of all VM instances and fine-grained CLONE and COMMIT support for individual instances. Additional code is required to translate these advanced features into corresponding control commands.

5. EVALUATION

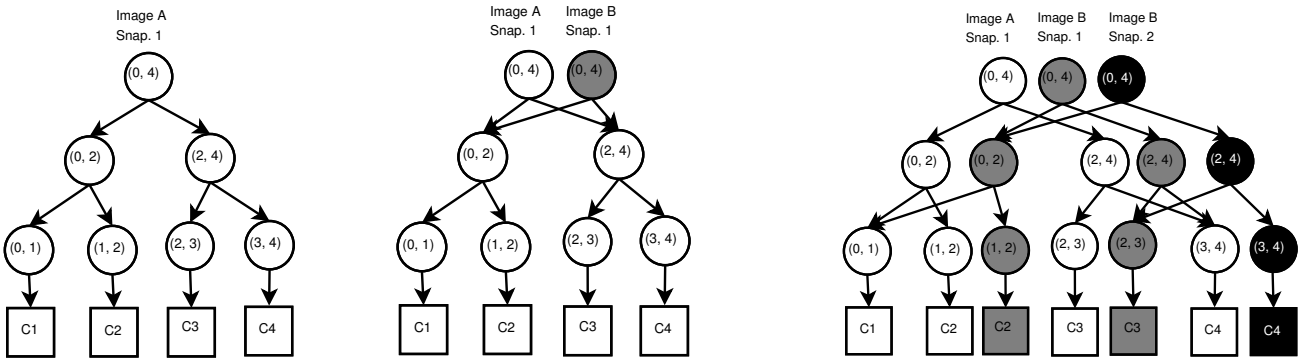
This section presents a series of experiments that evaluate how well our approach performs for both the multideployment and multisnapshotting patterns.

5.1 Experimental setup

The experiments were performed on Grid'5000 [15], an experimental testbed for distributed computing that federates nine sites in France. We used the 120-cluster system in Nancy, each with x86_64 CPUs offering hardware support for virtualization, local disk storage of 250 GB (access speed $\simeq$55 MB/s), and at least 8 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 $\simeq$0.1 ms). The hypervisor running on all compute nodes is KVM 0.12.5, while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 2 GB raw disk image file based on the same Debian Sid distribution was used.

5.2 Performance of multideployment

The first series of experiments evaluates how well our approach performs under the multideployment pattern, when a single initial VM image is used to concurrently instantiate a large number of VM instances.



(a) Segment tree and chunk composition of the initial VM image A

(b) Segment trees and chunk composition after VM image A was cloned into VM image B using CLONE

(c) Segment trees and chunk composition after VM image B was snapshotted two times consecutively using COMMIT

Figure 3: Cloning and shadowing by means of segment trees.

We compare our approach with two common techniques used to deploy and snapshot VM images:

Prepropagation.

Prepropagation [28, 31] is the most common method used on clouds. It consists of two phases. In the first phase the VM image is broadcast to the local storage of all compute nodes that will run a VM instance. Once the VM image is available locally on all compute nodes, in the second phase all VMs are launched simultaneously. Since in this phase all content is available locally, no remote read access to the repository is necessary. This approach enables direct local access to the image and eliminates contention to the repository, which is often implemented as a centralized NFS server, as discussed in Section 2.1. The downside of this approach, however, is the initialization phase, which potentially incurs a high overhead, in terms of both time and network traffic. To minimize this overhead, we use *taktuk* [10], a highly scalable broadcasting tool based on algorithms that follow the postal model [8] and is based on adaptive multicast trees that optimize the bandwidth/latency tradeoff. In our case, we broadcast the raw 2 GB VM image file, initially stored on a NFS server, to all nodes using taktuk. The NFS server is equipped with a regular 1 Gbit Ethernet interface, similar in configuration to the compute nodes.

Qcow2 over PVFS.

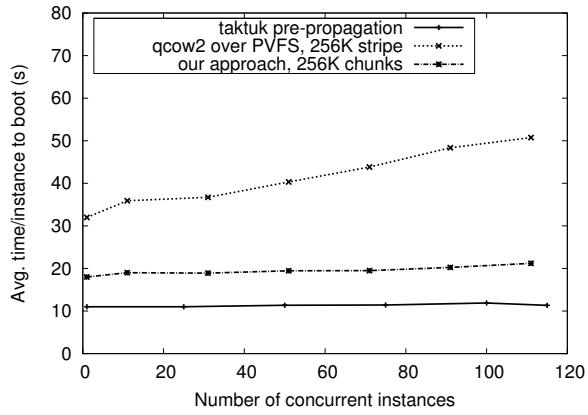
The second method we compare against is closer in concept to our own approach. We assume that the initial VM image is stored in a striped fashion on a distributed file system. We have chosen to use *PVFS* [9] to fill this role, as it is specifically geared to high performance and employs a distributed metadata management scheme that avoids any potential bottlenecks due to metadata centralization. PVFS is deployed on all available compute nodes, as is our approach, and is responsible for aggregating their local storage space in a common pool. To instantiate a new set of VM instances on the compute nodes, in a first initialization phase we create a new *qcow2* [12] copy-on-write image in the local file system of each compute node, using the initial raw 2 GB VM image stored in PVFS as the backing image (see Section 3.1.4 for details). Once the qcow2 image file is created on each compute node, in a second phase the hypervisor is launched by using the new qcow2 image as the underlying VM image, which automatically fetches data on demand from the backing raw image.

The comparison is based on the following performance metrics:

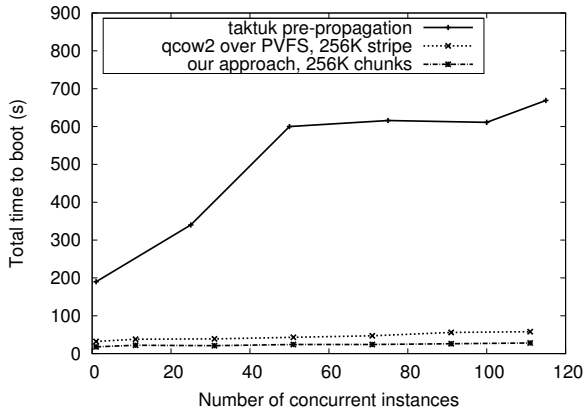
- *Average time to boot per instance.* This time is measured after the initialization phase completed: from when the hypervisor is launched up to the moment the operating system inside the VM has fully booted (more precisely, the /etc/rc.local script was executed). This parameter is relevant because it reveals the impact of remote concurrent reads (present in our approach and qcow2 over PVFS) vs. independent local reads (prepropagation) on the scalability of deploying the VMs in parallel.
- *Time-to-complete booting for all instances.* Essentially this metric measures the time taken by the slowest VM instance to finish booting. It is relevant because it measures the total time needed for the deployment to be ready, which is what the cloud user directly perceives from the outside.
- *Total network traffic.* generated throughout the execution of all VMs, including during the initialization phase (if applicable). This metric is relevant because it measures networking resource consumption and is proportional to the costs of running the deployment on the cloud.

The series of experiments consists in concurrently deploying an increasing number of VMs, one VM on each compute node. In the case of prepropagation, the initial VM image is stored on a local NFS server that serves as the source of the broadcast. In the case of our approach and qcow2 over PVFS, BlobSeer and PVFS respectively are deployed on the compute nodes, and the initial VM image is stored on them. For both BlobSeer and PVFS the chunk size was fixed at 256 KB, which we found to optimize the trade-off between having access to many small chunks and competing to access the same chunk, as discussed in Section 3.1.3. In order to obtain a fair comparison, chunks were not replicated.

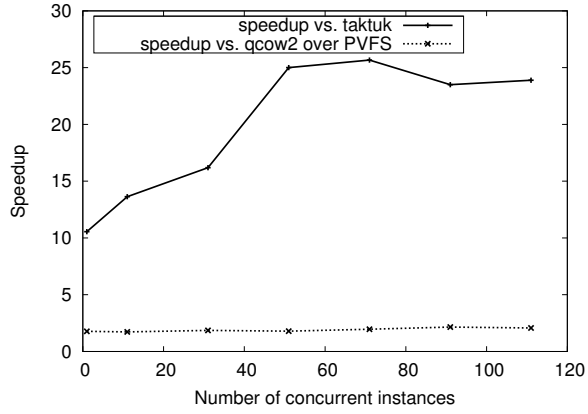
Figure 4(a), shows the average boot time per VM instance. As expected, in the case of prepropagation, average boot



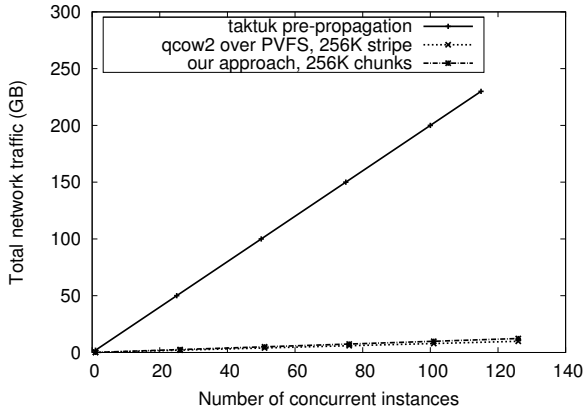
(a) Average time to boot per instance



(b) Completion time to boot all instances



(c) Speedup of the completion time to boot all instances for our approach



(d) Total generated network traffic

Figure 4: Multideployment: our approach compared with full prepropagation using broadcast/qcow2 images using PVFS as storage backend.

time is almost constant, since the data is already on the local file system and therefore no remote transfer is required. Both with qcow2 over PVFS and with our approach, boot times are significantly higher, as chunks need to be fetched remotely from the storage nodes on the fly during boot time. The more instances, the higher the read contention and thus the higher the boot times. As can be observed, however, this increasing trend resulting from increasing read pressure under concurrency is hardly noticeable for our approach compared with qcow2 over PVFS, which hints at better scalability. We can trace this back to the first optimization strategy introduced in Section 3.3, which avoids many small reads by prefetching whole chunks.

Figure 4(b) shows the total time to boot all VMs. As can be seen, prepropagation is an expensive step, especially since only a small part of the initial VM is actually accessed. This gives brings both our approach and qcow2 over PVFS a clear advantage. Looking at the speedup depicted in Figure 4(c), we can see an improvement of up to a factor of 25 with our approach vs. prepropagation, which slowly decreases because of increased read contention. On the other hand, the speedup vs. qcow2 over PVFS slowly increases thanks to our mirroring strategy, reaching a little over 2 at 110 instances.

Figure 4(d) illustrates the total network traffic incurred by both approaches. As expected, for all approaches the growth is linear and is directly proportional to the amount of data that was brought locally on the compute node. In the case of prepropagation, the network traffic is a little over 220 GB for 110 instances, as full content was transferred locally. Comparatively, our approach and qcow2 over PVFS bring huge savings in network traffic of around 13 GB and 12 GB, respectively. The slightly larger consumption in our case is due to the prefetching strategy in the mirroring module, which nevertheless is worthwhile considering the CPU time saved during the boot process for the whole deployment.

5.3 Multisnapshotting performance

This section evaluates the performance of our approach in the context of the multisnapshotting access pattern. Since it is infeasible to copy back to the NFS server the whole set of full VM images that include the local modifications done by

each VM instance, we limit the comparison of our approach with qcow2 over PVFS only.

The experimental setup is similar to the one used in the previous section: BlobSeer and PVFS are deployed on the compute nodes, and the initial 2 GB VM image is stored in a striped fashion on them, in chunks of 256 KB. The local modifications of each VM image are considered to be small, around 15 MB; this corresponds to the operating system and application writing configuration files and contextualizing the deployment, which simulates a setting with negligible disk access, as discussed in Section 2.3.

In the case of qcow2 over PVFS, the snapshot is taken by concurrently copying the set of qcow2 files locally available on the compute nodes back to PVFS. In the case of our approach, the images are snapshotted in the following fashion: first a CLONE, followed by a COMMIT is broadcast to all compute nodes hosting the VMs. In both cases, the snapshotting process is synchronized to start at the same time.

The average time to snapshot per instance is depicted in Figure 5(a). As can be observed, both in our approach and qcow2 over PVFS, average snapshotting time increases almost imperceptibly at a very slow rate. The reason is that an increasing number of compute nodes will always have at least as many local disks available to distribute the I/O workload, greatly reducing write contention. Since BlobSeer uses an asynchronous write strategy that returns to the client before data was committed to disk, initially the average snapshotting time is much better, but it gradually degrades as more concurrent instances generate more write pressure that eventually has to be committed to disk. The performance level is closing to the same level as qcow2 over PVFS, which essentially is a parallel copy of the qcow2 files.

Completion time for snapshotting, depicted in Figure 5(b), increases at a higher rate for both approaches because of the increasing striping overhead generated by the increasing number of concurrent instances. More specifically, more network connections need to be opened in parallel on each compute node, thus increasing latencies and the standard deviation of the snapshotting time. Overall, both approaches perform similarly; however, our approach avoids the generation of a new file for each snapshot and thus greatly improves manageability, as discussed in Section 3.1.4.

5.4 Local access performance: read-your-writes access patterns

As discussed in Section 2.3, some deployments involve virtualized web servers or other applications that need to maintain log files or object caches directly in the VM image, a requirement that leads to a read-your-writes access pattern to the VM image.

Therefore, in this section we evaluate the performance of our approach for such a scenario, comparing its overhead with the case when the hypervisor has direct access to the local filesystem of the compute nodes hosting the VM instances. This is the case for both prepropagation and qcow2 over PVFS. We found the overhead of writing in a qcow2 file vs. a raw file to be negligible. Therefore, we compare our approach only with the case when the hypervisor is using a raw image file that is fully available locally.

To generate a write-intensive scenario that also reads back written data, we use a standard benchmarking tool: Bonnie++ [21]. Bonnie++ creates and writes a set of files that fill a large part of the remaining free space of the disk, 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. Since data is first written sequentially and then read back, no remote reads are involved for our approach. This in turn means contention is not an issue, and therefore experimentation with a single VM instance is enough to predict behavior of multiple instances that run concurrently.

The experiment consists in booting the VM instance and then running Bonnie++ using both our approach and a locally available image directly. The total space written and read back by Bonnie++ was 800 MB out of a total of 2 GB, in blocks of 8 KB.

Throughput results are shown in Figure 6. As can be seen, reads of previously written data have the same performance levels for both approaches. This result is as expected, because previously written data is available locally for both approaches and therefore no additional overhead is incurred by our approach. Interestingly, write throughput and overwrite throughput are almost twice as high for our approach. The reason is that our mmap strategy triggers a more efficient write-back strategy in the host's kernel and overrides the default hypervisor strategy.

On the other hand, the extra context switches and management overhead incurred by FUSE for our approach become visible when measuring the number of operations per second. Figure 7 shows lower numbers for our approach, especially with random seeks and file deletion. However, since operations such as file creation/deletion and seeks are relatively rare and execute very fast, the performance penalty in real life is not an issue.

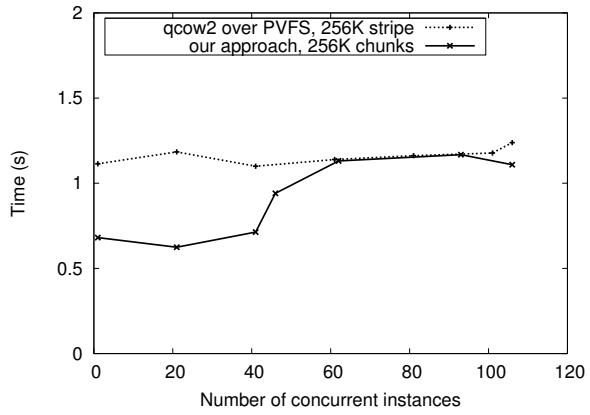
5.5 Benefits for real-life, distributed applications

As a last series of experiments, we illustrate the benefits of our approach for an application in the real world: Monte Carlo approximations. Such applications are a common type of high-performance computing (HPC) scientific applications that are loosely coupled and thus ideal candidates to run on the cloud without excessive performance loss due to virtualized networking infrastructure.

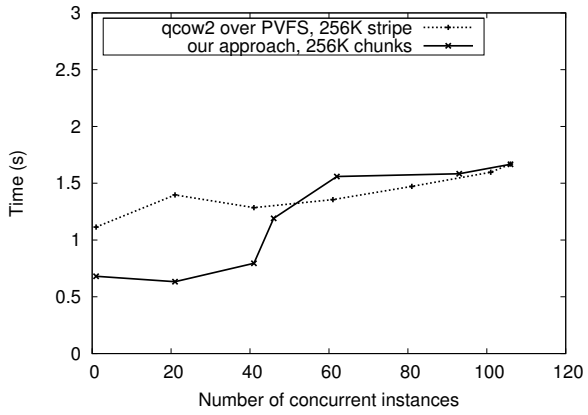
More specifically, these applications rely on repeated random sampling to compute their results and are often used in simulating physical and mathematical systems. Such an approach is particularly useful when it is infeasible or impossible to compute the exact result by using deterministic methods. In our case, we estimate the number π with high precision by sampling a large number of points in a square and calculating the number of points that fall inside the inscribed circle. This work is evenly distributed among 100 workers running as VM instances on the compute nodes. Each worker is programmed to save intermediate results in a temporary file inside the VM image (size per instance is $\simeq$10 MB).

In a first setting, the VM deployment is running uninterrupted for all three approaches: prepropagation, qcow2 over PVFS, and our approach. This setting highlights multideployment only. In all three cases, time to completion was measured.

In a second setting, we assume the VM deployment is snapshotted and terminated at some point during its execu-



(a) Average time snapshot an instance under concurrency



(b) Completion time to snapshot all instances

Figure 5: Multisnapshotting: our approach compared with qcow2 images using PVFS as storage backend. Diff for each image is 15 MB.

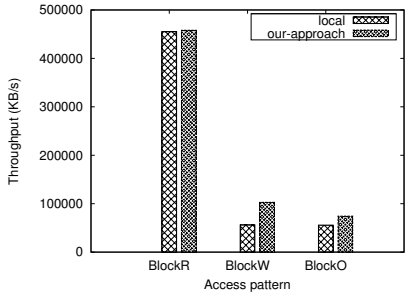


Figure 6: Bonnie++ sustained throughput: read, write, and overwrite in blocks of 8K (BlockR/BlockW/BlockO).

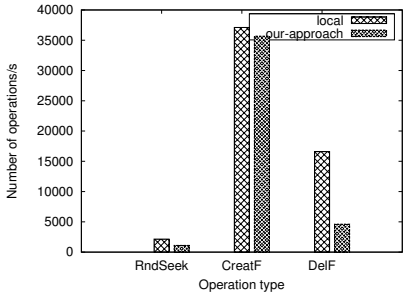


Figure 7: Bonnie++ sustained number of operations per second: random seeks (RndSeek), file creation/deletion (CreatF/DelF).

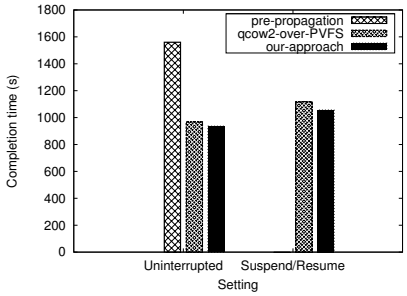


Figure 8: Benefits in the real world: time to finish a Monte Carlo simulation using 100 VM instances in various settings.

tion and then resumed on another set of nodes, which highlights both multideployment and multisnapshotting performance. In this case, our approach was compared with qcow2 over PVFS only. More specifically, once the VM instances are down, they are rebooted, and the application resumes from the last intermediate result saved in the temporary file. Each VM instance is resumed on a different compute node from the one where it originally ran, to simulate a resume on a fresh set of nodes where no parts of the VM images are available locally and therefore need to be fetched remotely. We measured time to completion for the whole cycle of multideployment, partial application execution, multisnapshotting, then multiredeployment, and at last final application execution for both cases.

We used the same hypervisor configuration to resume the application this experiment in order to be able to compare our approach with qcow2 over PVFS. We emphasize the practical importance of portability in this context: for our approach the new set of nodes may run a different hypervisor equally well, yet migration is still possible without any further effort, thanks to exposing snapshots as raw image files.

Results are shown in Figure 8. As expected from the results obtained in Section 5.2, in the first setting the initialization phase for prepropagation is extremely costly, limiting the on-demand provisioning performance on the cloud. On the other hand, our approach and qcow2 over PVFS perform significantly better, with our approach having better performance thanks to the advantage of a faster boot phase. This advantage increases in the second setting, since the boot phase is executed twice, effectively reducing the migration overhead as a whole. For a total computation time of about 1000 s, our approach enables the deployment to be resumed more rapidly: by almost 5%.

6. RELATED WORK

Multideployment that relies on full broadcast-based prepropagation is a widely used technique [28, 31, 14]. While this technique avoids read contention to the repository, it can incur a high overhead in both network traffic and execution time, as presented in Section 5.2. Furthermore, since the VM images are fully copied locally on the compute nodes, multisnapshotting becomes infeasible: large amounts of data are unnecessarily duplicated and cause unacceptable transfer delays, not to mention huge storage space and network traffic utilization.

In order to alleviate this problem, many hypervisors pro-

vide native copy-on-write support by defining custom VM image file formats [12, 26] specifically designed to efficiently store incremental differences. Much like our approach, this allows base images to be used as read-only templates for multiple logical instances which store per-instance modifications. However, lack of standardization and the generation of many interdependent new files limit the portability and manageability of the resulting VM image snapshots.

A different approach to instantiate a large number of VMs from the same initial state is proposed in [18]. The authors introduce a new cloud abstraction: VM FORK. Essentially this is the equivalent of the fork call on UNIX operating systems, instantaneously cloning a VM into multiple replicas running on different hosts. While this is similar to CLONE followed by COMMIT in our approach, the focus is on minimizing the time and network traffic to spawn and run, on the fly, new remote VM instances that share the same state of an already running VM. Local modifications are assumed to be ephemeral, and no support to store the state persistently is provided.

Closer to our approach is Lithium [13], a fork-consistent replication system for virtual disks. Lithium supports instant volume creation with lazy space allocation and instant creation of writable snapshots. Unlike our approach, which is based on segment trees, Lithium is based on log structuring [29], which can potentially degrade read performance when increasing the number of consecutive snapshots for the same image: the log of incremental differences starts growing, making it more expensive to reconstruct the image.

Cluster volume managers for virtual disks such as Parallax [22] enable compute nodes to share access to a single, globally visible block device, which is collaboratively managed to present individual virtual disk images to the VMs. While this enables efficient frequent snapshotting, unlike our approach, sharing of images is intentionally not supported in order to eliminate the need for a distributed lock manager, which is claimed to dramatically simplify the design.

Several storage systems, such as Amazon S3 [6] (backed by Dynamo [11]), have been specifically designed as highly available key-value repositories for cloud infrastructures. They can be valuable building blocks for block-level storage volumes [1] that host virtual machine images; however, they are not optimized for snapshotting.

Our approach is intended to complement existing cloud computing platforms, both from industry (Amazon Elastic Compute Cloud: EC2 [5]) and from academia (Nimbus [3, 17, 16], OpenNebula [4]). While the details for EC2 are not publicly available, it is widely acknowledged that all these platforms rely on several of the techniques presented above. Claims to instantiate multiple VMs in "minutes," however, are insufficient for meeting our performance objectives; hence, we believe our work is a welcome addition in this context.

7. CONCLUSIONS

As cloud computing becomes increasingly popular, efficient management of VM images, such as image propagation to compute nodes and image snapshotting for checkpointing or migration, is critical. The performance of these operations directly affects the usability of the benefits offered by cloud computing systems. This paper introduced several techniques that integrate with cloud middleware to efficiently handle two patterns: *multideployment* and *multisnapshotting*.

We propose a lazy VM deployment scheme that fetches VM image content as needed by the application executing in the VM, thus reducing the pressure on the VM storage service for heavily concurrent deployment requests. Furthermore, we leverage object versioning to save only local VM image differences back to persistent storage when a snapshot is created, yet provide the illusion that the snapshot is a different, fully independent image. This has two important benefits. First, it handles the management of updates independently of the hypervisor, thus greatly improving the portability of VM images and compensating for the lack of VM image format standardization. Second, it handles snapshotting transparently at the level of the VM image repository, greatly simplifying the management of snapshots.

We demonstrated the benefits of our approach through experiments on hundreds of nodes using benchmarks as well as real-life applications. Compared with simple approaches based on prepropagation, our approach shows a major improvement in both execution time and resource usage: the total time to perform a multideployment was reduced by up to a factor of 25, while the storage and bandwidth usage was reduced by as much as 90%. Compared with approaches that use copy-on-write images (i.e., *qcow2*) based on raw backing images stored in a distributed file system (i.e., PVFS), we show a speedup of multideployment by a factor of 2 and comparable multisnapshotting performance, all with the added benefits of transparency and portability.

Based on these encouraging results, we plan to explore the multideployment and multisnapshotting patterns more extensively. With respect to multideployment, one possible optimization is to build a prefetching scheme based on previous experience with the access pattern. With respect to multisnapshotting, interesting reductions in time and storage space can be obtained by introducing deduplication schemes. We also plan to fully integrate the current work with *Nimbus* [3] and explore its benefits for more complex HPC applications in the real world.

Acknowledgments

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/). This work was supported in part by Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

8. REFERENCES

[1] Amazon elastic block storage (ebs). http://aws.amazon.com/ebs/.
[2] File system in userspace (fuse). http://fuse.sourceforge.net.
[3] Nimbus. http://www.nimbusproject.org/.
[4] Opennebula. http://www.opennebula.org/.
[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. 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] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *SPAA '92: Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, New York, 1992. ACM.

[9] 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, GA, 2000. USENIX Association.

[10] B. Claudel, G. Huard, and O. Richard. Taktuk, adaptive deployment of remote executions. In *HPDC '09: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, pages 91–100, New York, 2009. ACM.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, New York, 2007. ACM.

[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] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *ATC '03: Proceedings of the 2003 USENIX Annual Technical Conference*, pages 283–296, San Antonio, TX, 2003.

[15] Y. Jégou, S. Lantéri, J. Leduc, M. Noredine, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[16] K. Keahey and T. Freeman. Science clouds: Early experiences in cloud computing for scientific applications. In *CCA'08: Proceedings of the 1st Conference on Cloud Computing and Its Applications*, 2008.

[17] K. Keahey, M. O. Tsugawa, A. M. Matsunaga, and J. A. B. Fortes. Sky computing. *IEEE Internet Computing*, 13(5):43–51, 2009.

[18] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 1–12, New York, 2009. ACM.

[19] X. Liu, J. Huai, Q. Li, and T. Wo. Network state consistency of virtual machine in live migration. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 727–728, New York, 2010. ACM.

[20] P. Marshall, K. Keahey, and T. Freeman. Elastic site: Using clouds to elastically extend site resources. In *CCGRID '10: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 43–52, Washington, DC, USA, 2010. IEEE Computer Society.

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

[22] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: Virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(4):41–54, 2008.

[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, 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, February 2011.

[25] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier. Blobseer: Bringing high throughput under heavy concurrency to Hadoop map/reduce applications. In *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, Atlanta, GA, 2010.

[26] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *VEE '08: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 111–120, New York, 2008. ACM.

[27] O. Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):1–27, 2008.

[28] A. Rodriguez, J. Carretero, B. Bergua, and F. Garcia. Resource selection for fast large-scale virtual appliances propagation. In *ISCC '09: Proceedings of 14th IEEE Symposium on Computers and Communications*, pages 824–829, 5-8 2009.

[29] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[30] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[31] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath. Image distribution mechanisms in large scale cloud providers. In *CloudCom '10: Proceedings 2nd IEEE International Conference on Cloud Computing Technology and Science*, Indianapolis, IN, 2010.