Low Latency Virtualization-based Fault Tolerance

Po-Jui Tsao, Yi-feng Sun, Li-Han Chen, Chuan-Yu Cho

Abstract—Virtualization technology has been widely adopted to reduce IT cost, to improve management and to increase service reliability by consolidating hardware servers and providing automatic virtual infrastructures. However, the reliability of virtual machines running on virtualized servers is threatened by hardware failures beneath the whole virtual infrastructure, but nosy hypervisors that essentially support virtual machines cannot be trusted. To protect virtual machine from hardware failures, virtualization-based fault tolerance system for an individual virtual machine is designed, implemented and evaluated. And we choice epoch-based fault tolerance method because it can support multi-core platform and it can save the backup machine performance overhead compared to log replay method. However, the epoch-based method will bring the long latency overhead, so we need to optimize processor usage and save backup bandwidth. We propose some optimization method such as tracking of dirty virtual device states to saving processor usage and fine-grained dirty region tracking to saving backup bandwidth. Furthermore, we solve the issue about the unexpected long time of snapshot using pending list method. And we also solve the TCP performance issues due to holding output buffer using fake ACK optimization. Finally, we do some experiment to show the performance result about our optimization and we can gain a low-overhead and low-latency virtualization-based fault tolerance system.

***Index Terms*—Fault Tolerance, Hypervisor, Live Migration, Optimization, Virtual Machine, Virtual Machine Monitors, Virtualization**

I. Introduction

Virtualization technology allows multiple VMs (Virtual Machines) running simultaneously on a physical server. In which, all physical resources are virtualized as resource pools of virtual CPU, memory, network card and various virtual devices. With virtualized resources, multiple operating systems could share a single set of physical hardware to not only improve the hardware utilization with less power consumption, but also dramatically enable the elastic software defined manageability, such as live migration, memory snapshot, dynamic provisioning, failed VM restart based high availability service and virtualization-based fault tolerance, …etc.

Among these software-defined management features, VM live migration [1] is considered the most important benefit brought by virtualization because it is the underlining supporting technology to enable zero-down time maintenance service level agreement (SLA). VM migration technology could move a running VM form one physical machine to another without interrupt its service during the complete of migration process. To minimize the migration time, shared network storage system are generally utilized to save the time for migrating storage system in modern virtualized infrastructure. A typical VM live migration configuration is shown in Fig. 1. This system can let VM to migrate to other physical machine quickly because only states of CPU, memory and devices are necessary to migrate to target machine while migration of bulk data on storage system is not needed. That is, to migrate a VM from a physical machine 1 to physical machine 2, it is not necessary to transfer the VM's disk image together; instead only running memory and device status is considered and as a result, it is possible to migrate a VM much faster than in a non-shared storage infrastructure. In general, live migration has been a common practice in virtualized infrastructure and is a powerful tool to save power and eliminate service interrupt during hardware upgrade.

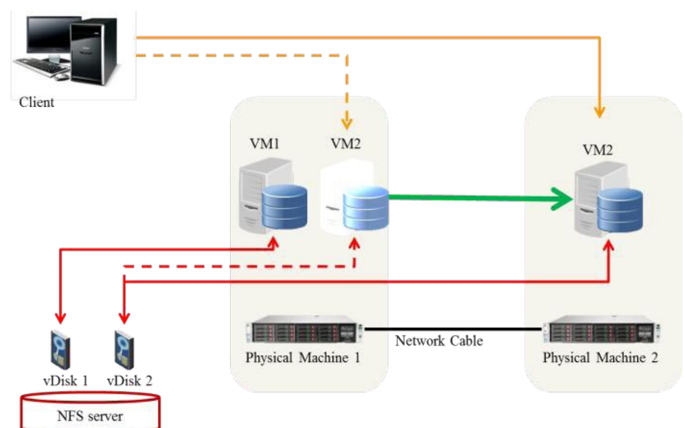

Fig. 1. VM live migration

However, unexpected service downtime may still occur because of hardware accidents, such as power failures or hardware failures, leaving service administrators no time to live migrate the affected VMs. As carrier grade services usually require highly available, uninterrupted service infrastructure, it becomes a new challenge while carriers start to migrate their services on to virtualization infrastructure, which is already a global trend to adopt network function virtualization infrastructure (NFVI) in telecomm datacenter. In this paper, we propose a virtualization-based fault tolerance architecture, which can provide fault tolerance service with controllable low latency while doing efficient and continuous memory status synchronization.

Virtualization-based fault tolerance system is shown in Fig. 2. It can provide uninterrupted VM services when VM stops unexpectedly. Consistent states of virtual machine are repeatedly synchronized to backup physical machine. We name

the running VM as master VM, the backup on backup physical machine as slave VM. When unexpected events happened, the slave VM can take over all ongoing jobs of the master VM because virtualization-based fault tolerance system provides a consistent view of service for clients. Any results on master VM without synchronizing to slave VM will not be exposed to master VM. If the hardware failure happened, the slave VM can take over the master VM's job and provide uninterrupted services for clients.

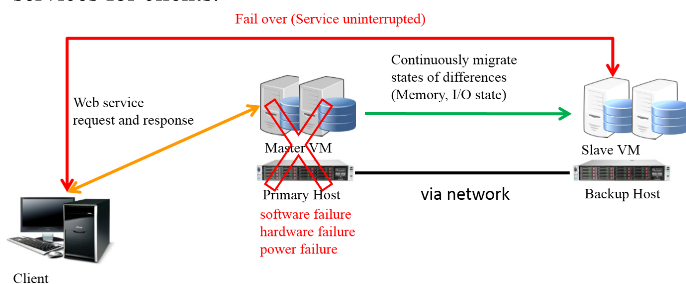

Fig. 2. Virtualization-based fault tolerance system

There are two ways to implement the virtualization-based fault tolerance system. One is to log all asynchronous events of the master VM and deterministic replay the logs on the slave VM [2]. Another one is snapshot mode in which snapshots of master VM are taken and backed up to slave VM frequently. In case of the failure of the master VM, the slave VM will take over and start executing.

The performance overhead which uses deterministically replaying logs on the slave VM with single VCPU is within 10%. To support replay mechanism with multiple VCPUs is difficult, as tracking memory access order for multiple VCPUs is not efficient. Because the slave VM runs as the master VM except it is not exposed like its outputs will not be transferred to users, it will take the same computing resources as master VM.

Remus [3] implemented epoch-based snapshot recovery on Xen [4]. The master VM executes during the epoch period and then it will be paused and its snapshot is taken. After that, master VM can continue running while its snapshot is transferred to slave VM simultaneously, which is named speculative execution. When master VM is running, its outputs are buffered In case of VM crashing the outputs will not be exposed to end users. At the time when the snapshot is synchronized to the slave VM, the outputs are flushed. Remus is based on live migration of VM [1][5].

One major overhead for epoch-based VM fault tolerance is the synchronization of memory states. Live migration faces the same issue when it tries to reduce the migration time. The pre-copy stage of live migration is similar to fault tolerance in that the hypervisor turns on dirty page tracking for VM and keeps transferring dirtied VM states to target VM. Live migration will stay in pre-copy stage until the dirtied pages become small or maximum number of iterations. Then the VM is suspended and the final dirtied pages are transferred. Memory deduplication is used by [6] to exploit the similarity in memory pages in order to avoid transferring redundant Data. Lu and Chiueh propose to speculatively transfer dirtied pages during replication in hope of the number of dirtied pages is reduced at the end of replication [7]. Svard et al. [8] and Du et al. [9] avoid to transfer hotspot pages during iteration.

To provide fault tolerance for VMs, Remus [3] implemented epoch-based fault tolerance on Xen [4] platform and Kemari [10] implemented event-based fault tolerance on KVM. The main point here is not to expose crashing to end user. That is, the Outputs (for example, network packets and disk writes) of the VM are hold of the VM is hold in a buffer until the states of VM of current epoch is backed up. If the VM crashes before the backup is finished, the outputs in the buffer will be simply discarded but the previous backup VM will continue running and produce new consistent outputs without end users' notice. Because the outputs are buffered and only flushed when backup is finished, the frequency of backup must be very high so as to reduce the latency. For event-based fault tolerance, we cannot predict what time we release the output buffer. In other words, we cannot control or bound the latency. So we choose the epoch-based fault tolerance method and optimize it to meet our expectations.

II. Technical Consideration to Deliver an Efficient Fault Tolerance Solution

A. *Epoch-based fault tolerance method system*

In an epoch-based fault tolerance method system, an epoch is a backup cycle and consists of four stages: running stage, snapshot stage, transferring stage and flushing stage, as shown in Fig. 3. The first stage is the running stage which a master VM receives requests from end users and produce outputs. The outputs in this stage cannot be exposed to end users or physical disks directly because if so, when the master VM crashes and the slave VM takes over, the slave VM may not produce the same outputs. The end users or the file system underneath may suffer from inconsistency by the different outputs. Thus the outputs must be hold in an output buffer. However, the output buffer cannot be held too long, because holding the buffer will increase the response latency.

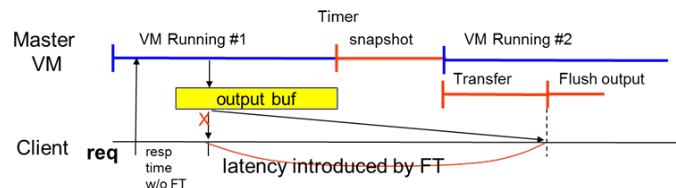

Fig. 3. Epoch-based fault tolerance stage

When the VM runs for an epoch time, an epoch timer will trigger the VM into snapshot stage. The purpose of snapshot stage is to make a local copy for the master VM so that the master VM can run again without waiting for finish of states synchronization so as to improve performance. Then, another thread will transfer the snapshot simultaneously while the master VM is running. This behavior is called speculative fault tolerance.

The simplest way to take snapshot is to copy the whole VM states into local memory. However, since the master VM is totally paused during the snapshot stage, the time length of

snapshot stage affects both output latency and throughput. Besides, the time to take a snapshot is about fixed. So the smaller the epoch sizes the lower throughput we will get. If epoch size is 5ms and snapshot stage is 1ms, we will lose at least 20% throughput from snapshot stage.

For transferring stage, our design is based on the assumption that there is a dedicated 10G bandwidth between the two physical machines where master and slave VMs are running, considering the huge size of data to be transferred in short time. It is not necessary to transfer whole snapshot to the slave VM, because the slave VM has already held the snapshot of the previous epoch. So during transferring stage, the fault tolerance thread need only transfer all the dirtied parts to the slave VM. When transferring finished, all the outputs in the output buffer will be flushed to outside, such as end users and physical disks. The last stage is output stage. As shown in Fig 3, the extra latency penalty is composed of all four stages, in which running stage and flushing stage are intrinsic to our design but snapshot stage and transferring stage can be shortened by optimizations.

B. Unexpected long time of snapshot

When we evaluate the network speed of VM on fault tolerance mode (FT mode), we use wget program to download file from VM to client and from client to VM. The result is shown on Table I.

TABLE I
THE RESULT OF VM NETWORK TRANSFER SPEED EXPERIMENT.

Speed (MB/s)	No blocked	blocked
wget (VM->C)	24.1	6.3
wget (C->VM)	17.7	6.1

In Table I, we use two columns to show the result because sometime the VM will stuck when downloading files and the transfer speed will become very slow. We call this situation is “blocked”. And we do the profiling of every stage to find which blocked reason is. And we find the snapshot time is very huge. The result is shown in Table II.

TABLE II.
THE RESULT OF SNAPSHOT TIME.

Normal	IO workload	Blocked
0.3~0.4 ms	0.5~0.6 ms	100 ms ~ 600ms

In TABLE II, we can see no matter it has workload or not, the snapshot time always less than 1ms (millisecond). But when the blocked happened, the snapshot time increase to 100ms to 600ms. That means some operations spend much time in snapshot stage blocked. After we profiled snapshot stage, we find out that the qemu_aio_wait function will use most of snapshot time when the blocked situation happened. The qemu_aio_wait function waits every aio (asynchronous input/output) device finished its request and call the callback function. If we do not wait for all aio requests, some aio requests will not be finished before the end of epoch. That means when doing failover, the unfinished aio requests will disappear, because the unfinished status will not be saved in any backup. Therefore, we need to find a backup mechanism to hold this information. We talk about this solution on next chapter.

C. TCP Performance issues due to holding output buffer

Because we flush all outputs until transfer stage is finished, the TCP performance would be also affected. For example, when the VM loads a web page, the overhead of response time is very large, as TABLE III shows:

TABLE III
VM WEB PAGE LOADING TEST

Response time (second) Webpage size	No FT	FT mode
7 KB	0.001	0.152
700 KB	0.008	1.490
7000 KB	0.067	3.615

According to Table III, we find that if the web page size is larger, the response time is slower. Transferring big web pages in FT mode is very slow. To find out the root cause, we do another experiment to measure the output requests released on every flush stage. The result is shown in Fig 4.

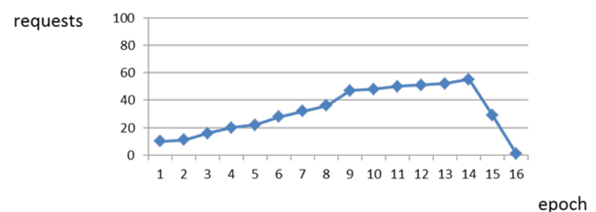

Fig. 4. The release output requests on every flush stage.

In Fig. 4, we can see it need to use 14 epochs for TCP to reach the highest speed, and for the beginning 13 epoch, TCP doesn't reach the full bandwidth the full bandwidth to transfer data. That means if the first to 13th epoch is able to transfer data with full speed, the transfer speed will be increased.

After reviewing the code, we find that the problem resides in TCP congestion window (CWND). In general, the transmission speed is limited by two factors: congestion window held by the sender and receive window held by the receiver. The sender maintains the value of congestion window. If the sender receives an ACK, it updates the value of congestion window. However, in FT mode, all output requests would be held until the backup finished, so the TCP packets are not sent to the receiver immediately, and the sender cannot receive ACKs from the receiver not as soon as non-FT mode Therefore, the congestion window would not increase quickly, so the TCP performance is bad.

III. OPTIMIZATION

As mentioned in the previous chapters, the snapshot stage and transferring stage can be shortened by adding some

optimizations. We propose some optimizations to reduce the time of snapshot stage and transferring stage in this chapter:

A. Tracking of dirty virtual device states

The major components of one VM include CPU states, memory states, virtual device states, external storage states as well as external environment states. The first three classes of states are self-explanatory. The external environment state is the recognition of the VM to other users and systems. For example, packets targeted to this recognized VM will be routed and switched to this VM. If all the states are backed up on the slave VM and the external environment states are set up properly, the slave VM will seamlessly take over master VM's role on the crash of master VM.

One essential step of taking snapshot for the master VM is to transfer all states of virtual devices to the slave VM. And the critical point is to take the snapshot for virtual devices in a very short time considering the high frequency the master VM is backed up to the slave VM (up to 200 times per second). In our testing environment, a VM has around 30 virtual devices, and the original migration codes take about 2ms to collect all the virtual device states. This means the master VM will be paused for 2ms for taking snapshots of virtual device states, which results in an unacceptable performance overhead. Another drawback is that the size of all the virtual device states is around 500Kbytes. For our chosen 5ms epoch time, it means that taking snapshots for virtual device states needs a backup bandwidth of about 100Mbytes/second.

A further observation is that most virtual device states keep unchanged within an epoch. So if only the modified virtual device states are transferred in each epoch, then the bandwidth waste will be reduced. A simple method is to save every virtual device state in a buffer and compare it with the state of this epoch. If these two buffers have the same contents, then we will not transfer it. If the contents are different, we can only transfer the difference between them. However, this method requires collecting all virtual device states in every snapshot stage, which is inefficient.

Another observation is that most virtual device states reside in userspace memory region of QEMU, as Fig. 5 shows. When the guest OS executes an I/O instruction, there will be an exception that causes a VMExit [11] [12]. The exception handler inside KVM module will give the control to QEMU, which will in turn deliver the control to virtual device codes. Virtual devices will act according to their current device states. This step varies for different virtual devices, and it requires a lot of engineering effort to track the virtual device states change. On the contrary, because virtual device codes accesses their virtual device states by QEMU's MMU, there is a workaround method to find out all the changed states. The x86 architecture manual [11] [12] tells us that in paging mode whenever a page is modified, the dirty bit in PTE (page table entry) will be set. So in our case, the virtual device state for a virtual device will be allocated exclusively in memory pages. At the end of each epoch, the dirty bit of PTEs will be scanned. If one dirty bit is set for PTEs of one corresponding virtual device states, then the dirty bit is cleared and TLB will be flushed to invalidated old PTE. The virtual states will be collected and sent to the slave VM.

In this way, only a few (4~5 for every 5ms) virtual device states are modified and backed up to slave VM. The time spent to collect the virtual device states is neglected, and the resulting size is reduced to around 30KB. The original migration codes, which were designed merely for migration and not for frequent backup, are still used to collect and load VM states, saving a lot engineering efforts. With the help of underlining operating system and the MMU hardware, the pause time of the master VM is reduced and the bandwidth are saved for other applications.

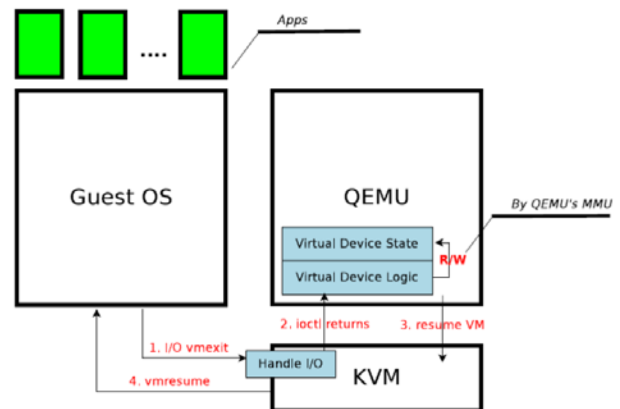

Fig. 5. The virtual device architecture in QEMU/KVM

B. Fine-grained dirty region tracking

In this section, we will first describe the characteristics of the memory page dirtying of a virtual machine: high number in short period and small partial modification within one page. Then the reason is explained why common compression methods cannot be applied here because of their lower compressing rate and high CPU usage. Then, the scatter-gather method used by our low latency virtualization-based fault tolerance system is discussed in the term of CPU usage and bandwidth saving. Also SSE instructions are used in the scatter-gather step to improve the performance.

Memory state is the main part of a virtual machine, holds all critical kernel data structures, user application data and other essential information for the guest OS. As in live migration, memory state handling takes the major part of time in fault tolerance. While the master virtual machine is running, it will process hardware inputs and produce outputs. In this process, a bunch of memory pages will be modified. The baseline design of our low latency virtualization-based fault tolerance system defines that at the end of each epoch, slave virtual machine will have the same memory states as the master virtual machine. As a result, the memory states on slave virtual machine will be synchronized with master virtual machine once per epoch.

The common way to synchronize memory pages from master virtual machine to slave virtual machine is to write protect all pages from guest OS and to track the Pages modified by guest OS. If master virtual machine does not make enough pages

dirty intensively, a normal virtual machine only writes a small portion of its memory pages in one epoch like 5ms. Fig. 6 shows the growing of the number of guest pages get dirtied as the guest OS runs. Both kernel compilation and SpecWeb2005 produce hundreds of dirtied pages within a short time like 5ms. To write protect pages for guest OS, hypervisor marks PTEs read-only in shadow page table. When guest OS tries to write the this page, page fault happens and hypervisor will mark this page as dirtied, then hypervisor makes this page as writable in shadow page table so that guest OS can write to this page. The dirtied pages can also be collected by hardware support like EPT Access/Dirty feature [11].

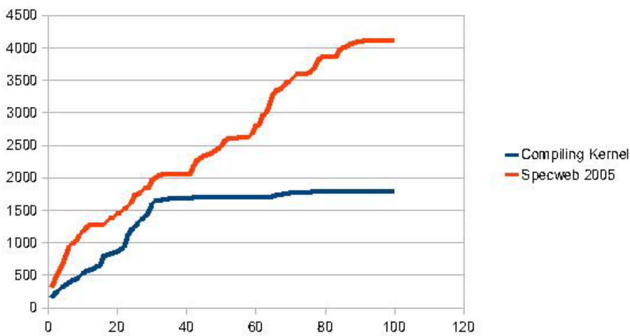

Fig. 6. The number of dirtied pages grows as VM is running.

The page dirtying behavior of virtual machine has two main characteristics: high number and partial modification. Fig. 6 shows that for kernel compilation and SpecWeb2005, the virtual machine writes to up to 800 pages within 5ms, or a rate of about 600Mbytes/second. The baseline design is to support fault tolerance for multiple virtual machines with a dedicated 10G NIC. But the dirty rate for one merely one VM lefts little room for other virtual machines. If the master virtual machine is not slowed down too much to sacrifice performance for bandwidth saving, one dedicated 10G NIC can support two master virtual machines. We know from experiments that averagely only around 12% of one memory page is modified within one 5ms epoch, as shown in Fig. 7. This finding means it is unnecessary to transfer whole dirtied page to slave virtual machine. If there exists a solution to transfer the only modified bytes in one page, then bandwidth requirement of one virtual machine can be reduced to as small as 75Mbytes/second, which theoretically will enable up to 13 virtual machines be fault tolerant with a dedicated 10G NIC.

Compression methods are used to reduce the transferred memory size in many proposals. Fig. 7 shows that after a page is exclusive-or with its original page, Zlib [13] can generate compressed buffer size comparable to the modified bytes of one page. However, commonly known compression algorithms like Zlib, lz4 and lz0 have two drawbacks under such circumstance: lower rate to process input and high CPU usage. According to experiments, these algorithms can compress around 2000 pages per second even configured with lowest compress level. In addition, all of them use all of one CPU time during compressing. Further experiments show that the compressing rate is severely affected when a lot memory copy happens during the process.

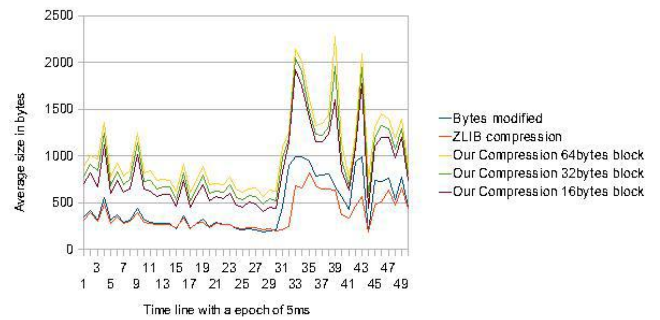

Fig. 7. Statistics of dirtied pages for average modified bytes, block size with scatter-gather and compression results.

Below we will describe a scatter-gather method that utilizes SSE related instructions to implement a fine-grained dirty region tracking algorithm, designed with zero memory copy in mind. Firstly we will introduce a data structure similar to radix tree, as Fig. 8 shows. A page (4096 bytes) is divided into continuous blocks, the size of which depends on how many bits the SSE instructions can handle on the running processor. Assume the block size is 512 bits or 64 bytes, and then a page can be divided into 64 blocks. If one block keeps unmodified during one epoch period, at the end of this epoch this block do not be transferred to the slave VM. Only those modified blocks will be transferred to the slave VM. The slave VM already received the pages at the previous epoch, so by applying the modified blocks to the reference page, the slave VM will get the same content of this dirtied page as on master VM. Alongside with the dirtied blocks, the master VM sends the radix tree data structure to the slave VM as the positions of the dirtied blocks. The radix tree is composed of two levels of dirty bit headers. Each dirty bit header is one byte long. Level one dirty bit header is one byte long, which means each bit represents data length of 4096 bytes / 8 = 512 bytes. If the ith 512 bytes inside the page are dirty, then the (i-1)th bit is set in the level one dirty bit header. For each dirtied data blocks of 512 bytes, there is a level two dirty bit header, each bit represents a data length of 512 bytes / 8 = 64 bytes. Similarly, if the ith block of size 64 bytes is dirtied within the 512 bytes, the (i-1)th bit is set in the level two dirty bit header. So one bit in level 1 dirty bit header corresponds to one level 2 dirty bit header. If one bit is not set in level one dirty bit header, there doesn't exist a corresponding level two dirty bit header for this bit, and there are no data blocks.

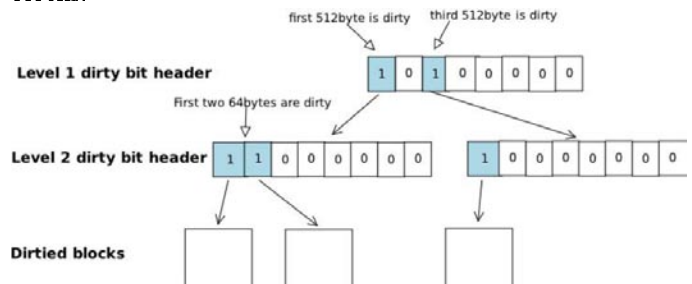

Fig. 8. The radix-tree like structure for gathering fine-grained dirtied block

The bit header is not necessarily a radix-tree like structure. It can be single bitmap. For a block size of 64 bytes, the length of the single bit header is 4096/64/8 = 8 bytes. The header size is small even if all 8 bytes are sent. The radix-tree is used for smaller block sizes. If the block size is 8 bytes, then the length of the single bit header is 64 bytes. An extra 64 bytes for each dirtied page will waste bandwidth.

To generate the radix tree and find out the dirtied blocks within one page, an efficient method is required to decide if one particular block is modified. For any guest physical page, a backup is copied before it is really modified by guest VM. Experiments show that all methods that require scanning the page content cannot reach satisfying processing rate. Our best highly optimized algorithm to compare blocks can only process 22,000 pages per second, while SpecWeb2005 can produce 100,000 dirtied pages per second. Not mention the comparison based on processor consumes 100% CPU usage. Further optimization depends on the SSE related instructions, which will process 16 bytes, 32 bytes or 64 bytes data each time. Firstly one block in the backup page is loaded into SSE register, then the corresponding block in the page being used by master VM is loaded into another SSE register. After these two loadings, these two registers are exclusive or and whether this block was modified or not based on the XOR result. The comparison based on SSE instructions can handle up to 90,000 pages per second, that is, 450 pages per 5ms. What is more, it consumes 40% CPU usage for one VM.

C. Aio request pending list to solve unexpected long time snapshot

As mentioned in previous chapters, if we disable qemu_aio_wait function, some aio requests will be lost because this unfinished status is not be saved on any backup. So we need a method to back up the unfinished aio request. Specifically, we need to record all aio requests until it finished. And we call this record list as pending list. First, we need to find out how many types of aio request we should record. And we find only disk write will produce a long flush time. So we only need to record the disk write events.

The steps of recording write request are shown below: first, we need to hold on every write request until flush stage because the output buffer need to be held until flush stage as described in chapter 2, and save these requests to a temporary list. Then, during flush stage, we issue all requests from temporary list and save all issued requests to pending list. Third, the temporary list and the pending list would be backed up to slave VM. Fourth, if there are new requests issued from temporary list, the request will append to the pending list. Finally, if the request finished, we will remove it from the pending list.

We can let the disk write aio requests be asynchronous, which means taking snapshot can be done without waiting for the disk write requests produced by previous epoch finished. Therefore, the qemu_aio_wait function can be disabled.

D. TCP performance optimization

To increase TCP transmission speed, we need to increase the congestion window in the guest OS. In order to increase window size of guest OS, senders in guest VM should receive ACKs without too long latency.

In order to increase the congestion window, we can send fake ACKs to deceive Guest OS, so the congestion window grows and the transmission speed increases. However, in this scenario, if any packet is lost, we have to deal with it by ourselves; otherwise, Receiver will never receive the lost packets. Therefore, we need to back up all packets and have a TCP stack in QEMU to handle all congestion events, such as packet lost. We implemented the prototype, and the result is shown in chapter IV.

IV. EXPERIMENTAL RESULTS

We evaluate our results to see the difference between FT without optimizations and with optimizations. The first experiment is to test the response time and the system performance overhead. The experiment environment is shown in Fig. 9.

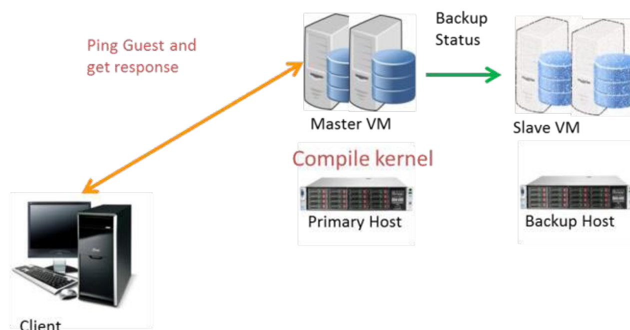

Fig. 9. The experiment environment

About the experiment environment, we use two physical machines, where one is the primary host, which is responsible for running the master VM and another is the backup host, which is for receiving backup status for slave VM. And we use a client machine to ping the master VM for measuring the response time. On the other hand, we run the kernel compiling process and measure its execution time on the master VM, because we need to evaluate the overhead of our low latency virtualization-based fault tolerance system. The result is shown in TABLE IV.

TABLE IV
THE EXPERIMENT RESULT

Version	Client ping Guest Ping Latency (ms)	Guest compile kernel (block device part) Compile kernel time (m's")
No FT	0.3	00'21"
FT No Optimization	430.6	41'38"
+SSE Optimization	64.2	02'36"
+SSE Optimization +Device state dirty tracking	9.8	11'39"
+SSE Optimization +Device state dirty tracking +Pending list	12.3	07'19"

In TABLE IV, we can see the response time is about 400 times slow compare with fault tolerance disabled without any optimization. And the compile time is about 40 times slow compared with fault tolerance disabled. But if we use the SSE optimization (Fine-grained dirty region tracking), we can reduce the response time to 64.2ms, and compile time is only 02'36". By using the version with the tracking of dirty virtual device states optimization, the response time is only 9.8ms, but the compile time will increase to 11'39". The reason is that our response time is reduced and the backup frequency is increased, so the overhead will also increase. Finally, we run the experiment by using the version with the pending list optimization. The response time is increase to 12.3ms, but we can get the compile kernel time only 07'19". Although the performance on VM is not good, the response time is still short.

We also test the TCP performance with our fake ACK optimization, and the result is shown in Fig. 10.

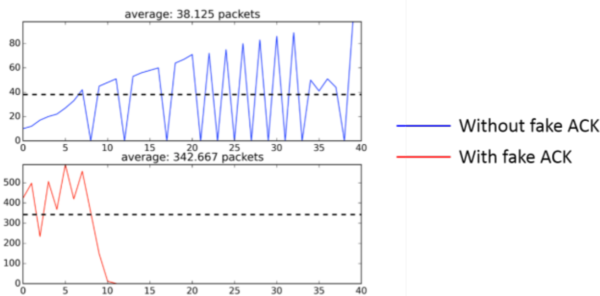

Fig. 10. The TCP performance with fake ACK optimization

In this experiment, we send 100Mbytes data from VM with fault tolerance to a client. With fake ACK optimization, we only need about 11 epochs to transmit all 100Mbytes. Without fake ACK optimization, it needs about 38 epochs. And the average number of packets transmitted in an epoch with and without optimization is 342.667 and 38.125, respectively. The TCP performance is largely increased with our fake ACK optimization.

V. CONCLUSION

To optimize the virtualization-based fault tolerance system, we need to consider reducing the snapshot time and transferring time. Reducing the snapshot time can let VM has more running time within an epoch, so it can increase the VM executing performance. Reducing the transferring time can reduce the response time of VM because the output will be buffered until the flush stage which means the backup finished. As mentioned in the previous chapters, the fault tolerance system latency is 430.6ms without any optimization. But with the SSE optimization, the latency is decreased by 85%. Furthermore, we add the tracking of dirty virtual device states optimization to our fault tolerance system, the latency is also decreased by 85% compared with the SSE optimization version. Although the latency is increased by 26% with pending list optimization, compared with the tracking of dirty virtual device states optimization version, the compiling time is still decreased by 37%. In conclusion, for the latest version of the fault tolerance system, the latency and the compiling time are decreased by 97% and 82%, respectively, compared with the version without any optimization. With several optimizations, the performance of our fault tolerance system is greatly improved. In addition, the VM would run any kind of workload such as CPU-bound or IO-bound. We need to let these workload not affect our fault tolerance backup job and our fault tolerance backup also not affect it, too. In this way, we can get a low overhead and a short response time on our virtualization-based fault tolerance system.

REFERENCES

[1] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew War eld. Live migration of virtual machines. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, pages 273-286. USENIX Association, 2005

[2] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. The design and evaluation of a practical system for fault-tolerant virtual machines. Technical report, Technical Report VMWare-RT-2010-001, VMWare, 2010.

[3] Brendan Cully, Geo rey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew War eld. Remus: High availability via asynchronous virtual machine replication. In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, pages 161-174. San Francisco, 2008.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew War eld. Xen and the art of virtualization. ACM SIGOPS Operating Systems Review, 37 (5):164-177, 2003.

[5] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In USENIX Annual Technical Conference, General Track, pages 391-394, 2005.

[6] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In Cluster Computing (CLUSTER), 2010 IEEE International Conference on, pages 88-96. IEEE, 2010.

[7] Maohua Lu and Tzi-cker Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on, pages 534-543. IEEE, 2009.

[8] Petter Svard, Johan Tordsson, Benoit Hudzia, and Erik Elmroth. High performance live migration through dynamic page transfer reordering and compression. In Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on, pages 542-548. IEEE, 2011.

[9] Yuyang Du, Hongliang Yu, Guangyu Shi, Jian Chen, and Weimin Zheng. Microwiper: E_cient memory propagation in live migration of virtual machines. In Parallel Processing (ICPP), 2010 39th International Conference on, pages 141-149. IEEE, 2010.

[10] Yoshi Tamura. Kemari: Virtual machine synchronization for fault tolerance using domt. Xen Summit, 2008, 2008.
[11] Intel 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.
[12] AMD64 Architecture Programmers Manual. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals
[13] Zlib http://www.zlib.net/

曹伯瑞 (Po-Jui Tsao) received his Master degree in Computer Science from National Cheng Kung University, Tainan, Taiwan, in 2010. He is now an associate software engineer of Datacenter System Software (Div-F) division hypervisor team, Information Comm. Research Lab at ITRI. His research interests include cloud computing, virtualization technology.

孫逸峰 (Yi-feng Sun) is a PhD candidate in Stony Brook University, with a focus on research that improves reliability of Virtual Machines in Cloud Computing.

陳立函 (Li-Han Chen) received his Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan in 2015. Currently, he is a software engineer of hypervisor team in Datacenter System Software (Div-F) division, Information Comm. Research Lab at ITRI. He is interested in system security, mobile security, cloud computing, and virtualization.

卓傳育 (Chuan-Yu Cho) received his Ph.D. degree in Computer Science from National Tsing Hua University, HsinChu, Taiwan, in 2006. He is now a senior software engineer and manager of Datacenter System Software (Div-F) division hypervisor team, Information Comm. Research Lab at ITRI. His research interests include cloud computing, virtualization technology, cyber security, image processing, video coding and streaming.