

V-Recover: Virtual Machine Recovery When Live Migration Fails

Dinuni Fernando, Jonathan Terner, Ping Yang, Kartik Gopalan

Live migration is a critical technology used in cloud infrastructures to transfer running virtual machines (VMs). When live migration fails, as it often does, it is critical that any VMs in transit are not lost. There are two primary live migration techniques – pre-copy and post-copy. Pre-copy transfers a VM’s memory to the destination before its virtual CPUs are transferred, whereas post-copy does the reverse. Both pre-copy and post-copy will lose the VM if the source machine fails during migration. Additionally, post-copy can lose the VM if the destination machine or network fail since the VM’s memory and execution state are split across the source and destination machines. We present V-Recover, an approach to recover a VM when the source, destination, or network fails during live migration. V-Recover consists of two techniques: (1) a *forward incremental checkpointing* (FIC) mechanism to handle source machine failure during both pre-copy and post-copy, and (2) a *reverse incremental checkpointing* (RIC) mechanism to handle destination or network failure during post-copy. We present the design, implementation, and evaluation of V-Recover in the KVM/QEMU virtualization platform. Our evaluations show that V-Recover effectively recovers a VM upon migration failure with acceptable overheads on migration metrics and application performance.

Index Terms—virtual machine, live migration, fault tolerance

1 INTRODUCTION

Virtual machine (VM) migration transfers a VM from a source physical machine to a destination physical machine. In *live* VM migration, the VM keeps executing during migration for all but a short duration, known as downtime, which is usually when its CPU execution state is transferred. Live VM migration is a key technology used in cloud platforms for routine server maintenance, load balancing, scaling to meet performance demands, and consolidation to save energy. Most major hypervisors such as VMware [1], [2], KVM [3], Xen [4], [5], [6], and Hyper-V [7] support live VM migration. Live VM migration is also used by many cloud platforms, such as in Google’s production infrastructure, which performs over a million migrations each month [8].

Existing live VM migration techniques aim to migrate VMs as quickly as possible to the destination with minimal performance impact on applications running inside the VM. There are two predominant live VM migration techniques: pre-copy [1], [9] and post-copy [10], [11]. The two techniques differ in whether a VM’s memory pages are transferred before or after the transfer of its virtual CPUs (VCpus).

In the rest of this section, we first provide a brief overview of pre-copy and post-copy live migration, then describe the problem of failure resilience in live migration, followed by a summary of our contributions in this paper.

1.1 Overview of Pre-copy and Post-copy

Figure 1 shows the timeline of traditional pre-copy and post-copy live migration techniques. In pre-copy migration, the VM to be migrated initially continues execution at the source machine while its memory contents are concurrently transferred to the destination over multiple rounds (or iterations). In the first round, all memory pages are transferred, whereas subsequent rounds transfer only the pages modified (or dirtied) by the VM during the preceding round. The downtime begins when the number of dirty pages remaining becomes less than a predefined threshold. During downtime, the source machine pauses the VM’s VCPUs and transfers all the dirty pages and VCPU states to the destination. The VM is then resumed at the destination and migration completes.

Pre-copy works well for VMs that mostly read from memory, i.e. read-intensive VMs. However, for write-intensive VMs, pre-copy rounds may not converge quickly to downtime, if ever, because each round would have substantial amount of dirty memory pages to transfer from the preceding round. Thus, for write-intensive VMs, pre-copy migration experiences a long total migration time (due to

- Dinuni Fernando is with University of Colombo, School of Computing.
- Jonathan Terner is with IBM, USA.
- Ping Yang and Kartik Gopalan are with Binghamton University, USA.
- Contact emails: {pyang,kartik}@binghamton.edu
- Research conducted at Binghamton University, USA

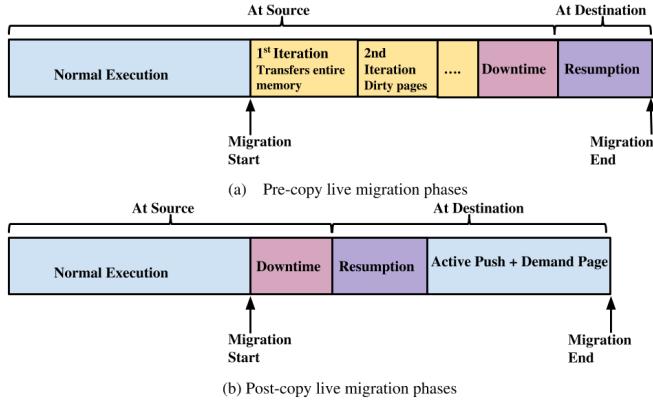


Fig. 1: Timeline of pre-copy and post-copy live migration.

many pre-copy rounds) and significant downtime (due to many dirty pages transferred during the downtime).

Post-copy migration solves the above deficiency of pre-copy in migrating write-intensive VMs. In post-copy, the VM is first suspended on the source machine and its VCPUs states are transferred to the destination. The VCPUs are immediately resumed at the destination even though the VM's memory pages have not yet been transferred. Concurrently, the memory pages are *actively pushed* from the source to the destination, with the expectation that most pages would reach the destination before they are accessed by the VM. If the VM accesses a page at the destination that has not yet been pushed from the source, then a page fault is triggered. The faulted page is then *demand paged* (explicitly requested) over the network from the source machine. Such remotely-serviced page faults may temporarily slow down the VM's progress at the destination until its full working set is transferred from the source. To reduce the number of page-faults, the source machine can prioritize pushing the VM's working set, such as pages around the location of the last page-fault, so that most pages can be sent before the VM faults on them.

Post-copy works equally well whether a VM's workload is read-intensive or write-intensive because each page is transferred over the network exactly once. In contrast, pre-copy transfers dirtied pages multiple times, causing increased network traffic and longer migration time. Google's data centers [8] use both techniques depending on whether the VM's workload is read-intensive or write-intensive.

1.2 Failure Resilience Problem of Live Migration

An important consideration in live VM migration, and the focus of this paper, is the robustness of the live migration mechanism. Specifically, the source, the destination, or the network can fail during the live migration. Since a VM encapsulates a cloud customer's critical workload, it is essential that the VM's execution state is preserved accurately and not lost due to failures during live migration.

In both techniques, the failure of the source machine during the migration could result in a permanent loss of the VM because some or all of the latest VM state resides at the source machine. Specifically, for pre-copy, the source contains the latest dirtied memory pages and execution state (VCPUs and I/O); for post-copy the source contains pages

that have not yet been transferred to the destination. A loss of the source in either case would leave the VM in an unrecoverable state.

However, pre-copy and post-copy differ in their resilience to failure of the destination machine or the network. It turns out that post-copy has worse failure resilience than pre-copy. For pre-copy, a failure of the destination machine or the network is not catastrophic, because the source machine still holds an up-to-date copy of the VM's memory and execution state and hence the VM is not lost. However, for post-copy, a destination or network failure is still catastrophic because the latest state of the VM is split between the source and the destination machines. The destination machine has a more up-to-date copy of the VM's execution state and some of its memory pages that have been transferred, whereas the source machine holds pages that are yet to be transferred. Thus, a destination or network failure during post-copy migration results in an irrecoverable loss of the VM.

To the best of our knowledge, the above failure scenarios involving a potential loss of the VM and its recovery has not been addressed by other researchers. The problem is important because a VM is particularly vulnerable during live migration. VM migration may last anywhere from a few seconds to several minutes, depending on a number of factors such as the VM's memory size, the applications running inside the VM, and other workload in the cluster. Thus the window of vulnerability is significant. In addition, because the VM is live during the migration, it might communicate over the network with remote entities, altering the external world's view of the VM's state. Hence, upon failure, one cannot simply revert the VM to an older snapshot that was saved before the migration began.

1.3 Contributions

We propose a technique called V-Recover to recover a VM upon the failure of the source, destination, or network during live migration. V-Recover has two components: (1) *forward incremental checkpointing* (FIC) to handle source machine failure, and (2) *reverse incremental checkpointing* (RIC) to handle destination or network failure.

The key idea behind FIC is as follows. Prior to migration, the source machine periodically saves forward incremental checkpoints of the VM to another machine (other than the source). Each forward checkpoint consists of the incremental state of the VM since the previous checkpoint. For pre-copy migration, the incremental checkpointing continues during the live pre-copy rounds. A final forward incremental checkpoint is performed for both pre-copy and post-copy just before downtime begins. If the source machine fails during live migration, then the lost VM state can be recovered by combining the VM states saved in the forward incremental checkpoint and the state already transferred to the destination.

RIC, on the other hand, can be used during post-copy to handle destination and network failures. Once a VM resumes execution at the destination, the destination transmits incremental checkpoints of the VM to an in-memory checkpoint store at another machine (either a third machine or the source itself). Reverse checkpointing proceeds concurrently,

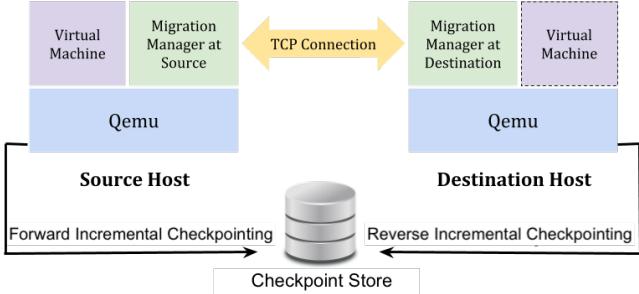


Fig. 2: The architecture of V-Recover.

and in coordination with, the forward post-copy migration from source to destination. These reverse checkpoints can be sent either periodically or upon external I/O activities of the VM (i.e., event-based checkpointing). If the destination or the network fails during the post-copy, then the source machine restores the VM from the last consistent reverse checkpoint received from the destination.

Note that the incremental checkpoints are much smaller than full-VM checkpoints because they consist of only the VM's modified memory pages since the last checkpoint, plus its VCPU and I/O states. We store the incremental checkpoints in an in-memory checkpoint store, instead of a disk, to reduce the time spent in transferring the checkpoints and to speed up the restoration process. For checkpoint consistency, packet transmissions from the VM to the external world are buffered between successive incremental checkpoints.

We implemented and evaluated a prototype of V-Recover in the KVM/QEMU virtualization platform. Our evaluations of the prototype using a variety of benchmarks show that V-Recover can effectively recover a VM upon live migration failure with acceptable overheads on live migration metrics and application performance.

A preliminary version of this paper [12] addressed destination and network failure during post-copy through reverse incremental checkpointing. This paper significantly extends our prior work by addressing source failures during both pre-copy and post-copy migration. We have also conducted extensive additional experiments to evaluate the effectiveness of V-Recover using various standard benchmarks such as STREAM, Sysbench and YCSB.

The rest of the paper is organized as follows. Sections 2 and 3 present the design and the implementation of V-Recover, respectively. Section 4 provides the evaluation results of V-Recover. Related work is discussed in Section 5 and Section 6 concludes the paper.

2 V-RECOVER DESIGN

V-Recover is designed to recover a VM when the source, the destination, or the network fails during live migration of the VM. Figures 2 and 3 provide V-Recover's architecture and operation timeline, respectively. As mentioned earlier, V-Recover consists of two main components: forward incremental checkpointing (FIC) and reverse incremental checkpointing (RIC). FIC can be used with both pre-copy and post-copy to handle source machine failure whereas RIC is specifically designed for post-copy to handle the failure

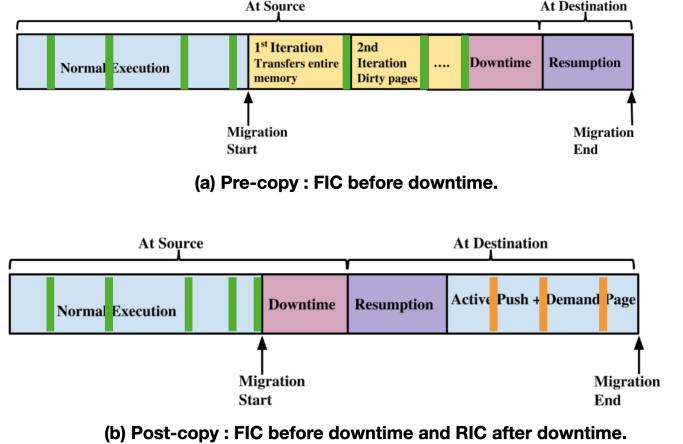


Fig. 3: Timeline of V-Recover in pre-copy and post-copy. Vertical bars in the timeline represent incremental checkpoints. FIC operates before downtime in both pre-copy and post-copy, whereas RIC operates in post-copy after downtime.

of the destination machine or the network. FIC operates prior to downtime in both pre-copy and post-copy, whereas RIC operates after downtime only in post-copy.

Upon source failure, for pre-copy, one must resort to non-live VM recovery using forward checkpoints. On the other hand, post-copy can perform live VM recovery upon source failure because the destination is executing the latest VM state and only the VM's missing memory pages need to be restored. Upon a destination or network failure, pre-copy migration is not affected as discussed earlier. However for post-copy, one must perform non-live VM recovery by loading the latest reverse checkpoints captured by RIC.

Our failure model in V-Recover is as follows. We assume that only one failure occurs before the VM is recovered, i.e. only one of the source machine, the destination machine, or the network fails. We assume that there are no additional failures during VM recovery. We further assume that the machine containing the incremental checkpoints (the checkpoint server) remains accessible over the network to the machine performing the recovery. Specifically, the destination machine should be able to access forward checkpoints to recover from source failures and the source machine should be able to access reverse checkpoints to recover from destination and network failures. Additionally, upon a network failure that prevents the source and destination from communicating, to prevent VM duplication by both the source and destination machines, we assume that the checkpoint server is placed such that it is accessible only by the source and not by the destination, so that only the source machine can perform the VM recovery. For instance, when migrating a VM across two racks, the above conditions can be met by placing the checkpoint server on the source rack, further assuming that there are no intra-rack network failures. We do not consider multiple failure scenarios in this paper even though some of them could be potentially handled by our techniques; for instance, the checkpoint server could potentially reconstruct a VM on its own in case both the source and destination machines fail together. Failures that occur before or after the live migration can be handled by

existing fault tolerance solutions [13] [14]. Storage failures can be mitigated using existing storage redundancy and recovery techniques [15]. Migration failures due to software bugs are not considered in this paper.

2.1 Forward Incremental Checkpointing (FIC)

FIC aims to recover a VM when the source machine fails during live migration. When a VM starts, FIC transfers an initial full VM checkpoint to a checkpoint store, which is an in-memory key-value store located at a third staging machine. Thereafter, during the VM’s normal execution at the source, FIC periodically saves partial incremental checkpoints of the VM’s modified memory pages to the checkpoint store. Just before downtime is to begin, the VM’s execution at the source is paused and FIC transfers a final and consistent incremental checkpoint of the VM to the checkpoint store.

Overhead Reduction: FIC periodically transfers incremental partial memory checkpoints during the normal execution of the VM with limits on the rate of memory transfer. The VM’s execution is not paused during partial checkpoints since only modified memory pages are transferred. Specifically, at every periodic time interval I (a few minutes or seconds) FIC transfers modified memory pages that haven’t yet been transferred to the checkpoint store. The number of pages transferred in each interval is bounded by either a maximum count N or a time limit T , whichever is reached first. All three parameters I , N , and T are configurable to reduce the overhead of FIC on the normal execution of VMs. These partial memory checkpoints together constitute progress towards a complete incremental checkpoint, when the VCPU execution state is also captured. To reduce the impact on application performance during VM recovery, the checkpoint store periodically merges the incremental checkpoints into a latest memory checkpoint.

Instead of performing incremental live checkpointing prior to migration, one could alternatively checkpoint the VM’s entire memory and execution state once just before live migration begins. If such a checkpointing was non-live, then it would introduce significant downtime. Additionally, network transfer to checkpoint store would compete for network bandwidth with live migration, thus prolonging the total migration time. Hence we chose a live FIC design that performs periodic partial incremental memory checkpointing to amortize the cost of FIC over the lifetime of the VM at the source.

Multiple VMs: In cloud environments, a physical machine often hosts multiple VMs, any of which can be migrated in advance. As a result, FIC must periodically checkpoint all VMs’ memory states to different checkpoint store instances. Partial memory checkpointing allows FIC to limit the rate at which memory pages are checkpointed and transferred periodically. It is also possible that FIC may be checkpointing one VM while another VM is being live migrated, leading to contention on the shared network interface. To reduce this contention, FIC monitors the available outgoing network bandwidth on the source machine prior to checkpointing and, if necessary, reduces the checkpointing frequency and the amount of memory checkpointed periodically.

Recovery from Source Failure: The source and destination machines use a heartbeat mechanism to monitor each other’s availability during live migration. When successive heartbeat messages are not acknowledged by the source, the destination concludes that either the source or the network has failed and the restoration manager at the destination triggers a VM recovery. If the checkpoint store is reachable from the destination, then the restoration manager loads any missing pages from the checkpoint store instead of the failed source machine.

The VM recovery process in post-copy is live, during which the VM is continuously running at the destination. Therefore, the recovery mechanism does not impose any significant additional downtime on post-copy, other than the time for switching page transfers from the source machine to the checkpoint store upon detecting source failure. On the other hand, pre-copy migration keeps the VM running at the source. Hence, if the source machine fails during pre-copy migration then the VM’s recovery must be performed non-live using the forward checkpoint saved in the checkpoint store.

2.2 Reverse Incremental Checkpointing (RIC)

RIC aims to recover a VM when the destination or the network fails during post-copy. (As mentioned earlier, these two failures during pre-copy can be trivially handled by continuing to run the VM at the source.) The first step of post-copy is to transfer the VCPU state of the VM to the destination. The VM is then resumed at the destination while concurrently receiving the VM’s memory pages from the source. RIC superimposes a reverse incremental checkpointing mechanism over this forward transfer of the VM state. Specifically, once a VM is resumed at the destination, RIC captures the VM’s initial execution state and modified memory at the destination and transfers them to a checkpoint store. Then onward, RIC saves any incremental changes in the VM’s state to the checkpoint store, including the execution state and any modified memory pages, either periodically or upon any external I/O activity by the VM. RIC stops once the migration succeeds.

To ensure the consistency of the reverse checkpoints, RIC buffers packet transmissions from the VM 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, after which any packets in the network buffer are transmitted and the VM is resumed. This ensures that the external world’s view of the VM does not change before the corresponding checkpoint is committed to the checkpoint store. Thus, if the destination or network fails during the migration, RIC guarantees that the latest committed checkpoint reflects a consistent state of the VM.

Overhead Reduction: To minimize impact on normal post-copy migration, RIC executes concurrently with the VM. The only time RIC affects the VM’s execution is when the VCUs are suspended briefly to capture the VM’s execution state. The active-push phase of post-copy from the source to the destination runs concurrently with the RIC mechanism even when the VCUs are paused. This

helps RIC to achieve similar total migration time as post-copy. Periodic checkpointing may also impact the performance of write-intensive VMs whose pages are dirtied often. To reduce this impact, RIC performs VM checkpointing in two stages. In Stage 1, RIC checkpoints only the modified memory pages of the VM, but not its execution state (i.e., the VM’s CPU and device states). The modified memory pages are checkpointed without pausing the VM to avoid interrupting the VM’s workloads. In Stage 2, the VM is paused briefly to capture the VM’s execution state, after which 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 version checkpointed in Stage 1 is overwritten by that in Stage 2 to ensure that the checkpoint contains the most up-to-date page. Checkpointing in two stages significantly reduces the performance impact compared to if the VM was fully paused during memory capture.

Recovery from Destination or Network Failure During Post-Copy: The source and destination machines use heartbeat messages to monitor the liveness and reachability of each other. When successive heartbeat messages are not acknowledged by the destination, the source concludes that the migration has failed, either due to a destination failure or a network partition, and the restoration manager at the source machine triggers a VM recovery. The source machine then recovers the VM by restoring the last consistent reverse checkpoint of each memory page from the checkpoint store onto the VM’s memory address space at the source. Pages not modified by the destination do not need to be overwritten. Finally, the VM is resumed at the source from the latest checkpointed execution state to complete the VM’s recovery.

3 IMPLEMENTATION DETAILS

We have implemented V-Recover in the KVM/QEMU [3], [16] virtualization platform. Each VM is associated with a userspace management process, called QEMU, which performs device emulation and various management functions, including live migration and checkpointing. A kernel module, called KVM, uses hardware virtualization features and coordinates with QEMU to execute the VM in guest mode (or non-root mode). We modify pre-copy and post-copy migration code in QEMU (about 1500 lines of new code) to implement both FIC and RIC. The guest OS and applications inside the VM are unmodified in our implementation.

3.1 FIC Implementation

FIC is implemented as a separate thread in QEMU on the source machine and executes concurrently with the normal execution of the VM prior to the start of downtime.

Dirty Page Tracking: FIC utilizes the dirty page tracking mechanism in KVM/QEMU to identify modified memory pages of a VM for forward checkpointing. The dirty page tracking mechanism represents the VM’s memory content as a bitmap, in which each bit specifies whether a guest page has been modified or not since the last check. FIC uses a separate bitmap, called *ft_bitmap*, to identify the VM’s memory pages modified during each checkpointing round. During a VM’s normal execution, FIC makes an *ioctl()* call to ask KVM to start dirty page tracking. In each forward

checkpointing cycle, FIC makes another *ioctl()* call to synchronize the *ft_bitmap* with the KVM’s bitmap to ensure that *ft_bitmap* reflects the latest VM state. FIC then captures the modified memory pages by reading *ft_bitmap* in QEMU and transfers the memory pages to the checkpoint store. Just before downtime begins, FIC uses the *ft_bitmap* to capture a final incremental checkpoint at the source.

Computing Available Bandwidth: As mentioned earlier, FIC controls the checkpoint transfer rate based on the available bandwidth. In FIC, the network usage of the source machine is measured by capturing the total data packets received and transmitted over the Ethernet interface using the *ifconfig* utility, which provides statistics about the network interface. The bandwidth monitoring module on the source machine uses message queues to send the available network bandwidth to the checkpoint transfer thread running in QEMU. The checkpoint transfer thread then estimates the maximum number of pages to transfer based on bandwidth availability.

Source Failure Detection and Live Recovery: The destination machine for a VM may be unknown until its migration is required. To reduce the recovery time, the checkpoint staging machine periodically pre-loads and merges incremental checkpoints from the checkpoint store to build up the latest memory state. Source failure is detected by the destination machine using heartbeat messages, as with destination failure described earlier. Upon source failure, the QEMU at the destination machine communicates with the checkpoint store to identify pages that have not yet been transferred to the destination by the failed source machine. The checkpoint store then concurrently transfers these missing pages to the destination, even as the VM continues to execute at the destination.

3.2 RIC Implementation

Like FIC at the source machine, RIC is implemented as a thread in QEMU on the destination machine. This thread executes concurrently with post-copy live migration and keeps track of all modified memory pages and execution states of the VM on the destination machine. Any pages modified by the VM during post-copy migration are transferred to the checkpoint store on another machine.

Capturing Modified Memory and Execution States: To begin, RIC at the destination inserts a network barrier to buffer outgoing network packets from the VM between successive incremental checkpoints. The checkpointing thread then periodically sends the incremental memory state of the VM to the checkpoint store. Unlike in FIC, where dirty page tracking is performed at the source machine to track modified memory pages, RIC performs dirty page tracking at the destination machine. We modified the default post-copy implementation in QEMU to perform dirty page tracking at the destination. Once the VM resumes at the destination during post-copy, the RIC thread in QEMU makes an *ioctl()* call to request KVM to start dirty page tracking. To identify any modified VM pages during each checkpointing cycle, RIC uses another *ioctl()* call to retrieve the latest dirty page bitmap from the KVM kernel module and updates another bitmap maintained by QEMU in user space. RIC then transfers the modified memory pages identified by the QEMU bitmap to the checkpoint store.

The execution state of a VM consists of its VCPU and I/O device states, which keep changing during the VM’s execution at the destination. At the end of each checkpointing cycle, RIC captures the execution state of the VM and writes to a channel buffer (a QEMU facility to perform buffered I/O operations) which then transfers the execution state to the checkpoint store on the staging machine.

Reducing Performance Impact of RIC: We have also implemented an *event-based reverse checkpointing* mechanism to reduce network packet buffering latency. The event-based approach checkpoints the VM’s state when either (a) an external event is triggered, such as an outgoing network packet transmission or other I/O from the VM, which might alter the external world’s view of the VM, or (b) when a significant amount of memory has been dirtied by the VM. As a result, our event-based approach pauses the VM for checkpointing only when necessary, as opposed to periodic checkpointing, and hence reduces the impact on VM’s performance.

The overhead of traversing the dirty bitmap and transferring each modified page to the remote checkpoint store can potentially affect the VM’s performance during migration. To reduce this performance impact, instead of sending checkpoints directly to the checkpoint store, the checkpoint is first stored in an in-memory local store called *checkpoint_stage* at the destination and then transferred to the external checkpoint store. This local *checkpoint_stage* is similar to Linux Kernel cache-slab [17], [18] and 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. First storing checkpointed state locally reduces the performance impact on the VM caused by synchronous network transmissions and provides the assurance of completeness in checkpoint. Since this local store contains the complete VM state, the VM can resume while RIC concurrently transfers the checkpointed state to the remote store and then releases any buffered network packets.

Destination Failure Detection and VM Recovery: The heartbeat module is implemented as a separate thread on both the source and destination machines to continuously monitor the availability of the other machine by sending periodic network packets to each other. If the heartbeat module does not receive a response for a specific timeout interval, then a VM recovery is triggered to recover the VM from the latest available consistent state.

Once a destination or network failure is detected in post-copy, the restoration process on the source machine initiates the VM’s recovery from the checkpoint store. The restoration mechanism is non-live by nature because the VM was running on the destination machine when the migration failed. The restoration process loads the incremental checkpoints to rebuild the VM’s consistent memory image, which is then memory mapped into QEMU’s address space. The restoration process finally loads the most recent VCPU state and resumes the VM.

3.3 Incremental Checkpoint Store

We consider several factors when selecting an external checkpoint store for V-Recover. In order to quickly and con-

sistently store incremental checkpoints, the checkpoint store should be an in-memory storage, provide duplicate filtering, and allow for checkpoint versioning. Each checkpoint also needs to be stored along with its version that represents the most recently committed checkpoint. That way we can discard incomplete checkpoints if a failure occurs in the middle of a checkpoint. The checkpoint store was implemented using the Redis [19] in-memory key-value store. The Redis clients reside on the source and destination machines while the Redis server resides on a checkpoint server which is neither the source nor the destination machine. The memory state of the checkpoint is stored in Redis as a key-value pair in the *map* data structure where the offset and the address of a page are used as a key to uniquely identify the page. Each complete checkpoint per cycle is separated with a version number to denote the checkpointing round.

As V-Recover may checkpoint multiple VMs running at the source and destination machines, we need to be able to distinguish the checkpoints for different VMs. To do so, we use a separate checkpoint store instance to maintain the memory state of each VM. The checkpoint store instances are created in advance. When a VM starts, V-Recover selects an available checkpoint store instance from the instance pool and updates the availability status of the checkpoint store to “unavailable.” When a VM terminates or completes the migration, the corresponding checkpoint instance is cleared and returned to the instance pool and the availability status is updated to “free.” V-Recover also transfers memory pages to Redis store in batches, rather than one at a time, to reduce synchronization overhead on write requests.

4 EVALUATION

In this section, we show that V-Recover can recover a VM from failures during live migration with acceptable performance overheads. We focus on the following metrics.

- *Total migration time:* Time taken to transfer a VM’s state entirely from the source machine to the destination machine.
- *Downtime:* Duration that a VM is not executing during the live migration.
- *Replication time:* Time taken to transfer the checkpoint to a checkpoint store.
- *Application performance:* Performance of applications running inside the VM during live migration.
- *Network bandwidth:* Network bandwidth during migration and checkpointing.
- *Recovery time:* The time taken to restore the VM from the last committed checkpoint after failure.

Our evaluation environment consists of dual six-core 2.1 GHz Intel Xeon machines with 128GB memory connected through a Gigabit Ethernet switch with 1Gbps full-duplex ports. To avoid network interference, separate network interfaces are used for VM-generated traffic and management traffic generated by live migration and checkpointing. VMs are configured with one VCPU unless specified otherwise. Virtual disks are accessed by VMs over a local area network from an NFS server. Due to space constraints, we use post-copy to evaluate both FIC and RIC. Each data point reported is an average over five runs of each experiment.

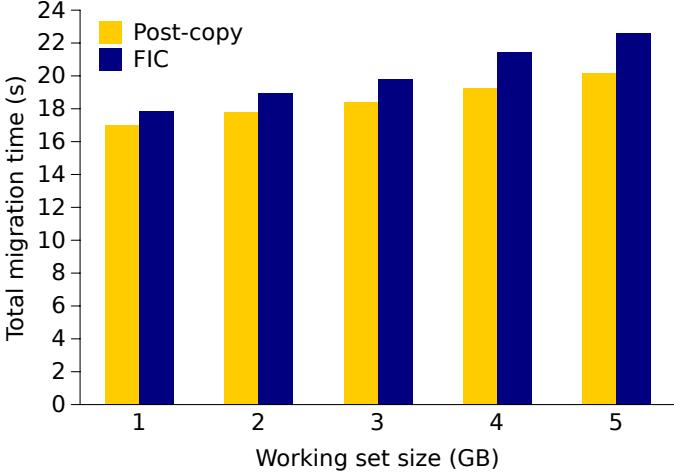


Fig. 4: Total migration time of write-intensive VM with FIC.

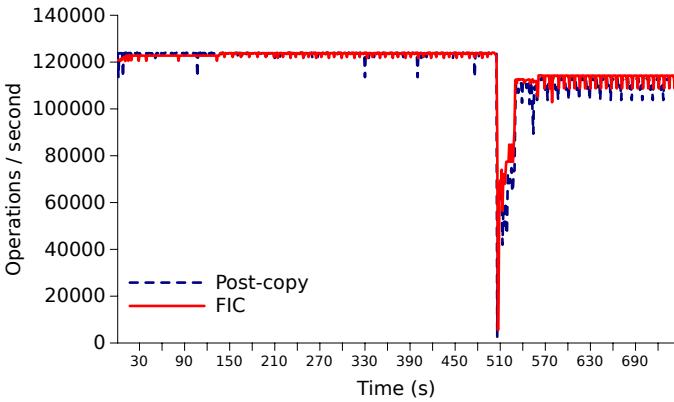


Fig. 5: The impact of FIC on the CPU intensive workload.

4.1 Performance of FIC

4.1.1 Live Migration Performance

We first evaluate the overhead of FIC on the source machine without triggering a migration failure. Figure 4 compares the total migration time of a write-intensive VM using post-copy with FIC versus vanilla post-copy. The figure shows that FIC imposes a very small overhead of 1%–4% on total migration time for write-intensive VMs. The corresponding overhead for migrating idle VMs (not shown) is only 0.2%–0.7%. The downtime with FIC is between 7.5ms to 11.7ms, which is only slightly higher than vanilla post-copy (7ms–11.6ms).

4.1.2 Impact on CPU-intensive Workload

We measured how FIC affects the performance of CPU-intensive applications running inside the VM using a Quicksort benchmark, which is a CPU-intensive application. The Quicksort benchmark repeatedly allocates 400MB of memory, writes random integers to the allocated memory segment, and sorts the integers using the Quicksort algorithm. Figure 5 shows that the number of sorting operations performed per second during migration is similar for both FIC and vanilla post-copy. At downtime there is a sharp but similar reduction in Quicksort performance for both FIC and post-copy. This shows that FIC does not have an observable impact on CPU-intensive workloads.

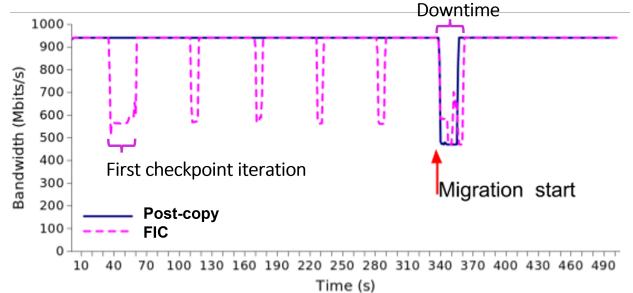


Fig. 6: Impact of FIC on iPerf bandwidth during migration.

4.1.3 Impact on Network-intensive Workload

To measure the impact of FIC on network-intensive VM workloads we used iPerf [20], a network-intensive benchmark, to measure the network throughput from the VM before and during the migration. The iPerf server runs on an external machine (i.e., neither source nor destination) in the same cluster and the iPerf client resides inside the VM being migrated. The iPerf client continuously sends data to the server through a TCP connection. The measured network bandwidth is reported by iPerf every second. Figures 6 shows the measured network bandwidth of iPerf client before, during, and after the migration, when migrating a 1GB VM. The checkpoint interval is set to 50 seconds for better visualization of the bandwidth fluctuations. The figure shows that, during the first checkpointing round of FIC, when the entire memory state is transferred to the destination, the bandwidth reported by iPerf client drops from 940Mbps to 580Mbps for about 20 seconds. Subsequent incremental checkpoints are shorter in duration and consequently the bandwidth drops are also shorter. In addition, just before the migration starts, the final checkpoint is transferred in parallel to the live migration, which leads to contention on the outgoing network link on the source machine. Therefore, we see a slightly longer network bandwidth drop in FIC than vanilla post-copy. We also measured the iPerf bandwidth for larger VMs on the source machine. The results are similar except that when the VM size increases, the duration of the network bandwidth drop in the first checkpointing round also increases.

4.1.4 Impact on Concurrent Migrations

Earlier, we discussed the possibility of FIC for one VM impacting the migration of another VM due to network contention. To address this issue, FIC dynamically adjusts its checkpointing speed based on the available bandwidth to minimize impact on other migrations. Figure 7 shows that the time taken to migrate an idle VM using post-copy during which another co-located VM is checkpointed using FIC. The checkpoint interval is 50 seconds, and the size of the VM ranges from 1GB to 8GB. The figure shows that FIC imposes 0.3%–1% overhead on the total migration time. For write-intensive concurrent VMs, FIC similarly imposes an overhead of 0.4%–1.9% on total migration time.

4.1.5 Recovering From Source Failure in post-copy

We now consider the scenario of recovering a VM when the source machine fails during post-copy migration. In this

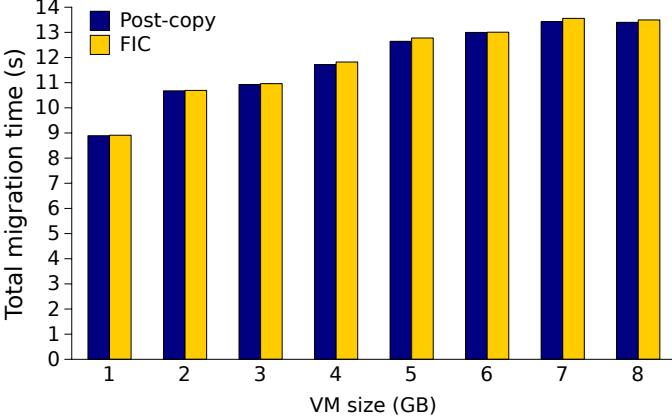


Fig. 7: FIC impact on concurrent migration of another VM.

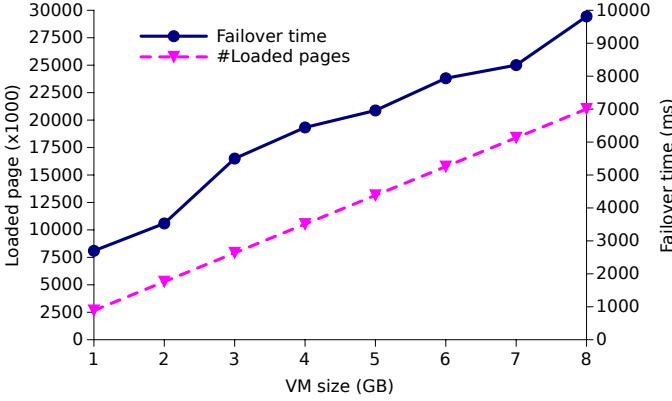


Fig. 8: Recovering a VM after source failure with FIC .

scenario, since the VM’s latest memory state is saved over multiple incremental checkpoints by FIC, the final VM’s memory state needs to be merged from all the incremental checkpoints. Figure 8 plots the recovery time of FIC and the number of loaded memory pages for migrating an idle VM. As expected, as the VM size increases, it takes longer time to transfer and merge the incremental checkpoints and hence the time taken to recover the VM also increases. The recovery time varies from around 3 seconds for a 1GB VM to around 10 seconds for a 8GB VM. Note that this VM recovery is live for source failures because the VM keeps executing at the destination node during recovery. To further hide the cost of recovery, one could also perform the merge ahead of time before a failure occurs, though at the expense of additional computation.

4.2 Performance of RIC

4.2.1 Live Migration Performance

Figure 9 compares the total migration time of a write-intensive VM using post-copy with RIC versus vanilla post-copy. The write-intensive VM executes a program that continuously writes random numbers to a large region of main memory. The working set size (i.e., size of the memory written) is varied from 1GB to 5GB. The figure shows that the total migration time of post-copy with and without RIC are almost the same. This is because in post-copy, the source machine actively pushes pages to the destination even when the VCPUs are paused at the destination (due to demand-paging or RIC), thus allowing these operations

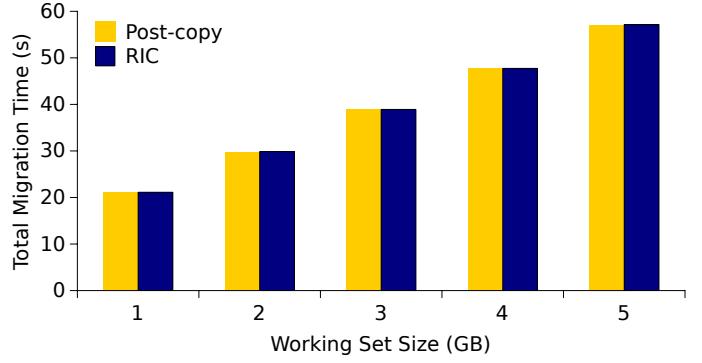


Fig. 9: Impact of RIC on total migration time when migrating a write-intensive VM.

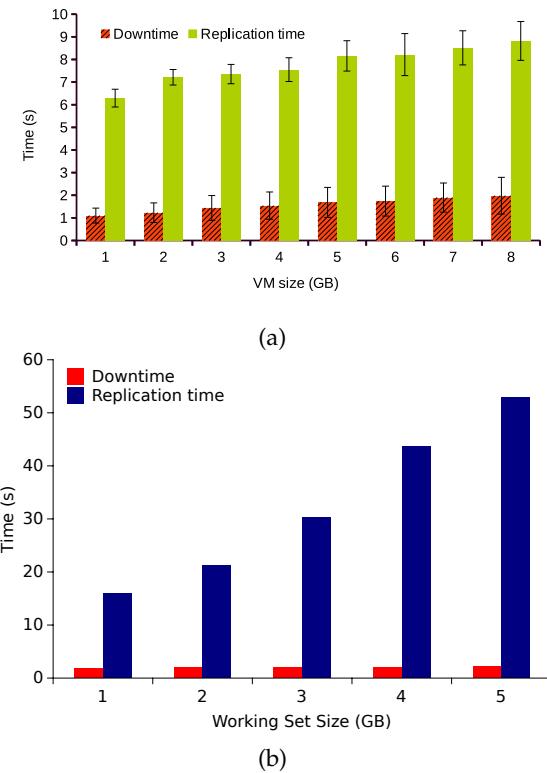


Fig. 10: Cumulative downtime and replication time of RIC for migrating (a) idle VM and (b) write-intensive VM.

to complete concurrently. Similar results are observed for migrating an idle VM and hence not shown.

Figure 10(a) shows the cumulative downtime and replication time of migrating idle VMs using RIC. The cumulative downtime is the sum of all the times that the VM is paused by RIC during migration. We purposely chose an aggressive checkpointing interval of $100\mu s$ to stress test RIC. The cumulative downtime with RIC ranges between 1.1s and 1.9s. For longer checkpointing intervals, the cumulative downtime will correspondingly reduce. As discussed earlier, RIC is split into two stages and only Stage 2 requires pausing the VM. Vanilla post-copy pauses the VM only once at the start of migration and hence it has a downtime of only 9ms–11.6ms. The figure also shows that the replication time increases with increasing VM size. Figure 10 (b) shows the cumulative downtime and replication time of RIC for migrating a write-intensive VM. The cumulative downtime

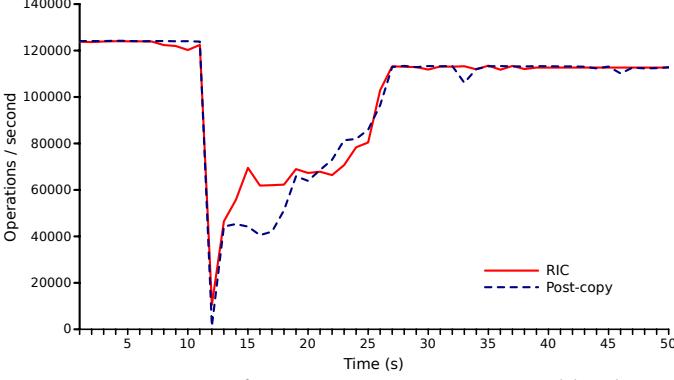


Fig. 11: Impact of RIC on CPU-intensive workload.

remains fairly stable between 1.8s and 2.2s. The figure also shows that the replication time increase when the working set size increases due to an increase in the number of dirty pages that need to be checkpointed by RIC.

4.2.2 Impact on CPU-intensive Workload

We measured how RIC affects the performance of applications running inside the VM using a Quicksort benchmark (described in Section 4.1.2). Figure 11 shows that the number of sorts performed per second is constant in both RIC and post-copy except that there is a sudden reduction in the performance at downtime. RIC and post-copy also have similar performance during the migration, which means that the reverse incremental checkpointing does not impose observable overhead on the application performance.

4.2.3 Checkpointing Frequency

Figure 12(a) shows the total migration time of migrating a VM running the STREAM [21] benchmark when the checkpointing interval is varied between 0.1ms and 100ms. Checkpointing interval 0 refers to the vanilla post-copy migration. STREAM is an industry standard for measuring the sustained memory bandwidth and the corresponding computation rate for simple vector kernels. STREAM allocates 1.5GB of array elements, runs 50 iterations on each vector kernel, and continuously executes during the VM migration. The figure shows that RIC does not incur any overhead compared to the post-copy. Figure 12(b) shows that when the checkpointing interval increases, the downtime decreases. As checkpointing is performed less frequently the overhead of bitmap synchronization, state transfer, network buffering, and pausing the VCPUs also decrease. The figure also shows that, when the checkpoint interval increases, more pages are dirtied during the interval. We also measured the impact of varying the checkpointing interval on the total migration time and downtime of migrating idle VM and write-intensive VM. The results are similar to the above. Finally, we measured the impact of the checkpointing interval on the performance of applications running inside the VM using the Sysbench [22] CPU-intensive workload during VM migration. Sysbench reports the time taken to find the 20,000th prime number. Our experimental results show that the checkpointing interval has little to no effect on the execution time of Sysbench (about 45s for all intervals).

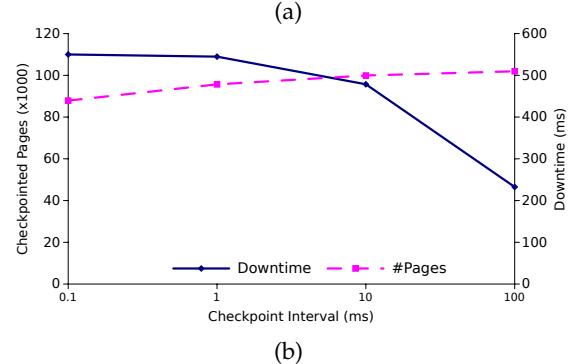
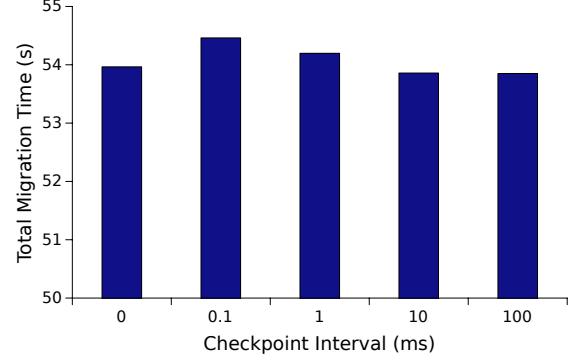


Fig. 12: The impact of checkpointing interval on (a) total migration time and (b) downtime and checkpointed page count for RIC with the STREAM benchmark.

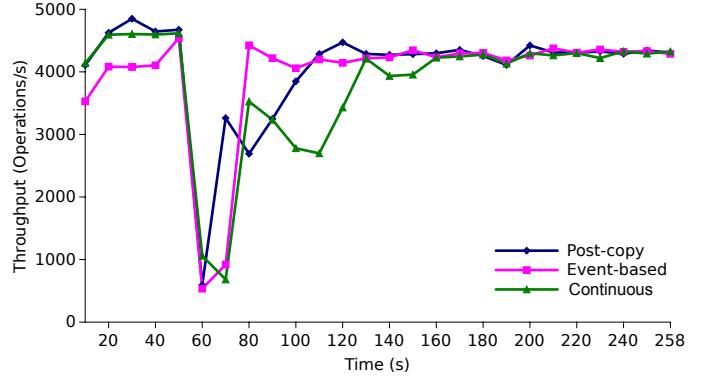


Fig. 13: YCSB throughput variation during migration.

4.2.4 Packet Buffering and Release

We now consider the impact of packet buffering and release on application throughput and latency when using RIC. Using Redis [19] and Yahoo Cloud Serving Benchmark (YCSB) [23], we generated outgoing and incoming network packets as query requests and responses. Redis is a key value in-memory database that runs on an external machine. YCSB is a database benchmark client that resides in the migrating VM and interacts with Redis. Redis first loads its database. Then YCSB client queries 1GB of data using update operations (read/update ratio is 50/50) while the VM is being live migrated. Figure 13 and Figure 14 show throughput and latency variation, respectively. Both figures show that, during the downtime, there is a sudden drop in the throughput and increase in latency. However, event-based checkpointing has more consistent throughput and lower latency degradation than continuous checkpointing. This is because, in event-based checkpointing, requests are

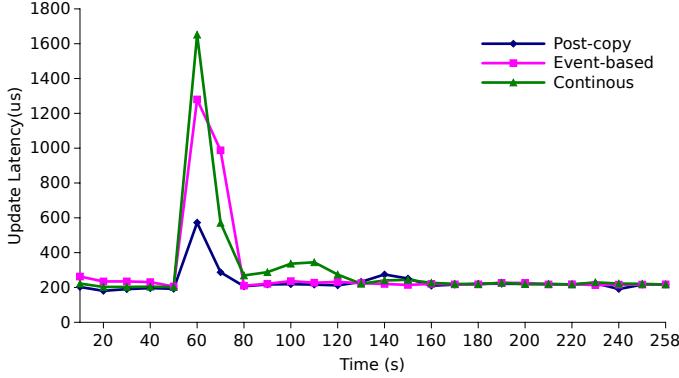


Fig. 14: YCSB latency variation during migration.

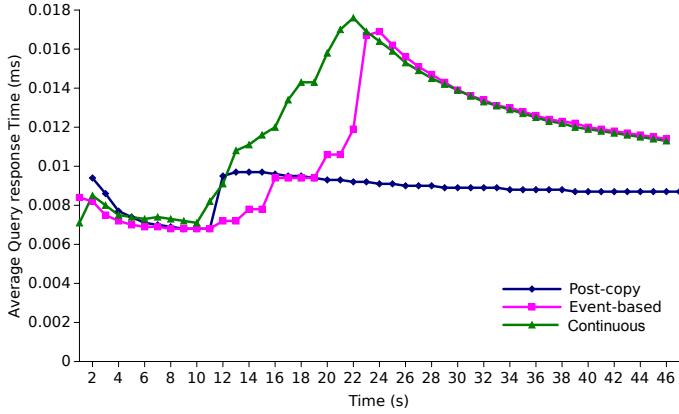


Fig. 15: Impact on Sysbench OLTP response time of post-copy with continuous and event-based RIC.

not buffered unless an event occurs, while in continuous checkpointing, requests are periodically buffered every checkpoint interval.

We also measured packet buffering and release overhead when running Sysbench Online Transaction Processing (OLTP) benchmarking tool. We ran a MYSQL server inside the VM and queried the database using a Sysbench client from an external host. Figure 15 shows the average query response time over 50 seconds during live migration. In this setting, query requests are incoming network requests as seen by the VM. Even though we do not buffer incoming network packets, query responses are treated as outgoing packets with respect to the VM. The figure shows that, during live migration, event-based checkpointing has lower average response time compared to continuous checkpointing. Once the migration completes, both response times gradually converge to that of vanilla post-copy.

4.2.5 Network Overhead and Recovery Time

Next, we evaluate the network overhead of RIC. Figure 16 shows total checkpointed page count when migrating a write-intensive VM with working set size of 5GB when the checkpointing interval is varied from 0.1ms to 100ms. The checkpointed page count reduces when the checkpointing interval increases. Since the same page may be dirtied multiple times, longer checkpointing interval reduces the number of times that a dirtied page needs to be checkpointed.

Next we varied the number of checkpointed pages with RIC and captured the time taken to restore the VM on the source when migrating a write-intensive VM. As expected, Figure 17 shows that the recovery time with RIC increases

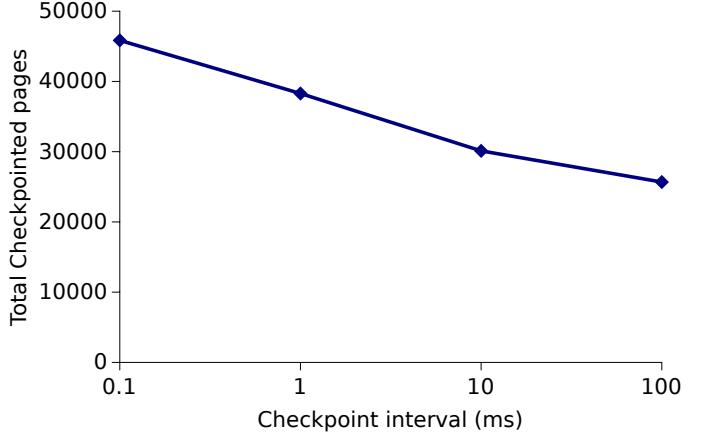


Fig. 16: Checkpointed pages vs. checkpoint interval in RIC .

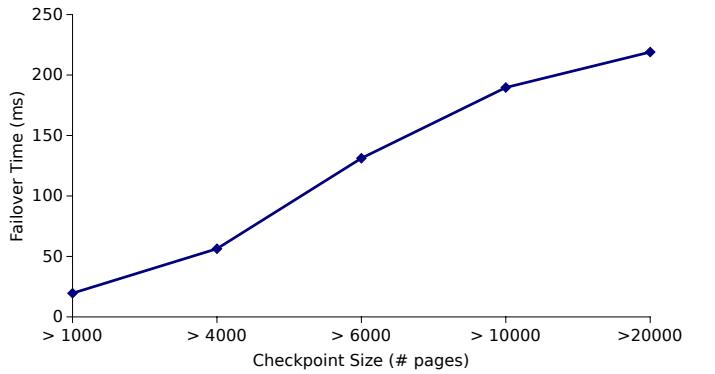


Fig. 17: Recovery time vs. checkpoint size with RIC.

when the checkpoint size increases. Even with 20,000 checkpointed pages, the recovery time is only around 200ms.

5 RELATED WORK

To the best of our knowledge, V-Recover is the first approach to address recovery from failures during live migration. Live migration mechanisms in Google’s datacenters [8] compare memory checksums of a migrating VM at the source and destination machines to detect memory corruption during migration. The VM is discarded if the checksums do not match. QEMU supports a feature called PostcopyRecovery [24] that allows an ongoing post-copy migration to be resumed once a broken network connection between the source and destination is restored. However it does not support recovering the VM when the source or the destination fail, or the network connection cannot be reestablished.

As mentioned earlier, our conference paper in IEEE Infocom 2019 [12] addressed the problem of destination and network failure during post-copy. This paper significantly extends the conference version by addressing source failures via FIC as well as conducting extensive additional evaluations using several benchmarks.

All major virtualization platforms [3], [4], [25] support VM checkpointing and restoration. Checkpointing can be performed at either the application-level [26], [27] or VM-level [13], [14], [28], [29], [30], [31], [32], [33], [34]. Compared to application specific checkpointing schemes, whole system checkpoints provide a cleaner and more complete encapsulation of application state. Checkpointing-based VM fault tolerance can be performed using active-passive and active-active approaches. In active-passive approaches [13], [35],

[36], a VM’s state is regularly checkpointed on a backup machine’s memory or disk. When the primary VM or machine fails then the VM image on the backup machine is activated. In active-active replication [14], [28], [37], [38], [39], [40], [41], [42], [43], [44], the primary and backup VMs execute in consistent lockstep. Record/replay mechanisms can record external events at a primary VM and replay those events on a backup VM to ensure that a consistent replica is always available. Event recording can be performed by the hypervisor but deterministic replay can be difficult on multi-core CPUs. In contrast to the above efforts which guard against failures during a VM’s normal execution, V-Recover uses incremental checkpointing to guard against failures during live migration.

Live migration itself can also be used as a fault-tolerance technique. One can quickly migrate a VM upon imminent failure of a source machine [45], [46], [47]. One can also use process migration [48], [49], [50], [51] for fault tolerance, though at the risk of leaving residual dependencies in the source machine. Some techniques [52], [53], [54] perform fast intra-host live migration to handle hypervisor updates/failures by co-mapping a VM’s memory to a new co-located VM. However the above techniques do not address the failure of live migration mechanism itself. Several techniques [55], [56], [57], [58], [59] estimate the required bandwidth for live migration based on the system’s resource usage. V-Recover also adapts to available bandwidth to avoid network contention with colocated VMs.

6 CONCLUSION

In this paper, we presented V-Recover, a mechanism to seamlessly recover a virtual machine when live migration mechanism fails due to source, destination, or network failure. V-Recover incorporates a forward incremental checkpointing mechanism to recover from source machine failures and a reverse incremental checkpointing mechanism to recover from destination machine or network failures. We described the design, implementation, and evaluation of V-Recover on the KVM/QEMU platform. Our evaluations show that V-Recover can effectively recover a VM upon migration failure with acceptable overheads on live migration metrics and performance.

7 ACKNOWLEDGEMENTS

This work is supported in part by National Science Foundation, USA, via awards 1320689, 1527338, and 1738929.

REFERENCES

- [1] M. Nelson, B. H. Lim, and G. Hutchins, “Fast transparent migration for virtual machines,” in *Proc. of USENIX Annual Technical Conference (ATC)*, April 2005.
- [2] VMWare Inc., “vSphere vMotion.” [Online]. Available: https://www.vmware.com/pdf/vmotion_datasheet.pdf
- [3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “KVM: The Linux virtual machine monitor,” in *Proc. of Linux Symposium*, June 2007.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, Oct. 2003.
- [5] 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 *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
- [6] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.
- [7] Microsoft Corporation, “Hyper-V Architecture.” [Online]. Available: <https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture>
- [8] A. Ruprecht, D. Jones, D. Shirav, G. Harmon, M. Spivak, M. Krebs, M. Baker-Harvey, and T. Sanderson, “VM Live Migration At Scale,” in *Proc. of ACM International Conference on Virtual Execution Environments (VEE)*, 2018, pp. 45–56.
- [9] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live Migration of Virtual Machines,” in *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.
- [10] M. Hines, U. Deshpande, and K. Gopalan, “Post-copy live migration of virtual machines,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 9, pp. 14–26, July 2009.
- [11] M. R. Hines and K. Gopalan, “Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning,” in *Proc. of ACM International Conference on Virtual Execution Environments (VEE)*, 2009.
- [12] D. Fernando, J. Terner, K. Gopalan, and P. Yang, “Live migration ate my VM: Recovering a virtual machine after failure of post-copy live migration,” in *Proc. of IEEE International Conference on Computer Communications (INFOCOM)*, 2019.
- [13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, “Remus: High availability via asynchronous virtual machine replication,” in *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008, pp. 161–174.
- [14] 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 *Proc. of ACM Symposium on Cloud Computing (SOCC)*, 2013.
- [15] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (RAID),” *ACM SIGMOD Record*, vol. 17, no. 3, pp. 109–116, Jun. 1988.
- [16] F. Bellard., “QEMU, a fast and portable dynamic translator,” in *Proc. of USENIX Annual Technical Conference (ATC)*, April 2005.
- [17] J. Mauro and R. McDougall, *Solaris internals: core kernel components*. Sun Microsystems Press, A Prentice Hall Title, 2001.
- [18] J. Bonwick and J. Adams, “Magazines and Vmem: Extending the slab allocator to many CPUs and arbitrary resources,” in *Proc. of USENIX Annual Technical Conference (ATC)*, 2001.
- [19] Redis Ltd., “Introduction to Redis.” [Online]. Available: <https://redis.io/docs/about/>
- [20] Iperf. [Online]. Available: <https://iperf.fr>
- [21] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.
- [22] Sysbench. [Online]. Available: <https://en.wikipedia.org/wiki/Sysbench>
- [23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. of ACM Symposium on Cloud Computing (SOCC)*, 2010.
- [24] QEMU Wiki, “PostcopyRecovery.” [Online]. Available: <https://wiki.qemu.org/Features/PostcopyRecovery>
- [25] C. Chaubal, “Architecture of VMware ESXi,” *VMWare White Paper*, Oct. 2008. [Online]. Available: <https://www.vmware.com/techpapers/2007/architecture-of-vmware-esxi-1009.html>
- [26] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, “Proactive process-level live migration in HPC environments,” in *Proc. of the ACM/IEEE Conference on Supercomputing (SC)*, 2008.
- [27] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghill, “Optimizing checkpoint sizes in the C3 system,” in *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.
- [28] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, “Proactive fault tolerance for HPC with Xen virtualization,” in *Proc. of the ACM International Conference on Supercomputing (ICS)*, 2007.
- [29] R. Jhawar, V. Piuri, and M. Santambrogio, “Fault tolerance management in cloud computing: A system-level perspective,” *IEEE Systems Journal*, vol. 7, no. 2, pp. 288–297, June 2013.
- [30] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, “Proactive fault tolerance using preemptive migration,” in *Euromicro*

- International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2009.
- [31] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *Proc. of ACM International Conference on Virtual Execution Environments (VEE)*, 2011.
 - [32] M. Lu and T. Chiueh, "Fast memory state synchronization for virtualization-based fault tolerance," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2009.
 - [33] A. Kangarloo, P. Eugster, and D. Xu, "VNsnap: Taking snapshots of virtual networked infrastructures in the cloud," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 484–496, 2012.
 - [34] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. of VLDB Endowment*, vol. 1, no. 1, Aug. 2008.
 - [35] M. Hines, "Micro checkpointing." [Online]. Available: <https://wiki.qemu.org/Features/MicroCheckpointing>
 - [36] C. Wang, X. Chen, W. Jia, B. Li, H. Qiu, S. Zhao, and H. Cui, "PLOVER: Fast, multi-core scalable virtual machine fault-tolerance," in *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
 - [37] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 30–39, 2010.
 - [38] 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 *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2017.
 - [39] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual machine synchronization for fault tolerance," in *Proc. of USENIX Annual Technical Conference (ATC)*, 2008, (Poster Session).
 - [40] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 1995.
 - [41] 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," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 211–224, Dec. 2002.
 - [42] G. W. Dunlap, "Execution replay for intrusion analysis," Ph.D. dissertation, University of Michigan, USA, 2006.
 - [43] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proc. of USENIX Annual Technical Conference (ATC)*, 2005.
 - [44] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proc. of International Symposium on Computer Architecture (ISCA)*, 2003.
 - [45] U. Deshpande, Y. You, D. Chan, N. Bila, and K. Gopalan, "Fast server deprovisioning through scatter-gather live migration of virtual machines," in *Proc. of IEEE International Conference on Cloud Computing*, 2014.
 - [46] U. Deshpande, D. Chan, S. Chan, K. Gopalan, and N. Bila, "Scatter-gather live migration of virtual machines," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 196–208, 2018.
 - [47] 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 *Proc. of International Conference on Utility and Cloud Computing (UCC)*, 2016, pp. 99–107.
 - [48] A. Barak and R. Wheeler, "MOSIX: An integrated UNIX for multiprocessor workstations," *International Computer Science Institute Technical Report TR 88-004*, Oct. 1988.
 - [49] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite network operating system," *Computer*, vol. 21, no. 2, pp. 23–36, Feb. 1988.
 - [50] R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network operating system kernel," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 1981.
 - [51] S. Mullender, G. van Rossum, A. Tananbaum, R. van Renesse, and H. van Staveren, "Amoeba: A distributed operating system for the 1990s," *Computer*, vol. 23, no. 5, pp. 44–53, May 1990.
 - [52] S. Doddamani, P. Sinha, H. Lu, T.-H. K. Cheng, H. H. Bagdi, and K. Gopalan, "Fast and live hypervisor replacement," in *Proc. of ACM International Conference on Virtual Execution Environments (VEE)*, 2019.
 - [53] H. Bagdi, R. Kugve, and K. Gopalan, "Hyperfresh: Live refresh of hypervisors using nested virtualization," in *Proceedings of Asia-Pacific Workshop on Systems (APSys)*, 2017.
 - [54] P. K. Sinha, S. S. Doddamani, H. Lu, and K. Gopalan, "mWarp: Accelerating intra-host live container migration via memory warp-ing," in *Proc. of IEEE International Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2019.
 - [55] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer, "Remedy: Network-aware steady state VM management for data centers," in *IFIP International Conference on Networking*, 2012, pp. 190–204.
 - [56] H. Wang, Y. Li, Y. Zhang, and D. Jin, "Virtual machine migration planning in software-defined networks," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2017.
 - [57] H. Liu and B. He, "VMbuddies: Coordinating live migration of multi-tier applications in cloud environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 4, pp. 1192–1205, 2015.
 - [58] D. Fernando, P. Yang, and H. Lu, "SDN-based order-aware live migration of virtual machines," in *Proc. of IEEE International Conference on Computer Communications (INFOCOM)*, 2020.
 - [59] H. Li, G. Xiao, Y. Zhang, P. Gao, Q. Lu, and J. Yao, "Adaptive live migration of virtual machines under limited network bandwidth," in *Proc. of ACM International Conference on Virtual Execution Environments (VEE)*, 2021.



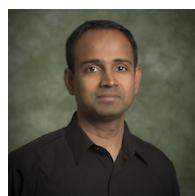
Dinuni Fernando is a Senior Lecturer at University of Colombo School of Computing. She received her Ph.D. from Computer Science Department at Binghamton University. Her research interests include virtualization, networks, and security.



Jonathan Terner received his B.S. in Computer Science from Binghamton University. He is presently a GNR Core Software Developer at IBM and is also pursuing an MS in Computer Science from Georgia Tech.



Ping Yang is an Associate Professor in Computer Science Department at Binghamton University and the Director of the Center for Information Assurance and Cybersecurity. She received her Ph.D. from Stony Brook University. Her research interests include virtualization and AI-based security.



Kartik Gopalan is a Professor in Computer Science Department at Binghamton University. He received his Ph.D. from Stony Brook University. His research interests are in virtualization, cloud computing, security, operating systems, and networks.