Live Migration Ate My VM: Recovering a Virtual Machine after Failure of Post-Copy Live Migration

Dinuni Fernando, Jonathan Terner, Kartik Gopalan, Ping Yang
Computer Science Department, State University of New York At Binghamton

***Abstract*—Post-copy is one of the two key techniques (besides pre-copy) for live migration of virtual machines in data centers. Post-copy provides deterministic total migration time and low downtime for write-intensive VMs. However, if post-copy migration fails for any reason, the migrating VM is lost because the VM's latest consistent state is split between the source and destination nodes during migration. In this paper, we present PostCopyFT, a new approach to recover a VM after a destination or network failure during post-copy live migration using an efficient reverse incremental checkpointing mechanism. We have implemented and evaluated our approach in the KVM/QEMU platform. Our experimental results show that the total migration time of post-copy remains unchanged while maintaining low failover time, downtime, and application performance overhead.**

***Index Terms*—virtual machine, live migration, fault tolerance**

I. Introduction

Live migration of a virtual machine (VM) refers to the transfer of an active VM's execution state from one physical machine to another. Live migration is a key feature and selling point for virtualization technologies. Users and service providers of a virtualized infrastructure have many reasons to perform live VM migration such as routine maintenance, load balancing, scaling to meet performance demands, and consolidation to save energy. For instance, vMotion [38] is a popular features of VMWare's ESX server product. Live migration is also extensively used in Google's production infrastructure to perform over a million migrations [35] per month.

Existing live migration mechanisms aim to move VMs as quickly as possible and with minimal impact on the applications and the cluster infrastructure, and indeed these goals have been extensively studied by both academia and industry. Two dominant live migration mechanisms underlie all migration techniques: pre-copy [6], [30] and post-copy [16], [17]. The two techniques differ in whether a VM's CPU execution state is transferred before or after the transfer of its memory pages. In pre-copy, the VM continues executing at the source while its memory contents are transferred to the destination over multiple iterations, at the end of which the CPU execution state is transferred and the VM is resumed at the destination. In contrast, post-copy first transfers the VM's CPU execution state to the destination, where the VM immediately resumes execution while, in the background, the memory pages are actively pushed from source and also retrieved upon page-faults at the destination. Pre-copy works well for applications that mostly read from memory whereas post-copy works well for write-intensive applications that would otherwise prevent pre-copy iterations from converging. Google's data centers [35] use both techniques depending upon the nature of a VM's workload.

An important consideration in live VM migration is the robustness of the migration mechanism itself. Specifically, the source, the destination, or the network itself can fail during live migration. Since a VM encapsulates a cloud customer's critical workload, it is essential that the VM's state is preserved accurately and not lost due to failures during migration. Let's consider a VM's recoverability after a failure during live migration. In both pre-copy and post-copy, the failure of the source node during migration results in a permanent loss of the VM because some or all of the latest state of the VM resides at the source during migration.

However, the two approaches differ in their response to failure of the destination node or the network. For pre-copy, either of these failures is not catastrophic because the source node still holds an up-to-date copy of the VM's execution state from which the VM can be resumed after migration failure. However, for post-copy, a destination or network failure has a more severe implication because the latest state of the VM is split across the source and destination. The destination node has a more up-to-date copy of the VM's execution state and a subset of its memory pages, whereas the source node has pages that have not yet been sent to the destination.

Thus, a failure of the destination or the network during post-copy migration also results in a complete loss of the VM. This failure scenario during post-copy live migration and the resulting loss of VM has not been addressed in existing literature. The problem is important because a VM is particularly vulnerable during migration. VM migration may last anywhere from a few seconds to several minutes, depending on factors such as the VM's memory size and load on the cluster. Thus the window of vulnerability can be large. Additionally, since the VM is live, it is executing code that might communicate over the network with remote entities, altering the external world's view of the VM's state.

In this paper, we propose a solution, called PostCopyFT, to recover a VM after the failure of destination node or network during post-copy live migration. The key idea is as follows. During post-copy, once a VM resumes execution at the destination, the destination concurrently transmits *reverse incremental checkpoints* of the VM back to the source node. This reverse checkpointing proceeds concurrently, and in coordination with, forward post-copy migration from source to destination. If the destination or the network fails during

migration then the source node recovers the VM from the last consistent checkpoint that it received from the destination.

PostCopyFT supports either periodic or event-based reverse incremental checkpointing, with different benefits. The reverse checkpoints are much smaller than full-VM checkpoints because they consist only of the VM's modified memory pages since the last checkpoint, and its CPU and I/O state. For checkpoint consistency, PostCopyFT buffers packet transmissions to external world between successive reverse checkpoints.

We present the implementation and evaluation of a PostCopyFT prototype in the KVM/QEMU virtualization platform. Our results show that, when using PostCopyFT, the total migration time of post-copy remains unchanged. There is a small increase in the cumulative downtime experienced by the VM when using periodic reverse incremental checkpoints, while event-based checkpoints further reduce this overhead.

The rest of the paper is organized as follows. Section II provides background of post-copy and pre-copy live migration techniques. Sections III and IV present the design and the implementation of PostCopyFT, respectively. Section V presents the evaluation of PostCopyFT using various workloads. Section VI presents related work and Section VII concludes the paper.

II. Background

Pre-copy: In pre-copy live migration [6], [30], the VM's memory pages are transferred to the destination host over multiple iterations. In the first iteration, the entire memory state of the VM is transfered and the subsequent iterations transfer only the modified memory pages. When the estimated downtime (the duration when the VM is suspended during migration) is less than a threshold, the VM on the source host is paused and the remaining dirty pages, the device state, and the CPU state are transfered to the destination. The VM is then resumed on the destination.

Post-copy: In post-copy live migration [16], [17], the VM is first suspended on the source host and the CPU state is transferred to the destination host where the VM is resumed immediately. The source then actively sends the memory pages to the destination. This stage is known as the *active push* phase, with the expectation that most pages would be received by the destination before they are accessed by the VM. If the VM accesses a page that has not yet received by the destination, then a page fault is triggered and the source sends the faulted page to the destination (called *demand paging*). During post-copy, after the VM resumes on the destination, the guest OS and all applications inside the VM continue execution on the destination machine.

VM Replication: High availability solutions, such as Remus [7], maintain a consistent replica of a VM during its normal execution. A checkpoint cycle in Remus consists of four stages: VM execution, VM replication, checkpoint transfer, and buffer release. In the first stage, the VM executes, the outgoing network packets of the VM are buffered, and the incoming network requests are served. The outgoing network packets are buffered to ensure that the state of the backup VM will be consistent with the external world during the restoration, if the primary VM crashes in the middle of a checkpoint. The buffer cannot hold the packets for too long as it increases the response latency of the network packets. When the epoch time, defined by the user, is reached, the VM is checkpointed (the second stage), which creates a replica of the primary VM. In this stage, the VM is paused in order to capture the entire system state accurately. The checkpoint is then transfered to the backup node (the third stage). Once the checkpoint is committed to the backup node, the buffered outputs are released.

III. Design

During the post-copy migration, the VM's state is split between the source and the destination nodes. As a result, a failure of the destination node or in the network during the migration may lead to the complete loss of the VM. PostCopyFT is developed to recover the VM in this failure scenario. Figure 1 shows the architecture of PostCopyFT.

A. Reverse Incremental Checkpointing

The first step of post-copy migration is to transfer the execution state of the VM, along with a minimal non-pageable memory state, to the destination. The VM is resumes at the destination while concurrently receiving the VM's pages from the source. PostCopyFT superimposes a reverse incremental checkpointing mechanism over this forward transfer of VM state. Specifically, once the migrating VM is resumed at the destination, PostCopyFT captures the the VM's initial execution state and memory at the destination and transfers them to a checkpoint store. This checkpoint store is an in-memory key-value store located at either the source node or a third staging node. Then onwards, PostCopyFT captures any incremental changes in the VM's state, including the execution state and any modified memory pages, either periodically or upon an I/O activity of the VM, and forwards these to the checkpoint store. This checkpointing mechanism stops once post-copy migration successful completes the VM's migration to the destination.

B. Failure Recovery

When the network or the destination node fails due to a hardware or software failure, the source node triggers a restoration process. The source node uses heartbeat messages to monitor the liveness and reachability of the destination node. When successive heartbeat messages are not acknowledged by the destination, the migration is considered to have failed. The source then recovers the VM by restoring the last consistent copy of each memory page from the checkpoint store on the VM's memory address space. Pages not modified by the destination do not need to be overwritten. Finally, the VM is resumed at the source from the latest checkpointed CPU execution state to complete the VM's recovery,

C. Network Buffering

To ensure the consistency of VM checkpoints, PostCopyFT buffers packet transmissions to external world between successive incremental checkpoints. The incoming network packets of the migrating VM are delivered to the VM immediately, but the outgoing network packets are buffered until the current reverse checkpoint is committed. The packets in the network buffer are then transmitted and the VM is resumed. This ensures that no network packets are transmitted before the corresponding checkpoint is committed to the checkpoint store. Thus, if the destination or network fails during the migration, PostCopyFT guarantees that the latest committed checkpoint reflects a consistent state of the VM to the external world.

D. Checkpointing Overhead Reduction

One of the requirements in PostCopyFT design is that the reverse incremental checkpointing mechanism should not significantly increase the total migration time if the migration succeeds. To satisfy this requirement, reverse checkpointing is implemented as a separate thread that runs concurrently with the VM. The only time this thread affects the VM's execution is when suspending the VCPUs briefly to capture their execution state. The active-push phase from the source to destination runs concurrently with the reverse checkpointing mechanism even when the VCPUs are paused. This helps PostCopyFT to achieve similar total migration time as post-copy live migration.

A second design requirement is that PostCopyFT's reverse checkpointing mechanism should not significantly increase the VM downtime. PostCopyFT introduces the following two optimizations for this purpose.

Performing incremental VM checkpointing in two stages: Periodic VM checkpointing may increase the downtime of migrating memory-write intensive VMs whose memory pages are dirtied rapidly. To reduce the downtime, PostCopyFT performs VM checkpointing in two stages. In Stage I, PostCopyFT checkpoints only the modified memory pages of the VM, but not its execution state, which includes the VM's CPU and device states. The modified memory pages are checkpointed without pausing the VM and hence Stage I does not increase VM downtime. In Stage II, the VM is paused briefly to capture the VM's execution state. Once Stage II is completed, the VM resumes its execution. The committed checkpoint contains the memory pages checkpointed in both stages. If a memory page is checkpointed in both stages, then the page checkpointed in Stage I is overwritten by that checkpointed in Stage II to ensure that the checkpoint contains the most up-to-date page. Our experimental results show that performing checkpointing in two stages significantly reduces the migration downtime, compared to if the VM was paused during memory capture.

Storing checkpoints locally: In Stage I, the VM's memory states are first copied to a local in-memory storage without waiting for the checkpoints to be transferred or synchronized with the staging node storing the checkpoints. After State II, the VM is resumed and the checkpointed memory and execution states are transferred to the staging node.

IV. Implementation

PostCopyFT was implemented on the KVM/QEMU [3], [22] virtualization platform version 2.8.1.1. The guest OS and applications running inside the migrating VM are unmodified. Each VM in KVM/QEMU is associated with a userspace management process, called QEMU, which performs I/O device emulation and various management functions such as VM migration and checkpointing. The userspace QEMU process communicates with a kernel-space hypervisor called KVM, which uses hardware virtualization features to execute the VM in guest mode (or non-root mode).

A. Capturing VM's Memory and Execution State

In parallel to forward post-copy migration, the reverse incremental checkpointing mechanism is implemented as a concurrent QEMU thread to capture consistent incremental checkpoints. To track modified memory pages for successive incremental checkpoints, PostCopyFT performs dirty page tracking on the VM executing at the destination node. The dirty page tracking mechanism represents the VM's memory content as a bitmap, in which each bit specifies whether a page is modified or not. KVM uses this bitmap to identify pages dirtied by the VM at a given instant. When the VM resumes on the destination host during post-copy migration, the reverse checkpointing thread informs KVM to mark all memory pages as read-only using an *ioctl* call. When the VM attempts to write to any of its resident memory pages, a write fault is triggered, upon which KVM marks the page as read-write and turns on the corresponding bit in the dirty bitmap. Successive writes to the same page do not trigger any bitmap updates until the bitmap is reset for the next checkpointing round. In each checkpointing iteration, the reverse checkpointing thread in QEMU retrieves the current dirty bitmap from KVM to userspace for better manageability. Once QEMU transfers dirty pages to the checkpoint store, the corresponding dirty bits are cleared.

The execution state of a VM consists of the CPU and the I/O device states, which keeps changing during the execution of the VM. PostCopyFT modifies QEMU's default method of capturing the VM's execution state to include the corresponding checkpoint version number (used to uniquely identify a checkpoint cycle) in an in-memory QEMUFile data structure. The QEMUFile structure is then transferred to the checkpoint store.

Between successive checkpointing rounds, a network barrier is inserted to buffer outgoing network packets of the migrating VM. We implemented an event-based VM checkpointing algorithm to reduce the network packet buffering latency. In the event-based approach, VM checkpointing is triggered only when an external event occurs, such as when the VM sends an outgoing network packet. As a result, the event-based approach captures the system state less frequently and hence reduces the migration downtime.

B. Reducing The Migration Downtime

Traversing the bitmap for modified pages and transferring each modified page to the checkpoint store increases the VM migration downtime due to the synchronization overhead of each write request. To address this issue, instead of sending the checkpoint directly to the checkpoint store on the source host, the incremental memory changes are first captured in an in-memory dynamically sized local data structure *checkpoint_stage*. Our *checkpoint_stage* is similar to the implementation of Linux Kernel cache-slab [4], [26]. It consists of a vector of pointers that point to contiguous memory chunks. Each memory chunk contains a series of page data and page keys. Once all chunks are filled, the list is doubled, and new chunks are allocated. Storing memory states locally reduces the VM downtime caused by the synchronous writes and provides the assurance of completeness in checkpoints.

C. Checkpoint Store

Once the VM's memory states are captured in a checkpointing cycle, memory states stored in *checkpoint_stage* are transferred to the checkpoint store at the source node. We considered several key factors when selecting a checkpoint store. In order to store incremental checkpoints, the checkpoint store should be an in-memory cache that provides duplicate filtering and allows for checkpoint versioning. During a failure situation, the checkpoint store may contain several checkpoints captured in different checkpoint iterations. The checkpoint store needs to maintain each VM checkpoint iteration separately along with a version number that represents the most recently committed checkpoint. That way, there is no ambiguity as to if a checkpoint is complete or not, and we can discard incomplete checkpoints if a failure occurs in the middle of a checkpoint. We used Redis [34], an in-memory distributed key-value store, to implement PostCopyFT's checkpoint store. The Redis client resides on the destination host while the Redis server resides on the source or the staging node. We store the checkpointed memory state in a map data structure in Redis. Each memory page is stored as a key-value pair using function *HMSET*, where the key is a unique page address consisting of page block id and page offset, and the value is the page content. Once the checkpoint is transferred to the checkpoint store, the checkpoint is marked as complete.

D. Detection of Destination/Network Failure

PostCopyFT uses the heartbeat mechanism as a simple detector to detect the destination/network failure. The heartbeat module was implemented as a separate thread on the source host that continuously monitors the availability of the destination node by periodically sending ping requests to the destination. If the heartbeat module cannot reach the destination host for a time interval, then the destination node is considered be in a failure state and the source host immediately triggers a failover.

E. VM Recovery after Failure

After detecting the migration failure, the restoration process on the source machine tries to restore the VM from the checkpoints previously received from the destination. Restoration was implemented as a non-live process. To build up the entire memory state, the restoration process loads the memory pages for each checkpoint version up to the latest committed complete checkpoint version number at once into a hashtable using the *HGETALL* function provided in Redis. The hashtable provides the capability to update memory pages with the most recently modified pages and merge duplicate pages. The restoration thread then places the page content on the allocated and mapped host memory address space using $mmap()$. Next, the restoration process loads the most recent execution state of the VM using the *GET* function and loads the execution state into a in-memory QEMUFile structure. This file is then passed to the modified *qemu_loadvm_section_start_full ()* function, which proceeds to unpack and load the execution state. Finally then VM is resumed.

V. Evaluation

This section presents the performance results of PostCopyFT using the following metrics:

- *Total migration time:* Time taken to transfer a VM's state entirely from the source to the destination host.
- *Downtime:* Duration that a VM is suspended during the migration.
- *Replication time:* Time taken to transfer the checkpoint to the checkpoint cache store.
- *Application performance degradation:* The performance impact on applications running inside the VM during migration and checkpointing.
- *Network Bandwidth degradation:* Reduction in network bandwidth during migration and checkpointing.
- *Failover time:* Time taken to restore the VM from the last committed checkpointing during a failure situation.

Our test environment consists of dual six-core 2.1 GHz Intel Xeon machines with $128GB$ memory connected through a Gigabit Ethernet switch with 1 Gbps full-duplex ports. To avoid network interference and contention between the migration traffic and the application-specific traffic, separate NIC interfaces are used for the migration and the application traffic. VMs in each experiment are configured with 1 vCPU and $8GB$ of memory with $4KB$ page size unless specified otherwise. Virtual disks are accessed over the network from an NFS server, which enables each VM to access its storage from both the source and the destination machines over local area network.

A. Baseline Comparison of PostCopyFT and Post-copy

To measure the total migration time and the downtime of PostCopyFT, the checkpointing thread ran continuously on the destination host without triggering a failure in the migration in order to evaluate the overhead of VM checkpointing. Figure 2 compares the time taken to migrate an idle VM using PostCopyFT and the KVM/QEMU vanilla post-copy

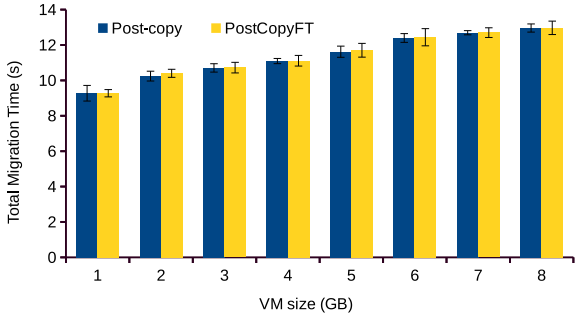

Fig. 2. The total migration time of PostCopyFT and Post-copy for migrating $1GB - -8GB$ idle VMs.

implementation [16], [17]. The size of the VM ranges from $1GB$ to $8GB$. As shown in Figure 2, the total migration time of PostCopyFT and post-copy is almost the same for idle VMs with different sizes.

Figure 3 shows the downtime and the replication time of migrating an idle VM using PostCopyFT. The downtime of PostCopyFT for migrating an idle VM ranges between 1.1 seconds and 1.9 seconds, which is higher than that of post-copy ($9ms - 11.6ms$). This is because the VM pauses continuously for every checkpointing interval ($100\mu s$) to checkpoint memory states. The figure also shows that the replication time is much higher than the downtime. This is because VM checkpointing is performed in two stages and only stage II pauses the VM (the details are given in Section III-D).

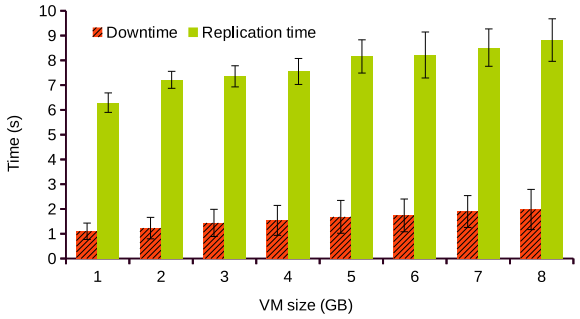

Fig. 3. The downtime and replication time for migrating idle VMs with different sizes.

Figure 4 shows the total migration time for migrating a memory-write intensive VM using PostCopyFT and post-copy. The memory-write intensive application running inside the VM is a C program that continuously writes random numbers to a large region of main memory. The size of the working set (i.e., the size of the memory written) ranges from $1GB$ to $5GB$. The figure shows that the total migration time of PostCopyFT is almost the same as post-copy. This is because, even when the VM is paused for checkpointing, the destination is actively gathering pages from the source host.

Figure 5 shows the downtime and the replication time of PostCopyFT for migrating a memory-write intensive VM.

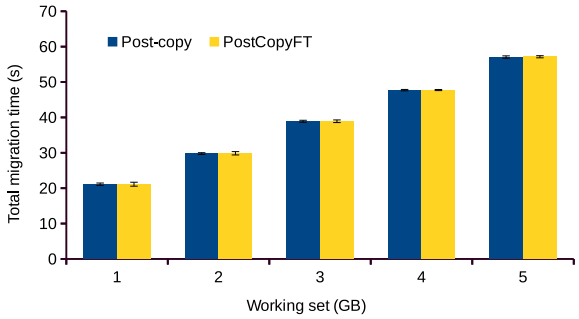

Fig. 4. The total migration time of migrating memory write-intensive VMs using post-copy and PostCopyFT.

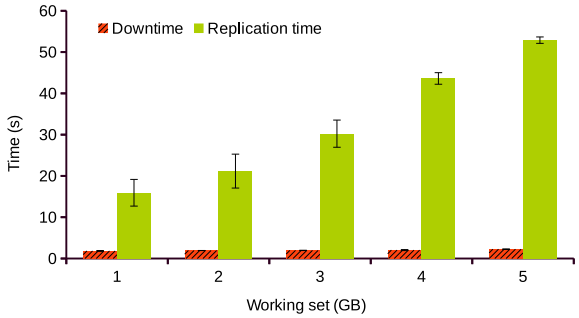

Fig. 5. The downtime and replication time for migrating memory-write intensive VMs using PostCopyFT.

The downtime of PostCopyFT ranges between $1.8s$ and $2.2s$, which is higher than that of post-copy ($7ms - 9ms$). The downtime respect to migration thread is almost constant for PostCopyFT and post-copy as the migration process is not interrupted by PostCopyFT during its checkpointing phase. But with the added overhead due to frequent checkpointing, the migration downtime of PostCopyFT increases. The figure also shows that the replication time increases when the size of the working set increases. This is because when the size of the working set increases, the number of dirty pages also increases.

B. Impact of Checkpointing Interval

This section evaluates the impact of the checkpointing interval on VM migration and the overhead incurred due to checkpointing. Figure 6 shows the total migration time of migrating an idle VM when the checkpointing interval varies between $0.1ms$ and $100ms$. Checkpointing interval 0 refers to the vanilla post-copy migration. PostCopyFT imposes $0.2\% - 0.9\%$ overhead compared to the post-copy migration and the overhead decreases linearly when the checkpointing interval increases. Figure 7 shows the downtime and the replication time of PostCopyFT by varying the checkpointing interval for idle VMs. The figure shows that when the checkpointing interval decreases, the downtime increases. This is because, when the checkpointing interval is low, the checkpointing is performed frequently with the overhead of

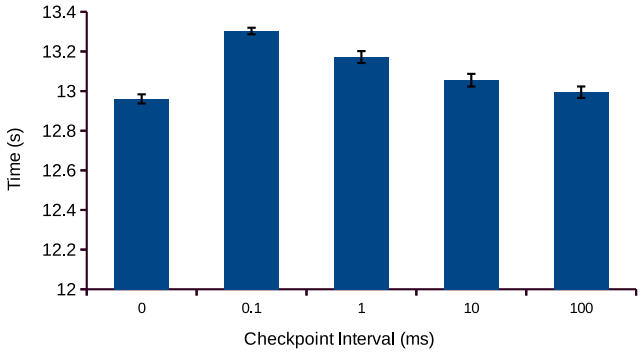

Fig. 6. The impact of checkpointing interval on the total migration time (idle VM).

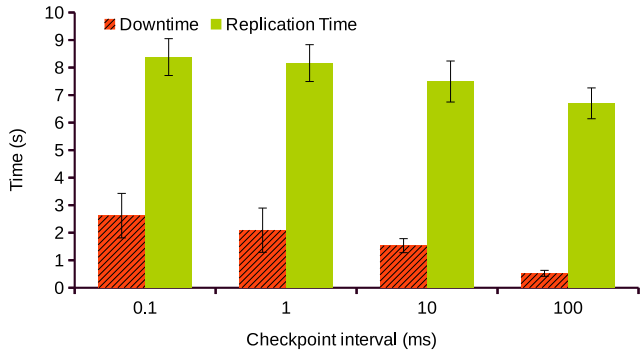

Fig. 7. The impact of checkpointing interval on the migration downtime and the replication time (idle VM).

bitmap synchronization, state transfer, network buffering, and the pause of the VM.

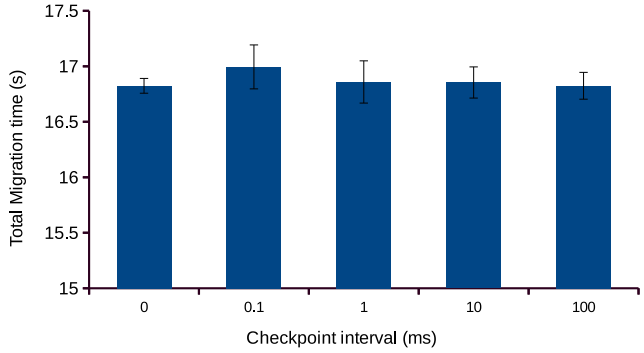

Fig. 8. The impact of checkpointing interval on the total migration time when migrating a $5GB$ memory-write intensive VM.

Figure 8 shows the total migration time of migrating a $5GB$ memory-write intensive VM when the checkpointing interval varies between $0.1ms$ and $100ms$. Checkpointing interval 0 refers to the post-copy migration. PostCopyFT imposes $0.1\% - 0.7\%$ overhead compared to the post-copy migration and the overhead decreases linearly when the checkpointing

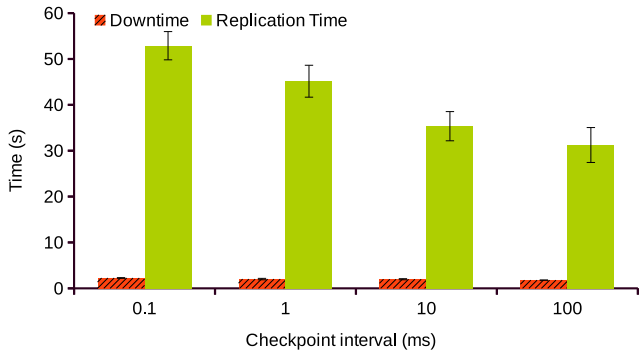

Fig. 9. The impact of checkpointing interval on downtime and replication time when migrating a $5GB$ memory-write intensive VM.

interval increases. Figure 9 shows the downtime and the replication time of migrating the $5GB$ memory-write intensive VM. The figure shows that the downtime and the replication time decrease when the checkpointing interval increases. This is because, when the checkpointing interval decreases, more memory pages get dirty, which results in higher bitmap synchronizations and larger checkpoint state.

C. Performance Impact on CPU-Intensive Workload

We measured how PostCopyFT and post-copy affect the performance of CPU-intensive workloads using the QuickSort benchmark, which repeatedly allocated 400 Bytes of memory, wrote random integers to the allocated memory segment, and sorted the integers using the QuickSort algorithm. We measured the number of sorts performed (i.e., the number of times the random integers are written to memory and are sorted) per second during the migration. Figure 10 shows that PostCopyFT has similar performance as post-copy and there is no observable adverse impact of the reverse incremental checkpointing mechanism during migration.

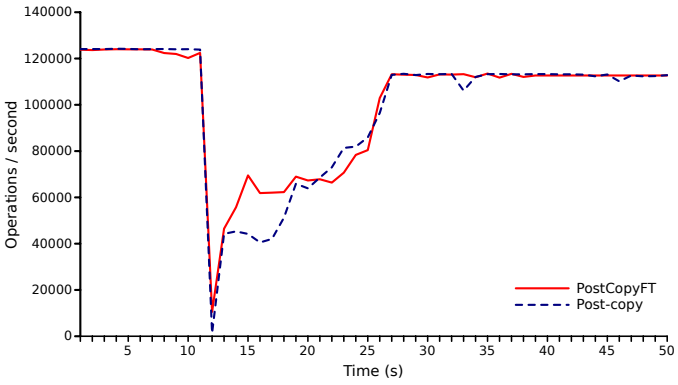

Fig. 10. The performance impact of post-copy and PostCopyFT on a CPU-intensive QuickSort benchmark.

D. Network Buffering Overhead

VM migration itself is a network intensive process. To mitigate the contention of network resources between the

migration process and the applications running inside the VM, PostCopyFT uses separate NIC interfaces for the migration traffic and the application traffic. This section presents the impact of PostCopyFT's periodic and event-based checkpointing mechanisms on network-intensive VMs. As the event-based checkpointing mechanism captures the VM's memory states only when the VM transmits an outgoing network packet, compared to the periodic checkpointing, the event-based checkpointing reduces the network latency and the time taken to pause/restart the VM and synchronize the dirty page bitmap. The event based checkpointing mechanism shows $2-3$ times lower downtime than the periodic checkpointing mechanism.

We used iPerf [18], a network intensive application to measure the outgoing and incoming network bandwidth of PostCopyFT, where the maximum network bandwidth is set to 100Mbit/s. The iPerf server runs on an external machine (i.e neither source nor destination host) in the same cluster and the iPerf client runs inside the migrating VM. During the migration, the client continuously sends data to the server through a TCP and UDP connection as shown in Figures 11 and 12. The network bandwidth is captured using iPerf every 0.1 second. Bandwidth fluctuations are seen for both TCP based and UDP based outgoing network traffic due to the impact of network buffering and releasing. Compared to the periodic checkpointing mechanism, the event-based checkpointing mechanism shows more frequent buffer releases and hence has higher bandwidth. Low bandwidth throughput is shown in the TCP based outgoing traffic due to TCP acknowledgments and timeouts.

When migrating an incoming network-intensive VM, the iPerf server runs inside the migrating VM and the iPerf client runs on an external machine in the same cluster. Figures 13 and 14 give the network bandwidth of incoming network intensive applications for TCP and UDP connections, respectively. For the TCP protocol, although PostCopyFT serves incoming network packets, the acknowledgment packets are being buffered as they are considered as outgoing network packet. Thus bandwidth fluctuations are seen in the incoming TCP network traffic workloads due to the acknowledgment packets. Compared to the periodic checkpointing mechanism, the event-base checkpointing mechanism shows much higher bandwidth.

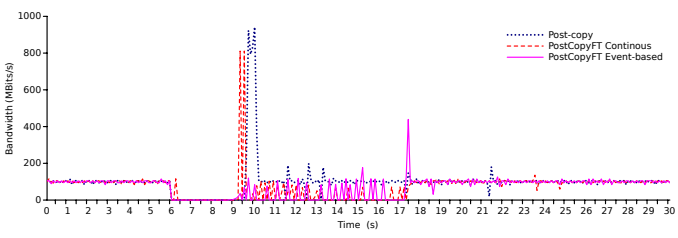

Fig. 11. Outgoing network bandwidth of post-copy, PostCopyFT with periodic checkpointing, and PostCopyFT with event-based checkpointing, when the migrating VM generates outgoing TCP traffic.

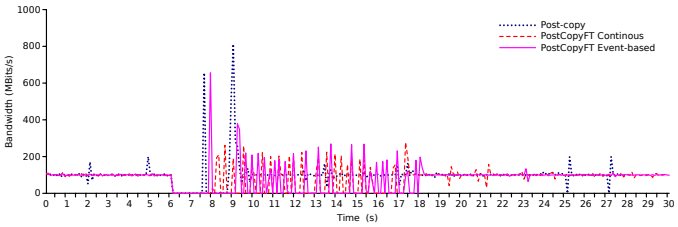

Fig. 12. Outgoing network bandwidth of post-copy, PostCopyFT with periodic checkpointing, and PostCopyFT with event-based checkpointing, when the migrating VM generates outgoing UDP traffic.

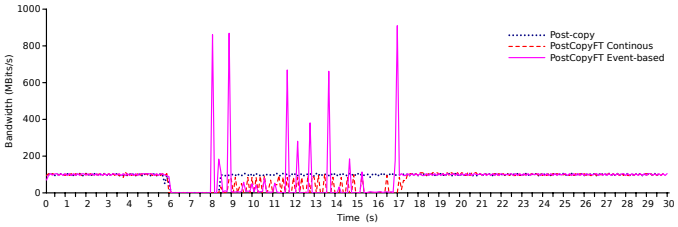

Fig. 13. Incoming network bandwidth of post-copy, PostCopyFT with periodic checkpointing, and PostCopyFT with event-based checkpointing, when the migrating VM receives incoming TCP traffic.

E. Failover Time

Upon detecting a failure during the migration of a VM, the time taken to recover the VM from the latest available committed state is called the *failover time*. Figure 15 shows the failover time when migrating a $5GB$ memory-write intensive VM using PostCopyFT. We varied the number of checkpointed pages and captured the time taken to restore the VM on the source host. The figure shows that, the failover time increases linearly when the size of the checkpoint increases.

VI. Related Work

This section reviews the related literature on VM fault-tolerance. In contrast to existing techniques which aim to recover a VM after failure during normal execution, PostCopyFT is the first approach to address the problem of VM recovery after failure during live migration.

Checkpointing based fault-tolerance: Almost all virtualization platforms [2], [22], [37] support VM checkpointing and restoration. In checkpointing, the memory state of a VM is captured and preserved locally or remotely. During a failure situation, the VM is rolled back/restored to the previously checkpointed state. However, after the restoration, the VM states between the last checkpoint and the time when the failure occurs are lost. Checkpointing can be done at either the application level [25], [41] or the whole system level [7], [10], [13], [19], [20], [23], [24], [27], [29], [32]. Several operating systems [1], [28], [31], [33] were developed to support process-level migration. The main challenge in process level migration for fault tolerance is that the migrated process leaves residual dependencies in the source machine and the solutions for high availability process level checkpoint/restoration have to deal with such dependencies. Compared to the application specific checkpointing schemes, whole system checkpoints

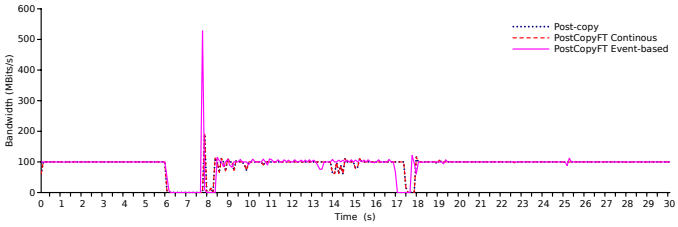

Fig. 14. Incoming network bandwidth of post-copy, PostCopyFT with periodic checkpointing, and PostCopyFT with event-based checkpointing, when the migrating VM receives incoming UDP traffic.

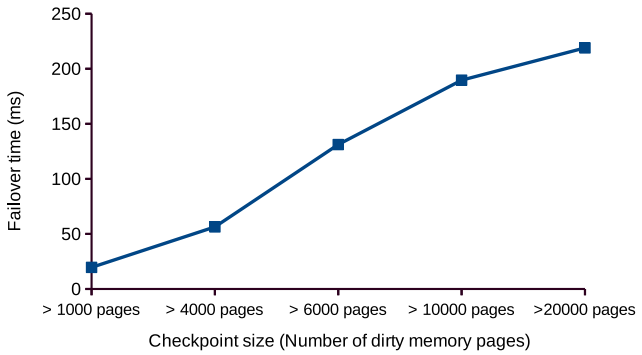

Fig. 15. The time taken for recovering a memory write-intensive VM from checkpoints with different sizes.

provide more reliability and higher availability with significantly higher cost.

The traditional way of achieving fault tolerance via checkpointing is an active-passive approach [7], [27], [39] where the backup node gets the control once the primary VM failed. However, the active-passive approaches suffer from extra network latency due to outgoing network packet buffering, large memory state transfer, and high checkpointing frequency. To provide seamless failover restoration, active-active replications [10], [40] are introduced in which the primary and backup VMs execute in parallel. These systems compare the responses to client requests to decide when the checkpoint should be triggered. If the output of both primary and backup VMs diverges, committing of network responses is withheld until the primary VM's memory state is synchronized with the backup VM. Network buffering latency is common for both active-passive and active-active approaches. Instead of buffering external events (i.e, network packets), Kemari [36] initiates the checkpoint of the primary VM when the hypervisor starts duplicating the external events to the backup VM. None of the above approaches applied periodic or event-based replication fault tolerance solutions to handle destination failure during live migration.

Logging and replay based fault-tolerance: Logging mechanisms can replay events at runtime to ensure the identical backup of the primary VM. Bressoud *et al.* [5] proposed a hypervisor-based fault tolerance approach in which the hypervisor logs each instruction level operation on the primary VM and replayed the logged state on the secondary VM. Although logging can be done in the virtual machine monitor (VMM) [5], [11], [21], [29], deterministic replay relies on the architecture of the VMM and cannot be easily viable to multi-core CPUs. Approaches such as Flight Data Recorder [42] sniff cache traffic to infer how shared memory is accessed. Dunlap [12] used CREW (concurrent read, exclusive write) protocol on shared memory to capture the access order. However, deterministic replay mechanisms work with high overhead.

Live migration as a VM fault-tolerance technique: Techniques have also been developed to quickly evict VMs from the source to the destination machine upon imminent failure of the source machine [8], [9], [14], [15]. However, none of them protect the VM against the failure of the live migration itself.

VII. Conclusion

In this paper, we addressed the problem of recovering a virtual machine when post-copy live migration fails due to destination or network failure. We proposed a reverse incremental checkpointing solution, called PostCopyFT, which proceeds concurrently with traditional post-copy migration and transfers incremental changes to a VM's memory and execution state from the destination back to the source, either periodically or upon external I/O events. Upon network or destination failure, PostCopyFT recovers the VM on the source node from the latest consistent checkpoint. Our implementation of PostCopyFT in the KVM/QEMU platform yields similar total migration time compared to the traditional post-copy live migration with low impact on application performance.

Acknowledgement

This work is supported in part by the National Science Foundation, USA, via awards 1320689 and 1527338.

References

[1] A. Barak and R. Wheeler. Mosix: An integrated multiprocessor unix. In *Mobility*, pages 41–53. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[2] 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. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. of USENIX Annual Technical Conference*, April 2005.

[4] J. Bonwick and J. Adams. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *USENIX Annual Technical Conference*, pages 15–33, 2001.

[5] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[6] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 161–174, 2008.

[8] U. Deshpande, D. Chan, S. Chan, K. Gopalan, and N. Bila. Scatter-gather live migration of virtual machines. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2015.

[9] U. Deshpande, Y. You, D. Chan, N. Bila, and K. Gopalan. Fast server deprovisioning through scatter-gather live migration of virtual machines. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 376–383, June 2014.

[10] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *the 4th Annual Symposium on Cloud Computing*, pages 1–16, 2013.

[11] 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. *SIGOPS Oper. Syst. Rev.*, 36:211–224, Dec. 2002.

[12] G. W. Dunlap, III. *Execution Replay for Intrusion Analysis*. PhD thesis, Ann Arbor, MI, USA, 2006.

[13] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott. Proactive fault tolerance using preemptive migration. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 252–257, 2009.

[14] D. Fernando, H. Bagdi, Y. Hu, P. Yang, K. Gopalan, C. Kamhoua, and K. Kwiat. Quick eviction of virtual machines through proactive live snapshots. In *the 9th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 99–107, 2016.

[15] D. Fernando, H. Bagdi, Y. Hu, P. Yang, K. Gopalan, C. Kamhoua, and K. Kwiat. Quick eviction of virtual machines through proactive snapshots. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 156–157, Sept 2016.

[16] M. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *In SIGOPS Operating Systems Review*, July 2009.

[17] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE*, pages 51–60, 2009.

[18] Iperf. The TCP/UDP Bandwidth Measurement Tool, http://dast.nlanr .net/Projects/Iperf/.

[19] R. Jhawar, V. Piuri, and M. Santambrogio. Fault tolerance management in cloud computing: A system-level perspective. *IEEE Systems Journal*, 7(2):288–297, June 2013.

[20] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked infrastructures in the cloud. *IEEE Transactions on Services Computing*, 5(4):484–496, 2012.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Annual Conference on USENIX Annual Technical Conference*, pages 1–1, 2005.

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: The linux virtual machine monitor. In *Proc. of Linux Symposium*, June 2007.

[23] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1(1), Aug. 2008.

[24] M. Lu and T. c. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, 2009.

[25] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghil. Optimizing checkpoint sizes in the c3 system. In *the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[26] R. McDougall and J. Maura. Solaris internals. In *Rachael Borden*, 2001.

[27] Micro Checkpointing. https://wiki.qemu.org/features/microcheckpointing.

[28] S. J. Mullender, G. van Rossum, A. S. Tananbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[29] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *the 21st Annual International Conference on Supercomputing*, pages 23–32, 2007.

[30] M. Nelson, B. H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proc. of USENIX Annual Technical Conference*, April 2005.

[31] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, Feb. 1988.

[32] E. Park, B. Egger, and J. Lee. Fast and space-efficient virtual machine checkpointing. In *the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 75–86, 2011.

[33] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. In *the Eighth ACM Symposium on Operating Systems Principles*, pages 64–75, 1981.

[34] Redis. In-memory data structure store , https://redis.io/.

[35] A. Ruprecht, D. Jones, D. Shiraev, G. Harmon, M. Spivak, M. Krebs, M. Baker-Harvey, and T. Sanderson. Vm live migration at scale. In *the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 45–56.

[36] Y. Tamura, K. Sato, S. Kihara, and S. Moriai. Kemari: Virtual machine synchronization for fault tolerance using domt, ntt cyber space labs. Technical report, Proc. USENIX Annual Technical Conference, 2008.

[37] VMWare Inc. VMWare Inc- Architecture of VMware ESXi , http://www.vmware.com/files/pdf/esxi architecture.pdf.

[38] VMWare Inc. VMWare vMotion https://www.vmware.com/pdf/vmotion _datasheet.pdf.

[39] VMWare Inc. VMWare vSphere 6 Fault Tolerance https://www .vmware.com/files/pdf/techpaper/vmware-vsphere6-ft-arch-perf.pdf.

[40] C. Wang, X. Chen, Z. Wang, Y. Zhu, and H. Cui. A fast, general storage replication protocol for active-active virtual machine fault tolerance. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 151–160, Dec 2017.

[41] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008.

[42] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *the 30th International Symposium on Computer Architecture*, pages 122–135, 2003.

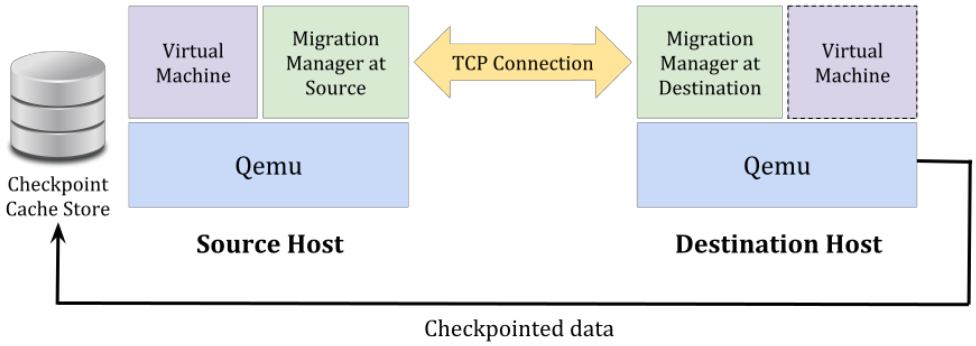

Fig. 1. The architecture of PostCopyFT.