

Enhancing Reliability for Virtual Machines via Continual Migration

Wenchao Cui, Dianfu Ma, Tianyu Wo, Qin Li

School of Computer Science and Engineering

Beihang University

Beijing 100191, China

cuiwc@act.buaa.edu.cn, dfma@buaa.edu.cn, {woty,liqin}@act.buaa.edu.cn

Abstract—Our approach is to design and implement a continual migration strategy for virtual machines to achieve automatic failure recovery. By continually and transparently propagating virtual machine’s state to a backup host via live migration techniques, trivial applications encapsulated in the virtual machine can be recovered from hardware failures with minimal downtime while no modifications are required. Deployment agility is considered so that initiating continual migration can be done with no time consuming preparations. Experimental results show that virtual machine in a continual migration system can be recovered in less than one second after a failure is detected, while performance impact to the protected virtual machine can be reduced to 30%.

Keywords-virtualization; availability; fault-tolerance

I. INTRODUCTION

Virtualization has been widely adopted by data centers for transparent load balancing, application mobility, server consolidation and secures computing [1]. With one physical machine hosting many virtual machines, a single node failure may result in more severe disruption to hosted services, which brings great challenge for automatic failure recovery in virtualized computing environments.

One of the most general solutions for failure recovery is to replicate the states of the protected virtual machine to a backup host, with which the virtual machine may be recovered from host failures. Although virtual machine replication can be done in various ways, its consistency and efficiency are not guaranteed. Most virtual machines maintain memory and external storage states in separate ways, which may result in an inconsistency when replicated to the backup host. In the mean time, fast and transparent failure recovery requires the backup to be synchronized with the primary virtual machine; however, synchronizing on every change brings too much performance tradeoffs [2], [3].

In this paper, we present a continual migration mechanism for virtual machine based fault tolerant systems by continually and transparently propagating virtual machine’s states to a backup host via live migration [4] techniques. Unlike execution replay based solutions [2], [5], [6], [7], continual migration do not rely on instruction and non-deterministic event sequence logged by hypervisor, which enables continual migration to run on heterogeneous platforms. Compared

with reboot based high available systems [8], continual migration reduces machine downtime by eliminating boot time of virtual machines, and prevents state loss by preserving application state via migration protocols. While other migration based high available systems [9] maintains a secondary disk image file on the backup host, continual migration propagates disk stats via network attached storage, which is more common in a data center configuration and provides better agility in both configuration and deployment. Some research work achieves failure recovery by taking and restart from checkpoints [10], [11], [12]. However, current checkpoint techniques take up considerable time and space, which is not quite suitable when checkpointing in a high frequency like continual migration.

Continual migration aims to provide transparent fault tolerance for commodity applications in data centers or virtual computing environments [13], [14] without specialized hardware or operating systems. Experimental results on our prototyped continual migration system based on the KVM [15] hypervisor demonstrate that virtual machines protected by continual migration mechanism can recover from hardware failures in less than one second, while the performance impact can be reduced to 30%.

The following section introduces related work, while Section III describes the concept and major components of continual migration. Section IV provides the architecture of our prototyped continual migration system, Section V gives the experimental results and finally, Section VI summarizes the paper.

II. RELATED WORK

Cully’s Remus [9] is the first migration based high available system. Based on the Xen hypervisor, Remus can asynchronously replicate memory and disk state changes to a backup system in a very high frequency, thus creating a mirror of the current virtual machine to recover from. Remus maintains a pre-copied disk image file on the backup host to receive disk updates, while write operations to disk device are duplicated and flushed to both files on the primary and backup hosts. The major contribution of Remus is to provide a migration based high available system architecture and prototype, however, pre-copying disk image is a time and space consuming operation, which prevents Remus from

rapidly redeploying after backup virtual machine is brought up. Besides, Remus doesn't support hardware virtualizations.

Kemari [3] achieves virtual machine replication by event driven synchronizations. Unlike Remus, Kemari migrates disk images via network attached storage, which may result in an inconsistency when failure happens. Kemari solves this problem by synchronizing virtual machine's memory state every time before disk operations are issued. However, when running I/O intensive workloads, Kemari suffers a severe performance penalty due to high frequency synchronizations.

Ta-Shma [12] uses a continuous data protection (CDP) enabled file system to preserve virtual machine disk image files. In a CDP enabled file system, every write operation is logged and revertible, which enables the virtual machine to move to any historical states for debugging, intrusion detection or fault tolerance. Though historical data is valuable for system analysis, it is not quite suitable for fault tolerant purpose due to tremendous storage overheads.

III. CONTINUAL MIGRATION

The basic concept of continual migration is to continually and transparently migrate virtual machine's states to the backup host until failure is detected. For most virtual machines, two parts of states are required to migrate: the internal state such as memory, registers and/or internal device buffers, and external states such as attached hard disk images. In a live migration, internal states are propagated via migration data stream transferred through network, while external states are shared in network attached storage. With such configuration, virtual machines are able to migrate between hosts within local network rapidly and agilely, without needing to pre-copy large disk image files. Continual migration adopts this configuration but extends the migration protocol so that states can be transferred continually to backup hosts as shown in Figure 1.

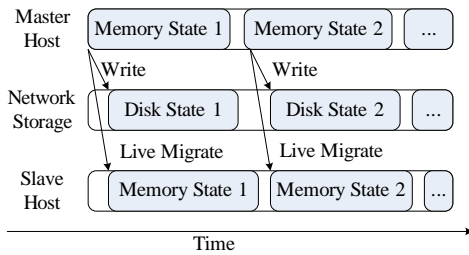


Figure 1. Continual migration mechanism

A. Migration issues

The major difference between live and continual migration is that the virtual machine on the target host is not guaranteed to be consistent during a live migration, while continual migration must ensure that even an ongoing migration is interrupted by a failure on the source, there must be a valid

and consistent state on the backup system to recover from. This brings three major challenges to our work:

Migration continuity. Although live migration provides minor machine downtime, its overall migration time is significant. To achieve fast and frequent checkpoints, continual migration must reduce overall migration time to a matter of tens of milliseconds. Besides, migration procedures should be changed so that ongoing migrations will not corrupt previous migrated consistent internal states on the backup host.

External state consistency. Continual migration participants share disk image via network attached storage, thus the consistency of the image file must be guaranteed even if cached disk data fails to flush to disk as failure happens.

Coherent consistency. When applications are rolled back to the last migrated state, the corresponding disk image may have been changed, which will result an incompatibility between memory and disk states on the backup host. Continual migration must ensure that whenever a failure happens, the backup host can load a consistent virtual machine with compatible internal and external states.

B. Internal state migration

In continual migration, internal states are replicated via live migration data stream. Continual migration extends live migration with following modifications:

Scheduled migration. The major change in continual migration is to periodically issue outgoing migrations. This is done by scheduling a migration immediately after previous migration is completed. One of the most simple and straightforward strategies is to issue migrations with static time intervals.

Lightweight migration. Compared to live migration protocols, continual migration introduces a lightweight migration strategy for reducing migration iteration time. For the first live migration iteration, whole memory blocks are transferred the same way as traditional migration protocols, while after the migration is completed, continual migration continues to track dirtied pages, as following live migration iterations will only transfer dirtied pages.

Buffered migration. Continual migration buffers migration data to preserve consistent internal states. On the source host, migration data is buffered until the current migration iteration is completed, and then this piece of migration stream is transferred to the backup host together with the length and checksum. Incoming migration streams on the backup host are buffered and verified before merging into migrated states, so that incomplete migration data will not affect the current consistent states.

C. Buffered block device

For most virtual machines, the disk image file is the only external storage that preserves the states of the virtual machine. To ensure the consistency of the image file stored

in shared network storage after failure is occurred, continual migration introduces a buffered block device as shown in Figure 2.

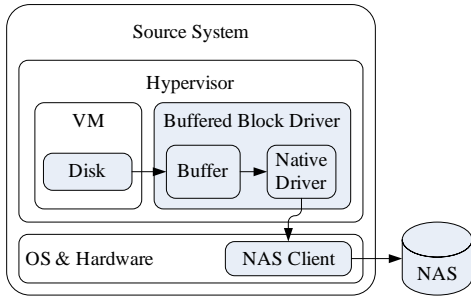


Figure 2. Buffered block device

A buffered block device consists of a memory buffer for queuing write requests issued to the disk image and an attached native driver for underlying image files like vmdk [16] or qcow2 [17]. Buffered block device works in two modes: sync and async. Under sync mode, any read or write operation is passed directly to the attached native driver without translating or buffering for higher performance. When switched to async mode, writing to a disk sector is buffered in the memory, while reading operations will firstly try to find and return specific sector in the memory buffer. A switch from async to sync mode will flush buffered sectors to the image file in a cache write through way.

Buffered block device aims to protect disk image file from being corrupted by failing write operations. By default, hypervisor uses a cache write back strategy for better I/O performance. In this manner, write operations will not immediately affect disk image files but rather change the states of the cache managed by hypervisor or hosting machines. Thus there is a chance that when the memory states of the virtual machine has migrated to the backup host, issued disk operations are still in the cache. When failure occurs at this specific time, virtual machine on the backup host will find that issued disk operations seem to be 'forgotten', which will lead to unexpected behaviors. Disabling cache could fix the problem; however, the performance tradeoff is unacceptable. With buffered block device, writing to memory won't impact the I/O efficiency, while flushing to disk with cache disabled can be done asynchronously, which eliminates this problem.

D. Three-phase commits

When failure happens, the internal states are rolled back to last migrated states; however, the external states in the network attached storages needs more discussing.

By buffering write operations in memory, buffered block device ensures the consistency inside the disk image itself; however, the time to flush memory buffer to external image files is crucial. When failure happens, internal states are rolled back to the last migration state, while external states

in the network attached storage won't, which may result in an incompatibility between memory and disk states on the recovered virtual machine.

Continual migration introduces a Three-phase commits mechanism for this kind of coherent consistency problem, as shown in Figure 3.

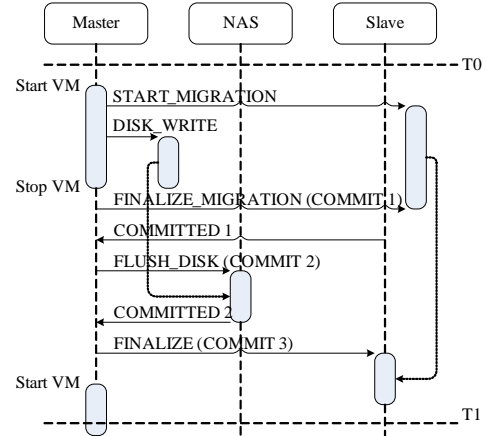


Figure 3. Three-phase commits

Beside the migration channel, an event channel was established between the source and the destination hosts. Commit requests and responses are exchanges in the event channel, so that both the source and the destination will get a clear view of the migration status, depend on which the destination will decide how to get a consistent virtual machine up and running.

Consider a time designated as T_0 , when both the source and the destination holds a consistent virtual machine states. In the first phase, live migration is issued to the backup host, when memory and processor updates are copied to the backup host and buffered for verification. The attached buffered block device is switched to async mode, in which the write operations are buffered without affecting the states in the network attached storage. When this round of live migration comes to an end, a COMMIT 1 message is posted through the event channel, to which the backup host will response to complete the first phase.

After receiving the first COMMIT response, the source switches the disk to sync mode to flush buffer back to disks. This is considered as the second phase.

In the last phase, a COMMIT 3 message is posted to the backup host, upon which the backup host will merge buffered internal states to the backup virtual machine. Both the source and the destination now enter a new consistent state designated as T_1 .

Three-phase commits ensures that at any time, the backup host can find a consistent virtual machine state to recover from failure. For failures between T_0 and commit 2, both the internal states and external states are untouched, thus backup virtual machine can be easily recovered to T_0 state.

For failures between commit 2 and commit 3, external states are updated to newer version, while complete internal state updates have already been transferred to the backup host, with which the internal states can be updated to match the external states. Failures after commit 3 won't affect backup consistency since both internal and external states are updated to T1.

IV. SYSTEM ARCHITECTURE AND IMPLEMENTATION

We implement our prototyped continual migration system on top of the KVM hypervisor. KVM is part of the Linux kernel which provides full and hardware virtualization support. The whole system architecture is demonstrated in Figure 4.

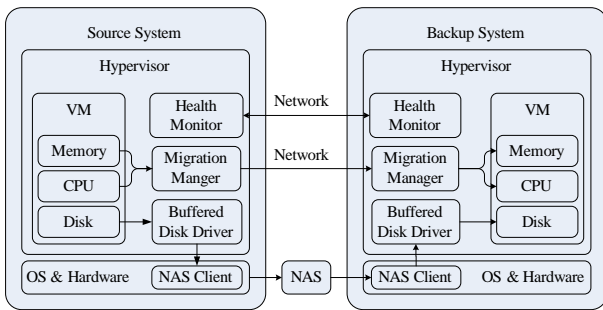


Figure 4. System architecture

Migration manager is the key component for scheduling continual migrations. Migration manager issues live migrations by creating and updating a QEMU timer, which is bound to the migration request function. Every time this migration timer expires, the bounded function issues a new migration; then the timer is reset with the checkpoint interval parameter, to schedule the next migration iteration.

Buffered block driver aligns disk data to a sector of 512 bytes. A disk write operation is indexed by the offset of the first modified sector and the count of modified sectors. Operation indexes are maintained by link lists, while buffered data are allocated from a memory pool for better performance.

We adopt a simple strategy that only monitors the network link between the source and backup system for failure detection. When a connection loss is detected, the backup virtual machine is brought up. In a production environment, the health monitor could be modified to provide more accuracy.

V. EVALUATION

We deploy our system on two desktops connected with gigabyte networks, both equipped with dual core Intel E8400 CPU and 4 GB memory. An extra box with NFS server is attached to the same network to provide shared storage. The virtual machine being tested is configured to have single processor, 512 MB memory and 10 GB storage. The virtual machine is bridged to the physical network so that external

clients can connect to applications inside virtual machine directly.

The experiments are categorized in two types: the functional and performance tests. Functional tests concentrate on the transparent recovery capabilities of our system, while performance tests measure the performance tradeoffs of our implementation.

A. Functional tests

First of all, we install the Gnome desktop and VNC Server on the tested virtual machine, and issue a connection to the VNC Server on an external machine. The x-clock and glxgears are started to frequently refresh the desktop. After everything is ready, we mandatorily kill the process of the source virtual machine to simulate a system failure. On the other host, the backup virtual machine is brought up almost instantly, while the VNC connection keeps connected without interruption, with only a slight pause around one second.

Later, we use ping to measure the machine down time at different migration intervals. Ping is configured to send packages every 50 milliseconds, and we use the time interval between the last ping transaction before failure and the first ping transaction after failure to represent the machine downtime. This experiment is conducted five times at each migration interval, and the average ping interval is used to represent the machine downtime at each migration interval. Figure 5 demonstrates the results.

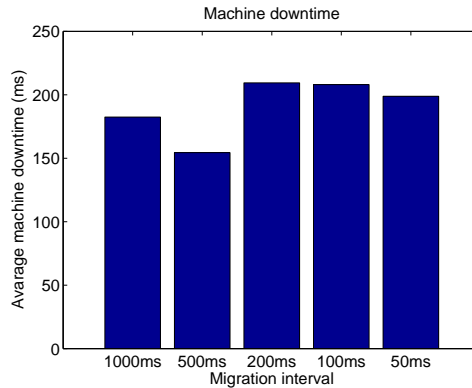


Figure 5. Machine downtime by ping

As shown in Figure 5, the average machine downtime is between 150ms and 209ms, while the longest machine downtime measured in all tests is 396ms, which means that continual migration can recover failing virtual machine in less than one second. We believe that for most virtualized systems this level of MTTR [18] is tolerable.

B. Performance tests

In this section, we evaluate the performance tradeoffs of our implementation for continual migration system. We conducted three experiments to evaluate the disk, network

and computing drawbacks after continual migration is enabled with different migration frequencies. Dbench, Apache Benchmark and package compiling are used to measure these performance metrics, respectively.

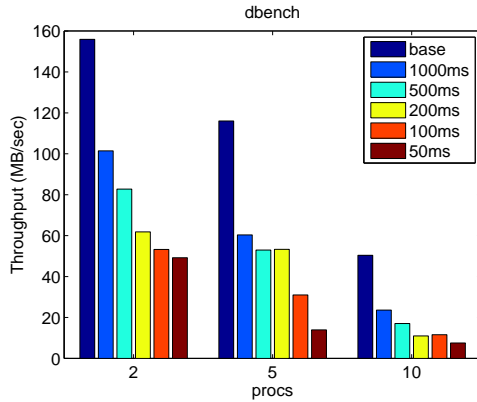


Figure 6. Disk throughput by dbench

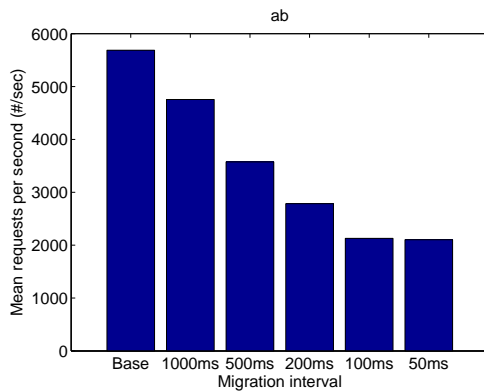


Figure 7. Network throughput by ab

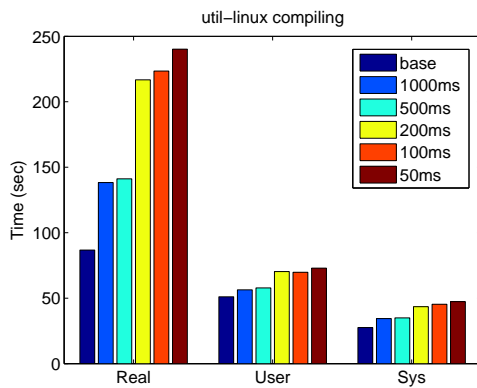


Figure 8. Overall performance by compiling

Figure 6 shows the disk throughputs with different migration interval compared to non-migrating systems. Due

to frequent pausing and resuming of virtual machines, continual migration will certainly affect the performance of virtual machines, and the performance penalty goes greater as migration frequency grows. When migrating in one round per second, disk performance drops down to 65% compared to normal systems; and this value drop to 34% when migration frequency increases to twenty times per second.

Apache Benchmark (ab) is used to measure the mean request counts that Apache running inside the tested virtual machine can handle per second. Figure 7 shows the experimental results. Similar to the dbench results, Apache performance drops as migration frequency grows. In the best configuration, Apache’s capability of handling requests drops to 84% compared to non-migrating system; while in the worst case, this indicator drops to 38% when migrating every 50ms.

Finally, we use a package compiling task to measure the computing and overall performance of the tested system, for compiling job will not only require a significant amount of CPU time, but also issue disk operations and memory modifications frequently. We conducted this experiment by building the Debian package util-linux. Figure 8 demonstrates the result. When migrating in relatively low frequency like one or two rounds per second, the amount of migration data generates the major performance hits. However, as migration frequency increases, the percentage of pausing time increases, which reduces memory refreshing rate but consumes more CPU time.

The above experiments demonstrates that continual migration performs better for network centric workloads compared to disk I/O intensive workloads, and the overall performance tradeoff is around 30% in certain configuration.

VI. CONCLUSION AND FUTURE WORK

In this paper, we demonstrate the design and implementation of our hypervisor based fault tolerant systems via continual migration mechanism. By encapsulating trivial applications in a virtual machine, continual migration can easily provide transparent fault tolerance on top of commodity hardware without requiring redesigning or modifying these applications. We also build our system specifically on shared disk image systems for better agility and shorter migration preparation time. Experimental results show that our prototyped system can reduce machine downtime to less than one second, while maintaining tolerable performance tradeoffs.

We have demonstrated that virtual machine performance is highly affected by migration frequency, thus finding the best migration frequency at which the performance tradeoff is minimized while machine downtime caused by virtual machine rollback is tolerable will be our major target in the future works.

Continual migration is part of our virtual computing environment, which aims to virtualize a set of machines

connected with virtual networks. Our next step is to protect a whole virtual machine cluster against hardware failures. Additional locks and network buffers will be required to achieve this [19].

ACKNOWLEDGMENT

This work is partially supported by grants from China 863 High-tech Program (Project No. 2007AA01Z120, 2009AA01Z419), China 973 Fundamental R&D Program (No. 2005CB321803) and National Natural Science Foundation of China (Project No. 90818028). We would also like to thank members in Network Computing Research team in Institute of Advanced Computing Technology of Beihang University for their hard work.

REFERENCES

- [1] 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," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.
- [2] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1995, pp. 1–11.
- [3] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: virtual machine synchronization for fault tolerance," in *USENIX '08 Poster Session*, San Jose, CA, USA, 2008.
- [4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.
- [5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. New York, NY, USA: ACM, 2002, pp. 211–224.
- [6] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2008, pp. 121–130.
- [7] VMware. (2009) VMware fault tolerance. [Online]. Available: <http://www.vmware.com/products/vsphere/>
- [8] ——. (2008) VMware high availability. [Online]. Available: <http://www.vmware.com/products/vi/vc/ha.html>
- [9] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 161–174.
- [10] A. Agbaria and R. Friedman, "Virtual-machine-based heterogeneous checkpointing," *Softw. Pract. Exper.*, vol. 32, no. 12, pp. 1175–1192, 2002.
- [11] G. Vallee, T. Naughton, H. Ong, and S. L. Scott, "Checkpoint/restart of virtual machines based on xen," in *High Availability and Performance Computing Workshop (HAPCW06)*, 2006.
- [12] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor, "Virtual machine time travel using continuous data protection and checkpointing," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 1, pp. 127–134, 2008.
- [13] J. Huai, Q. Li, and C. Hu, "Research and design on hypervisor based virtual computing environment," *Journal of Software*, vol. 18, no. 8, pp. 2016–2026, 2007.
- [14] X. Lu, H. Wang, and J. Wang, "Internet-based virtual computing environment (ivce): Concepts and architecture," *Science in China (Series E)*, vol. 36, no. 10, pp. 1081–1099, 2006.
- [15] kvm. Kernel based virtual machine. [Online]. Available: <http://www.linux-kvm.org/>
- [16] VMware. Virtual machine disk format (vmdk). [Online]. Available: <http://www.vmware.com/interfaces/vmdk.html>
- [17] QCOW2. The qcow2 image format. [Online]. Available: <http://www.gnome.org/markmc/qcow-image-format.html>
- [18] K. S. Trivedi, *Probability and statistics with reliability, queuing and computer science applications*. Chichester, UK: John Wiley and Sons Ltd., 2002.
- [19] A. Kangarlou, D. Xu, and P. Eugster, "Vnsnap: Taking snapshots of virtual networked environments with minimal downtime," Center for Education and Research, Purdue University, Tech. Rep. 47907-2086, November 2008.