VirtCFT: A Transparent VM-Level Fault-Tolerant System for Virtual Clusters

Minjia Zhang, Hai Jin, Xuanhua Shi, Song Wu
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@hust.edu.cn

Abstract—**A virtual cluster consists of a multitude of virtual machines and software components that are doomed to fail eventually. In many environments, such failures can result in unanticipated, potentially devastating failure behavior and in service unavailability. The ability of failover is essential to the virtual cluster's availability, reliability, and manageability. Most of the existing methods have several common disadvantages: requiring modifications to the target processes or their OSes, which is usually error prone and sometimes impractical; only targeting at taking checkpoints of processes, not whole entire OS images, which limits the areas to be applied. In this paper we present *VirtCFT*, an innovative and practical system of fault tolerance for virtual cluster. *VirtCFT* is a system-level, coordinated distributed checkpointing fault tolerant system. It coordinates the distributed VMs to periodically reach the globally consistent state and take the checkpoint of the whole virtual cluster including states of CPU, memory, disk of each VM as well as the network communications. When faults occur, *VirtCFT* will automatically recover the entire virtual cluster to the correct state within a few seconds and keep it running. Superior to all the existing fault tolerance mechanisms, *VirtCFT* provides a simpler and totally transparent fault tolerant platform that allows existing, unmodified software and operating system (version unawareness) to be protected from the failure of the physical machine on which it runs. We have implemented this system based on the Xen virtualization platform. Our experiments with real-world benchmarks demonstrate the effectiveness and correctness of *VirtCFT*.**

Keywords-Fault Tolerance; Coordinated Checkpointing; Virtual Machine; High Availability

I. INTRODUCTION

In recent years, internet services have been growing in number and functionality, and the increasing demand for computing power and reducing of the running cost have led to the widespread of virtualization technology [1].

In this way, using virtual cluster provided by virtual machine to decrease the number of physical machines leads to better utilization of resources like space, electric power, maintenance, and management [1]. A virtual cluster is consisted of multiple *virtual machines* (VMs) distributed across physical hosts. A virtual cluster can execute distributed application such as client-server systems and transaction processing; or an isolated, private *sandbox-like* environment with little performance reduction; or a flexible computing environment customized by the users. In fact, Amazon's *Elastic Compute Cloud* (EC2) already uses the virtual cluster to provide customers with a completely customized environment on which to execute their computations [21].

However, with the increased use of virtual clusters, fault-tolerance has also become a major issue. As the virtual cluster consists of large counts of virtual machines functioning as compute nodes, faults are becoming common place. Even more unfortunately, the failure of a single virtual machine usually causes a significant crash of fault of the other related part of the virtual cluster to fail. Because the running state is not stored redundantly, loss of any data is catastrophic. Especially, the scientific computing is quite time consuming. It often executes hours even days to get the result, it is unbearable to start it all over again just because fault happens. Thus, the large computing potential of virtual cluster is often impeded by its susceptibility of failures.

In order to bring high availability to virtual cluster, it is highly desirable to provide a mechanism of fault tolerance to protect the entire virtual environment from failure. It can save running states of entire VMs as well as network communication. This mechanism needs to ensure the entire virtual cluster recovery to the correct state when fault is detected.

In this paper, we propose *VirtCFT*, a fault tolerance system for virtual cluster based on the *Virtual Machine Monitor* (VMM) Xen. *VirtCFT* runs beneath target virtual cluster and can provide fault tolerance to arbitrary applications like executing MPI programs. *VirtCFT* initiates coordination of virtual machines, continuously takes and replicates checkpoints of individual VMs to additional backup host for fault resiliency. When a virtual machine crashes for any reason, the latest taken checkpoints will be immediately restored on a backup physical node and the backup VM will replace the crashed primary VM to provide services until the primary VM recovers.

Different from all the existing distributed checkpointing fault tolerant techniques, *VirtCFT* aims to transparently backup the entire OS at the virtual machine level, while all the other checkpoint-recovery methods are only concerned with taking checkpoints of target processes. Besides, *VirtCFT* does not require any modification to applications as well as the guest operating system (GuestOS).

Our system can be extended to platform of cloud computing for providing fault-tolerance. Further, our system can be easily developed to include supporting for job schedulers like PBS, and monitor of virtual cluster like Open-Nebulas. With such functionality, migration and load balance can be used to reduce checkpoint overhead while still providing fault-tolerance.

The rest of the paper is organized as follows. Section 2 discusses the related work of fault tolerance mechanism. Section 3 details the design and implementation of *VirtCFT* on top of Xen. In section 4, we test *VirtCFT* with several popular benchmarks and analyze the experiments. We draw the conclusion of the paper in section 5.

II. Related Works

A. *Fault Tolerance Mechanism*

Over the years, computer scientists and commercial companies have developed numerous practical mechanisms to achieve fault tolerance for cluster. Many of techniques are proposed to deal with distributed applications, yet very few have addressed the need for providing fault tolerance to an entire cluster environment, and even fewer to a virtual cluster with the virtualization technology.

One of the main components of fault tolerance mechanism is checkpoint/rollback. Traditional fault tolerance can be classified by several levels of checkpointing, each of which balances generality and efficiency differently. These techniques can be mainly classified into application-level, OS-level (e.g. [6, 7]), and library-level (e.g. [8–10]) fault tolerance. Although these solutions work fine in their own rights and perform well in specific scenarios, each has its own limitations.

Application-level checkpointing is the most common one. It requires the programmer to manually identify the live data structures, the accurate points in the application where it is possible to take the checkpoint. It is error-prone and needs access to source code of application which is impossible under certain circumstances. OS-level checkpointing often requires modifications to the OS kernel or requires new kernel modules, which may bring more errors and instability. Besides, it will introduce incompatibility to other system-level applications and make them unable to execute. Similarly, only a certain kinds of applications can benefit from linking to a specific checkpointing library. This is due to that the checkpointing library is usually developed as part of the message passing library (such as MPI) which not all applications can use. Moreover, many of these solutions cannot maintain communication of network connected and adjust dependencies of application on local resources such as process identifiers (PIDs), IP addresses, MAC addresses, and file descriptors. The problems of dependencies are usually the main obstacles that prevent a checkpoint from being recoverable on a backup physical host.

In addition to different granularities, fault tolerance can also be categorized as automatic methods (checkpoint-based or log-based) [11, 12, 13] and non-automated approaches [9, 14]. Checkpoint-based methods usually rely on a combination of OS support to checkpoint a process image (e.g., via *Berkeley Labs Checkpoint Restart* (BLCR) Linux module [16]) combined with a coordinated checkpoint protocol handling the complexities arising from each process. Log-based methods generally rely on logging messages and possibly their temporal ordering, then rollback and replay input deterministically when a fault occurs. Yet it is impractical for real-time operation, especially in a multi-processor environment. Non-automatic approaches generally need to manually invocate checkpoint routines.

B. *Fault Tolerance with Virtualization Technology*

Virtualization technology has appeared as a solution to decouple the complex dependency of application execution, checkpointing and restoration from the underlying physical infrastructure. Recently, three solutions have been proposed based on Xen migration. Paper [2] advocates using migration and anticipation as a proactive method to move VM from *unhealthy* nodes to healthy ones in a high performance computing environment. Though this method can be used for predictable failure scenarios, it does not provide protection against unexpected failures nor restore distributed execution states in the event of such failures. Remus [3] is a practical high-availability service that provides a running system to transparently continue execution on an alternate physical host in the face of failure with only seconds of downtime. However, the focus of Remus is individual VMs whereas *VirtCFT* focuses on virtual cluster and need to address problem of coordination. Another solution is Kemari [4], which offers a feasible synchronous approach to fault tolerance based on logging and replay. However, it cannot maintain the network connection after the failover, which is not suitable for virtual cluster performing as science computing like running MPI program.

III. System Design

In order to enhance the utility of virtual clusters, some form of fault tolerance must be presented. Based on Xen, we propose a fault tolerant solution—*VirtCFT*, a virtual machine level, virtualization solution with coordinated distributed checkpointing.

The main challenge to provide such fault tolerance to virtual cluster lies in that in order to recover the virtual cluster to the correct state, the checkpoint of each VM in the cluster need to be coordinated to compose a globally consistent state. Such phase of coordination is essential in that the correctness and consistency of the VM execution and communication states need to be guaranteed when doing recovery in the future. The other challenge is the efficiency; in other words, overhead caused by the fault tolerance mechanism need to be concerned. To address the first challenge, we adopt a modified global coordinated algorithm with synchronous checkpointing to ensure that all VMs are coordinated their checkpoint actions so that global consistent checkpoint is guaranteed. To address the second challenge, we involve an optimized virtualization technique by which checkpoint of individual VM is continuously sent to the backup host so that the redundant data need to save is sharply reduced. As such, we are able to failover the entire virtual cluster with both correctness and efficiency.

After we have analyzed the main difficulties, we then come to the detail design of *VirtCFT*. In order for virtual cluster to do fault tolerance with the property of correctness, we need to ensure the computing execution on virtual cluster come out the correct result after the fault resiliency, which means the captured checkpoints should be globally consistent [17]. If we could simultaneously save checkpoints of all virtual machines and the states of virtual network, we would have a consistent global state. It is notoriously difficult to capture globally synchronized clocks cross physical machines, so taking simultaneous checkpoints is impractical. A possible globally consistent state given by Chandy and Lamport [17] can be regarded as a relaxation of the requirement. Informally, we can describe a globally consistent state composed by the distributed entities (e.g., processes or VMs) that do not contain a message which is recorded as being received by some VMs but not logged as being sent out by any other VM.

We use Figure 1 to show an example and illustrate why globally consistent state composed by virtual machines is vital as well as how the consistent state consisting of four VMs is reached. Messages exchanged between VMs are marked by arrows going from the sender to the receiver. The execution line of the VMs is separated by a cut. A cut is consist of a sequence of events – one cut event at the execution line of each VM that separates each VM's timeline into two parts. The part on the left of the cut corresponds to state before the cut event (past) and the part on the right of the cut corresponds to state after the cut event (future). A cut is globally consistent only if there are no messages passed from the future to the past. As a result, if all VMs come to an agreement on a globally consistent state before a checkpoint is set, we can say that checkpoints taken at this point together make a valid globally consistent state. In *VM2*'s checkpoint taken at time *B*, message *m3* is recorded as being received, but *VM1*'s checkpoint at time *A* has no record that the message *m3* has been sent out. Then the state is inconsistent because *VM2* receives a message which is sent by no one. By avoiding messages like *m3*, which is also called the orphan message [19], we can say that checkpoint *A*, *B*, *C*, *D* consist of a globally consistent state.

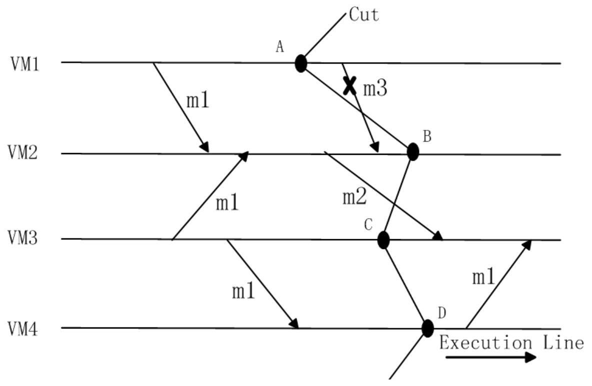

Figure 1. The correctness definition

In our approach, under the assumption of a reliable communication channel, we adopt two-phase commit coordinated-blocking algorithm [18] for FIFO communication channels to reach the globally consistent state by avoiding orphan messages (a post-checkpoint network message). It is mainly used to prevent network message generated by a VM whose checkpoint has been taken from being received by a VM whose checkpoint has not been completed. Therefore, the VMs' checkpoints will form a globally consistent state [7] which is guaranteed to be safely restored to the correct state.

The overview of our coordinated algorithm can be described as below:

Coordinated checkpointing assumes single coordinator, as opposed to multiple coordinators concurrently invoking the algorithm to take checkpoints. The coordinated process occurs as a sequence of phases:

Phase 1: The checkpoint coordinator broadcasts checkpointing request CHKP_REQ to all virtual machines. Then the coordinator waits to collect ACK.

Phase 2: When the VM receives the CHKP_REQ signal, it then checks whether it is ready to take a checkpoint to save the current state of the VM. If the VM is ready, it buffers all the outgoing messages and sends confirmation YES_ACK to the coordinator.

Phase 3: After the coordinator receives YES_ACK from all the VMs, the coordinator then sends valid signals to all the VMs to take tentative checkpoints. Each domain informs coordinator whether it is succeeded in taking a tentative checkpoint. If coordinator can ensure that all the VMs have successfully taken tentative checkpoints, the coordinator decides that all tentative checkpoints should be changed into permanent status and sends VALIDATE signals to all VMs; otherwise the coordinator decides that all the tentative checkpoints should be discarded and sends INVALID messages to all VMs.

Phase 4: When the VM receives the VALIDATE signal, it changes its previous tentative checkpoint into permanent status, sends the CHKP_SUC message to the coordinator and unblocks all the connections.

Phase 5: The coordinated operation is completed when the coordinator receives the CHKP_SUC messages from all other VMs.

The whole progress can be demonstrated as Figure 2. Arrows represent coordinated signals. The coordination process begins when the coordinator broadcasts the CKPT_REQ. At the global state *C1* (*A*, *B*, *C*), the virtual machines receive CKPT_REQ. At this time, each virtual machine tries to do tentative checkpointing to save the local state. At the global state *C2* (*D*, *E*, *F*), the virtual machines change their tentative checkpoints to permanent checkpoints. As a result, all the virtual machines can only send messages again later than *C2*. Therefore, there will be no orphan messages return from the future to the past. *C1* is hereby the globally consistent state.

Our protocol based on this algorithm is different from other existing coordinated protocols in that it is targeting at transparently taking checkpoints of entire operating system, while all the other coordinated protocols are only concerned with taking checkpoints of target processes. This goal requires us to modify and instantiate our protocol underneath the target operating system which leads us to the virtual machine technology.

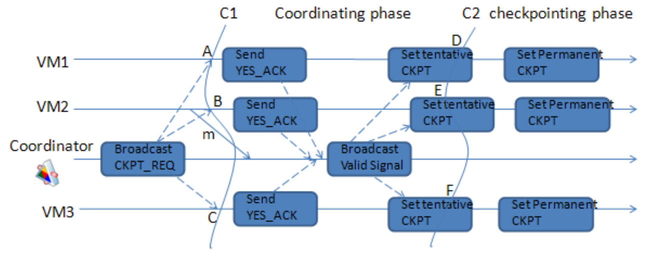

Figure 2. Two-phase coordinated-blocking protocol

Based on all the analysis above, we can give the specific framework of our multiple virtual machine fault tolerant system and show how each component of our system interacts with other part. As shown in Figure 3, within the dashed box, VM checkpoint coordinator, fault sensor, and recovery scheduler are essential and are supposed to be stable. Fault sensor and recovery scheduler are used to handle occasional fault caused by virtual machines. The coordinator where the coordinated protocol is implemented masters the whole condition of virtual cluster and interacts with local communication daemon on each host via the network. The local daemon has its own independent coordination daemon and replication daemon. These daemons can help the distributed host to control all the virtual machines running above it to coordinate with other virtual machines across hosts as well as creating replications of each virtual machine.

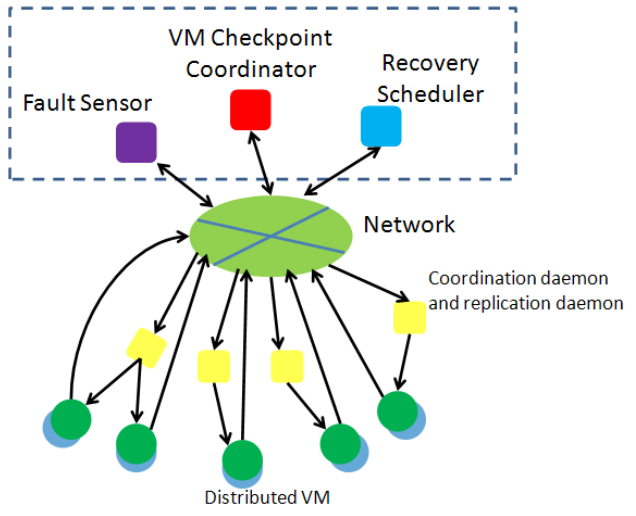
VM Checkpoint Coordinator
Fault Sensor
Recovery Scheduler
Network
Coordination daemon and replication daemon
Distributed VM

Figure 3. Framework of *VirtCFT*

IV. SYSTEM IMPLEMENTATION

We have described the basic idea of *VirtCFT* and the mechanism of our fault tolerance for virtual cluster. Now we present the detail implementation of each component. These components, including VM-level coordinator, replication daemon with incremental checkpointing, and recovery scheduler, are all implemented based on Xen virtualization platform.

A. Overview

VirtCFT is designed and implemented as a virtual machine level, coordinated fault tolerant system. Different from all the previous related systems which are concerned with taking checkpoints of processes, our system aims at restoring the entire virtual cluster to the previous correct state set by continuously taking incremental checkpoints of entire OS images of virtual cluster with virtualization technology.

The system architecture of VirtCFT can be demonstrated as Figure 4. It is mainly composed of following parts: VM-level coordinator which consists of coordinated module and network control module used for coordinating virtual machines to reach the globally consistent state. Replication daemon is responsible for backup redundant data to the additional peer host with incremental checkpoint which can tremendously reduce the overhead of checkpointing. Fault sensor and recovery scheduler restore the virtual cluster from failure to correct state when a fault occur. The fault sensor continuously probes the state of the virtual cluster until a fault being detected, and then the recovery scheduler will roll back the related virtual machines to reach the former correct consistent state established by coordinator and allow the entire system to keep running from that point. The more detail design can be described as following.

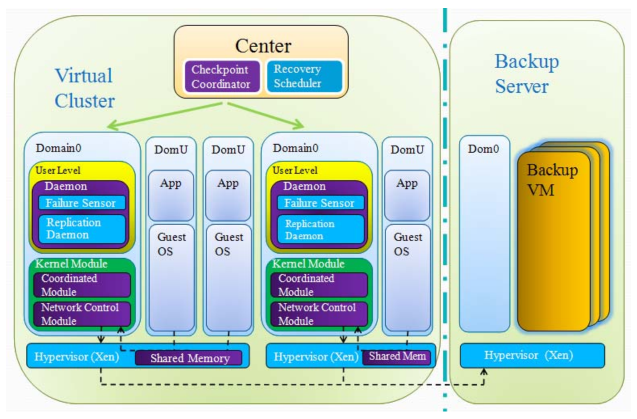

Figure 4. System architecture

B. VM-level Coordinator

Considering that our coordinated checkpointing protocol is required to be totally transparent to the process and the OS running in guest domains, we have to implement our protocol beneath the guest OS. In Xen, domain 0 is the special control domain of the Xen system and more importantly, all the network messages heading to the guest domains will go through domain 0 first, all of which makes the domain 0 a perfect choice to implement our coordinated checkpointing protocol.

For the virtual cluster, there are two types of communication which can be referred to as inter-host and intra-host communications. For the VMs in the same physical host, they communicate through a Linux ethernet bridge as it is widely implemented in a typical Xen setup, so the coordination process can be accomplished simultaneously. For communication across hosts, we add a module between VM and Linux ethernet bridge which can be used to carry our coordinated signals to address process of coordination. The coordination protocol is implemented as the description in the above section of the system design.

When the coordinator spots the global consistent state, we need to block and later unblock all the outgoing network messages of a guest domain. According to our algorithm, when a virtual machine receives a message from the coordinator commanding to set the tentative checkpoint, all the outgoing messages of the virtual machine must be blocked before we take a checkpoint of the virtual machine, and after finishing the checkpointing, we need to unblock these blocked connections. We implement these blocking and unblocking functionalities by adding a new kernel module in the traffic control [20] module of the domain 0 kernel, shown in Figure 5. This module waits the coordinating signals to determine whether it is time to block the network messages. If so, the module blocks all the outgoing messages of this virtual machine until the checkpointing process has finished and receives an unblock command. The advantage of this method is that all the network messages communicating with VM need to go through domain 0 first, thus block the outgoing messages in domain 0 will simultaneously block all the other guest operating systems above the virtual machine monitor. In addition, a buffering scheme also needs to be added so that it can preserve the messages dropped during the process of blocking.

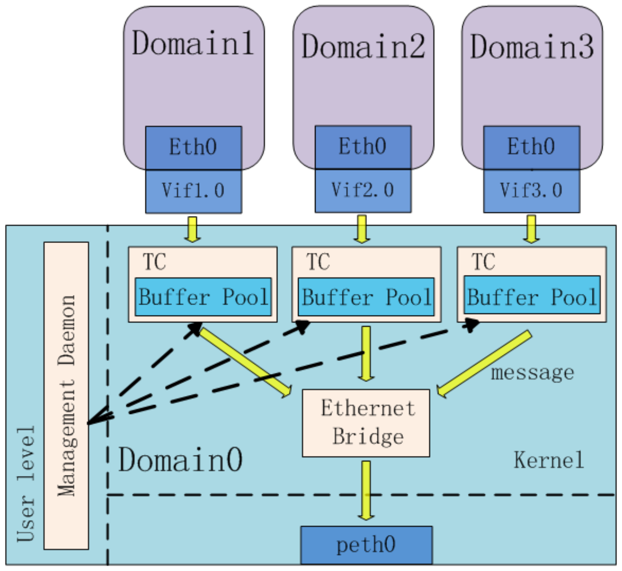

Figure 5. Traffic control in domain0 with bridge mechanism and TC

C. Replication Daemon with Incremental Checkpointing

To provide an effective fault tolerant system, we need a mechanism that enables a VM to run on additional physical host with minimum possible overhead. More importantly, the network connections should not be disconnected while failover is in progress. The states need to be preserved include information of VCPU, memory, disk, I/O device, and network communicate.

Xen provides a capability of *live migration*, which enables the guest VM to be transferred from one physical host to another [5]. During migration, it will preserve the state of all the processes on the guest, which effectively allows the VM to keep executing without interruption.

This technique is extended to transparently (to the GuestOS) mark all VM memory pages as read only. The Xen hypervisor is then able to capture all writes that a VM makes to memory and update a bitmap of pages of *Shadow Page Table* that have been dirtied since the previous epoch. In each epoch, the Xen hypervisor atomically reads and resets this bitmap, and the iterative process of sending dirty pages goes until progress can no longer be made. Finally, the live migration process suspends execution of the VM and enters a final phase of *stop-and-copy*, where any remaining pages including VCPU states are transmitted and execution resumes on the destination host.

To fulfill our need, we modify Xen as repeatedly executing the final step of live migration: each round, the guest is paused while the replication daemon interacts with the source host in obtaining the dirtied memory pages and VCPU state and putting them into a buffer. The guest then resumes execution on the current host, rather than restoring on the destination host. Another primary change required to be added for support is that the VM needs to keep running after it has been suspended. Previously, Xen will terminate the VM after saving and sending the states out. To implement such migration, we create a replication daemon of checkpointing to obtain only newly-dirty memory each round of VM. Instead, the original VM will not be destroyed after its state has been copied. The VM will keep schedulable even after being suspended, that means, the VM will resume continuing the computation.

In addition to the states saved above, disk state is also needed to backup. Disk checkpointing is yet not implemented in Xen, however, we can also incrementally save the VM's file system by using the LVM snapshot capability. The LVM snapshot records changes made to a logical volume after the snapshot has been made. In *VirtCFT*, the LVM snapshots are taken during the stop-and-copy phase when a VM is suspended. It can be processed and submitted asynchronously to the backup host after the VM resumes executing.

D. Recovery Scheduler

In this paper, we make the assumption that failures follow the fail-stop model. That is, one or more virtual machines crash or stop sending or receiving network messages. Because the main feature of our system is the state capture and replication of the whole system, error produced by software will also be saved which is the consequence of providing transparently system-level fault tolerance.

In order to trigger the recovery phase, we currently use a failure sensor set in user space to watch possible failures. It periodically sends messages to inquiry other virtual machines whether they are in normal condition. A timeout of the virtual machine responding to inquiry requests will result in the failure sensor assuming that that virtual machine has crashed and give a failure report to the failure sensor.

When faults occur, the fault sensor detects the failed VM and triggers the recovery scheduler to send messages to all the other domains to inform this failure and initiate recovery process. It is needed to roll back the related virtual machines to the latest globally consistent state and drop all the uncommitted states. After the recovery, in order to ensure the newly established system state not to be interrupted by communication events, which will result in message losing and inconsistencies in the global state, it still need to report to the recovery scheduler in hypervisor and wait the recovery scheduler responding the request. After that the entire virtual cluster can continue to execute correctly again.

We implement the recovery scheduler based on Xen's xm_restore which is the destination side of live migration process. It will introduce the advantages of virtualization technology that the backup virtual machine is able to maintain its network address unchanged. With this feature, it is greatly benefit the recovery process. For example, a running MPI computation needs not to update their address caches or any process IDs. Since the entire VM is checkpointed, the MPI job sees the environment as it is prior to checkpointing.

V. PERFORMANCE EVALUATION

In this section, we first measure the overhead brought by our system, and then give a thorough test of our design and implementation by using NPB MPI benchmarks.

A. *Experimental Setup*

We build our experimental environment on a pair of two-socket servers (server1 and server2) connected by a one gigabit Ethernet network, each sockets have 4 Intel Xeon 1.6GHz CPUs. Both servers have 4GB DDR RAM and 150GB hard disk. We use Linux 2.6.18 with Xen 3.4.0 installed as the operation system. Identical images of VM exist on both the primary and backup host and the path is assumed to be the same on the backup as it is on the primary. In all cases the VM is configured to have a single CPU with 512MB of RAM, and installed a Red Hat Enterprise Linux 5.3 as guest OS. To execute evaluation, we create our test environment as virtual Linux clusters of 4 VMs. All the virtual machines and physical machines are connected with each other based on the bridging mechanism provided by Xen. The experiment environment is shown in Figure 6.

B. *Performance Overhead*

The purpose of the evaluation for the performance of *VirtCFT* is to compute the overhead of *VirtCFT* introduced into the virtual cluster as well as the overhead of coordination phase. In order to ensure the accuracy of our evaluation, we reboot all virtual machines ready for test so that virtual machines can run with low load. In order to demonstrate the effectiveness of checkpointing the file system, no extraordinary measures are taken to reduce the GuestOS image size. We use the most recent version of the Xen-3.4-testing version for all tests.

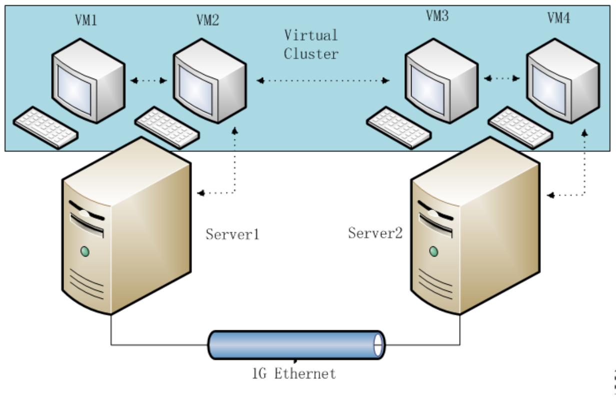

Figure 6. The experiment enviroment

NAS Parallel Benchmarks (NPB) [15] developed at the *NASA Advanced Supercomputing* (NAS) contains a combination of computational kernels. For our analysis, we choose to use EP and IS. EP is a compute-bound MPI benchmark with a few network communications. IS is an IO-bound MPI benchmark with large amounts of network transaction.

We first measure these benchmarks' runtime when they are executed on 4 virtual machines (the configuration of these physical machines is described above) without any fault tolerant functionality. Then we initiate the *VirtCFT* without coordinating process and record their runtime. By comparing these two groups of runtime, we find out that the ability of providing fault tolerance to each VM without coordinated checkpointing itself will incur approximately 20%~40% performance penalty than that generated in 4 virtual machines environment without fault tolerance. This is a necessary consequence to provide both transparency and generality by saving redundant states of the entire VM.

After this, we start our *VirtCFT* with coordinated checkpiointing and evaluate the runtime with different coordinating intervals, including 400ms, 200ms, and 100ms. Shown in Figure 7, we choose EP and Class A and record its runtime under different situations. We first initiate 4 processes; each process is pinned to one of the 4 VMs and has full access to a single processor. To test the performance of *VirtCFT* accurately, we then repeat the EP benchmarks 10 times. We find out when the coordinating interval is 100ms, the runtime is relatively high to fault tolerance of the virtual cluster without coordinated checkpointing, but when we increase the coordinating interval, the runtime overhead caused by our system begins to drop. The reason is that when we increase the interval, the whole virtual cluster will do less coordinated checkpointing, which means the whole system will suspend less time during a period of time and can spend more time on computing. When we update the interval to 400 seconds, the runtime increase is already less than 30 percent comparing with the *FT enabled without coordination* situation. We are sure if we continue increasing the coordinating interval, the runtime overhead caused by *VirtCFT* will keep dropping.

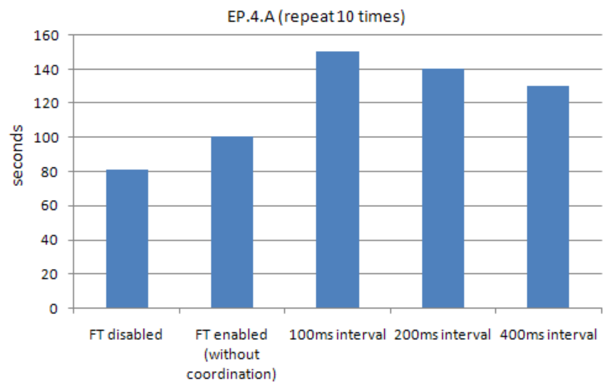

Figure 7. EP.4.A runtime

After evaluating the performance of EP.A.4, we then choose Class B (the Class B problems are roughly four times larger than the Class A problems) to see how *VirtCFT* will perform if we large the problem size. As shown in Figure 8, the results show a proportionately overhead comparing with EP.4.A, the overhead of runtime does not go up sharply as the problem size rises. Besides, the runtime overhead still gradually decreases as we increase the coordinating interval. Thus we can say that *VirtCFT* is suitable for larger task or job that need longer time to compute without extra overhead of runtime.

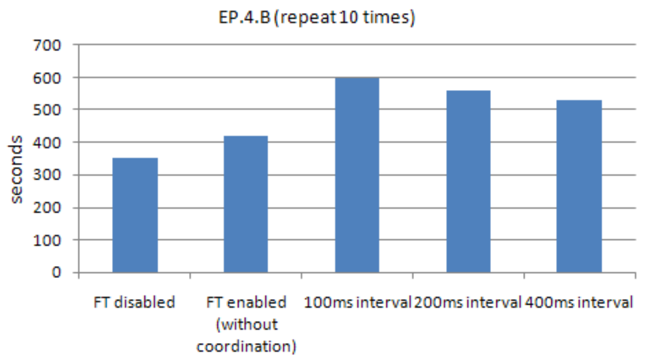

Figure 8. EP.4.B runtime

The runtime overhead of our system is reasonable on dealing with compute-bound benchmark since this is an inevitable consequence to provide fault tolerance to the whole virtual cluster by saving the entire running state of each VM. We still need to test some extreme cases with I/O intensive applications. We then choose IS, a NPB benchmark without including computation of floating point, but mainly with significant network exchanging communications.

As shown in Figure 9, the runtime dramatically increases if we set the coordinating interval to 400ms, but when we decrease the interval, the runtime overhead caused by *VirtCFT* begins to reduce. The sharply rising overhead in this case is largely due to output-commit delay on the network interface and as we decrease the interval, the impact caused by network delay can be partly mitigated. Another reason is that workloads produced by benchmark are considerably more intensive than that expected in a typical virtual cluster. In fact, even though *VirtCFT* is sensitive to network latency, there are still several potential ways of optimization to reduce the network delay.

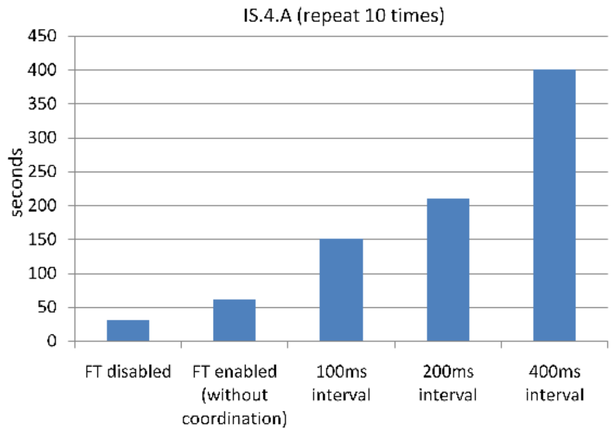

Figure 9. IS.4.A runtime

In Figure 10, we show the time needed to restore a computation for the EP and IS benchmark with different coordinating intervals. In the case of recovering, the dominating factors include the time of activation of backup VMs to their checkpointed states, the coordinating intervals, and the type of application running on the virtual cluster. With the increasing of the interval of coordinated checkpointing, the downtime to resume the virtual cluster slightly grows. This can be explained by that the interval is longer, so does the time to detect the fault is longer.

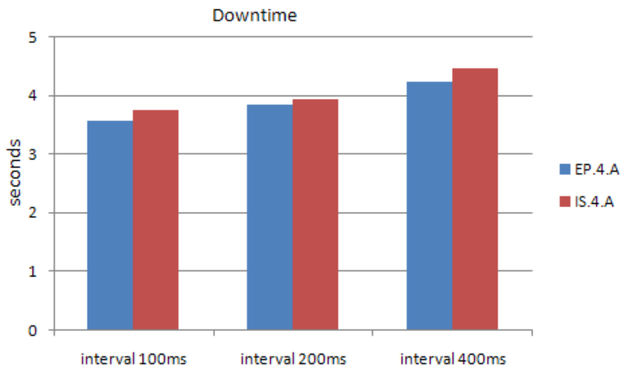

Figure 10. Time to restart from a failure

To verify the correctness of *VirtCFT*, we run the NAS MPI benchmark programs in our experimental environment. By comparing the outputs of the following: (1) an uninterrupted MPI benchmark execution generated in the ordinary environments, (2) the same benchmark execution under the protection of *VirtCFT* and (3) the same benchmark execution recovered by *VirtCFT* after randomly injecting network failures or power off, we confirm that all executions generate the same results.

VI. Conclusion

In this paper, we present the design and implementation of a transparently virtual machine level Fault-Tolerant system: *VirtCFT*. Comparing to all the existing fault tolerant systems, *VirtCFT* is different in that it is aiming at recovering the entire virtual cluster to the previous correct state when fault occurs by transparently taking incremental checkpoints of virtual machine images coordinately. To make *VirtCFT* totally transparent to the target virtual machines, we choose to implement our mechanism in the control domain of the Xen virtualization platform. We modify the Xen source code to implement the *live checkpoint* so that the primary host can continually transfer its updated information to the backup host without distinction. We implemented a coordinated checkpoint protocol, including adding control info, and modules severing as blocking and unblocking outgoing messages. Besides, we also implement a set of user level daemons which are the management unit in our implementation. We apply several popular benchmarks to verify the correctness of *VirtCFT* and evaluate its overhead.

ACKNOWLEDGMENT

This work is supported by National 973 Basic Research Program of China under grant No.2007CB310900, NSFC under grant No.61073024 and 60973037, NCET under grant 07-0334, and Wuhan Chenguang Program under grant No.201050231075.

REFERENCES

[1] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends", *IEEE Computer Magazine*, May 2005.

[2] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive Fault Tolerance for HPC with Xen Virtualization", *Proc. ACM International Conference on Supercomputing*, 2007.

[3] B. Cully, G. Lefebvre, D. Meyer, M. Freeley, N. Hutchinson, and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication", *Proc. USENIX NSDI*, 2008.

[4] Y. Tamura, K. Sato, S.Kihara, and S. Moriai, "Kemari: virtual machine synchronization for fault tolerance", *Proc. USENIX'08* Poster Session, San Jose, CA, USA, 2008.

[5] I. Philp, "Software failures and the road to a petaflop machine", *Proc. the 1st Workshop on High Performance Computing Reliability Issues*, 2005.

[6] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments", *Proc. USENIX OSDI*, 2002.

[7] H. Zhong and J. Nieh, "Linux Checkpoint/Restart As a Kernel Module", *Technical Report CUCS-014-01*, Department of Computer Science, Columbia University, 2001.

[8] J. Sankaran, J. M. Squyres, B. Barret, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing", *Proceedings of the LACSI Symposium*, 2003.

[9] G. E. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World", *Proc. the 7th European PVM/MPI User's GroupMeeting*, LNCS, Vol.1908, 2000.

[10] Y. Chen, J. S. Plank, and K. Li, "CLIP: A Checkpointing Tool for Message-Passing Parallel Programs", *Proc. IEEE Supercomputing*, 1997.

[11] G. Stellner, "CoCheck: checkpointing and process migration for MPI", *Proc. of IPPS'96*, 1996.

[12] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing", *Proc. LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[13] G. Bosilca, A. Boutellier, and F. Cappello, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes", *Proc. Supercomputing*, Nov. 2002.

[14] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski, "Architecture of LA-MPI, a network-fault-tolerant MPI", *Proc. International Parallel and Distributed Processing Symposium*, 2004.

[15] NPB, http://www.nas.nasa.gov/Resources/Software/npb.html.

[16] J. Duell, "The design and implementation of berkeley lab's linux checkpoint/restart", *Technical Report*, Lawrence Berkeley National Laboratory, 2000.

[17] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems", *ACM Transactions on Computing Systems*,Vol.3, No.1, pp.63-75, February 1985.

[18] B. S. Boutros and B. C. Desai, "A two-phase commit protocol and its performance", *Proc. the 7th International Workshop on Database and Expert Systems Applications*, 1996.

[19] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguezb, and F. Cappello, "Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI", *Proc. the 2006 ACM/IEEE conference on Supercomputing*, Nov, 2006.

[20] W. Almesberger, "Linux network traffic control implementation overview", *Proc. of 5th Annual Linux Expo*, 1999, Raleigh, NC, pp.153-164.

[21] A. Weiss, "Computing in the clouds", ***netWorke***, pp.16-25, November 2007.