BlobCR: Virtual Disk Based Checkpoint-Restart for HPC Applications on IaaS Clouds

Bogdan Nicolae, Franck Cappello

BlobCR: Virtual Disk Based Checkpoint-Restart for HPC Applications on IaaS Clouds

Bogdan Nicolae[a], Franck Cappello[b,c]

[a] *IBM Research, Ireland*
[b] *INRIA Saclay*
[c] *University of Illinois at Urbana-Champaign*

Abstract

Infrastructure-as-a-Service (IaaS) cloud computing is gaining significant interest in industry and academia as an alternative platform for running HPC applications. Given the need to provide fault tolerance, support for suspend-resume and offline migration, an efficient Checkpoint-Restart mechanism becomes paramount in this context. We propose *BlobCR*, a dedicated checkpoint repository that is able to take *live incremental* snapshots of the whole disk attached to the virtual machine (VM) instances. BlobCR aims to minimize the performance overhead of checkpointing by persisting VM disk snapshots asynchronously in the background using a low overhead technique we call *selective copy-on-write*. It includes support for both application-level and process-level checkpointing, as well as support to roll back file system changes. Experiments at large scale demonstrate the benefits of our proposal both in synthetic settings and for a real-life HPC application.

Keywords: IaaS clouds, high performance computing, checkpoint-restart, fault tolerance, virtual disk snapshots, rollback of file system changes

1. Introduction

Infrastructure-as-a-Service (IaaS) clouds have gained significant attention over the last couple of years due to the proposed pay-as-you-go model that enables clients to lease computational resources in form of virtual machines from large datacenters rather than buy and maintain dedicated hardware. With increasing interest in High Performance Computing (HPC) applications (both in industry and academia) such clouds have the potential to provide a competitive replacement for leasing time on leadership-class facilities where HPC applications are typically run. This potential results from the fact that leadership-class facilities rely on expensive supercomputers, which are not readily available for the masses.

Email addresses: bogdan.nicolae@ie.ibm.com (Bogdan Nicolae), fci@lri.fr (Franck Cappello)

Despite efforts to define a HPC cloud market paradigm [41] and adopt it in practice (such as Amazon Web Services' HPC offering [7] or science cloud initiatives [3, 42]), the HPC community has been reluctant to embrace cloud computing. To date, the mainstream cloud application patterns have typically been "embarrassingly parallel."

This is not without good reason: porting HPC applications to clouds is a challenging task. Although there is evidence of increasing improvement in the scalability and performance of cloud-based HPC systems [26, 24], many obstacles are still problematic, because of architectural differences specific to IaaS clouds: multi-tenancy, overhead due to the virtualization layer, poor networking performance [29], lack of a standardized storage stack, etc. These differences lead to a situation where well established HPC approaches cannot be easily adapted to IaaS clouds and need to be redesigned.

One critical challenge in this context is *fault tolerance*. With increasing demand in scale and the emergence of exa-scale, the number of components that can fail at any given moment in time is rapidly growing. This effect is even more noticeable in IaaS clouds [47], since they are mostly build out of commodity hardware [7]. Thus, an assumption about complete reliability is highly unrealistic: at such large scale, hardware component failure is the norm rather than the exception.

Fault tolerance is a well studied aspect of cloud computing. However, due to the embarrassingly parallel nature of mainstream cloud applications, most approaches are designed to deal with fault tolerance of individual processes and virtual machines. This either involves restarting failed tasks from scratch (e.g., MapReduce) or using live migration approaches to replicate the state of virtual machines on-the-fly in order be able to switch to a backup VM in case the primary instance has failed [15]. HPC applications on the other hand are *tightly coupled*: processes depend on each other to exchange information during the computation and the failure of one process destroys the coherence of the other processes, ultimately leading to a situation where the whole application needs to be restarted. This is known as the *domino effect*.

To address this issues, *Checkpoint-Restart (CR)* [18] was proposed to provide fault-tolerance for HPC applications. Fault tolerance is achieved by saving recovery information periodically during failure-free execution and restarting from that information in case of failures, in order to minimize the wasted computational time and resources. Although replication of processes has been considered before [19] in the context of supercomputing architectures (and a similar approach could be imagined using virtual machine replication), this involves a high performance overhead to synchronize the process replicas, not to mention an explosion in resource usage (at least double the amount of resources needed to run the application using CR). Thus, replication is of little practical interest in the cloud context, as an important target besides performance is to minimize operational costs.

Furthermore, the potential benefits of CR in the context of IaaS clouds go well beyond its original scope, enabling a variety of features that bring substantial reduction in operational costs under the right circumstances: *suspend-resume* (i.e. suspending the computation when the price of resources fluctuates and they become expensive or the budget is tight), *migration* (i.e. moving a computation to a new cloud provider without losing progress already made and paid for), *debugging* (i.e. reducing application development and testing costs by capturing and replaying subtle bugs at large scale close to the moment when they happen).

In this paper we propose *BlobCR* (BlobSeer-based Checkpoint-Restart), a checkpoint-restart framework specifically optimized for HPC applications that need to be ported to IaaS clouds. Our solution introduces a dedicated checkpoint repository that is able to take incremental snapshots of whole disks attached to the virtual machine (VM) instances where the HPC application is writing its checkpointing data. This mechanism can be leveraged either at application-level by directly requesting virtual disk snapshots, or at system-level by using modified transparent checkpointing protocols normally implemented in the middleware that is employed by the application (e.g. message passing libraries such as MPI [22]).

We summarize our contributions below:

- We present a series of design principles that facilitate checkpoint-restart on IaaS clouds and show how they can be applied in IaaS cloud architectures. Unlike conventional approaches, our proposal introduces support for an important feature: the ability to roll back I/O operations performed by the application.
- We complement our previous work [36] with support for *live snapshotting*. In this context, we contribute with a scheme specifically optimized for asynchronous transfers of checkpointing data.
- We introduce an algorithmic description and show how to implement and integrate it in practice on top of *BlobSeer*, BlobSeer, a versioning storage service specifically designed for high throughput under concurrency [32, 34].
- We demonstrate the benefits of our proposal in a series experiments, conducted on hundreds of nodes provisioned on the Grid'5000 testbed, using both synthetic benchmarks and real-life applications.

2. Checkpoint-restart on IaaS clouds

In short, CR is a mechanism that periodically saves the state of an application to persistent storage (referred to as *checkpoints*) and offers the possibility to resume the application from such intermediate states.

2.1. Application model

CR treats an application as a collection of distributed processes that communicate through a network in order to solve a common problem. Communication is performed by exchanging messages between the processes using a message passing system.

In addition to message passing, the processes have access to a persistent storage service (typically a parallel file system, such as Lustre, GPFS, PVFS, etc.) that is guaranteed to survive failures and is used by the application to read input data, write output data and possibly save logging information or other intermediate data.

The CR mechanism also relies on the storage service to save the state of the application in a persistent fashion. Upon restart, it is assumed that the machines where the application is launched have access to the intermediate state previously saved on the storage service in order to initialize the application from it.

2.2. Desired features of CR

Several important properties need to be considered when designing CR mechanisms. We detail these properties below:

Performance. CR does not come for free: saving the application state periodically to persistent storage inevitably introduces a runtime-overhead and consumes storage space. Therefore, it is crucial to design a CR mechanism that aims to minimize the interruption time of the application during normal execution, as well as the amount of information necessary to capture the application state. Moreover, a restart needs to be able to quickly initialize the application from a previously saved checkpoint with minimal overhead.

Scalability. An important property of CR approaches is scalability: the ability to control and keep the performance overhead of the checkpointing process at acceptable levels, even when the application grows in size and complexity. Note that this property is independent of the scalability of the application itself: a poorly designed CR system can easily turn an otherwise scalable application into a non-scalable application.

Transparency. There are two basic approaches to CR: *application-level* and *system-level*. In the case of application-level checkpointing, it is the responsibility of the user to explicitly decide what variables need to be saved for each process and when the right moment has come to do so. In this case, a careful design can minimize the application state that needs to be saved for each checkpoint and thus overall overhead of both checkpointing and restart. However, with great power comes great responsibility: writing efficient CR code is a laborious and error-prone task that is becoming increasingly difficult as the application gains in complexity. At the opposite end is system-level checkpointing: this approach is completely transparent and requires no modification at application-level. However, since no information is available about the application, all variables and and data structures need to be saved for each process. This can lead to suboptimal checkpoint sizes and increased runtime overhead, but greatly simplifies application design. Overall, advocating for one approach or another is not trivial and is highly application-dependent. For this reason, it is important to provide support for both approaches.

Portability. Using CR for migration raises portability issues: the checkpoints should be easy to move from one platform to another. This does not simply imply being able to run the CR mechanism on a different platform: what is really needed is the ability to checkpoint the application on one platform, transport the checkpoints to a different platform and then restart the application there. In the context of IaaS clouds, this problem is not trivial: there are differences in hardware and virtualization technologies employed by cloud providers. This can introduces incompatibilities (e.g. incompatible virtual machine image formats due to different hypervisors) that must be dealt with.

Manageability. In order to optimize the checkpointing process, many approaches introduce optimizations that decompose the checkpoints into smaller, inter-dependent pieces [48, 39]. This is done in order to speed up the checkpointing performance, at

the expense of having to reconstruct the checkpoint at restart time. Since restarts are considered to occur much more seldom than checkpointing, it is an acceptable trade-off. However, many small pieces and their dependencies are difficult to manage. For example, if a piece is accidentally lost, all checkpoints that depend on that piece become corrupted and cannot be used to successfully restart the application. This can become a complex issue, especially when storage space needs to be reclaimed by deleting old checkpoints. Thus, in order to ease the management of checkpoints it is desirable to work with checkpoints as *first-class objects*, i.e. single independent entities.

Rollback of changes to persistent storage. Applications often interact during the computation with the persistent storage to save logging information or other intermediate data. On restart, all these interactions with the "outside world" become undesired side-effects of the computation that followed after the checkpoint. In some cases, these side-effects are harmless (e.g. the application will overwrite files with the same contents). However, this is not always the case: for example, a common pattern is to append data to files as the computation progresses (such as status updates in log-files), which can lead to inconsistencies that affect the final results. Thus, a CR mechanism should be able to support rollback of changes to the underlying storage.

3. Challenges of CR on IaaS clouds

IaaS clouds are typically built on top of clusters made out of loosely-coupled commodity hardware [7]. Each node is equipped with local disk storage in the order of several hundred GB, while interconnect is provided by main-stream networking technology, such as Ethernet. Users leverage these resources in form of virtual machine instances (VMs) that are hosted on the nodes of the cluster. Each node typically has hardware virtualization support and runs a hypervisor that takes advantage of it for VM hosting.

Given this configuration, there are two challenges that play an important role in the design of CR mechanisms.

3.1. How to provide persistency.

In order to provide persistent storage, clouds typically employ a dedicated repository that is hosted separately, either in a centralized or distributed fashion. At a first glance, it may seem as if the cloud repository could play the role of a parallel file system and provide persistency for the application.

However, repositories on clouds mostly offer a different access model for user data (e.g. key-value stores using a REST-ful access API [8], database management systems [27], message queues [13] etc.). Such differences in access model can pose a serious problem: they may require significant changes to the application, which are not always feasible, either because of technical issues (e.g. no support for Fortran, which is widely used for HPC applications) or prohibitive development costs. This problem is also accentuated by the lack of standardization: different cloud providers offer different data access models that limit portability. Furthermore, cloud repositories do not offer out-of-the-box support to roll back changes, which (as explained in Section 2.2) is an important feature.

Thus, there is a need for a persistency option that overcomes these obstacles.

3.2. How to capture the state of the application.

In the most general case, the state of the computation is defined at each moment in time by two main components: (1) the state of application process; and (2) the state of the communication channels between them (opened sockets, in-transit network packets, virtual topology, etc.).

Since the applications we consider rely on message passing, there are complex inter-process dependencies that make it difficult and expensive to capture the state of the communication channels into the global state. For this reason, (2) is typically avoided in favor of alternative schemes. In the case of application-level CR, a synchronization point is typically used right before checkpointing in order to guarantee that all messages have been consumed. A similar technique is also widely leveraged in practice for system level checkpointing that uses a coordinated protocol [14], both for the blocking and non-blocking case. More recently, uncoordinated checkpointing protocols, which previously received little attention in practice due the cost and complexity introduced by message logging [6] have been increasingly considered for certain classes of HPC applications [23].

In this paper we do not focus on the techniques used to deal with (2), as they are widely covered in the literature and can be used to complement our work. Thus, for the purpose of our work we focus only on (1): we assume the global application state is reduced to the sum of the states of all its processes. There are two approaches to capture the state of a process in a transparent fashion:

Take a snapshot of its virtual machine instance. Several methods have been established in the virtualization community to capture the state of a running VM instance (RAM, CPU state, state of devices, etc.). An advantage of this option is the fact that it captures not only the state of the process itself, but the context of the operating system as well, which ultimately means that a reinitialization of the environment is avoided on restart (boot VM instance, configure application, etc.). However, at the same time it has an important disadvantage: the VM instance snapshots can explode to huge sizes (e.g. saving 2 GB of RAM for 1,000 VMs consumes 2 TB of space), which can lead to undesired consequences: (1) unacceptably high storage space utilization for a single one-point-in-time checkpoint; (2) performance degradation because of large data transfers to and respectively from persistent storage.

Use a process-level checkpointer. In this case, only the process state (process context, allocated memory regions, etc.) is saved, while the context of the virtual machine instance is discarded. Several approaches have been proposed to achieve this (e.g. *BLCR* [17]): essentially they dump the whole process state into a regular file (called process image) and are able to restore the process from that file. Since the state of the virtual machine instance is discarded, there is an additional overhead on restart, as the virtual machine instance needs to be redeployed. However, a much smaller checkpoint size is generated, which has a three-fold benefit: (1) it lowers overall storage and bandwidth costs; (2) during checkpointing it reduces the performance overhead because of smaller data transfers to persistent storage; (3) during restart it compensates for the

overhead of rebooting the VM instances by having to read less data from persistent storage.

Our previous work [36] shows that using a process-level checkpointer can save storage space in the order of hundreds of MB per VM instance, with a checkpointing overhead of up to 8x smaller. Furthermore, it can reach an overall restart speed-up of up to 6x, despite the need to reboot VM instances. Starting from these findings, we advocate for the use of process-level checkpointers for system-level CR.

4. Our approach

This section details our approach: Section 4.1 insists on the key ideas of our approach; Section 4.2 illustrates how our approach integrates in an IaaS infrastructure; Section 4.3 provides an algorithmic description for the proposed design principles; finally, Section 4.4 insists on some implementation details.

4.1. Design principles

Our approach relies on a series of key design principles, detailed below.

4.1.1. Rely on virtual machine disk-image snapshots

The key idea of our proposal is to save both the output data and state of the application to the virtual disks attached to the VM instances and then take persistent snapshots of the images corresponding to those disks. This approach solves both challenges presented in Section 3 simultaneously, as discussed below.

First, it provides a persistency solution (as discussed in Section 3.1) by enabling each process to rely on the file system of their virtual machine instance for all I/O. This option is not completely equivalent to using a parallel file system: in addition to message passing, processes running on different nodes could theoretically also synchronize by sharing files, a feature that is not available in our case. However, in practice this feature is not needed: for scalability reasons (in particular to avoid I/O bottlenecks), each process typically manipulates its own set of files independently of the other processes.

Starting from this assumption, the requirements identified in Section 3.1 are satisfied: (1) transparency is guaranteed because the file system of each VM instance is implicitly POSIX-compliant and thus there need to change the application; (2) portability is guaranteed because the filesystem is under the direct control of the guest operating system, which is independent of the physical host where the VM is running (e.g. the virtual disk can be safely migrated to a different cloud provider without raising incompatibility issues); and (3) reverting to a virtual disk snapshot implicitly rolls back all file system changes made after the snapshot was taken.

Second, it provides an efficient means to capture the application state, both in the case of application-level checkpointing and system-level checkpointing. This is achieved in a two-stage procedure: first the process state is saved as files into the file system of the VM instance, then a snapshot of the virtual disk is taken immediately after. In the case of system-level CR, we rely on a process-level checkpointer to capture the process state into a file. As discussed in Section 3.2, this choice saves considerable amount of storage space and bandwidth, while bringing important performance improvement when compared to full VM instance snapshots.

Synchronizing the VM instance with its host: the CHECKPOINT primitive. Note that the virtual disk must be snapshotted outside of the VM instance, while the application is running inside the VM instance. These two environments run concurrently and are isolated one from another. Thus, it is impossible to determine from the outside when it is safe to take a snapshot (i.e. when the first stage of the checkpointing procedure has completed). To solve this issue, a synchronization mechanism is necessary that enables each VM instance to request a snapshot of its disk to the outside and then wait for an acknowledgment to know when it is safe to continue. To fill this role, we introduce the CHECKPOINT primitive, which must be integrated into the checkpoint protocol and must be called either directly at application-level (for application-level CR) or inside the message passing system (for system-level CR).

4.1.2. Leverage local disk storage available on compute nodes

In most cloud deployments [7, 4, 5], the disks locally attached to the compute nodes are not exploited to their full potential. These disks have a capacity of hundreds of GB that normally serves as scratch space for the VM instances, yet only a fraction of it is actually used. Starting from this observation, we propose to aggregate parts of the storage space from all compute nodes in order to build a distributed checkpoint repository specifically designed to store VM disk-image snapshots persistently. Each snapshot is stored in a striped fashion: it is split into small, equal-sized chunks that are evenly distributed among the local disks of the checkpoint repository. Using this load-balancing strategy effectively distributes the I/O workload among the local disks, guaranteeing that no local disk becomes a bottleneck due to heavier load compared to others.

Furthermore, each snapshot is locally mirrored: it 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.

Using this scheme, our approach achieves the high scalability requirement presented in Section 2.2. First, one can observe that a growing number of compute nodes automatically leads to a larger checkpoint repository, which is not the case when using dedicated storage resources. Furthermore, there is no limit on the total I/O bandwidth except the limit of the interconnect between compute nodes itself. Second, data striping greatly enhances the scalability of read and write accesses under concurrency, as the global I/O workload is evenly distributed among the local disks. Finally, local mirroring conserves overall I/O bandwidth because data is stored on the local disk and written remotely only when the CHECKPOINT primitive is invoked.

Since virtual disk snapshots need to be stored in a persistent fashion, fault tolerance becomes a critical concern. Considering that we use unreliable local disks to store the chunks, there is a need to introduce a resilience mechanism. One simple solution is to rely on *replication*, i.e. store the same chunk more than once on different local disks.

Besides addressing fault tolerance, replication also increases chunk availability, because concurrent reads can be served by different local disks independently. However, one major drawback of replication is the extra storage space and network bandwidth necessary to store and maintain multiple chunk copies, which implicitly also leads to higher checkpointing overhead. In the context of CR, this drawback is particularly important, because checkpoints are frequently written but only seldom read back (i.e. only during restart). Furthermore, there is no need to read the same checkpoint concurrently, because each VM needs to access its own checkpoint. For this reason, replication can lead to an inefficient use of available resources without bringing significant benefits for CR through high availability. To address this issue, we explored in our previous work [21] the use of *erasure codes* to provide resilience. Compared to replication, such an approach can reach up to 50% higher checkpointing throughput and 2x lower bandwidth / storage space consumption for the same reliability level.

4.1.3. Live incremental snapshotting using selective copy-on-write

Not all parts of the VM disk are touched between consecutive checkpoints. Therefore, saving the full disk for each VM instance unnecessarily generates duplicated data, leading to an explosion of storage space utilization, as well as an unacceptably high snapshotting time and I/O bandwidth consumption. To avoid this issue, a well known optimization is *incremental* snapshotting, i.e. to store persistently only the chunks that have changed since the previous snapshot.

Even when relying on such an optimization, the modified chunks have to be replicated and stored remotely, which can become a lengthily process. If the VM instance is permitted to run at the same time while this is happening, some chunks may be modified before they are persisted, which in turn may lead to inconsistencies. Therefore, an important property that needs to be obeyed by snapshotting is *atomicity*. A simple solution to provide atomicity is offline snapshotting, i.e. to stop the VM instance for the duration of the snapshotting. However, this approach can lead to high downtime, which negatively impacts application performance. Therefore, it is important to be able to take snapshots atomically without interrupting the execution of the VM instance. We refer to this ability as *live snapshotting*.

Live snapshotting is still an open issue. One potential solution to address it is *copy-on-write*: whenever a chunk needs to be modified, it is copied to an alternate location and all modifications are performed on the copy. The advantage of this approach is a minimal impact on application performance: the application is never interrupted, only writes are delayed by the time required to copy the chunks. However, there is also a disadvantage: copying the chunks to alternate locations increases fragmentation, which in turn decreases the performance of subsequent I/O, as there is need to jump from one location to another in order to access the required chunks. This may not even possible in some scenarios if the chunks are expected to be in a contiguous region, which can further complicate the adoption of copy-on-write (e.g. it may require some additional copies to create contiguous regions, which again has a negative impact on performance of I/O).

In order to deal with this disadvantage, we propose a principle that we call *selective copy-on-write*. Our goal is to eliminate fragmentation but still take advantage of the benefits provided by copy-on-write. To achieve this goal, we leverage the fact that

the snapshotting process does not access all chunks simultaneously. Therefore, when a write conflicts with a chunk that has not been persisted yet, two cases are possible: either the chunk is actively accessed by the snapshotting process or it is scheduled for access in the future. In the first case, it is necessary to wait for the snapshotting process in order to avoid an inconsistency. However, in the second case this can be avoided: it is still necessary to copy the chunk to an alternate location, but this time the snapshotting process can be redirected to the copy while the write can modify the original chunk. Thus, fragmentation is avoided at the expense of dealing with the first case. Since the first case is a rare occurrence, this can reduce the space overhead and provide better optimization opportunities compared to traditional copy-on-write. We detail an algorithmic description of how this works in Section 4.3.

4.1.4. Shadowing and cloning

Copy-on-write is typically implemented in traditional approaches through custom VM image file formats [20]: the incremental differences are stored as a separate file, while leaving the original file corresponding to the base disk image untouched and using it as a read-only backing file. Such copy-on-write images can depend themselves on other copy-on-write images, thus representing successive snapshots as a long chain of "patches".

This approach has the advantage of being easy to implement on top of conventional file systems, however, at the same time it presents two important disadvantages. First, it generates a chain of files that depend on each other, which, as discussed in Section 2.2, raises a lot of issues related to manageability. For example, accidental removal of one file in the chain essentially corrupts the whole set of incremental snapshots, rendering the checkpoints unusable. Second, a custom image file format is not portable and 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.

Thus, an approach is needed that addresses both the manageability and portability requirements. To this end, we leverage two features used by versioning systems: *shadowing* and *cloning* [32].

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, if a small part of a large file needs to be updated, shadowing enables the user to see 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. It is similar in concept to the fork system call.

With this approach, snapshotting can be performed in the following fashion. The first time a snapshot is built, for each VM instance a new checkpoint image is cloned from the initial backing image. Subsequent local modifications are written as incremental differences to the checkpoint image and shadowed as a new snapshot. For the rest of this paper, we denote this process using two primitives: CLONE and, respectively, COMMIT. In this way all snapshots of all VM instances share unmodified

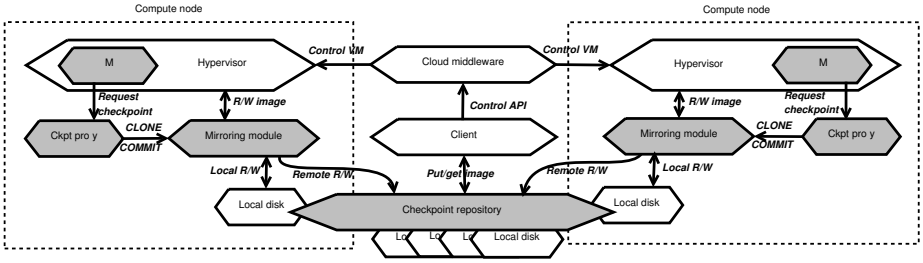


Figure 1: Our approach (dark background) integrated in an IaaS cloud.

content among one another and still appear to the outside as independent, *first-class* disk-images. Thus, they hide all dependencies introduced by incremental snapshotting from the user, which makes them much easier to manage. Furthermore, when using a simple raw image file as the initial backing image, all snapshots will themselves represent raw images, which are understood by most hypervisors and thus make our approach highly portable.

4.1.5. Lazy transfers and adaptive prefetching

Since our approach avoids saving the whole state of the VM instances, a restart implies that the instances are re-deployed and rebooted using the disk snapshots of the last checkpoint, after which the state of the processes is restored from the files. To optimize the performance of this process, both in terms of run-time and I/O bandwidth consumption, we introduce two optimizations.

First, as VM instances typically access only a small fraction of the VM image throughout their run-time, fetching only the necessary parts on-demand can reduce this overhead considerably [35]. Therefore, we propose the use of a "lazy" transfer scheme that fetches only the hot content of the disk image (i.e. the checkpoint files and any other files directly accessed at runtime by the guest operating system and the application).

Second, since the disk snapshots store only incremental differences, large parts of the images are shared and potentially need to be read concurrently by the VM instances during the boot process. In order to limit the negative impact of this issue, we exploit small delays between the times when the VM instances access the same chunk from the checkpoint repository (due to jitter in execution time) in order to prefetch the chunk for the slower instances based on the experience of the faster ones [38].

4.2. Architecture

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

A *checkpoint repository* that survives failures and supports cloning and shadowing is deployed on the compute nodes. The checkpoint repository aggregates part of the

storage space provided by the local disks of the compute nodes and is responsible to persistently store both the base and the disk images snapshots.

The *cloud client* has direct access to the checkpoint repository and is allowed to upload and download the disk images. Typically the user downloads and uploads base disk images only, however, thanks to shadowing and cloning, our approach enables the user to see and download checkpoint images as standalone entities as well. This feature that can become useful in a scenario where the checkpoints need to be inspected and even manually modified. Moreover, the cloud client interacts with the *cloud middleware* (the frontend of the user to the cloud) through a control API that enables deployments of a large number of VM instances starting from an underlying set of disk images.

Each compute node runs a *hypervisor* that is responsible to launch and execute the VM instances. The VM instances run in a modified guest environment that implements an extended CR protocol, which is able to ask the hosting environment to take a snapshot of its virtual disk. This is done through the *checkpointing proxy*, a special service that runs on the compute nodes and accepts checkpoint requests. Both for security and scalability reasons, the checkpointing proxy is not globally accessible: it accepts checkpoint requests only from the VM instances that are hosted on the same compute node.

All reads and writes issued by the hypervisor are trapped by the *mirroring module*, responsible to fetch the hot contents of the base disk image remotely from the repository and cache it locally. Local modifications to the base disk image triggered by writes are stored on the local disk as incremental differences. Whenever a checkpoint request is issued for the first time, the checkpointing proxy asks the mirroring module to create a checkpoint image that is derived from the base image (CLONE). This initial checkpoint image shares all contents with the base image. Then, the local modifications are committed to the checkpoint image as an incremental snapshot (COMMIT). Any subsequent checkpoint request will commit the local modifications recorded since the last checkpoint request as a new incremental snapshot into the checkpoint disk image.

Thanks to shadowing, it is possible to garbage-collect old local modifications that were overwritten by newer ones, despite the chain of dependencies introduced by the incremental snapshots. To this end, the CR protocol implementation can mark old snapshots as obsolete, which in turn enables the garbage collector to delete all changes that no subsequent snapshots depend upon.

A mapping between each successful checkpoint request and the resulting incremental snapshot together with its corresponding checkpoint image is maintained by the cloud middleware. In case of a failure or when the whole application needs to be terminated and resumed at a later point, all VM instances are re-deployed using a recent snapshot from their corresponding checkpoint image as the underlying virtual disk. It is the responsibility of the CR protocol implementation to pick a set of snapshots for the VM instances such that the application can roll back to a globally consistent state.

4.3. Algorithms

This section materializes the design principles presented in Section 4.1 into a series of algorithms that describe the interactions between the various building blocks presented in the previous section.

As a convention, we consider each disk-image snapshot as a set of chunks, each of which covers a well defined region of the virtual disk, delimited by *offset* and *size*. The initial image, configured by the user and used to deploy the virtual cluster is denoted *BaseImage*. This initial image is mirrored locally on the host of the virtual machine instance by the mirroring module.

The set of chunks that were either read or written during the lifetime of the VM instance is denoted *LocalMirror*. Each chunk of the *LocalMirror* is in a state that describes its relationship to the snapshotting process: it is either *Idle* (i.e. it is not part of any snapshotting request in progress), *Scheduled* (i.e. a snapshotting request runs in the background and the chunk is part of it, but the chunk was not accessed by the snapshotting process so far), or *Pending* (i.e. a snapshotting request runs in the background and actively accesses the chunk). All chunks are initially in the *Idle* state.

The CHECKPOINT primitive, exported by the checkpointing proxy is presented in Algorithm 1. Essentially it invokes remotely on the mirroring module the COMMIT primitive, responsible to commit all modified chunks since the last checkpoint request as a new snapshot of *CheckpointImage*, which in turn is cloned from *BaseImage* if the checkpoint request was issued for the first time.

Algorithm 1 Request a disk-image snapshot the VM instance

```
1:  function CHECKPOINT
2:      if first checkpoint request then
3:          CheckpointImage ← CLONE(BaseImage)
4:      end if
5:      M ← ∅
6:      for all c ∈ LocalMirror such that c was modified since last checkpoint request
    do
7:          M ← M ∪ {c}
8:      end for
9:      return COMMIT(M)
10: end function
```

Both CLONE and COMMIT are exported by the mirroring module. The CLONE primitive involves only a minimal metadata overhead that essentially gives the base image another name and enables it to evolve in a different direction. We do not detail this primitive here. The COMMIT primitive is detailed in Algorithm 2. It adds all chunks that were modified since the last checkpoint request to the *ScheduledSet* set, changing their state to *Scheduled*. After this step completed, it creates a new snapshot of *CheckpointImage* that is uniquely identified in the system by $snapshot_{id}$. Then it starts the snapshotting process (represented by the BACKGROUND_PERSIST primitive) in the background. Finally, it informs the checkpointing proxy of the snapshot corresponding to its checkpoint request through the return value $snapshot_{id}$. Having obtained this value, the checkpoint proxy acknowledges the checkpoint request back to the VM instance, signaling it that it is safe to continue.

At this point, the snapshotting process run concurrently with the VM instance inside the mirroring module. Its role is to store all chunks in the *ScheduledSet* set persistently to the checkpoint repository. This is presented in Algorithm 3 as an iterative process: a

Algorithm 2 Commit local modifications into a new VM disk-image snapshot

```
1: function COMMIT(M)
2:     ScheduledSet ← M
3:     for all c ∈ ScheduledSet do
4:         state[c] ← Scheduled
5:     end for
6:     snapshot_id ← generate new id
7:     start BACKGROUND_PERSIST(snapshot_id)
8:     return snapshot_id
9: end function
```

chunk is extracted from the set, put into the *Pending* state, transferred and replicated to the checkpoint repository and finally put into the *Idle* state. After all chunks have been successfully persisted, they are consolidated as a new snapshot of *CheckpointImage* using shadowing, after which the checkpoint is marked as stable and can be used for a restart from that point on.

Algorithm 3 Persist committed modifications to the checkpoint repository

```
1: procedure BACKGROUND_PERSIST(snapshot_id)
2:     while ScheduledSet ≠ ∅ do
3:         c ← extract chunk from ScheduledSet
4:         state[c] ← Pending
5:         write c persistently to repository
6:         ScheduledSet ← ScheduledSet\{c}
7:         state[c] ← Idle
8:     end while
9:     consolidate chunks using shadowing
10:    mark snapshot_id as stable
11: end procedure
```

A graphical illustration of the calls issued in parallel by the entities during a checkpoint request, from the initial checkpoint request of the VM instance to the moment when the checkpoint becomes stable, is depicted in Figure 2. Each call is represented as an arrow. Solid arrows are used to represent interactions between different entities, while a dotted pattern is used to represent interactions between components that run within the same entity.

To enable live snapshotting, write requests use selective copy-on-write, as explained in Section 4.1. More precisely, before modifying any chunk c that is involved in the write request, first a verification is made to check whether c needs to be persisted by the snapshotting process. If this is the case and c is in the *Pending* state, then the write request waits for the snapshotting process to finish persisting c. Otherwise, c is copied to an alternative location c', which replaces c in $ScheduledSet$, while the write continues normally on c itself. This process is illustrated in Algorithm 4.

For the rest of this section, we briefly analyze the proposed algorithms with respect

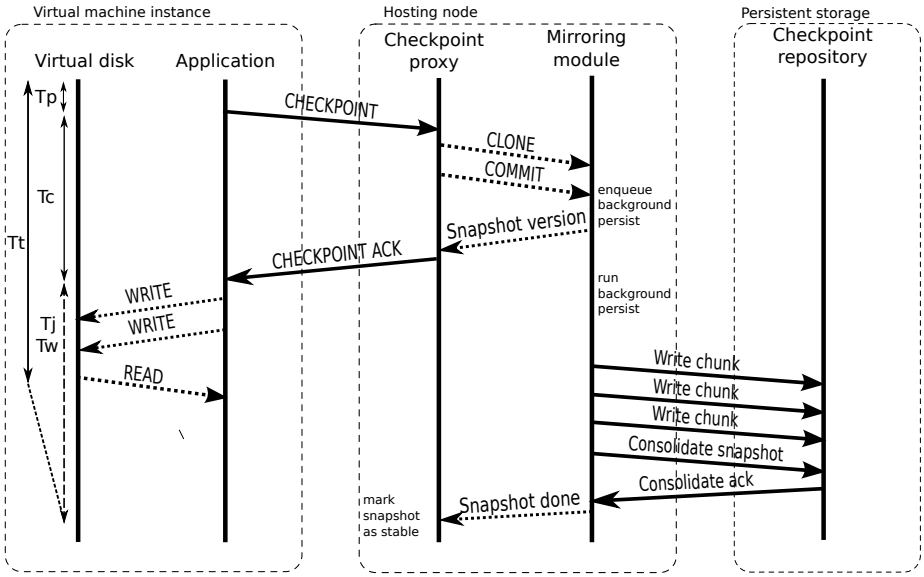


Figure 2: Overview of the checkpointing process

to the performance overhead they incur on the application.

Let's denote the total checkpointing overhead T_t. In our context, $T_t = T_p + T_c + T_w + T_j$, where T_p is the "preprocessing" overhead (i.e. the overhead of writing checkpointing data to the virtual disks), T_c is the downtime caused by calling the CHECKPOINT primitive, T_w is the downtime of write requests due to selective copy-on-write and T_j is the jitter caused by the background snapshotting. Figure 2 illustrates these variables in the context of our algorithms to make the model more intuitive. Note that T_w and T_j, although spread throughout the whole duration of the snapshotting, contribute only partially to T_t because of the overlapping with the application runtime: this is depicted as a dotted line that has a smaller corresponding solid section in T_t.

T_c can be traced down to cloning a base image if necessary and putting all locally modified chunks into the *Scheduled* state. Since these operations incur only a minimal metadata overhead compared to the rest of the snapshotting process (tens to hundreds of ms), we consider T_c negligible. Therefore, $T_t \approx T_p + T_w + T_j$. Based on this result, we conclude that three main factors dominate the checkpointing overhead perceived by the application: (1) how long it takes to dump the checkpointing data to the virtual disk (T_p); (2) the amount of snapshotted data overwritten by the application in the time window between the moment when a snapshot is requested and the moment when it has been persistently stored – in other words writes that cover chunks still in the *ScheduledSet* set (proportional to T_w); and finally (3) the amount of data written to the virtual disk since the last checkpoint (which determines how long it takes to persist the chunks in the background and thus is proportional to T_j).

Algorithm 4 Write using selective copy-on-write

```
1:  function WRITE(buffer, offset, size)
2:      for all c ∈ LocalMirror such that c ∩ (offset, size) ≠ ∅ do
3:          if c ∈ ScheduledSet then
4:              if state[c] = Pending then
5:                  wait until state[c] = Idle
6:              else if state[c] = Scheduled then
7:                  c' ← copy of c
8:                  ScheduledSet ← ScheduledSet\{c}
9:                  ScheduledSet ← ScheduledSet ∪ {c'}
10:             end if
11:         end if
12:         write from buffer to c ∩ (offset, size)
13:     end for
14:     return success
15: end function
```

4.4. Implementation

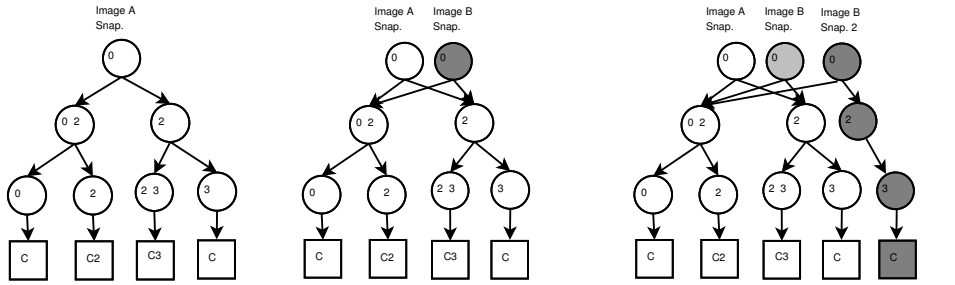
We implemented our approach into the preliminary *BlobCR* prototype introduced in [36]. In this section, we briefly describe its building blocks (presented in Section 4.2).

The distributed checkpoint repository was implemented on top of *BlobSeer* [32, 34], a distributed storage service specifically designed to efficiently support incremental updates by means of cloning and shadowing. More specifically, each VM snapshot is stored as a large sequence of bytes (BLOB) that is automatically striped into chunks and stored in a distributed and resilient fashion (either replicated or erasure coded) among the participating nodes. All metadata that describes the composition of BLOBs is itself striped and stored in a distributed fashion.

The *mirroring module* was implemented on top of *FUSE* (File System in UserspacE) [2], and relies on our previous work presented in [35]. It exposes each checkpoint image as a directory and the associated snapshots as files in that directory, accessible from the outside using the regular POSIX access interface. Any reads and writes issued by the hypervisor are trapped by FUSE and treated in a special fashion.

Reads implement the lazy transfer and adaptive prefetching mechanism introduced in Section 4.1.5: the BLOB is accessed on-demand only and all needed content is cached on the local disk for faster future reads of the same regions. Furthermore, prefetching hints piggybacked on each read are used to pre-read and cache contents that is expected to be accessed in the near future. Writes implement the selective copy-on-write strategy introduced in Section 4.1.3. 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



(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.

possible through *distributed segment trees* that are enriched with versioning information [34]. More precisely, each tree node covers a region of the BLOB, with leaves covering chunks directly and non-leaf nodes covering the combined range of their left and right children. Non-leaf 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. CLONE is a trivial operation in this context: 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.

Note that although the mirroring module is subject to failures, all VM disk snapshots and their associated metadata are stored in a resilient fashion into BlobSeer. Thus, if a failure occurs while the VM disk snapshotting is in progress, it is easy to identify the VM disk snapshots of the latest checkpoint confirmed to be fully committed to BlobSeer and restart from there.

The *checkpointing proxy* was implemented as a service that listens on a specified port for incoming TCP/IP connections originating from VM instances that resides on the same compute node where the proxy is deployed. If application-level checkpointing is desired, the checkpointing proxy can be directly contacted from within the application code. In order to enable process-level checkpointing, the user must install a modified MPI [22] library implementation based on *mpich2*. The checkpointing protocol itself is an extension of the original implementation available in mpich2. It uses the scheme presented in Section 3.2, relying on *blcr* [17] to dump the checkpoint of the MPI processes into files. We added extra code to this checkpointing protocol that flushes all guest caches to the virtual disk and then invokes the CHECKPOINT primitive right after BLCR has finished saving the process image.

5. Evaluation

This section evaluates the benefits of our proposal both in synthetic settings and for real-life applications.

5.1. Experimental setup

The experiments were performed on Grid'5000 [11], an experimental testbed for distributed computing that federates nine sites in France. We used 90 nodes of the graphene cluster from the Nancy site, each of which is equipped with a quadcore Intel Xeon X3440 x86_64 CPU with hardware support for virtualization, local disk storage of 278 GB (access speed ≃55 MB/s using SATA II ahci driver) and 16 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of ≃0.1 ms).

The hypervisor running on all compute nodes is Qemu/KVM 1.0, while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 4 GB raw disk image file based on the same Debian Sid distribution was used as the guest operating system. Inside this guest OS, we installed a modified mpich2 library (based on the 1.3.x development branch) that integrates our approach.

5.1.1. Methodology

The experiments we perform involve a set of VM instances, each of which is running on a different compute node. Inside the VM instances, we run either a synthetic benchmark or a real application that is checkpointed at regular intervals. The checkpointing process has an application-specific part (detailed for each experiment) that writes checkpointing data to the virtual disks, after which it calls the CHECKPOINT primitive for each VM to initiate the snapshotting process. We compare five snapshotting approaches, listed below:

Live incremental disk snapshotting using our approach. In this setting we rely on BlobCR to store base disk image and on the FUSE-based mirroring module to expose a locally modifiable view of the disk image to the hypervisor. BlobSeer is deployed on all compute nodes and stores the initial base disk image (4 GB) in a distributed fashion, using a stripe size of 256 KB (which from our previous experience is large enough to avoid excessive fragmentation overhead, yet small enough to avoid contention under concurrent read accesses). All chunks are replicated three times in order to guarantee resilience. Any CHECKPOINT request is handled using selective copy-on-write, as presented in Section 4.3. For the rest of this paper, we refer to this setting as our−approach.

Live incremental disk snapshotting using traditional copy-on-write. This setting is very similar to the previous setting, except for the way the CHECKPOINT primitive is handled: rather than using selective copy-on-write, we use traditional copy-on-write. More specifically, writes that occur during the snapshotting process create a copy of the involved chunks and perform the writes there. After the snapshotting process has completed, all newly generated chunks replace the old chunks. We refer to this setting as live−cow.

Live incremental disk snapshotting using local pre-copy. The same configuration is used for this setting as well, however this time we avoid copy-on-write altogether in favor of a full local pre-copy strategy: all modified chunks are first copied to the local disk and then persisted to the checkpoint repository in the background. This way, writes never conflict with the snapshotting process and can be treated normally. We refer to this setting as live−precopy.

Offline incremental disk snapshotting. This setting differs from the previous configuration only in that chunks are directly persisted to the checkpoint repository rather than copied locally and persisted in the background. It essentially avoids any jitter caused by background transfers at the expense of higher downtime. We refer to this setting as offline−disk.

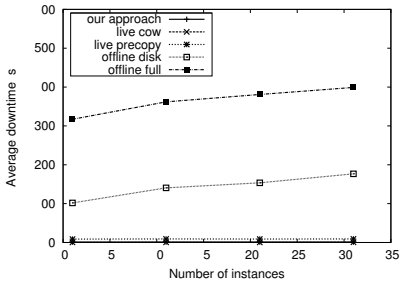
Offline incremental full VM snapshotting. Finally our last setting takes a complete incremental snapshot of the VM, including memory and CPU state besides the disk. In order to store these VM snapshots, we deploy *PVFS*, a highly popular parallel file system. We fix the the chunk size to 256 KB, the same size used for BlobSeer. To achieve incremental snapshotting, we build a derived *qcow2* image for each VM instance and then use the savevm QEMU monitor command when the CHECKPOINT primitive is called. We refer to this setting as offline−full.

These approaches are compared based on the following metrics:

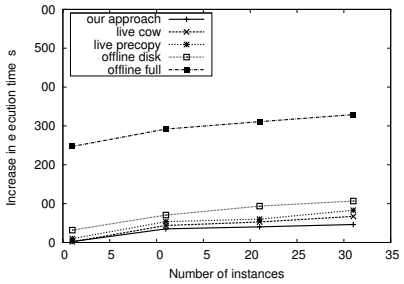
- *Average snapshotting downtime*: is the average time per VM instance required to execute the CHECKPOINT primitive (i.e. T_c). This metric is important because it shows the degree by which the application execution can be overlapped with the VM disk snapshotting process. Such an insight is helpful in understanding what the maximal theoretical benefit of live approaches is when compared to offline approaches in an ideal situation where there is no interference because of background transfers (i.e. $T_j = T_w = 0$).

- *Impact on application performance*: represents the increase in application execution time compared to the baseline (i.e. when no checkpointing happens). This metric shows the real benefits of each approach when taking all overhead sources into consideration. More specifically, it offers an insight into the degree by which T_j (zero for offline, nonzero for live) and T_w (potentially non-zero for our−approach and live−cow, zero for the rest) influence the application execution ($T_j + T_w$ = Impact - T_c).

- *Average snapshotting completion time*: represents the average time per VM instance required to transfer and store the chunks in a persistent fashion. This metric shows how much it takes until the snapshot is successfully committed and can be safely used for restart. It also indicates the period of time over which the effects of T_j and T_w are spread making it easier to understand the implications of overlapping application runtime with snapshotting.



(a) Average VM downtime due to checkpointing requests (lower is better)



(b) Time to complete benchmark (lower is better)

Figure 4: Benchmarking results using a 1 GB data buffer per VM instance. Memory size is 4 GB/instance

5.2. Synthetic benchmarks

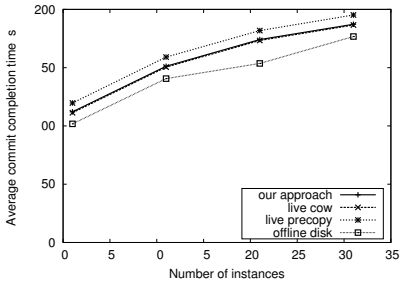
The first series of experiments evaluates the scalability of our proposal in controlled synthetic settings.

To this end, we implemented a simple benchmarking application that consists of a configurable number of processes, each of which runs in a dedicated VM instance. Each process independently allocates a fixed amount of memory as a data buffer and fills it with random data. The checkpointing process itself consists in saving the memory buffer to the virtual disk, after which the CHECKPOINT primitive is called. In order to create a non-trivial context that generates conflicts between the snapshotting process and subsequent writes, we start a second iteration immediately after returning from the CHECKPOINT primitive.

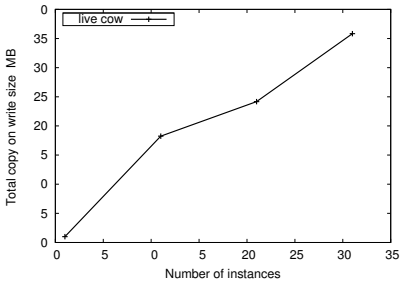
The experiment itself consist in deploying an increasing number of VM instances and then concurrently launching a benchmarking application process in each of the instances.

Results are shown in Figure 4. As expected, the downtime (Figure 4(a)) caused by CHECKPOINT is almost negligible for our−approach and live−cow: it remains close to constant in the order of hundreds of milliseconds. In the case of live−precopy, we observe a close to constant downtime too, however it stabilizes at almost two orders of magnitudes higher (around 10s) due to the initial local copy of modifications. Finally, for offline−disk and offline−full an explosion of downtime is clearly visible, with an increasing trend due to growing I/O pressure under concurrency on the checkpointing repository and PVFS respectively.

Comparing the increase in application execution time to complete the benchmark reveals that all live snapshotting approaches have a large advantage over the offline approaches, thanks to the fact that the background transfers are overlapping with the benchmark execution. At the extreme, our−approach is almost 4x faster than offline−full and 80% faster than offline−disk. Figure 5(a) reveals why in greater detail: the overhead of copy-on-write is very small for all three asynchronous approaches compared to offline−disk and thus enables a high degree of overlapping between the computation and the application runtime. Furthermore, compared to live−cow, we achieve a smaller



(a) Average COMMIT completion time for the asynchronous approaches compared to offline−disk (lower is better)



(b) Aggregated live−cow fragmentation size due to chunks that triggered copy-on-write (chunk size is 256KB)

Figure 5: Under-the-hood benchmarking measurements using a 1 GB data buffer per VM instance. Memory size is 4 GB/instance

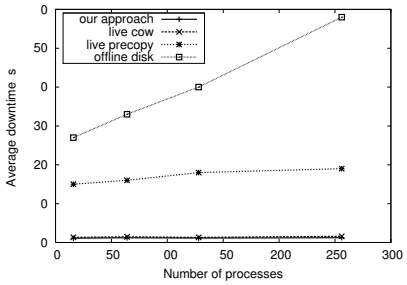
overhead due to the fact that we avoid fragmentation: in this case, our approach is almost 20% faster. To better understand this effect, Figure 5(b) depicts the aggregated size of all chunks from all VMs that triggered a copy-on-write in the case of live−cow, leading to fragmentation. As expected, we can see an increasing fragmentation in the system, which has a negative impact on the performance of the benchmark. Thanks to selective copy-on-write, our approach avoids this fragmentation alltogether, which explains the lower increase in execution time when compared to live−cow.

Finally, we measured the storage space and bandwidth consumed by the checkpointing process. All BlobCR approaches generate VM disk snapshots of little more than 1 GB, which is more than 3x less than offline−full. Since these have to be stored remotely, the consumed bandwidth is proportional to the checkpoint size.

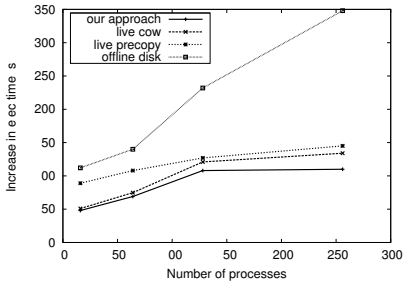
5.3. Real life application case study: CM1

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

CM1 is representative of a large class of scientific applications that model a phenomenon in time which can be described by a spatial domain that holds the value of fixed parameters in each point (temperature, pressure, etc.). Starting from such an initial spatial domain, the application calculates the evolution of the values of the parameters in each point according to a set of governing equations that involves the previous values of the parameters in that point and eventually its neighborhood. The problem is solved iteratively in a distributed fashion by splitting the spatial domain into subdomains, each of which is managed by a dedicated MPI process. At each iteration, the MPI processes calculate the values for all points of their subdomain, and then exchange the values at the border of their subdomains with each other.



(a) Average downtime per MPI process due to checkpointing requests (lower is better)



(b) Total increase in execution time (lower is better)

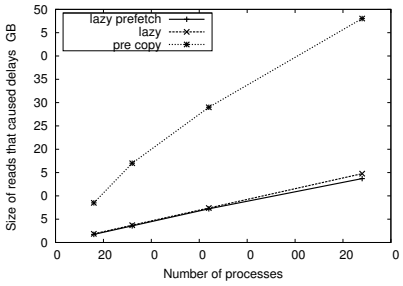
Figure 6: Checkpointing results using a real life HPC application: CM1. Each VM instance runs 4 MPI processes. Subdomain size per process is 200x200. Memory size is 4 GB/VM instance

CM1 is able to take application-level checkpoints by synchronizing the MPI processes to dump the contents of the subdomains into files. Each MPI process independently writes its own checkpoint file. Furthermore, at each fixed number of iterations, all MPI processes write intermediate summary information about the subdomains, again into independent files. For the purpose of this work, we have chosen a 3D hurricane that is a version of the Bryan and Rotunno simulations [12].

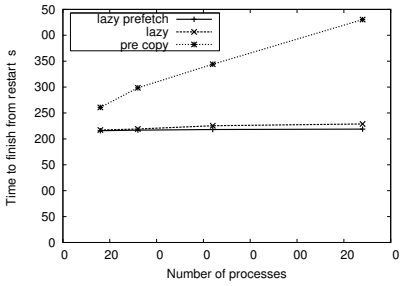
We study the weak scalability of our approach by solving the same problem using a different precision, in such way that the size of the subdomain solved by each process remains constant at 200x200, which roughly corresponds to 1 GB of new application-level checkpointing data per VM. The experiment consists in deploying an increasing number of quad-core VM instances, each of which hosts 4 MPI processes, one per core. We take three global checkpoints evenly spaced throughout the execution time and compute the average downtime and increase in execution time.

Results are shown in Figure 6. Again, the average downtime (Figure 6(a)) due to checkpointing is negligible in the case of our−approach and live−cow, growing to two orders of magnitude higher in the case of live−precopy. In all live approaches, a stable trend is noticeable, which hints at excellent scalability. On the other hand, offline−disk experiences a sharp increase in downtime due to remote I/O pressure on the checkpoint repository, which is hidden in the case of live snapshotting by the background transfers. Since CM1 is a memory-hungry application, the VM snapshot size obtain by offline−full was almost 5x larger than the rest, which in turned caused unacceptably high downtime that led to communication errors inside CM1. For this reason, the curve for offline−full was omitted.

Figure 6(b) depicts the performance overhead of all three live approaches compared to offline−disk. As can be observed, the reduction in downtime compared to offline−disk cannot be fully leveraged to reduce the performance overhead, because the asynchronous background transfers negatively impact the performance of the application: not only do they cause jitter and/or trigger copy-on-writes, but they also steal bandwidth away from the MPI processes themselves. Since CM1 is a bandwidth-



(a) Total size of reads that caused waiting for I/O, excluding prefetched data (lower is better)



(b) Total time to boot VM instances and restart CM1 from latest checkpoint to completion (lower is better)

Figure 7: Restart performance using a real life HPC application: CM1. Each VM instance runs 4 MPI processes. Subdomain size per process is 200x200. Memory size is 4 GB/VM instance

hungry application, the three live approaches perform closer than in the synthetic setting. Nevertheless, at less than 66% of the overhead of offline−disk, our approach leverages the downtime up to 20% better than live−cow which is very closely followed by live−precopy. With respect to scalability we note a downward trend in all three approaches, which hints at better opportunities to leverage the downtime as the I/O pressure on the checkpoint repository increases. We also note an increasing difference between our−approach and live−cow. This happens because of longer background snapshotting, which increases fragmentation overhead.

Finally, we run a series of experiments that evaluate the restart performance of our approach, in particular the lazy transfer and adaptive prefetching techniques introduced in Section 4.1.5. To this end, we use the VM disk snapshots of the last checkpoint from the previous experiment in order re-deploy all VM instances (i.e. boot the OS) and subsequently restart CM1. We compare three approaches: (1) lazy, which implements our on-demand read strategy complemented with local caching; (2) lazy−prefetch, which extends lazy with additional prefetching hints piggybacked on each read in order to pre-read and cache contents that is expected to be accessed in the near future; and (2) pre−copy, the traditional approach that pre-copies the whole VM disk content on the local disk before booting and running the VM instance.

Results are shown in Figure 7. The total size for reads that caused waiting for I/O is depicted in Figure 7(a). For lazy, this includes all reads intercepted by the FUSE interface for all VM instances. For lazy−prefetch, it includes all reads that were not already prefetched based on the hints. For pre−copy the total size represents the sum of the sizes of the individual VM disk snapshots. As can be observed, pre−copy needs to transfer 3 times more data than the other two approaches, thus confirming that only a fraction of the VM disk snapshot is actually needed on restart. Thanks to prefetching, it can be observed that lazy−prefetch causes less blocking reads compared to lazy. This results is also reflected in the total time to boot and restart CM1, which is depicted in Figure 7(b). As can be observed, both lazy and lazy−prefetch maintain good scalability,

with slightly less overhead for lazy−prefetch. As expected, pre−copy needs to transfer large amounts of data corresponding to the full VM disk snapshots before restarting CM1, which generates a large overhead compared to the other two approaches: the completion time is almost 2x larger.

6. Related work

An alternative mechanism to CR is provided Remus [15]: it enables asynchronous replication of state very similar to live migration for individual VM instances. Although this could be extended in a manner similar to [19], such an approach is not feasible in our context as it would lead to high performance overhead and an explosion of resource usage and thus operational costs.

There are previous efforts to build a dedicated checkpoint repository specifically designed to optimize for the CR access patterns, such as PLFS [10], proposed by Bent et al., which is a layer of indirection that remaps an application's preferred data layout into one which is optimized for the underlying parallel file system. A similar effort closer to our own approach, CRFS [40], relies on FUSE to intercept the checkpoint file write system calls and aggregate them into fewer bigger chunks which are then asynchronously written to the underlying file system. Unlike our approach (which naturally integrates with a dedicated storage layer), both proposals are heavily dependent on the performance characteristics of the underlying generic file system and its limitations.

Optimizations such a incremental checkpointing are commonly used both at application level and system level. However, unlike our approach, differences to previous checkpoints are stored as separate files, which raises manageability issues. Approaches such as [48], attempt to compensate for this effect using a hybrid CR mechanism that relies on incremental checkpoints to complement full checkpoints, with the purpose of avoiding indefinite accumulation of differences. Our approach avoids this problem altogether, thanks to shadowing.

The idea of departing from synchronous checkpointing in order to overlap the application execution with the checkpointing process has been exploited in several contexts. Quasi-synchronous checkpointing algorithms such as Manivannan et al. [28] limit contention to stable storage by staggering checkpoint requests in order to diminish the degree of concurrent I/O transfers. Another widely used approach is multi-level checkpointing [9, 31, 16], i.e. dump the checkpointing data on fast local storage and then asynchronously flush this data to globally persistent storage.

In the context of virtualization, several CR approaches based on full VM snapshots have been proposed [46, 45, 49]. This choice however comes at a high price: in addition to a large performance overhead, it generates an explosion of storage space and bandwidth utilization. To our best knowledge, we are the first to propose a CR framework for HPC applications based on incremental disk snapshots, which has the potential to drastically reduce the storage space utilization at the cost of minimal intervention inside the guest environment.

Many hypervisors provide native copy-on-write support using custom VM image file formats, such as *qcow2* [20] and *Mirage* [43]. This enables base images to be used as read-only templates for multiple VM disk snapshots that store per-instance modifications. However, unlike our approach, support for live incremental snapshotting is

currently not available. Furthermore, lots of files representing incremental differences need to be generated and shared through a parallel file system, which raises manageability and performance issues at large scales.

Several other approaches have been proposed in order to snapshot virtual disks, however we are not aware of any work that specifically targets CR. Lithium [25] is one such approach that supports fork-consistent, instant volume creation with lazy space allocation, instant creation of writable snapshots, and tunable replication. While this can prove a valuable building block that offers a viable alternative to cloning and shadowing, it is based on log-structuring [44], which can potentially incur a high read overhead the more incremental snapshots are taken. Parallax [30] enables 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. Amazon EBS [1] provides block level storage volumes that can be attached to Amazon EC2 [7] instances. Such volumes outlive the VM instances that mount and use them, which makes them a potential target to store the process state and all other intermediate files. Snapshotting is supported, however it is implemented over Amazon S3 [8], a key-value store not specifically optimized for this purpose.

7. Conclusions

With increasing interest in HPC applications among the mainstream community, cost-effective solutions that are affordable to the masses are highly desirable. In this context, IaaS clouds are a promising alternative to leadership-class supercomputers. However, due to differences in architecture and consumer needs, porting HPC applications to IaaS clouds is a challenging task that requires rethinking of several well established HPC approaches. One challenge in this context is the need to provide a high-performance, resource-friendly and scalable Checkpoint-Restart mechanism.

We proposed *BlobCR*, a dedicated checkpoint repository that is able to take *live incremental* snapshots of the whole disk attached to the virtual machine (VM) instances. Our approach supports both application-level and process-level checkpointing and includes the unique ability to implicitly roll back file system changes.

Compared to approaches that capture the whole VM state, our approach shows large performance gains and much lower bandwidth/storage space utilization. Based on our results with real life HPC applications, we incline to believe that full incremental VM snapshots at large scale are unfeasible in practice. Furthermore, by persisting VM disk snapshots asynchronously in the background, we show large reductions in checkpointing downtime (up to two orders of magnitude) compared to offline disk snapshotting. Finally, we show how to efficiently leverage this reduction in downtime to improve performance by means of selective copy-on-write: it significantly reduces the negative impact on the application due to background transfers compared to conventional copy-on-write. All these benefits are demonstrated not only using synthetic benchmarks but also through a real-life HPC application case study.

In future work we plan to explore the use of adaptive compression schemes [33] and/or deduplication bring further reductions in overhead and resource utilization of

CR. Furthermore, we are interested in the possibility of complementing checkpoint-restart with pro-active live migration: if failures can be predicted with a relatively high confidence level, then we could reduce the checkpointing frequency in favor of migrating unstable VMs to more safer nodes. We already explored the possiblity of live migration of local storage [37] with encouraging results.

Acknowledgments

This work was supported in part by the Agence Nationale de la Recherche (ANR) under Contract ANR-10-01-SEGI. 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] Amazon Elastic Block Storage (EBS). http://aws.amazon.com/ebs/.

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

[3] Nasa nebula. http://nebula.nasa.gov.

[4] Nimbus. http://www.nimbusproject.org/.

[5] Opennebula. http://www.opennebula.org/.

[6] Lorenzo Alvisi and Keith Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Trans. Softw. Eng.*, 24(2):149–159, February 1998.

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

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

[9] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *SC '11: Proceedings of 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 32:1–32:32, Seattle, USA, 2011. ACM.

[10] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the 22nd Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Portland, USA, 2009.

[11] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and

highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, 20:481–494, November 2006.

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

[13] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP '11: Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 143–157, Cascais, Portugal, 2011. ACM.

[14] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC '06: Proceedings of the 19th International Conference for High Performance Computing, Networking, Storage and Analysis*, Tampa, Florida, 2006. ACM.

[15] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, San Francisco, USA, 2008.

[16] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 8(2):6:1–6:29, June 2011.

[17] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Future Technologies Group, 2002.

[18] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[19] Kurt Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *SC '11: Proceedings of 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 44:1–44:12, Seattle, USA, 2011.

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

[21] Leonardo Bautista Gomez, Bogdan Nicolae, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Scalable Reed-Solomon-based Reliable Local Storage for HPC Applications on IaaS Clouds. In *Euro-Par '12: 18th International Euro-Par Conference on Parallel Processing*, Rhodes, Greece, 2012.

[22] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1999.

[23] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. In *IPDPS '11: 25th IEEE International Parallel and Distributed Processing Symposium*, pages 989–1000, 2011.

[24] Abhishek Gupta, Laxmikant V. Kal, Dejan S. Milojicic, Paolo Faraboschi, Richard Kaufmann, Verdi March, Filippo Gioachin, Chun Hui Suen, and Bu-Sung Lee. Exploring the Performance and Mapping of HPC Applications to Platforms in the Cloud. In Dick H. J. Epema, Thilo Kielmann, and Matei Ripeanu, editors, *HPDC '12: 21th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 121–122, Delft, The Netherlands, 2012. ACM.

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

[26] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. Case study for running hpc applications in public clouds. In *HPDC '10: Proceedings of the 19th International Symposium on High Performance Parallel and Distributed Computing*, pages 395–401, Chicago, USA, 2010.

[27] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[28] D. Manivannan, Q. Jiang, Jianchang Yang, and M. Singhal. A quasi-synchronous checkpointing algorithm that prevents contention for stable storage. *Inf. Sci.*, 178(15):3109–3116, August 2008.

[29] Piyush Mehrotra, Jahed Djomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. Performance evaluation of Amazon EC2 for NASA HPC applications. In *ScienceCloud '12: Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, page 4150, Delft, The Netherlands, 2012. ACM.

[30] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(4):41–54, 2008.

[31] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system.

In *SC '10: Proceedings of the 23rd International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, New Orleans, USA, 2010. IEEE Computer Society.

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

[33] Bogdan Nicolae. On the Benefits of Transparent Compression for Cost-Effective Cloud Data Storage. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 3:167–184, 2011.

[34] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, 71:169–184, February 2011.

[35] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In *HPDC '11: The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 147–158, San Jose, USA, 2011.

[36] Bogdan Nicolae and Franck Cappello. Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In *SC '11: 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 34:1–34:12, Seattle, USA, 2011.

[37] Bogdan Nicolae and Franck Cappello. A Hybrid Local Storage Transfer Scheme for Live Migration of I/O Intensive Workloads. In *HPDC '12: 21th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 85–96, Delft, The Netherlands, 2012.

[38] Bogdan Nicolae, Franck Cappello, and Gabriel Antoniu. Optimizing multi-deployment on clouds by means of self-adaptive prefetching. In *Euro-Par '11: 17th International Euro-Par Conference on Parallel Processing*, pages 503–513, Bordeaux, France, 2011.

[39] Xiangyong Ouyang, Karthik Gopalakrishnan, Tejus Gangadharappa, and Dhabaleswar K. Panda. Fast checkpointing by write aggregation with dynamic buffer and interleaving on multicore architecture. In *HiPC '09: Proceedings of the 16th International Conference on High Performance Computing*, pages 99–108, Kochi, India, 2009.

[40] Xiangyong Ouyang, Raghunath Rajachandrasekar, Xavier Besseron, Hao Wang, Jian Huang, and Dhabaleswar K. Panda. Crfs: A lightweight user-level filesystem for generic checkpoint/restart. In *ICPP '11: Proceedings of the 40th International Conference on Parallel Processing*, pages 375–384, Washington, DC, USA, 2011. IEEE Computer Society.

[41] Tran Vu Pham, Hani Jamjoom, Kirk Jordan, and Zon-Yin Shae. A service composition framework for market-oriented high performance computing cloud. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 284–287, Chicago, USA, 2010.

[42] Lavanya Ramakrishnan, Piotr T. Zbiegel, Scott Campbell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, and Anping Liu. Magellan: experiences from a science cloud. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 49–58, San Jose, USA, 2011.

[43] Darrell Reimer, Arun Thomas, Glenn Ammons, Todd Mummert, Bowen Alpern, and Vasanth 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, Seattle, USA, 2008.

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

[45] G. Vallée, T. Naughton, H. Ong, and S.L. Scott. Checkpoint/restart of virtual machines based on Xen. In *HAPCW '06: Proceedings of the High Availability and Performance Workshop*, Santa Fe, USA, 2006.

[46] Oreste Villa, Sriram Krishnamoorthy, Jarek Nieplocha, and David M. Brown, Jr. Scalable transparent checkpoint-restart of global address space applications on virtual machines over Infiniband. In *CF '09: Proceedings of the 6th ACM Conference on Computing Frontiers*, pages 197–206, Ischia, Italy, 2009.

[47] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204, Indianapolis, USA, 2010.

[48] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Hybrid Checkpointing for MPI Jobs in HPC Environments. In *ICPADS '10: Proc. of the 16th International Conference on Parallel and Distributed Systems*, pages 524–533, Shanghai, China, 2010. IEEE Computer Society.

[49] Minjia Zhang, Hai Jin, Xuanhua Shi, and Song Wu. VirtCFT: A transparent vm-level fault-tolerant system for virtual clusters. In *ICPADS '10: Proceedings of the 16th International Conference on Parallel and Distributed Systems*, pages 147–154, Shanghai, China, 2010. IEEE Computer Society.