# Checkpoint-based Fault-tolerant Infrastructure for Virtualized Service Providers

Íñigo Goiri, Ferran Julià, Jordi Guitart, and Jordi Torres
Barcelona Supercomputing Center and Technical University of Catalonia
Jordi Girona 31, 08034 Barcelona, Spain
{igoiri,fjulia,jguitart,torres}@ac.upc.edu

***Abstract*—Crash and omission failures are common in service providers: a disk can break down or a link can fail anytime. In addition, the probability of a node failure increases with the number of nodes. Apart from reducing the provider's computation power and jeopardizing the fulfillment of his contracts, this can also lead to computation time wasting when the crash occurs before finishing the task execution. In order to avoid this problem, efficient checkpoint infrastructures are required, especially in virtualized environments where these infrastructures must deal with huge virtual machine images.**

**This paper proposes a smart checkpoint infrastructure for virtualized service providers. It uses Another Union File System to differentiate read-only from read-write parts in the virtual machine image. In this way, read-only parts can be checkpointed only once, while the rest of checkpoints must only save the modifications in read-write parts, thus reducing the time needed to make a checkpoint. The checkpoints are stored in a Hadoop Distributed File System. This allows resuming a task execution faster after a node crash and increasing the fault tolerance of the system, since checkpoints are distributed and replicated in all the nodes of the provider. This paper presents a running implementation of this infrastructure and its evaluation, demonstrating that it is an effective way to make faster checkpoints with low interference on task execution and efficient task recovery after a node failure.**

## I. Introduction

The emergence of Cloud Computing has encouraged a lot of enterprises to rely on external providers to supply the services that they need to support their business processes. From the business point of view, the service provider agrees with its customers the Quality of Service (QoS) to be delivered through a Service Level Agreement (SLA), which is a bilateral contract between the customer and the service provider that states the conditions of service and details the penalties that the provider must reimburse when the service is not satisfied.

In order to be profitable, service providers tend to share their resources among multiple concurrent services used by different customers, but at the same time, they must guarantee the agreed SLA. This consolidation of services while supporting isolation from other services sharing the same physical resource has been accomplished by means of virtualization.

In addition to have several services running concurrently onto the same physical resources, service providers must be able to deal with hardware failures in order to fulfill their agreed SLAs (i.e. they must be fault-tolerant). Crash and omission failures are common in service providers: a disk can break down or a link can fail anytime. In addition, the probability of a node failure increases with the number of nodes. Apart from reducing the provider's computation power and jeopardizing the fulfillment of his contracts, this can also lead to computation time wasting when executing medium and long-running tasks and the crash occurs before finishing the task execution. For instance, if a task that takes 24 hours is executing in a provider's node, and this node crashes 5 minutes before the task execution ends, almost one day of execution will be wasted and the SLA will probably be violated.

In order to avoid this problem, checkpoint mechanisms have been proposed. These mechanisms record the system state periodically to establish recovery points. Upon a node crash, the last checkpoint can be restored, and task execution can be resumed from that point. However, making checkpoints is expensive in terms of performance. This is especially noticeable in virtualized service providers, considering that checkpoint mechanisms must deal with huge virtual machine images that must be also be saved and restored. For this reason, these environments require efficient checkpoint infrastructures.

This paper proposes a smart checkpoint infrastructure for virtualized service providers, which uses Another Union File System (AUFS) to differentiate read-only from read-write parts in the virtual machine image. In this way, read-only parts can be checkpointed only once, while the rest of checkpoints must only save read-write parts. Furthermore, read-write parts are incrementally checkpointed, that is, only modifications from last checkpoint are stored. This reduces the time needed to make a checkpoint and, as a consequence, the interference on task execution. The checkpoints are stored in a Hadoop Distributed File System. Using this system, the checkpoints are distributed and replicated among all the nodes of the provider, and eliminates any single point of failure. In addition, there is not any performance bottleneck in the checkpoint mechanisms, because the checkpoint can be concurrently recovered from different nodes. This allows resuming faster task execution after a node crash under contention.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 describes the architecture and functioning of our checkpoint infrastructure. Section 4 presents the evaluation results. Finally, Section 5 presents the conclusions of the paper and the future work.

## II. Related Work

Failure management in high-performance computing systems and clusters has been widely studied in the literature [1],

978-1-4244-5367-2/10/$26.00 ©2010 IEEE

[2], [3], [4]. Different characterization approaches of these failures have been made. For instance, Fu and Xu predict the failure occurrences in HPC systems through the spatial and time correlation among past failure events [5]. Gokhale and Tivedi [6] forecast the software reliability representing the system architecture using Markov chains. Zang et al. [7] evaluate the performance implication of failures in large clusters. What we can extract from most of these works is that the systems failures occur, involve some performance penalty, and are difficult to predict. Even if we could perfectly predict them, we still need to apply some solution.

Cloud computing and virtualization have opened a new window in the failure management. Pausing, resuming, and migrating VMs [8], [9] are powerful mechanisms to manage failures in such environments. A VM can be easily migrated to another node when a failure is predicted or detected. Although migration is a good solution in order to deal with failures, it has two main problems. First, it has considerable overhead, especially if entire VM images have to be migrated. Second, the prediction and advance detection of failures is key issue. We cannot start the migration of a VM if its running node has already failed. Our approach overcomes these two problems by making checkpoints only of delta images and distributing and replicating them in the checkpoint storage.

The checkpoint and rollback technique [10] has been widely used in distributed systems. We can offer high availability by using it, while adding bearable overhead to our system. In our case, this overhead comes mainly from the time needed to save the state and the memory of a VM. This time is determined by the virtualization technology, and mechanisms for reducing is are out of the scope of this paper.

Different solutions using checkpoints have been proposed in the literature. For example, [11] proposes using a union file system in order to save time storing VM checkpoints. It also introduces a remote storage in order to store the checkpoints and introduces the storage of VM disk space during the checkpoint phase. In addition, it also proposes to store only the differences in the disk. Nevertheless, it does not provide any implementation and just presents a theoretical approach. Opposite to this, we present a working implementation and we evaluate its performance.

Ta-Shma et al. [12] present a CDP (Continuous Data Protection) with live-migration-based checkpoint mechanism. They use a central repository approach and intercept migration data flow to create the checkpoint images. Although the authors say that it has good performance, no experimentation is presented. The architecture presented does not seem to be able to make checkpoints of the VM disk data.

Parallax [13] developed by Warfield et al. is a storage subsystem for Xen to be used in cluster Xen Virtual Machines. The solution proposed by the authors makes coupled checkpoints of both memory and disk using a Copy-on-Write mechanism (CoW) to maintain the remote images. There is no real experimentation and no performance results of that prototype. It also seems to be a solution based on a centralized storage server, having then a single point of failure.

Finally, [14] presents a working architecture that makes the SLURM job scheduler able to support virtualization capabilities such as checkpointing. In order to evaluate their implementation, the authors make checkpoints of a MPI application multiple times. Nevertheless, they only focus on memory checkpoints, neglecting disk. In addition, they do not evaluate in detail the checkpoint mechanisms.

## III. Architecture of the Checkpoint Infrastructure

Our checkpoint mechanism is built on top of a virtualized service provider using Xen [15]. In order to execute the tasks, this provider uses Virtual Machines (VM) that are created on demand. The creation of a VM image involves dealing with a large amount of data, including the installation of the guest operating system and the deployment of the required software. A basic architecture of a virtualized service provider, which provides VM creation and efficient live-migration, is fully described and evaluated in [16]. The following sections detail how to implement a checkpoint infrastructure upon a virtualized service provider based on the described architecture.

### A. Checkpoint Content

Making a checkpoint of task running within a VM can imply moving tens of gigabytes of data, since it must include all the information needed to resume the task execution in another node: the task context, the memory content, and the disks. However, just a small amount of data is really changing with respect to the VM startup. According to this, our checkpoint mechanism mounts the base system as a read-only file system and stores the user modifications in an extra-disk space called *delta disk*. The distinction between read-only and read-write parts was initially proposed in [11], and has been also used in other environments such as Linux LiveCDs, which store the modifications external devices since it is not possible to modify or add any information to the CD. Different implementations for providing this kind of file system exist. Our checkpoint mechanism uses Another Union File System (AUFS) [17].

In order to apply this idea in a virtualized service provider, we need two different disks: a read-only one containing the base system and the *delta disk*, granted with read-write permissions, which will contain only the user space. These two disks have to be merged to create a root file system before starting the VM. This process is done by the ramdisk as proposed in [18]. Finally, when the VM has been already booted, the user can work with the file system in a transparent way without worrying about its underlying structure.

In order to make a checkpoint, we need to save the task current status (basically its memory and disk contents). Using our checkpoint infrastructure, the system only needs to save the system disk once the first checkpoint is done and then store the *delta disk* (i.e. the user disk) each time a checkpoint has to be performed. As [11] proposes, delta images just store the changes performed after the last checkpoint. Notice that this process reduces considerably the time needed to make the checkpoint. Reducing this time is especially important considering that a task should be stopped when doing a checkpoint because if we allow the task to perform changes on

its status while the checkpoint is being done, this could result in a checkpoint inconsistency, due to concurrent accesses.

Finally, once the checkpoint is ready, and while the VM is still stopped, it could be directly uploaded to the distributed file system. Nevertheless, as the checkpoint size can be large, the upload time can be large too, increasing the VM stop time. For this reason, our mechanism merges the read-only disk with the last changes (as it is shown in Figure 1), generates a new delta disk for storing future changes and resumes the VM execution immediately, and then starts uploading the checkpoint to the distributed file system in parallel with the VM execution.

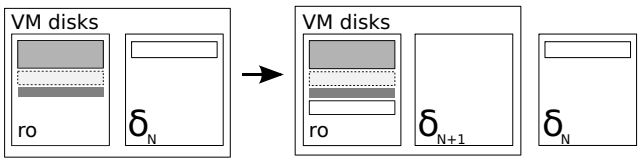


Figure 1. Checkpoint making process

Moreover, the creation of the new delta disk that will contain the future changes can be performed before initiating the checkpoint or when it has already finished. Therefore, this does not contribute to the time necessary to do a checkpoint.

*1) Checkpoint compression:* As the *delta disk* only contains the modifications that have been made from the last checkpoint and the rest of the image just contains zeros, it can be highly compressed. Considering this, we could replace the direct upload of the delta disk checkpoint to the distributed file system, with the compression of the checkpoint, and then, the upload of the compressed version. However, this must only be carried out if it allows saving time.

If there are not many changes, the *delta disk* can be highly compressed in a fast way. Nevertheless, if this disk contains many changes, it would be faster just to upload it than compress it and upload this compressed file to the distributed file system. Therefore, it is worth compressing the checkpoint while:

$$t\_compress + t\_upload_c < t\_upload_u \quad (1)$$

In this formula, $t\_compress$ refers to the time needed to compress the checkpoint, $t\_upload_c$ refers to the time needed to upload the compressed checkpoint, and $t\_upload_u$ refers to the time needed to upload the uncompressed checkpoint. Therefore, our mechanism checks the amount of data contained in the disk in order to decide whether it has to be compressed or not. $t\_compress$ can be estimated using the compression times of previous checkpoints, the size of current *delta disk*, and the amount of data (not zeros) within this *delta disk*. Furthermore, the upload times can be also estimated using the time needed to upload the previous checkpoints and the size of current *delta disk*.

Our system will just upload the checkpoint when it evaluates that this is faster than compressing and uploading it. Notice that the compression of the checkpoints is especially intended for applications with low disk consumption, since their *delta disks* will be more compressible. This compression also allows the system reducing the size of the checkpoints that must be stored in the file system. In order to save space in case too many delta images have been stored, they can be merged.

In the future, we plan to incorporate the preferences of the application in order to decide when to compress the checkpoints. For instance, an application could prefer to wait a little more, but use less resources from the network.

### B. Distributed File System

In this section, we discuss which the best alternative for storing the checkpoints is. Obviously, they cannot be stored in the node where the task is running, because if this node crashes, the checkpoint will not be accessible. According to this, the checkpoints must be stored in a remote disk space. However, storing the checkpoints in a single remote node is not a good solution either. This node would become a bottleneck for the performance of the checkpoint infrastructure. In addition, it would be a single point of failure, resulting in a not fault-tolerant solution.

In order to overcome these problems, our mechanism uploads the checkpoints to a distributed file system which supports also replication, namely the Hadoop Distributed File System (HDFS) [19]. This system splits the stored data in blocks that are distributed among the different nodes comprising the file system. Furthermore, it allows specifying the degree of replication of the blocks in the system.

As shown in Figure 2, we have decided that every node in the service provider becomes part of the distributed file system and stores some of the blocks of the checkpoints. Both the checkpoints of the base system and the *delta disks* are replicated and distributed among the different nodes. In this way, if one node crashes, the checkpoint can be restored from the other nodes using the different files and merging the different delta disks. In addition, as the checkpoint is replicated, it can be concurrently obtained from several nodes. Notice that the base system only needs to be uploaded to HDFS once when the first checkpoint is done, while the *delta disk* and the memory are uploaded for every checkpoint.

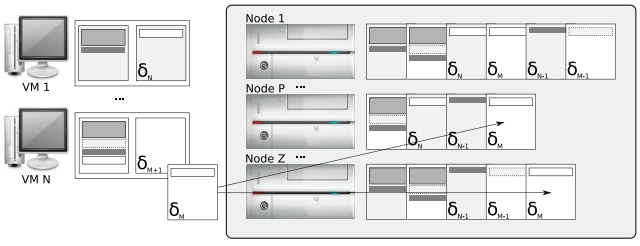


Figure 2. Checkpoint storage architecture

## IV. Evaluation

This section evaluates the performance of the presented checkpoint infrastructure and compares it with other alternatives. The service provider used in the evaluation consists of three different nodes, *Node A* which is a 64-bit architecture

with an Intel Xeon CPU at 2.6GHz and 16 GiB[1] of RAM memory, *Node B* which is a 64-bit architecture with 4 Intel Xeon CPUs at 3.0GHz and 16 GiB of RAM memory, and *Node C* which is a 64-bit architecture with 2 Intel Pentium D at 3.2GHz with 2 GiB of RAM. All the machines are connected through a Gigabit Ethernet.

Virtual machines are executed in *Node A* and *Node B*, while *Node C* contains the storage services. Furthermore, when using HDFS, all the nodes act as data nodes. Another important issue is the disk speed since it is a key issue when storing and copying checkpoints. The disk speeds are respectively: 117 MiB/s, 83 MiB/s, and 58 MiB/s.

The size of the different disks needed by a VM are shown in Table I. The size of the memory image depends on its size and the size of the user disk image depends on the user requirements and the usage of this image. Furthermore, a system that does not support the delta disks does not require the ramdisk, and the user disk only acts as a place to store user files: it does not save any changes of the system, these are directly stored in the base system image.

| | Size (MiB) |
|---|---|
| Configuration file | 0 |
| Ramdisk | 5.3 |
| Kernel | 1.6 |
| Base system | 769 |
| Memory (m) | $1 + m$ |
| User disk ($\delta$) | $8 + \delta$ |

Table I
SIZE OF VM DISKS

The application used in this experimentation consists of a CPU intensive application that can be executed with a memory size of 256 MiB. The execution of this application in *Node A* takes around 3221 seconds to be completed.

### A. Checkpoint Upload Protocol

Table II shows the time to upload the checkpoint of a VM using different storage protocols. This checkpoint, which has a size of 1496 MiB, is composed of the base OS image, the delta disk, the memory and VM state, the configuration file, the kernel, and the ramdisk of the VM.

| Storage | Time |
|---|---|
| HDFS 3 replicas | 53.185" |
| HDFS 2 replicas | 31.227" |
| HDFS 1 replica | 16.751" |
| FTP | 18.636" |
| SFTP | 33.25" |
| NFS | 26.73" |

Table II
CHECKPOINT UPLOAD TIME WITH DIFFERENT PROTOCOLS

As shown in this table, SFTP is slower than the other approaches due to encryption of the channel, whereas other centralized solutions such as FTP and NFS are faster than the distributed ones (based on HDFS). However, centralized solutions also imply a single point of failure and a bottleneck in order to store/recover multiple checkpoints concurrently. Furthermore, HDFS when using just one replica per file is similar to the centralized approaches, but it also stores parts of the file in the same node. Meanwhile, HDFS with a higher number of replicas has worse performance but distributes the checkpoints among different nodes, giving the system better fault tolerance and better performance.

[1]We use GiB (aka. Gibibyte, $2^30$) just for avoiding ambiguity with GB (aka. Gigabyte), which is sometimes used as $10^9$.

### B. Time to Make a Checkpoint

The time to make a checkpoint mainly depends on two different aspects: creating the checkpoint and uploading it to the checkpoint storage. This section evaluates the cost of making a checkpoint depending on the size of the memory of a VM, the size of user disk and its usage.

As commented in Section III-A, it is important to minimize the time needed to make a checkpoint as during this time the VM must be stopped. In these experiments, the VM is stopped during the categories 'Save memory', 'Save user disk', and 'Restart VM'.

*1) Depending on VM memory size:* Figure 3 shows the time needed to make a checkpoint and uploading it depending on the amount of memory of the VM. We have measured these times in a just created VM with a *delta disk* of 100 MiB that only contains the changes required to boot the VM. The base system has been already uploaded to the checkpoint storage, thus only regular checkpoints must be uploaded. Tests have been executed at *Node A*.

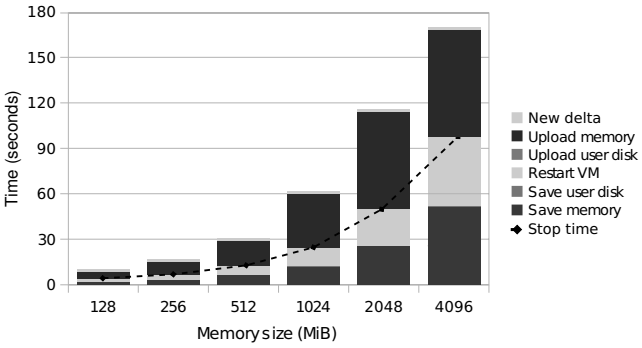


Figure 3. Checkpoint time depending on VM memory size

As shown in this figure, the time needed to make a checkpoint increases linearly with the size of the VM memory, basically due to the time required to save the memory and restart the VM execution after the checkpoint has been done. In addition, the size of the memory checkpoint is coupled with the memory size. Therefore, the time to upload the checkpoint of the memory is proportional with the memory size.

*2) Depending on user disk size:* Figure 4 shows the time required to make a checkpoint of a VM with 128 MiB of memory depending on the size of the user disk. Like in the previous experiment, the base system has been already uploaded to the checkpoint storage, thus only regular checkpoints must be uploaded. Tests have been executed at *Node A*. As shown

in this figure, the time to make a checkpoint increases with the size of the user disk, because of the time needed to recreate the delta image. Nevertheless, as the user disk has low usage, it can be rapidly merged with the base disk. In addition, it can be highly compressed independently of its size. For this reason, the time required to upload the user disk is just the time to compress it. In the following section, we will discuss the time needed to create a checkpoint depending on the user disk usage, which determines the compressibility of the checkpoint.

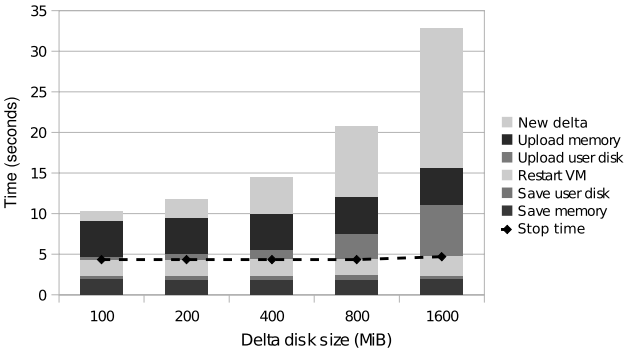


Figure 4. Checkpoint time depending on user disk size

*3) Depending on user disk compressibility:* As commented, we propose compressing the user disk checkpoint in order to save space and reduce the upload time. In order to perform this operation we use "gzip" without any compression, which just removes the consecutive zeros of this image. For example, in an image of 800 MiB which contains 554 MiB of data, making the compression takes around 34 seconds in *Node A* while just copying the whole image is around 11".

Nevertheless, compressing the checkpoint is not always the best solution, as we have discussed in this paper. As the usage of the user disk increases, its compressibility decreases. This makes that from a given user disk usage, it is more efficient just to upload the checkpoint than to compress and upload it. We evaluate if it is worth doing it with Equation 1.

Figure 5 compares the time needed to make a checkpoint of a VM with 128 MiB of memory depending on the usage of a user disk of 800 MiB when using three different techniques: compress always, copy always, and our mixed technique. As shown in the figure, always compressing the user disk is a good approach when the usage of the disk is low, but it becomes more and more expensive as the usage grows. On the other hand, always copying the checkpoint is expensive when the disk usage is low, but it is more efficient with high disk usage. Obviously, the mixed solution combines the benefits of the other two. In particular, when the usage of the user disk is lower than 200 MiB, our system compresses the disk checkpoint, since it estimates that this is faster than working with the original disk. With greater disk usages, our system switches from compressing the checkpoint to just uploading it without any compression. Using this mixed approach allows making the checkpoint of the user disk in the most efficient way independently of its usage.

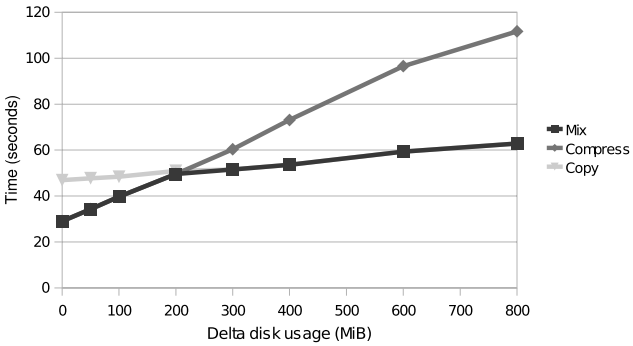


Figure 5. Checkpoint time using different compress/copy techniques

*4) Depending on the number of checkpoints:* Previous results regarding the time needed to make a checkpoint assumed that the system disk had been already uploaded to the checkpoint storage. As commented before, this should be done when the first checkpoint is performed. Figure 6 shows the required time to make a series of checkpoints (one every 100 seconds) of VMs with different memory sizes (in MiB). As shown in the figure, the first checkpoint is more expensive than the rest, as all the disks must be uploaded to the checkpoint storage, while in the rest of checkpoints only the modifications on the user disk and the memory must be uploaded.

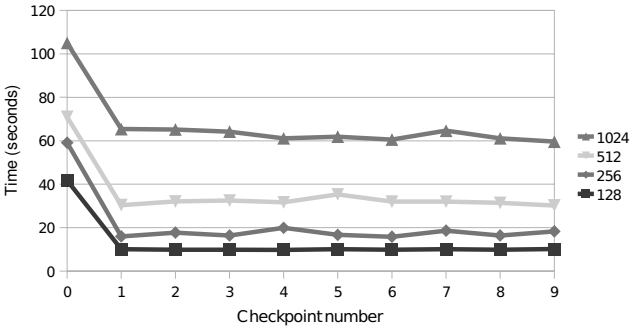


Figure 6. Time to make a series of checkpoints

If the checkpoint infrastructure would not use the *delta disk* approach, it would be required to upload all disks for every checkpoint. Hence, the time needed to perform a checkpoint would always be the time needed for the first checkpoint.

Finally, after testing our checkpoint making process with different parameters, we have seen the memory and user disk usage are key factors that determine the time that the VM is not available. In addition, our delta image technique makes the VM stop time stable if small changes are performed in the file system. This is the most typical situation, where the user just writes some bytes in the disk space between two different checkpoints.

### C. Comparison of Checkpoint Infrastructures

In this section, we compare the performance in a real environment of our checkpoint infrastructure regarding other

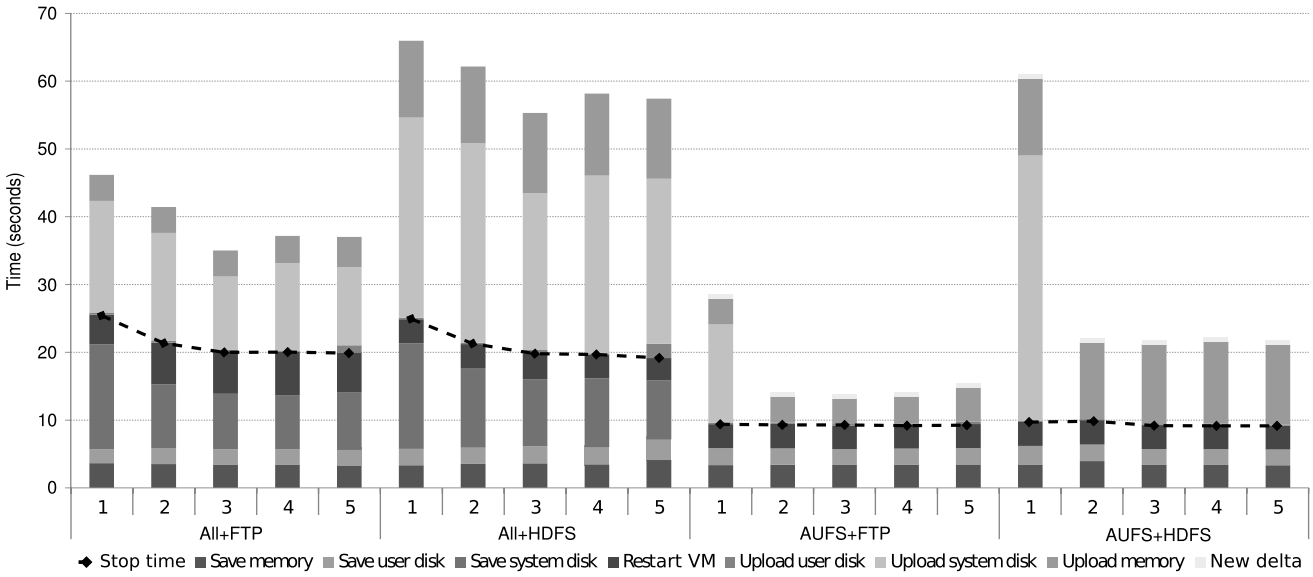


Figure 7. Checkpoint time with different checkpoint infrastructures

possible solutions. We execute the afore-mentioned task within a VM with 256 MiB of memory and 100 MiB of user disk, which is enough to store the required files (around 10 MiB).

*1) Regarding the time needed to make a checkpoint:* First, we evaluate the performance of the different solutions regarding the time needed to make a checkpoint. The experiment consists of submitting the task and, after 300 seconds of execution, making one checkpoint every 600 seconds, until the task finishes. Therefore, five checkpoints will be performed.

Compared alternatives include using AUFS to support checkpoints or storing always all the disks, and storing the checkpoints in a HDFS system with 3 replicas or a centralized storage server accessed using FTP. Figure 7 shows the time needed to make a checkpoint for each one of the five checkpoints when using the different checkpoint infrastructures.

When using *All+FTP*, it takes around 20 seconds in order to save all the disks of the VM. Once the checkpoint is created, it is uploaded to the FTP server used. Notice that there is no appreciable difference among the different checkpoints, since all of them must save and upload all the disks of the VM. This also occurs when using *All+HDFS*. The time to save the disks is basically the same as *All+FTP*, but the upload time has increased. As shown in Section IV-A, this occurs because the checkpoints are being replicated and distributed across HDFS nodes and this requires more time to be accomplished than using FTP. However, this does not affect the time that the VM is stopped, which is the same as *All+FTP*. In addition, the time to recover a VM is reduced when using HDFS, as will be demonstrated in Section IV-C3.

When using *AUFS+FTP* and *AUFS+HDFS*, the system disk is only uploaded with the first checkpoint. This can be clearly appreciated in the figure. Notice that the first checkpoint takes much longer than the rest, as it must upload the system disk. The next checkpoints only need to upload the user disk and the memory. The distinction between using FTP and HDFS described in the previous paragraph also applies in this case.

| | Time | Diff. | Stop Time |
|---|---|---|---|
| No checkpoint | 3201” | 0” | - |
| All + FTP | 3307” | 106” | 107” |
| All + HDFS | 3312” | 110” | 113” |
| AUFS + FTP | 3264” | 43” | 46” |
| AUFS + HDFS | 3265” | 44” | 47” |

Table III
TASK EXECUTION TIME IN SECONDS

This experiment has demonstrated that using the AUFS approach has better performance than making always a checkpoint of all the disks. In addition, FTP gives better performance, but it does not replicate information among different nodes, becoming a single point of failure and a possible bottleneck if different nodes try to store/recover checkpoints concurrently. This will be discussed in the Section IV-C3.

*2) Regarding the time needed to execute the task:* Another interesting result of the previous experiment refers to the time needed to execute the task when using the different checkpoint infrastructures, which is summarized in Table III. Notice that for all the solutions, the task execution time is basically increased with the time that the VM has been stopped making checkpoints. Obviously, approaches with lower checkpoint time (i.e. those using AUFS) benefit from having lower task execution time.

*3) Regarding the time needed to resume task execution:* Finally, we evaluate the performance of the different solutions when a checkpoint wants to be recovered in order to resume the execution of a task in another node. In terms of checkpoint recovering, there is no significant difference between using AUFS or not, as the memory and all the disks must be recovered in any case. Hence, in this section we compare only two alternatives: recovering checkpoints stored in a FTP server, and stored in a HDFS distributed file system.

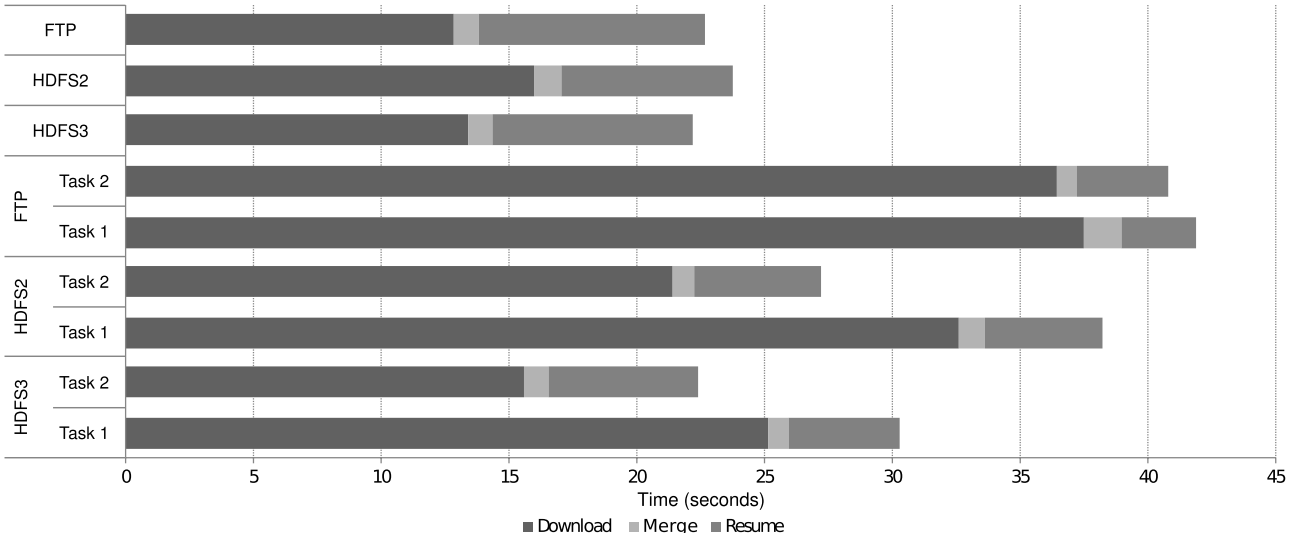


Figure 8. Time to resume tasks with different checkpoint infrastructures

Figure 8 shows the amount of time needed to resume some tasks execution when recovering a checkpoint stored in different storage systems. The infrastructure has to download all the checkpoints, merge all the delta disks and finally, resume its execution. As this application only uses around 10 MiB of data, this process is very fast. The top part of this figure shows the time needed when only one task at a time wants to be restored. In this case, the three compared approaches (FTP, HDFS with 2 replicas, and HDFS with 3 replicas) behave similarly.

Nevertheless, when two tasks have to be recovered concurrently, differences arise. We have evaluated this by running two different tasks in two different VMs in a single node (i.e. *Node A*) that suddenly crashes. At this point, these are resumed in two other nodes: Task 1 will be resumed at *Node B* and Task 2 at *Node C*. As shown in the bottom part of Figure 8, FTP is considerably slower than HDFS when recovering the tasks, since the FTP server becomes a bottleneck for performance when accessed in parallel. Using HDFS allows resuming the tasks faster as their checkpoints can be recovered from different nodes at the same time. In this case, having more replicas of the checkpoint allows even faster recovery. This is denoted when comparing HDFS with 2 and 3 replicas. Apart from the replication ratio, the time needed to recover a task depends also on HDFS internals (e.g. resuming Task 1 takes longer than resuming Task 2 with HDFS 2).

### D. Use Case

Finally, we present a usage example of the whole infrastructure in order to give a proof of concept of the presented checkpoint system. The use case consists of the execution of the afore-mentioned task within a VM with 256 MiB of memory and 100 MiB of user disk space. AUFS and HDFS with 3 replicas are used to manage checkpoints.

Figure 9 shows the CPU usage of the task, which is initially executing at *Node A*. Our system starts making checkpoints after 300 seconds of task arrival, and a checkpoint is performed every 600 seconds. At second 1800, *Node A* crashes and it is decided to resume this task at *Node B*. When the task execution is resumed at *Node B*, this node continues making checkpoints.

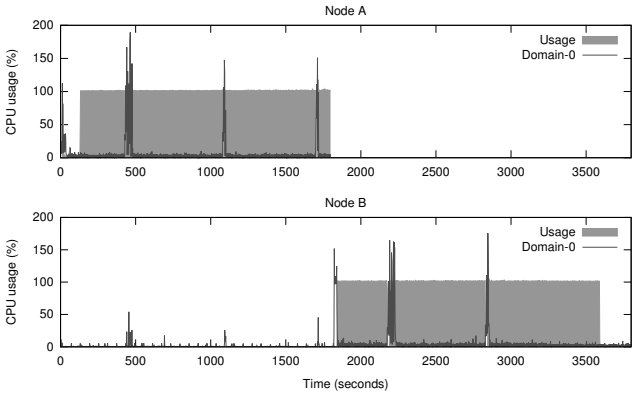


Figure 9. Use case of our checkpoint infrastructure

This figure shows how each checkpoint has a noticeable impact on the load of the VM hypervisor (i.e. Xen Domain-0). It also shows that making a checkpoint in one node also implies some load in the other nodes, as they have to store certain blocks of the checkpoint (because of using HDFS). However, after crashing, *Node A* does not do any CPU consumption at all. As the node has crashed 100 seconds after the last checkpoint, the execution performed in this time has been wasted. Nevertheless, the system can recover the previous 1700 seconds of execution.

The duration of this task if executed at Node A without making checkpoints is around 3220 seconds. However, the task duration is this experiment (which includes checkpoints and recovering the task at *Node B*) has been 3460 seconds. 60 seconds of this time has been used to make the checkpoints.

100 seconds have been spent between the last checkpoint and the crash of *Node A*. 40 seconds are required to realize that *Node A* has crashed and to recover the task at *Node B*. The additional 40 seconds of execution correspond to the lower computing capacity of *Node B* with respect to *Node A*.

## V. Conclusion

In this paper, we have proposed a smart checkpoint infrastructure for virtualized service providers. We have provided a working implementation of this infrastructure that uses Another Union File System (AUFS) to differentiate read-only from read-write parts in the VM image. In this way, read-only parts can be checkpointed only once, while the rest of checkpoints must only save the modifications in read-write parts, thus reducing the time needed to make a checkpoint and, as a consequence, the interference on task execution. The checkpoints are compressed (only if this permits saving time) and stored in a Hadoop Distributed File System. Using this system, the checkpoints are distributed and replicated in all the nodes of the provider.

As demonstrated in the evaluation, the time needed to make a checkpoint using our infrastructure is considerably lower by using AUFS. The checkpoint upload time is higher when using HDFS, but this does not increase the time that the VM is stopped, and it is far compensated when resuming tasks. On the other side, the time needed to resume a task execution is comparable to other approaches when only one task is resumed, and significantly lower when resuming several tasks concurrently. This occurs because the checkpoint can be concurrently recovered from different nodes. Furthermore, this makes our checkpoint mechanism fault-tolerant, as any single point of failure has been eliminated.

Our future work consists of adding CoW mechanisms for memory and disk checkpointing to our proposal. We also plan to incorporate this checkpoint mechanism to our infrastructure for autonomic management of service providers. Our main goal is having service providers able to autonomously react to unexpected situations such as a node failure while fulfilling their agreed contracts. As making checkpoints has some cost in performance, we will work on decision policies to determine when it is worth using this capability and how often it must be used. This includes determining what applications can benefit from having regular checkpoints (we are thinking on stateful medium to long-running tasks) and the best checkpoint frequency to compensate the incurred overhead while maintaining the advantages.

## VI. Acknowledgements

This work is supported by the Ministry of Science under contract AP2008-02641 and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the Generalitat de Catalunya under contract 2009-SGR-980, and the European Commission under FP6 IST contract 034556 (BREIN).

## References

[1] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," in *International Conference on Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, USA, June 25–28 2006, pp. 249–258.

[2] R. Sahoo, M. Squillante, A. Sivasubramaniam, and Y. Zhang, "Failure data analysis of a large-scale heterogeneous server environment," in *International Conference on Dependable Systems and Networks (DSN 2004)*, Florence, Italy, June 28 - July 1 2004, pp. 772–781.

[3] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, and M. Gupta, "Filtering failure logs for a bluegene/l prototype," in *International Conference on Dependable Systems and Networks (DSN 2005)*, Yokohama, Japan, June 28 - July 1 2005, pp. 476–485.

[4] P. Yalagandula, S. Nath, H. Yu, P. Gibbons, and S. Seshan, "Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems," in *USENIX Workshop on Real, Large Distributed Systems (WORLDS'04)*, San Francisco, CA, USA, December 5 2004.

[5] S. Fu and C. Xu, "Exploring event correlation for failure prediction in coalitions of clusters," in *2007 ACM/IEEE conference on Supercomputing*, Reno, NV, USA, November 10–16 2007.

[6] S. Gokhale and K. Trivedi, "Analytical Models for Architecture-Based Software Reliability Prediction: A Unification Framework," *IEEE Transactions on Reliability*, vol. 55, no. 4, pp. 578–590, 2006.

[7] Y. Zhang, M. Squillante, A. Sivasubramaniam, and R. Sahoo, "Performance implications of failures in large-scale cluster scheduling," *Lecture Notes in Computer Science*, vol. 3277, pp. 233–252, 2005.

[8] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[9] A. Nagarajan, F. Mueller, C. Engelmann, and S. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *21st annual international conference on Supercomputing*, Seattle, WA, USA, June 16–20 2007, pp. 23–32.

[10] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 23–31, 1987.

[11] G. Vallee, T. Naughton, H. Ong, and S. Scott, "Checkpoint/Restart of Virtual Machines Based on Xen," in *High Availability and Performace Computing Workshop (HAPCW 2006)*, Santa Fe, NM, USA, October 17 2006.

[12] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor, "Virtual machine time travel using continuous data protection and checkpointing," *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 127–134, 2008.

[13] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, "Parallax: Managing storage for a million machines," in *10th Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, USA, June 12-15 2005, pp. 1–11.

[14] R. Badrinath, R. Krishnakumar, and R. Rajan, "Virtualization aware job schedulers for checkpoint-restart," in *13th International Conference on Parallel and Distributed Systems (ICPADS'07)*, vol. 2, Hsinchu, Taiwan, December 5–7 2007, pp. 1–7.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[16] I. Goiri, F. Julia, and J. Guitart, "Enhanced Data Management for Virtualized Service Providers," in *17th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Weimar, Germany, February 18–20 2009, pp. 409–413.

[17] "AUFS: Another Union File System," http://aufs.sourceforge.net. [Online]. Available: http://aufs.sourceforge.net

[18] "AUFS root file system on USB flash," ubuntu Community Documentation, https://help.ubuntu.com/community/ aufsRootFileSystemOnUsbFlash. [Online]. Available: https://help.ubuntu.com/community/aufsRootFileSystemOnUsbFlash

[19] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," *Document on Hadoop Wiki*, 2008, http://hadoop.apache.org/common/docs/current/hdfs_design.html.