Going Back and Forth: Efficient Virtual Machine Image Deployment and Snapshotting on IaaS Clouds

Bogdan Nicolae, John Bresnahan, Kate Keahey, Gabriel Antoniu

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Going Back and Forth: Efficient Virtual Machine Image Deployment and Snapshotting on IaaS Clouds

Bogdan Nicolae, John Bresnahan, Kate Keahey,Gabriel Antoniu

N° 7482

October 2010

ISSN 0249-6399 ISRN INRIA/RR--7482--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Going Back and Forth: Efficient Virtual Machine Image Deployment and Snapshotting on IaaS Clouds

Bogdan Nicolae*, John Bresnahan†, Kate Keahey†,Gabriel Antoniu‡

Thème : Calcul distribué et applications à très haute performance
Équipe-Projet KerData

Rapport de recherche n° 7482 — October 2010 — 25 pages

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 providerâĂŹs datacenter for a short time by deploying virtual machines (VMs) on those resources. This new model raises new challenges in the design and development of IaaS middleware. On of those challenges if 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. This paper addresses both challenges by proposing a series of optimization techniques minimizing resource consumption (execution time, network traffic and storage space) which translate into lower end-user costs. While conventional approaches transfer the whole VM image contents between the persistent storage service and the computing nodes, we propose a lazy transfer scheme based on object-versioning that transfers only the needed content on-demand: this greatly reduces total time for execution time, network traffic and storage space. We demonstrate these benefits through experiments operating on hundreds of nodes which show improvements in time to process application execution by a factor of up to 25, while at the same time reducing storage and bandwidth usage by as much as 90% when compared with conventional approaches.

Key-words: large scale; virtual machine deployment; virtual machine image storage; lazy propagation; snapshotting

* University Rennes 1, IRISA, France.
† Argonne National Laboratory, USA.
‡ INRIA Rennes-Bretagne Atlantique, IRISA.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Déploiement et Snapshotting Efficaces des Images des Machines Virtuelles sur les Clouds IaaS

Résumé : Le cloud computing de type Infrastructure-as-a-service (IaaS) a rvolutionné notre manière d'acquérir les ressources de calcul. Cette révolution est due à un principe très simple: permettre aux utilisateurs de réserver des ressources de calcul à partir d'un centre de données fourni par un cloud provider pour une courte durée de temps. L'exploitation de ces ressources se fait en déployant des machines virtuelles (VMs). Ce nouveau modèle apporte de nouveaux défis vis-à-vis la conception et le développement des middlewares des IaaS. Un de ces défis est le besoin de déployer un grand nombre (une centaine ou milliers) d'instances de machines virtuelles simultanément. Une fois ces instances déployées, un autre défi est de simultanément prendre un snapshot de plusieurs images et de les transférer à un stockage persistant afin de rendre les taches de gestion telles que suspend-resume et la migration plus efficaces. Ce papier traite ces deux défis en proposant un ensemble de techniques d'optimisation qui minimisent la consommation des ressources (temps d'exécution, le trafic réseau et l'espace de stockage) ce qui réduit le cout pour l'utilisateur final. Bien que les approches conventionnelles transfèrent le contenu de l'image de la machine virtuelle dans sa totalité entre le service de stockage persistant et les noeuds de calcul, on propose un schémas 'lazy transfert' basé sur le versioning d'objets qui transfère seulement le contenu nécessaire sur demande. Cette approche réduit d'une façon significative le temps total d'exécution, le trafic réseau et l'espace de stockage. On démontre ces gains en conduisant des expérimentations sur une centaine de noeuds. Ces expérimentations ont montré une réduction jusqu'à 25 fois du temps d'exécution du processus de l'application, toute en réduisant le stockage et l'utilisation de la bande passante par 90% par rapport des approches conventionnelles.

Mots-clés : grande échelle; déploiement des machines virtuelles; stockage des images des machines virtuelles; propagation de type “lazy”; snapshotting

Contents

1 Introduction

In the recent years, Infrastructure-as-a-Service (IaaS) cloud computing [1] has emerged as a viable alternative to the acquisition and management of physical resources. Using IaaS, users are able to 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 [2, 3], the user can purchase a resource for a short time, which 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 lease attractive, enabling users to elastically expand or shrink the required resources according to their computational needs, e.g. use external resources to complement their local resource base [4].

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 data-center 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.

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 GB up to more than 10 GB). A typical deployment consists of hundreds if not thousands of such images: just propagating (transfering) them to the nodes on which a VM instance will be deployed can take hours. This would make the response time of IaaS installation much longer than acceptable and erase the on-demand benefits of cloud computing. A similar challenge applies to snapshotting images (saving them to capture VM state for later use): many images that were locally modified need to be rapidly transferred to storage.

In addition to incurring significant delays, both propagation and snapshotting have the potential to generate significant network traffic that may interfere with the execution of applications on leased resources. Moreover, they consume huge amounts of storage space, which ultimately means higher costs for the user.

This paper proposes a set of techniques that aim to significantly reduce the execution time, as well as storage space and network traffic consumption for both the propagation and snapshotting in an IaaS infrastructure. More precisely, we consider the following patterns:

- *The multi-deployment pattern* occurs when multiple VM images (or a single VM image) are deployed on many nodes at the same time. In such a scenario where massive concurrent accesses increase the pressure on the storage service where the images are located, it is interesting to avoid full propagation to the nodes that will host the VMs. At the minimum, when the image is booted, only parts of the image that are actually accessed by the boot process need to be transferred. This saves us the cost of moving the image and makes deployment fast while reducing the risk for a bottleneck on the storage service where images are stored. However, such a "lazy" transfer will make the boot process longer, as some necessary parts of the image may not be available locally. We exploit this tradeoff to achieve a good balance between deployment and application execution.

- *The multi-snapshotting pattern* occurs when many images deployed on many nodes in a datacenter are saved to a storage system at the same time. The interesting property of this pattern is that most of the time, only small parts of the image are modified by the VM instance. Therefore, image snapshots share large amounts of data among each other, which can be exploited both to reduce execution time, as well as to reduce storage space and bandwidth consumption.

These patterns are complementary and for this reason we investigate them in conjunction.

We have investigated these patterns in the context of the Nimbus compute cloud infrastructure [5] which provides cloud computing capabilities for science

applications. However, the series of techniques we propose in this paper are generic and can be applied to any IaaS cloud computing infrastructure. Our contributions are summarized as follows:

- We introduce a series of design principles that optimize multi-deployment and multi-snapshotting patterns and describe how our design can be integrated with IaaS infrastructures (Section II and III)
- We show how to materialize these design principles in practice 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 [6, 7].
- 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

Clouds typically are built on top of clusters made out of loosely-coupled commodity hardware that minimizes per unit cost and favors low power over maximum speed [2]. Disk storage (cheap hard-drives with capacities in the order of several hundred GB) is attached to each processor, while processors are interconnected with standard Ethernet links. A part of those nodes is employed as compute nodes that run the VMs of users. Their disk storage is not persistent and is wiped after the VM finished running. Another part of these nodes is employed as storage nodes, which are responsible to host a distributed storage service, such as S3 [8], that persistently stores both user VM images and application data. In many commercial clouds, the ratio of storage nodes to compute nodes is not officially disclosed, but with the recent explosion of data sizes, (for example, Google grew from processing 100 TB of data a day with MapReduce in 2004 [9] to processing 20 PB a day with MapReduce in 2008 [10]), we estimate that the storage nodes will soon have to outnumber the compute nodes to cope with these increasing storage needs.

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 between 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 state of the communication channels is difficult and still an open problem [11]. 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). Even so, while this is perfectly feasible for one single VM instance and widely used in practice, for a large number of VMs the necessary storage space explodes to huge sizes. For example, saving 2 GB of RAM for 1000 VMs consumes 2 TB of space, which is unacceptable for a single one-point-in-time application state.

Therefore, Model 2 can further be simplified such that the VM state is represented only by the virtual disk attached to it, which is stored as an image file in the local file system of the VM host (Model 3). Thus, the application state is saved by persistently storing all disk-image files (locally modified by the VM). While this approach requires the application to be able to save and restore its state explicitly to disk (for example as a temporary file), it has two important practical benefits: (1) huge reductions in the size of the state, since the contents of RAM, CPU registers, etc. 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.

In this work, for clarity, we assume VM state is represented using Model 3. It is however easy to extend the applicability of our approach to Model 2 by considering that VM image files include not only the contents of the virtual disk attached to the VMs, but also the state of the devices.

2.3 Application access pattern

VM typically do not access their whole initial images. For example, they may never access some applications and utilities that are installed by default. In order to model this it is useful to analyze the life-cycle of a VM instance, which consists of three phases:

- *Booting*, which involves reading configuration files and launching processes which translates to random small reads and writes from the virtual machine image acting as the initial state.
- *Running the user application*, which generates application-specific access patterns:
  - *Negligible disk-image access*, which applies to CPU-intensive applications or applications that use external storage services. Examples for this case are large-scale simulations.
  - *Read-your-writes*, which applies to applications that write temporary files or log files and eventually read them back (e.g. web servers).
  - *Read-intensive*, which applies to application that read (most often sequentially) input data stored in the image. Examples here are data mining applications. Results are typically presented as a single aggregated value, so generated writes are negligible.
- *Shutting down*, which generates negligible disk access to the image.

3 Our approach

We propose an approach that enables both efficient propagation of the initial virtual image contents to the VMs when the application is deployed, as well as efficient snapshotting while the VM is running.

3.1 Design overview

We rely on three key principles:

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

Before the VM needs to be instantiated, an initially empty file of the same size as the image is created on the local file system of the compute node that hosts the VM. This file is then passed to the hypervisor running on the compute node for use as the underlying VM image. 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 (either by a previous read or write), results in fetching the missing content remotely from the repository, creating a local copy of it and redirecting the read to the local copy. If the whole region is available locally, no remote read is performed. This relates closely to *copy-on-reference*, first used for process migration in the V-system [12]. Writes on the other hand are always performed locally.

3.1.2 Reduce contention by striping the image

Each virtual image is split into small equally-sized chunks that are distributed among the storage nodes of the repository. When a read request triggers a remote fetch from the repository, the chunks that hold this content are first determined. Then, the storage nodes that hold the chunks are contacted in parallel and the data is transferred back to the compute node. Under concurrency, this scheme effectively enables the distribution of the I/O workload among the storage nodes, because accesses to different parts of the image are served by different storage nodes.

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 because of various reasons: different hypervisor initialization overhead, 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), etc. For example, when booting 150 VM instances simultaneously, we measured two random instances to have on the average a skew of about 100ms between the times they access the boot sector of the initial image. This skew grows higher the more 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, i.e. 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.

3.1.3 Optimize snapshotting by means of shadowing and cloning

Since on-demand mirroring does not bring the whole contents of the initial image locally, taking a snapshot of the image by saving the associated local file persistently to the repository is not possible. Even under the assumption that all local files have transferred and represent fully consistent images, storing all of them concurrently is not advantageous either, both because too much storage space and network traffic is generated on one side, and because of high write contention to the repository on the other side, which leads to unacceptably high snapshotting time.

For this reason, most hypervisors implement custom image file formats that enable storing incremental differences to support efficiently multiple snapshots of the same VM instance in the same file. For example, KVM introduced the QCOW2 [13] format for this purpose, while other work such as [14] proposes the Mirage Image Format (MIF). This effectively enables snapshots to share unmodified content, which lowers storage space requirements. However, in our context we need to support efficiently multiple snapshots of *different* VM instances that share an initial image. This requirement limits the applicability of using such a custom image file format. Moreover, a custom image file format also 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 two requirements simultaneously: (1) store only the incremental differences between snapshots; and (2) represent each snapshot as an independent, raw image file that is compatible with most hypervisors.

We propose a solution that addresses these two requirements by leveraging two features commonly available in versioning systems: *shadowing* and *cloning* [15]. Shadowing basically 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 aforementioned illusion is upheld. This effectively means that from the user 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 than the original, but physically shares all initial content with the original.

If a versioning system is deployed on the storage nodes that efficiently supports shadowing and cloning, snapshotting can be easily performed in the following fashion: the first time a snapshot is build, 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. This

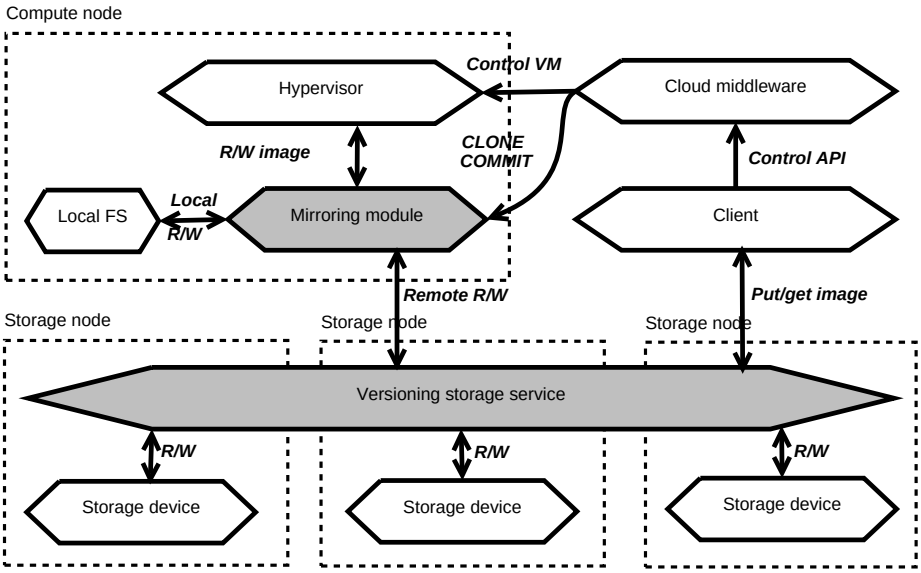

Figure 1: Cloud architecture that integrates our approach (dark background)

way all snapshots of all VM instances share unmodified content among each other, and still appear to the outside as independent raw image files.

3.2 Applicability in the cloud

The simplified architecture of a cloud which 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 *versioning storage service* that supports cloning and shadowing is deployed on the storage nodes and manages their storage devices. 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 also interacts with the *cloud middleware* through a control API that enables a variety of management tasks: deploying an image on a set of compute nodes, dynamically adding or removing compute nodes from that set, snapshotting individual VM instances or the whole set, etc. The cloud middleware in turn coordinates the *compute nodes* to achieve the aforementioned management tasks. Each compute node runs a *hypervisor* that is responsible to run 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 file system* and 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 the local modifications to it (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 when 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 user level through the control API for fine grain control over snapshotting. This enables leveraging snapshotting 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 virtual image snapshots are independent entities, they can be either collectively or independently analyzed and modified in an attempt to fix the bug. Once this is done, the application can safely resume from the point where it left. If the attempt was not successful, this 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 whether the requested contents is locally available or not. While this approach certainly works, its performance is rather questionable. More specifically, many small remote read requests generate significant network traffic overhead (because of the extra networking information encapsulated with each request), as well as a 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. A lot of fragments however incur a significant management overhead, negatively impacting access performance.

For this reason, we propose two heuristics that aim to limit the negative impact of small reads and writes. The first heuristic is based on the empirical observation that reads tend to be locally correlated: a read on one region is probably followed by a read "in the neighborhood". This is especially true for sequential read access, which is a common access pattern. For this reason, the heuristic tries to minimize the negative impact of small reads by forcing remote reads to be at least as large as a chunk. Subsequent reads that fall within the same chunk are served locally, thus greatly improving throughput in this case.

The second heuristic we propose eliminates 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 the contiguous region need to be maintained for each chunk, which makes fragment management overhead negligible. Moreover, it is better than plain copy-on-reference which would read the whole chunk before applying a write, since in the case of sequential writes no remote read needs to be performed at all.

4 Implementation

In Section 3.2 we illustrated how to apply our approach in the cloud by means two basic building blocks: a *versioning storage service* that supports *cloning* and *shadowing* and is responsible for the management of the repository, as well as a *mirroring module* that runs on each compute node and is responsible to trap 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 way that they achieve the design principles introduced in Section 3.1 on one side, and are easy to integrate in the cloud on the other side.

We have chosen to implement the versioning storage service on top of *BlobSeer*, presented in Section 4.1. This choice was motivated by three factors. First it offers out-of-the-box support for shadowing, which enables easy implementation of our approach. Second, BlobSeer supports transparent data striping of large objects and fine-grain access to them, which enables direct mapping between BlobSeer objects and virtual machine images and therefore eliminates the need for explicit chunk management. Finally it offers support for high throughput under concurrency, which enables efficient parallel access to the image chunks. Since our approach requires cloning support as well, additional effort was undertaken to introduce it in BlobSeer.

The mirroring module was implemented on top of *FUSE* (FileSystem in UserspacE) [16]. FUSE is a loadable kernel module for Unix-like operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module acts only as an intermediate to the actual kernel interfaces. This approach brings several advantages in our context. First, it enables portability among hypervisors by exposing a POSIX-compliant file system access interface to the images. POSIX is supported by most hypervisors and enables running the same unmodified mirroring module on all compute nodes, regardless of what hypervisor is installed on them. Second, FUSE takes advantage of the kernel-level virtual file system, which brings out-of-the-box support for advanced features such as cache management. Finally it avoids the need to alter or extend the hypervisor in any way, which effectively reduces implementation time and maintenance costs. The downside of 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 5.3.

4.1 BlobSeer

BlobSeer is a distributed versioning data management service designed to deal with the needs of data-intensive applications: *scalable aggregation of storage space* from the participating nodes with minimal overhead, support to store *huge data objects*, *efficient fine-grain access* to data subsets and ability to sustain a *high throughput under heavy access concurrency*.

Data is abstracted in BlobSeer as long sequences of bytes called BLOBs (Binary Large OBject). These BLOBs are manipulated through a simple access interface that enables creating a blob, reading/writing a range of *size* bytes from/to the BLOB starting at a specified *offset* and appending a sequence of *size* bytes to the BLOB. This access interface is designed to support versioning explicitly: each time a write or append is performed by the client, a new snapshot of the blob is generated that acts as a first class object rather than overwriting any existing data (but physically stored is only the difference). This snapshot is labeled with an incremental version and the client is allowed to read from any past snapshot of the BLOB by specifying its version.

BlobSeer relies on *data striping*, *distributed metadata management* and *versioning based concurrency control* to avoid data-access synchronization and to distribute the I/O workload at large-scale both for data and metadata. This is crucial in achieving a high aggregated throughput for data-intensive applications, as demonstrated in [6, 17, 7].

4.2 Modus Operandi

BlobSeer is leveraged as a versioning storage service by mapping each virtual image to a BLOB. Local modifications are commited simply by using the BlobSeer write primitive to update the BLOB. In order to support cloning, a new primitive was introduced for this purpose. Since BlobSeer shares both data *and* metadata between unmodified snapshots, the BLOB cloning primitive is very efficient, involving only minimal metadata manipulations that associate the new BLOB to the same data and metadata of the original BLOB.

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 to track what contents is available locally and the *R/W translator*, responsible to translate each original read and write request into local and remote reads and writes, according to the strategy presented in Section 3.3.

Whenever a BLOB snapshot is opened for the first time, an initially empty file of the same size is created on the local file system. This file is then used to mirror the contents of the BLOB snapshot. For performance reasons, extents are used to create it instantaneously. As soon as this operation completed, the whole local file is *mmap*-ed in the host's main memory for as long as the snapshot is still open. This allows local reads and writes to be performed directly as memory access operations which enables *zero-copy* (i.e. avoiding unnecessary copies between memory buffers). Moreover, local writes are also optimized this way because they benefit from the built-in asynchronous mmap write strategy.

Finally, 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

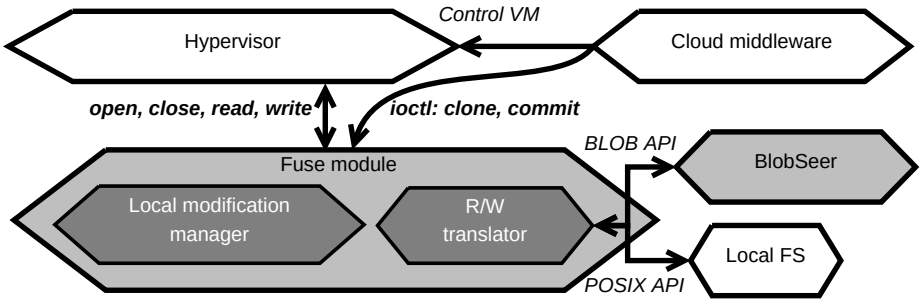

Figure 2: Implementation details: zoom on the FUSE module

modifications (what chunks are available locally, what chunks have been altered, etc). If the same snapshot is reopened at a later time, this extra metadata is used to restore the state of the local modifications.

CLONE and COMMIT are implemented as *ioctl* system calls. Both rely on the clone and write primitives natively exposed by BlobSeer. CLONE simply calls the BlobSeer clone primitive and binds all local modifications to the newly cloned BLOB, while the COMMIT primitive writes all local modifications back to the BLOB as a series of BlobSeer writes. Care is taken to minimize the amount of issued BlobSeer writes by aggregating consecutive "dirty" (i.e. locally modified) chunks in the same write call. Once the all dirty chunks have been successfully written, the state is reset and all chunks are marked as "clean" again.

For the purpose of this work, we did not integrate the CLONE and COMMIT primitives with a real cloud middleware. We implemented a simplified service instead that is responsible to coordinate and issue these two primitives in a series of particular scenarios that are described in the next section.

However, we plan to integrate these primitives with *Nimbus* [18, 3], an open source cloud middleware that allows turning clusters into *Infrastructure-as-a-Service (IaaS)* clouds. A Nimbus deployment consists of a *central service* that is installed on a dedicated node, and a series of *control agents* that are 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*. The control 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: global snapshotting of all VM instances, fine-grain CLONE and COMMIT support for individual instances, etc. Finally, additional code is required to translate these advanced features into corresponding control commands.

5 Evaluation

5.1 Experimental setup

The experiments presented in this work have been performed on Grid'5000 [19], an experimental testbed for distributed computing that federates 9 different sites in France. We have used the clusters located in Nancy. All nodes of Nancy, numbering 200 in total, are outfitted with x86_64 CPUs, local disk storage of 250 GB (access speed $\tilde{5}5$ MB/s) and at least 2 GB of RAM. Unless otherwise stated, we fix 50 nodes to act as storage nodes and we employ a variable number of compute nodes, up to the rest of 150. This accounts for a storage-to-compute ratio that is at least 1:4. The hypervisor running on all compute nodes is KVM 0.12. For all experiments, a 2 GB raw disk image file based on a recent Debian Sid distribution was used.

5.2 Scalability of initial VM image content distribution under concurrency

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

We compare our approach to a common technique used by cloud middleware to distribute the image content on the node that hosts the VM: *pre-propagation*. Pre-propagation consists of two phases: in a 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 the second phase all content is available locally, no remote read access to the repository is necessary. This enables direct local access to the image and does not depend or use the network connection to the storage nodes. This is based on the assumption that access to the local file system of the compute nodes is faster than access to remote storage, which is the case for most large clusters in the industry that are build from commodity hardware, as mentioned in Section 2.1.

The downside of this approach is however the initialization phase, which potentially incurs a high overhead, both in terms of latency and network traffic. In order to minimize this overhead, we use TakTuk [20], a highly scalable tool based on broadcasting algorithms in the postal model [21], which builds adaptive multicast trees that optimize the bandwidth/latency tradeoff in order to efficiently broadcast a file to a set of machines. In this setting, it is assumed that the root of the multicast tree is a NFS file server which is hosted by a dedicated storage node and which holds the initial image.

We will compare the two approaches according to the following metrics:

- *Average execution time per instance*: the average time taken to execute an application in the VM. This time is measured after the initialization phase (if applicable) has been completed, between the time the VM is launched and the time the VM is terminated. This parameter is relevant because it reveals the impact of remote concurrent reads (present in our approach) vs. independent local reads (pre-propagation) on the scalability of running VMs in parallel.

- *Time-to-completion for all instances*: the time taken to complete the initialization, launching, and execution of applications for all VMs. This time is measured between when the request to launch the VMs is received in the system and when the last VM is terminated. This parameter is relevant because it measures the total time needed to execute the application and obtain a final result, which is what the user directly perceives.
- *Total network traffic*: the total network traffic generated throughout the execution of all VMs, including during the initialization phase (if applicable). This parameter is relevant because it is proportional to the cost of running the application.

The series of experiments consists in deploying an increasing number of VMs using both our approach and pre-propagation. We start with one VM and increase the number of deployed VMs in steps of 25 up to 150, applying the metrics defined above at each step. In the case of pre-propagation, the initial image is stored on the local NFS server which serves the cluster and which is used as the source of the multicast. In the case of our approach, BlobSeer is assumed to be deployed on the 50 reserved storage nodes and the initial image stored in a striped fashion on it.

Note that the variation of average execution time needs not be represented explicitly. This results from two reasons: (1) in the case of pre-propagation all data is available locally after the initialization phase and therefore the variation is negligible; and (2) in the case of our approach there is no initialization phase and therefore the completion time coincides with the time to run the slowest VM, which measures the maximal deviation from the mean.

5.2.1 Boot performance

In the first experiment, the only activity carried out in the VM is fully booting the operating system. This corresponds to the behavior of an application that performs minimal access to the underlying virtual image, which is for example the case of CPU-intensive applications. Under such circumstances, almost all access to the virtual image is performed during the boot phase.

Since in our approach the image is striped into chunks, an important aspect is to evaluate the impact of the chunk size on performance and network traffic. As discussed in Section 3.1.2, a small chunk size minimizes the amount of unnecessary data that is prefetched and thus minimizes network traffic. However, lowering the chunk size too much means that many chunks have to be fetched and thus this can have a negative impact on performance. We evaluated various chunk sizes and found the best performance was delivered at 512 KB. For completeness, we include both results obtained with a higher size (1 MB), and a lower size (256 KB).

Figure 3, shows the average boot time per VM instance. As expected, in the case of pre-propagation, average boot time is almost constant, as data is already on the local file system and therefore no transfer from the storage nodes is required. In the case of our approach, boot times are 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. A breaking point is noticeable at 50 instances, where the increase in average time becomes steeper. This is because after 50 instances, the storage

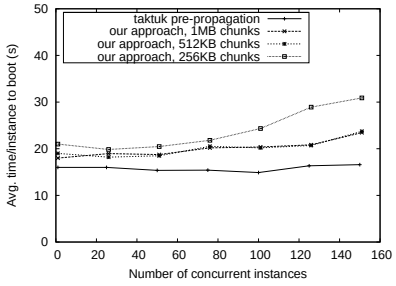

Figure 3: Average time to boot per instance

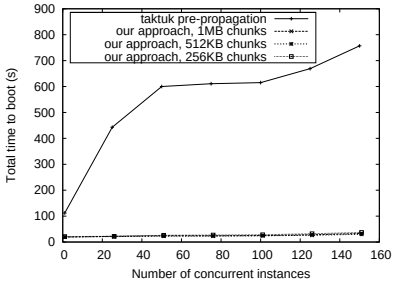

Figure 4: Completion time to boot all instances

nodes are outnumbered by the VM instances and therefore the probability of concurrent access to the same storage node increases even when reading different chunks.

Figure 4 shows the total time to boot all VMs. As can be seen, the pre-propagation is an expensive step, especially when considering that only a small part of the initial virtual is actually accessed. This brings our approach at a clear advantage, with the speedup depicted in Figure 5. The speedup is obtained as the completion time to boot all instances of the pre-propagation approach divided by the completion time of our approach. As expected, the highest speedup is obtained when the number of VM instances is the same as then number of storage nodes, which enables the optimal distribution of I/O workload for our approach. Performance-wise, it can be observed that a 512 KB chunk size brings the highest speedup and scales well even after the breaking point is reached.

Finally, Figure 6 illustrates the total network traffic incurred by both approaches. As expected, 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 pre-propagation, the network traffic is as expected a little over 300 GB for 150 instances. In the case of our approach, the smaller the chunk size, the smaller amount of total network traffic. However, the total amount of network traffic is not directly proportional to the chunk size. Lowering the chunk size from 1 MB to 512 KB results in a network traffic drop from 30 GB to 21 GB, while lowering from 512 KB to 256 KB in a drop from 21 GB to 15 GB only. The smaller the chunk gets, the smaller the benefits from avoiding network traffic are. This happens mainly because of access locality: consecutive reads issued by the hypervisor fall inside a region that is more likely to be covered by consecutive chunks, which makes the benefit of small chunks sizes smaller.

5.2.2 Full read performance

The second experiment considers the a complementary case to the one presented in the previous section, namely when the whole content of the virtual image needs to be read by each VM instance. This represents the most unfavorable read-intensive scenario that corresponds to applications which need to read input data stored in the image. In this case, the time to run the VM corresponds to the time to boot and fully read the whole virtual image disk content

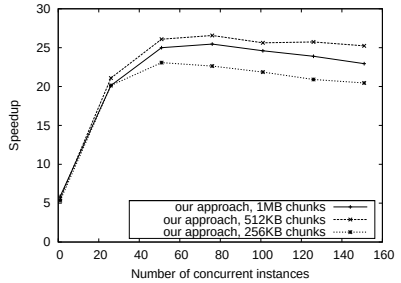

Figure 5: Performance of VM boot process: speedup vs. pre-propagation under concurrency

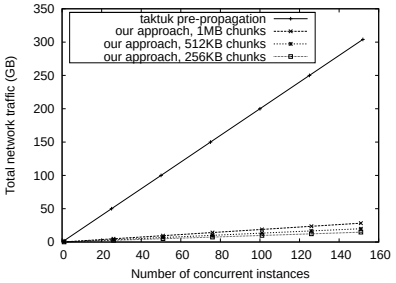

Figure 6: Total network traffic generated by our approach vs. pre-propagation

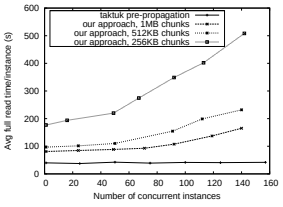

Figure 7: Average time to boot and fully read image per instance (lower is better)

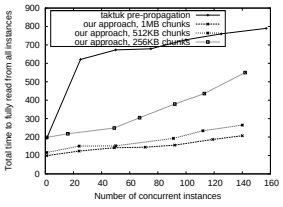

Figure 8: Completion time to boot the image and fully read its content from all instances (lower is better)

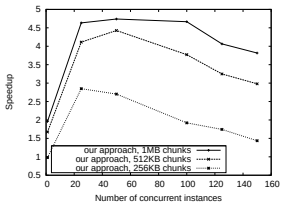

Figure 9: Performance of boot followed by full read: speedup vs. pre-propagation

(by performing a "cat /dev/hda1", where hda1 is the virtual disk corresponding to the image). Again, the evaluation is performed for three chunk sizes: 1 MB, 512 KB and 256 KB.

The average time to boot and fully read the initial disk content is represented in Figure 7. As expected, in the case of pre-propagation, this time remains constant as no read contention exists. In the case of our approach, almost perfect scalability is also noticeable up to 50 storage nodes, despite read concurrency. This is so because there are enough storage providers among which the I/O workload can be distributed. After the number of instances outnumber the storage nodes, the I/O pressure increases on each storage node, which makes the average read performance degrade in a linear fashion. On a general note, the read performance is obviously worse in our approach, as the data is not available locally.

However, when considering the completion time for booting and fully reading the image from all instances 8, our approach is at a clear advantage. The main reason for this is the fact that the pre-propagation approach needs to touch the data twice: once in the initialization phase, when it needs to be copied locally and then in the execution phase when it needs to be read by the guest. Since in our approach the initialization phase is missing, the total time is better even for 1 instance.

The actual speedup is represented in Figure 9. The peak is as expected at 50 when the number of instances reaches the number of storage nodes. Then, the speedup gradually starts falling as the I/O pressure on the storage nodes

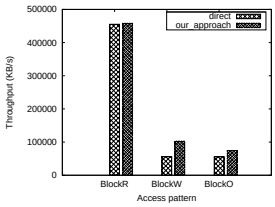

Figure 10: Bonnie++ throughput: read, write and overwrite in blocks of 8K (BlockR/BlockW/BlockO)

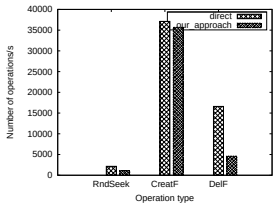

Figure 11: Bonnie++ number of operations per second: random seeks (RndSeek), file creation/deletion (CreatF/DelF)

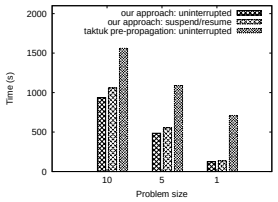

Figure 12: Monte-Carlo PI calculation: time to finish estimating PI using 100 instances that each pick 1, 5 and 10 billion points

increases. Since the image needs to be fully read by each instance, the total generated network traffic is not represented explicitly, as it is roughly the same for both approaches and was already depicted in Figure 6 as the curve corresponding to pre-propagation. Compared to the speedup obtained for the boot-only experiment, it can be noted that the peak speed-up is much lower at 4.7 and degrades faster when the number of compute nodes outnumber the storage nodes. This is explained by the fact that in the boot-only experiment, our approach had the advantage of fetching only the necessary parts of the image, which does not apply any longer.

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

While the previous section evaluates first time read performance of the initial image contents, this section focuses on local read and write performance, i.e. when the accessed regions are presumed to be already available locally. This scenario is representative of read-your-writes applications, that need to write large amounts of data in the virtual image (e.g., log files) and then eventually read all this information back.

For this purpose, we compare our approach to the ideal scenario: when a copy of the virtual image is already available on the local file system of the compute node and can be used directly by the hypervisor. We aim to evaluate the overhead of our approach which needs to trap reads and writes and needs to manage local modifications, as opposed to the ideal case when the hypervisor can directly interact with the local file system. In order to generate a write-intensive scenario that also reads back written data, we use a standard benchmarking tool: Bonnie++ [22]. 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++ both using our approach and using a locally available image directly. This experiment is repeated 5 times and the results of Bonnie++ are averaged. 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 10. As can be noticed, reads of previously written data have the same performance levels for both approaches. This results is as expected, because previously written data is available locally for both approaches and therefore no additional overhead is incurred by our approach. Interestingly enough, write throughput and overwrite throughput is almost twice as high for our approach. This is explained by the fact that mmap triggers a more efficient write-back strategy in the host's kernel and overrides the default hypervisor strategy, which comes into play when it has direct access to the image.

On the other hand, the extra context switches and management overhead incurred by our approach become visible when measuring the number of operations per second. Figure 11 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 negligible.

5.4 Snapshotting performance

This section evaluates the performance of our approach in the context of the multi-snapshotting access pattern. We assume a large number of VM instances that need to concurrently save their corresponding VM image, which suffered local modifications, persistently on the storage nodes.

The experimental setup is similar to the one used in the previous sections: an initial RAW image, 2 GB large, is striped and stored on 50 storage nodes in three configurations: 1 MB, 512 KB and 256 KB chunks. In order to underline the benefits our our approach better, we consider that local modifications are small, which is consistent for example with checkpointing a CPU-intensive, distributed applications where the state of each VM instance can be written as a temporary file in the image. In order to simulate this behavior in our experiments, we have instantiated a variable number of VM instances from the same initial image and then instructed each instance to write a 1 MB file in its corresponding image, after which the hypervisor was instructed to flush all local modifications to the VM image. Once all hypervisors finished this process, all images were concurrently snapshotted by broadcasting a CLONE, followed by a COMMIT command to all compute nodes hosting the VMs. The local modifications captured by COMMIT include not only the temporary file, but also file system operations performed during the boot phase (i.e. creating configuration files, writing to log files, etc.)

The execution time on each compute node, as well as the total storage space consumed by all snapshots is recorded and used to calculate the following metrics: the average snapshotting time per instance, the completion time to snapshot all VMs, and the overall occupied storage space on the storage nodes (which is roughly equivalent to the total generated network traffic).

Execution time results are depicted in Figure 13 and Figure 14. As can be observed, average snapshotting time does not experience a significant growth

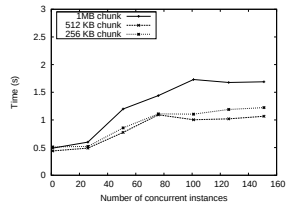

Figure 13: Average time to snapshot an instance under concurrency

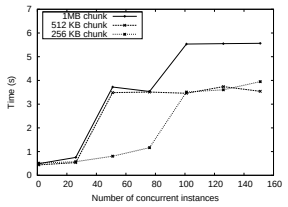

Figure 14: Completion time to snapshot all instances concurrently

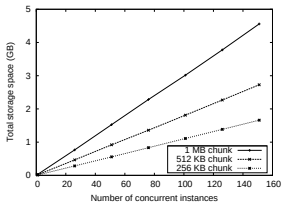

Figure 15: Total aggregated storage space occupied by all snapshots

when the storage nodes outnumber the compute nodes, and from that point on a linearly-shaped growth. The completion time to snapshot all images follows the same trend, but grows faster and experiences a more discrete evolution, which is explained by the higher deviation from the mean that is present to the increased write pressure on the storage nodes. Nevertheless, both the average time as well as the completion time are just a tiny fraction when compared to traditional approaches that store all images fully in the cloud storage, a process that can easily extend to hours. The same conclusion can be drawn when looking at the storage space and network traffic: Figure 15 shows a total occupied storage space well below 2 GB for 150 instances, when the chunk size is 256 KB, and about 4.5 GB for a 1 MB chunk size. For comparison, fully transferring the images to the storage nodes takes well over 300 GB for 150 instances.

5.5 Benefits for real-life, distributed applications

We illustrate the benefits of our approach for a common type of CPU-bound scientific applications: Monte-Carlo approximations. Such applications rely on repeated random sampling to compute their results and are often used in simulating physical and mathematical systems. This approach is particularly useful when it is unfeasible or impossible to compute the exact result using deterministic methods. In our particular case, we estimate the number π by choosing random points in a square and calculating how much of the points fall inside of the inscribed circle. In this case, $\pi = 4 \times (points\_inside)/(total\_points)$.

This is an embarrassingly parallel application, as a large number of workers can independently pick such points and verify their belonging to the circle. In a final step, the results are aggregated and PI is calculated. We spawn 100 instances to solve this problem for a variable number of total points: 10^{11}, 5×10^{11} and 10^{12}. The work is evenly distributed among instances. The workload is mostly CPU-intensive, with each instance programmed to save intermediate results at regular intervals in a temporary file (I/O is negligible).

For each number of total points, the problem is solved in three different ways: using our approach, using pre-propagation as described in Section 5.2, and using our approach but suspending the application and then resuming each VM instance on a different node as where it originally ran. The suspend-resume is performed in the following fashion: CLONE, followed by a COMMIT, is broadcast to all VM instances. Immediately after all snapshots are successfully written to persistent storage, the VM instances are killed. Once all 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 than the one where it originally ran, to simulate a realistic situation where the original compute nodes have lost all their local storage contents or there is a need to migrate the whole application on a different set of compute nodes.

Results are shown in Figure 12. As expected from the results obtained in Section 5.2.1, since this is a CPU-intensive application, the initialization phase for pre-propagation is extremely costly. This effect is more noticeable when fewer work is done by each VM instance. While the overhead of snapshotting is negligible in all three cases, the suspend-resume cycle overall grows slightly higher the more work needs to be performed by the VM instances, as resuming on different nodes with potentially different CPU creates a larger variance in longer VM execution times. Overall however, this accounts for less than 10% overhead in the worst case of 10^{12} total points.

6 Related work

Several mechanisms to disseminate the content of the initial virtual image exist. Commonly in use is full virtual image pre-propagation to the local storage of the compute nodes before launching the VM. For this purpose, efficient broadcast techniques have been proposed. TakTuk [20] is a highly scalable tool inspired by broadcasting algorithms in the postal model [21]. It builds adaptive multi-cast trees to optimally exploit bandwidth and latency for content dissemination. Multi-casting is also employed by Frisbee [23], a tool used by EmuLab [24] to apply disk images to nodes. Scp-wave [25], in use by OpenNebula [26] is another such tool. It carries out the transfers by starting with a single seed and increases the number of seeders as more content is transferred, in order to obtain a logarithmic speed-up versus a sequential transfer. A similar idea is implemented in [27] as well. While these approaches avoid read contention to the repository, they can incur a high overhead both in network traffic and execution time, as presented in Section 5.2.

A different approach to instantiate a large number of VMs from the same initial state is proposed in [28]. 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 local 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 [29], a fork-consistent replication system for virtual disks. Lithium supports instant volume creation with lazy space allocation, instant creation of writable snapshots, and tunable replication. While this achieves the same as CLONE and COMMIT, it is based on log-structuring [30], which incurs high sequential read and maintenance overhead.

Content Addressable Storage (CAS) [31, 32] was also considered for archival of virtual machine images [33] and disk image management [34, 35]. While the focus is on providing space-efficient disk image storage mainly by exploiting

deduplication, concurrency issues are again not addressed or not part of the original design.

Cluster volume managers for virtual disks such as Parallax [36] 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 unsupported 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 [8] (backed by Dynamo [37]) have been specifically designed as highly available key-value repositories for cloud infrastructures. This objective however is achieved at the cost of limiting the client to read and write full objects only, which limits the applicability in our context.

Finally, our approach is intended as a means to complement existing cloud computing platforms, both from the industry (Amazon Elastic Compute Cloud: EC2 [2]) and academia (Nimbus [18, 3], OpenNebula [26]). 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 is insufficient for the performance objectives of our work, which is why we believe it is a welcome addition in this context.

7 Conclusions

In the context of an increasing popularity of cloud computing, efficient management of VM images such as image propagation to compute nodes and image snapshotting for checkpointing or migration are critical. The performance of these operations directly impact the usability of the elastic features brought forward by cloud computing systems. This paper introduced several techniques that integrate with cloud middleware to efficiently handle two patterns: *multi-deployment* and *multi-snapshotting*. 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 an important benefit in that 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.

We demonstrate the benefits of our approach through experiments on 100s of nodes using benchmarks as well as real-life applications. Compared to simple approaches based on pre-propagation, our approach shows improvements both in execution time and resources usage (i.e., storage space and network traffic): we show that the time to process application execution in an IaaS cloud can be improved by a factor of up to 25, while at the same time reducing storage and bandwidth usage by 90%. These results have impact on the final user costs, as costs are directly proportional to the amount of consumed resources and the time the user spends waiting for an application to finish.

Our approach relies on configurations with multiple storage nodes in order to scale. However, with the explosion of data sizes and resulting growth in storage

demand, datacenters dedicate more and more resources to storage, which places our approach in a favorable context. Although it has been developed in the context of the Nimbus toolkit, our approach is explicitly designed to be versatile and integrate well with any IaaS cloud middleware.

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/).

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

[3] K. Keahey and T. Freeman, "Science clouds: Early experiences in cloud computing for scientific applications. cloud computing and applications," 2008.

[4] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *CCGRID*, 2010, pp. 43–52.

[5] "Nimbus," http://www.nimbusproject.org/.

[6] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "Blobseer: Next generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, 2010, in press.

[7] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier, "Blobseer: Bringing high throughput under heavy concurrency to hadoop map-reduce applications," in *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, Atlanta, USA, 2010, pp. 1–11.

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

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[10] ——, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] 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*. New York, NY, USA: ACM, 2010, pp. 727–728.

[12] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the v-system," in *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1985, pp. 2–12.

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

[14] 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 fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2008, pp. 111–120.

[15] O. Rodeh, "B-trees, shadowing, and clones," *Trans. Storage*, vol. 3, no. 4, pp. 1–27, 2008.

[16] "File system in userspace (fuse)," http://fuse.sourceforge.net.

[17] B. Nicolae, G. Antoniu, and L. Bougé, "Enabling high data throughput in desktop grids through decentralized data and metadata management: The blobseer approach," in *Euro-Par '09*, Delft, The Netherlands, 2009, pp. 404–416.

[18] K. Keahey, M. O. Tsugawa, A. M. Matsunaga, and J. A. B. Fortes, "Sky computing," *IEEE Internet Computing*, vol. 13, no. 5, pp. 43–51, 2009.

[19] 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*, vol. 20, no. 4, pp. 481–494, November 2006.

[20] 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*. New York, NY, USA: ACM, 2009, pp. 91–100.

[21] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," in *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1992, pp. 13–22.

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

[23] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, "Fast, scalable disk imaging with frisbee," in *Proc. of the USENIX Annual Technical Conference*, San Antonio, TX, 2003, pp. 283–296.

[24] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale virtualization in the emulab network testbed," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 113–128.

[25] "Fuse," http://code.google.com/p/scp-wave/.

[26] "Opennebula," http://www.opennebula.org/.

[27] A. Rodriguez, J. Carretero, B. Bergua, and F. Garcia, "Resource selection for fast large-scale virtual appliances propagation," in *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, 5-8 2009, pp. 824–829.

[28] 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*. New York, NY, USA: ACM, 2009, pp. 1–12.

[29] J. G. Hansen and E. Jul, "Lithium: virtual machine storage for the cloud," in *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*. New York, NY, USA: ACM, 2010, pp. 15–26.

[30] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[31] J. K. Hollingsworth and E. L. Miller, "Using content-derived names for configuration management," in *SSR '97: Proceedings of the 1997 symposium on Software reusability*. New York, NY, USA: ACM, 1997, pp. 104–109.

[32] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *FAST '02: Proceedings of the Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2002, pp. 89–101.

[33] P. Nath, M. A. Kozuch, D. R. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups, "Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines," in *In Proc. of the USENIX Annual Technical Conference*, 2006, pp. 363–378.

[34] A. Liguori and E. V. Hensbergen, "Experiences with content addressable storage and virtual disks," in *Workshop on I/O Virtualization*, 2008.

[35] S. Rhea, R. Cox, and A. Pesterev, "Fast, inexpensive content-addressed storage in foundation," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 143–156.

[36] 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.*, vol. 42, no. 4, pp. 41–54, 2008.

[37] 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 twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 205–220.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399