A checkpointing mechanism for virtual clusters using memory-bound time-multiplexed data transfers

Jumpol Yaothanee, Kasidit Chanchio
Department of Computer Science, Faculty of Science and Technology, Thammasat University, Pathumthani, Thailand

Article Info

Article history:

Received Mar 24, 2023
Revised Jul 30, 2023
Accepted Sep 6, 2023

Keywords:

Cloud fault tolerance
Distributed snapshot algorithm
Hypervisor-level checkpointing
In-memory checkpointing
Virtual cluster checkpointing

ABSTRACT

Transparent hypervisor-level checkpoint-restart mechanisms for virtual clusters (VCs) or clusters of virtual machines (VMs) offer an attractive fault tolerance capability for cloud data centers. However, existing mechanisms have suffered from high checkpoint downtimes and overheads. This paper introduces Mekha, a novel hypervisor-level, in-memory coordinated checkpoint-restart mechanism for VCs that leverages precopy live migration. During a VC checkpoint event, Mekha creates a shadow VM for each VM and employs a novel memory-bound timed-multiplex data (MTD) transfer mechanism to replicate the state of each VM to its corresponding shadow VM. We also propose a global ending condition that enables the checkpoint coordinator to control the termination of the MTD algorithm for every VM in a VC, thereby reducing overall checkpoint latency. Furthermore, the checkpoint protocols of Mekha are designed based on barrier synchronizations and virtual time, ensuring the global consistency of checkpoints and utilizing existing data retransmission capabilities to handle message loss. We conducted several experiments to evaluate Mekha using a message passing interface (MPI) application from the NASA advanced supercomputing (NAS) parallel benchmark. The results demonstrate that Mekha significantly reduces checkpoint downtime compared to traditional checkpoint mechanisms. Consequently, Mekha effectively decreases checkpoint overheads while offering efficiency and practicality, making it a viable solution for cloud computing environments.

Corresponding Author:

Kasidit Chanchio
Department of Computer Science, Faculty of Science and Technology, Thammasat University
99 Moo 18 Paholyothin Road, Klong Nueng, Klong Luang, Pathumthani 12121, Thailand
Email: kasiditchanchio@gmail.com

1. INTRODUCTION

Cloud computing enables the creation of a virtual cluster (VC) or cluster of virtual machines (VMs) to run distributed and parallel applications. Users can select desired operating systems, software stacks, and computing environments for their applications, making the VC an attractive platform for high-performance computing (HPC) applications [1]. However, as cloud providers scale out hardware resources to meet user demands, the number of failures increases [2]–[4], making fault tolerance mechanisms essential for large-scale cloud computing environments. Checkpoint-restart [5] is a widely used technique for mitigating failures by periodically saving the state of processes to storage and restarting the computation from the saved state when a failure occurs. Most checkpoint-restart mechanisms are implemented at the application [6]–[8], user [9]–[11], and operating system [12]–[14] levels. These mechanisms have been integrated with message passing interface (MPI) libraries and runtime systems to support MPI applications; however, using them is not straightforward. Users often need to modify source codes, programming libraries, or runtime systems.

Hypervisor-level checkpoint-restart [15]–[17] is an alternative to the previously mentioned solutions. This approach saves and restores the state of VMs instead of application processes. The advantage of this approach is that the hypervisor can perform checkpoint-restart operations of a VM independently and transparently from the VM's computation. Users do not have to modify application source code, programming libraries, runtime systems, or guest operating systems (guest OSes).

In the past, a few coordinated checkpoint systems have been created on top of hypervisor-level checkpoint-restart mechanisms. These systems allow users to run MPI applications on guest OSes without modification. We will discuss them in more detail in Section 5. Nevertheless, these systems suffer from the following drawbacks: i) They require complex communication subsystems. In some works, users have to modify the network stack of the host [18] or create a virtual network [19] to provide global consistency and prevent message loss. Such an endeavor is complicated and necessitates major software development work; ii) They can cause long downtimes. Some early works such as those in [20]–[22] pause the computation of every VM in a VC simultaneously, save VMs state to checkpoint storages, and then resume VMs computation. We call this approach the “stop-and-save approach”. This approach causes long checkpoint downtimes, especially when VMs in the VC run memory-intensive applications; iii) They can cause long checkpoint latency. Works such as [19], [23], [24] apply traditional precopy live migration of VMs to reduce the checkpoint downtimes. We call this approach the “traditional-precopy approach”. This approach encounters two problems. First, the traditional precopy live migration may perform for a long time due to high VM memory updates [25]. Second, the checkpoint operation duration of each VM in a VC may be imbalanced; thus, the latency of a VC checkpoint operation can be prolonged by an extremely long checkpoint operation of a single VM.

This paper proposes Mekha, a coordinated hypervisor-level checkpointing system for a VC. Since Mekha is implemented at the hypervisor level, it is highly transparent to applications and guest OS. We have developed a novel coordinated checkpointing protocol using barrier synchronizations to guarantee global consistency. We propose a rendezvous time synchronization mechanism to provide fine-grained barrier synchronizations for the protocols. Additionally, we introduce the concept of time virtualization and leverage the network data retransmission mechanism in the guest OS to effectively handle message loss situations.

Mekha is designed to solve the prolonged precopy duration problem and the imbalance of precopy durations among VMs. On a VC checkpoint operation, Mekha creates a new VM (namely a shadow instance) for each active instance and uses it as an in-memory storage. To transfer state of each active instance to its shadow, we employ a novel precopy live migration mechanism called memory-bound time-multiplexed data transfer (MTD). The MTD overcomes the prolonged checkpoint duration problem by finishing the VM state transfer within a memory-bound period of time. We also introduce the concept of global ending condition in the MTD mechanism and Mekha's checkpoint protocol to handle imbalanced precopy durations among active instances.

We implemented a prototype of Mekha and conducted extensive experiments to compare it with two other VC checkpoint-restart mechanisms: the stop-and-save approach and the traditional precopy approach. We performed checkpoint operations on VCs that were running a block tridiagonal (BT) solver, a class D MPI benchmark program from the NASA advanced supercomputing (NAS) parallel benchmark [26]. Our results showed that: i) Mekha does not require modifications to applications, MPI implementation, and the guest OSes of the VM. Our protocol can provide global consistency and effectively handle message loss without the need to modify the communication subsystem of the guest OS and host OS, ii) Mekha's checkpoint overheads and downtimes are up to 90% lower than those of the stop-and-save mechanism when saving checkpoint files on low write bandwidth storage area network (SAN) storage. On the other hand, Mekha's checkpoint latency is comparable to other checkpoint mechanisms when using the same type of checkpoint storage, iii) Mekha's checkpoint overheads, downtimes, and latencies are comparable to those of the manually finetuned traditional precopy approach, which could be higher without finetuning, iv) Mekha's global ending condition can effectively handle the imbalance of precopy duration among VMs, leading to lower checkpoint overheads and latencies than the traditional precopy approach.

The contents of this paper are organized as follows. Section 2 proposes the Mekha system. Section 3 discusses its implementation. Section 4 elaborates on the experimentation and discusses experimental results. Section 5 discusses related works. Section 6 describes the limitation of Mekha and future improvement. Finally, Section 7 summarizes the paper.

2. MEKHA SYSTEM

2.1. Architecture

Mekha consists of a checkpoint coordinator, agents, VM instances, and networks, as illustrated in Figure 1. The coordinator is responsible for controlling the checkpoint and restart protocols and monitoring all events occurring in the VC when performing VC checkpoint and restart operations. It controls VMs by sending instructions through the agents, which are daemon processes running on every host and sitting between the coordinator and VMs. There are two types of VM instances in Mekha: active and shadow instances. These instances are spawned, destroyed, and monitored by the agent running on the same physical host. Active instances are primary VMs that execute user tasks and are members of a VC. When Mekha starts a VC checkpoint operation, it spawns a shadow instance for an active instance; this shadow instance serves as a transient in-memory storage to store the state of the corresponding active instance. After the checkpoint operation is completed, the coordinator instructs these shadow instances to lazily save their state to persistent storages before terminating them once the state-saving process finishes.

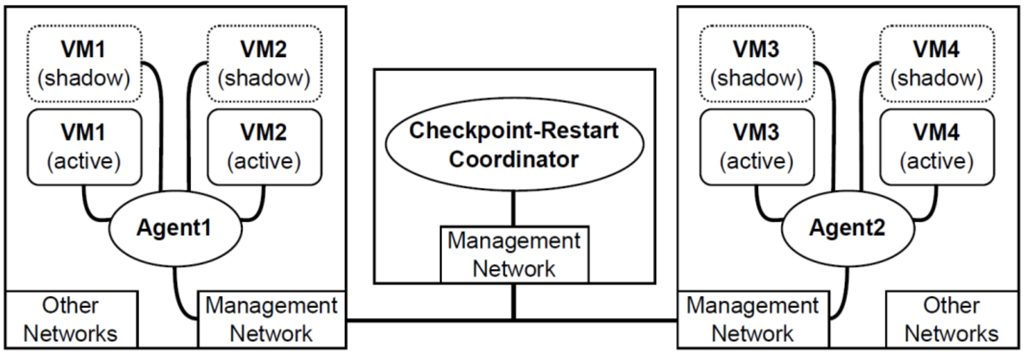


Figure 1. Architecture of Mekha

2.2. VC checkpoint protocol

This section describes the checkpoint protocol of Mekha, as illustrated in Figure 2. The figure contains a coordinator and a VC with three active instances, each running on a different host. The active instance and its corresponding shadow instance are both running on the same host. We assume that this protocol operates during a failure-free period; for simplicity, we omit the checkpoint agents from the figure. We will discuss how the checkpoint protocol of Mekha guarantees global consistency and maintains reliable communication in Section 2.6 and 2.7 respectively. The checkpoint protocol comprises the following steps:

1) Spawning shadow instances: The coordinator broadcasts a `SPAWN` command to all checkpoint agents (at “r” in the figure), instructing them to spawn a shadow instance for each active instance on the same host. All shadow instances are suspended from VM computation from the start.
2) Starting MTD and performing the precopy stage: After receiving notifications that all shadow instances have been launched successfully. The coordinator broadcasts (at “u”) the `START-MTD` command to the hypervisors of all active instances to start the MTD algorithm, a modified version of the memory precopy algorithm of VM live-migration (to be fully described in Section 2.3) Mekha uses this MTD algorithm to reduce checkpoint downtime duration; in Figure 2, the boxes labeled “pre” in the timeline of every active instance represent the *precopy stage* of the MTD. In the MTD design, there are two types of active instances: the starter and non-starter. The *starter instances* are the majority of active instances that end the precopy stage by themselves. These instances (e.g., the instance 1 and 2 in Figure 2) will send the `MTD-EMPTY` message to inform the coordinator and enter the extended precopy stage (“ext” in Figure 2). During this stage, the MTD will continue transferring dirty pages of the active instance to the shadow instance until it receives the `END-MTD` message from the coordinator. On the other hand, the remaining active instances (e.g., the instance 3 in Figure 2) are the *non-starter active instances* that will continue performing the precopy stage until they receive the `END-MTD` message from the coordinator.
3) Evaluating the global ending condition: The global ending condition allows the coordinator to collaborate with the MTD algorithm in determining when to terminate the MTD precopy phase for all active instances. The coordinator evaluates the global ending condition every time it receives the `MTD-EMPTY` message from the active instances. In our design, the global ending condition is satisfied if and only if the coordinator receives the `END-MTD` message from the *majority* of the active instances in the VC. We will discuss this issue further in Section 2.4.
4) Suspending the VC and performing stop-and-copy stage: When the global ending condition is met, the coordinator initiates a *rendezvous time barrier synchronization* to simultaneously suspend the VM's computation of all active instances. In this method, the coordinator calculates the rendezvous time (at

Marker "A" on the coordinator's timeline in Figure 2) and sends the `END-MTD` command, along with the calculated rendezvous time information, to all active instances. The calculation of the rendezvous time will be discussed in Section 2.5. Upon receipt of the `END-MTD` command, the MTD algorithm on each active instance will enter the "stop-and-copy" state. In this state, the algorithm will defer the suspension of the active instance for a period of time ("df1" in the figure), until the local time reaches the rendezvous time, t_s. Then, the VM of the active instance will be suspended ("pause" in the figure). After the suspension, the MTD algorithm on each active instance transfers the remaining memory pages ("transfer remaining" in Figure 2) from the active instances to the shadow instance (see Section 2.3). Then, the MTD algorithm terminates.

5) Creating new overlay disk images: After the MTD algorithm completes, the hypervisor of each active instance sends a status report to the coordinator. The coordinator then sends the `NEW-OVERLAY` message back to the hypervisor, as shown in Figure 2. It is important to note that the coordinator does not wait to receive reports from every instance. Instead, it responds to each active instance individually. Upon receiving the `NEW-OVERLAY` message, the hypervisor changes the current disk image to an immutable backing file and creates a new, copy-on-write, overlay disk image on top of it. The new overlay disk image will store all disk updates that the active instance will make when the VM computation is resumed in the future. The backing file represents the disk image snapshot that is consistent with the VM state stored in the shadow instance. Finally, the hypervisor informs the coordinator that the new image has been created.
6) Resuming the VC: The coordinator launches another *rendezvous time barrier synchronization* to resume the VM's computation of the active instances. After receiving image creation notifications from every active instance in the VC, the coordinator calculates a rendezvous time (at Marker "B" on the coordinator's timeline in Figure 2) and broadcasts the `RESUME-VM` command with the rendezvous time, t_r, to all active instances. Upon receiving this message, the hypervisor of each active instance waits for a "df2" duration before resuming the active instance at time t_r. In Mekha's design, hypervisors must wait until computation of the VM is resumed before notifying the coordinator. As resumption times of VMs depend on VM-specific properties such as workloads, the lengths of "resume" periods may be varied for each instance as shown in Figure 2.
7) Saving state of shadow instances: The coordinator sends the `SAVE-VM` message to the shadow instances, instructing them to save their state to persistent checkpoint storage after all active instances in the VC have been resumed. Once the saving process is complete, the hypervisor of each shadow instance sends a report of the results back to the coordinator. The coordinator then sends the `TERM-VM` message to terminate the shadow instances. The checkpoint protocol ends when all shadow instances in the VC have been successfully saved to persistent checkpoint storage (at time "z").

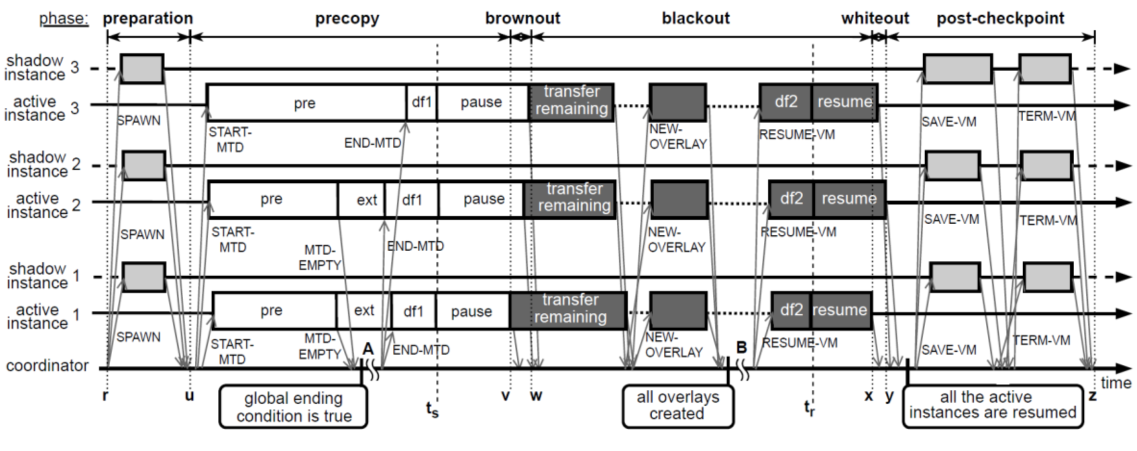


Figure 2. Mekha's checkpoint protocol

2.3. Memory-bound time-multiplexed data transfer

The development of the VM state saving mechanism is a crucial part in the creation of a VC checkpoint mechanism. According to the VC checkpoint protocol of Mekha in Section 2.2, the hypervisor of every active instance must transfer the state of its VM to the shadow instance while the VM is running.

Although we can use traditional precopy live migration to transfer VM state and minimize VM downtime at the same time, it is unsuitable for VMs running memory-intensive workloads [27]. Since workloads of the VM can be unpredictable, it is difficult to determine an appropriate *maximum acceptable downtime* value, the default parameter for ending the precopy stage of the traditional precopy live migration in QEMU [28]. If this value is too low, the precopy algorithm may continue transferring memory pages for an extended period until the applications have terminated. The default value of this parameter in QEMU is 300 milliseconds, which is too low for numerous memory-intensive applications [27].

We propose a new memory-bound time-multiplexed data transfer (MTD) algorithm to address the prolonged precopy duration problem previously discussed. In MTD, the VM state consists of the CPU state, memory state, and device state. Since the memory state is generally large, MTD transfers the memory state to the shadow instance while the active instance is running. MTD separates memory pages into three sets: initial, clean, and dirty. *Initial pages* are memory pages in the *initial set* that have never been transferred to the shadow instance. *Clean pages* are memory pages in the *clean set* that have been sent to the shadow instance and have not been modified by the VM after being sent. *Dirty pages* are memory pages in the *dirty set* that have been modified after being sent to the shadow instance. Note that a memory page can only belong to one set at a time. The MTD algorithm transfers the initial and dirty pages from the active instance to the shadow instance in a time-multiplexed style. It alternates between transferring the initial pages and the dirty pages using two user-defined timeout parameters: the *initial-set timeout* and the *dirty-set timeout*. These parameters determine the duration of time that the MTD spends transferring the initial and dirty pages to the shadow instance, respectively.

At the start of the transfer, the hypervisor of the active instance creates a migration thread to run the MTD algorithm concurrently with the VM's computation. Initially, the algorithm assigns every memory page to the initial set, leaving the clean and dirty sets empty. It also shortly pauses the VM to activate the memory update tracking mechanism, which starts monitoring memory updates made by the VM from that point forward. In our implementation, we utilize the memory tracking mechanism of KVM. Since this tracking mechanism can cause overhead for VM execution, it will be turned off once the MTD finishes.

Figure 3 illustrates the FSM diagram of the MTD algorithm, which consists of four states. These states are associated with three stages of the MTD operation. The first stage is the *precopy stage*, which comprises the operations in the "copy-initial-set" and "copy-dirty-set" states. The second stage is the *extended precopy stage*, which corresponds to the operation in the "wait-for-EMD-MTD" state. Finally, the stop and copy stage represents the operation in the "stop-and-copy" state.

The MTD begins by performing memory page transfer operations associated with the "copy-initialset" state, as shown in Figure 3. During this stage, the algorithm transfers the contents of the initial pages to the shadow instance until the initial set is empty or the initial-set timeout expires. The transmitted pages are then moved from the initial set to the clean set. If the initial set is empty, the MTD will send an `MTD-EMPTY` message to inform the coordinator and change to operate at the "wait-for-END-MTD" state. On the other hand, if the initial-set timeout expires, the MTD algorithm moves to operate at the "copy-dirty-set" state, as shown in Figure 3.

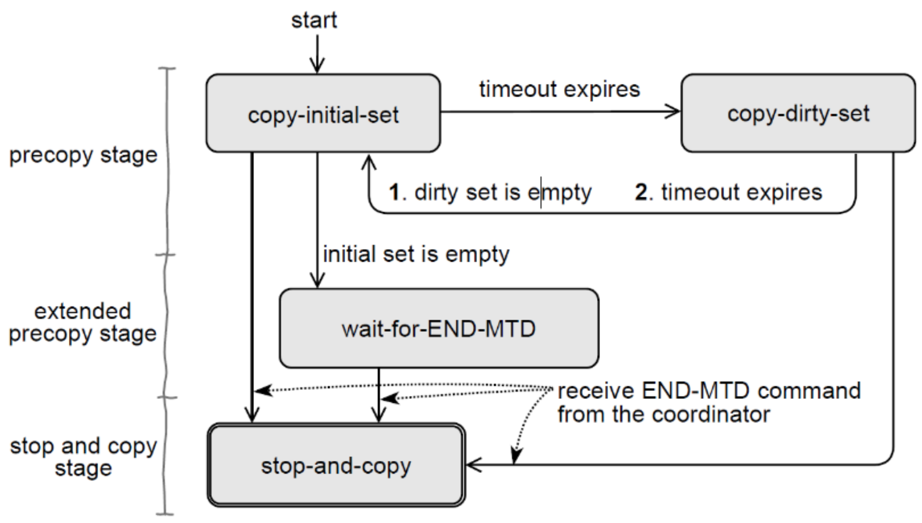


Figure 3. An FSM diagram of the MTD algorithm

At the “copy-dirty-set” stage, the MTD transfers the contents of dirty pages to the shadow instance until it changes operational state. The MTD moves every transmitted dirty page from the dirty set to the clean set. If the VM modifies any page in the clean set, the memory tracking module will keep track of them. According to the FSM diagram in Figure 3, the MTD will keep transferring dirty pages until the dirty-set timeout expires before transitioning back to the “copy-initial-set” state. However, if the dirty set is emptied before the timeout, the MTD would retrieve all the modified clean pages from the memory tracking module and add them to the dirty set. Then, it will continue transferring the dirty pages until the dirty set is emptied for the second time or the dirty-set timeout expires. In the former case, if there is some remaining time before the dirty-set timeout expires, the remaining time will be added to the initial-set timeout for the next operations at the “copy-initial-set” state. In practice, we recommend setting the initial-set timeout to be greater than the dirty-set timeout to speed up the completion of the precopy stage since the number of initial pages is generally greater than that of dirty pages. We discuss these timeout values used in our experiments in Section 4.2.

In the “wait-for-END-MTD” state, the MTD will repeatedly transfer the contents of dirty pages to the shadow instance. The transferred dirty pages will be moved from the dirty set to the clean set, as usual. If the dirty set is empty, the MTD will move the clean pages updated by the VM from the clean set to the dirty set. It will then continue transferring the dirty pages to the shadow instance. On the other hand, if the dirty set is empty but there are no updated clean pages, the algorithm will wait and periodically poll the memory tracking module until newly updated clean pages are found. The MTD algorithm in the “wait-for-END-MTD” state keeps performing the dirty page transfers and polling described above until it receives the `END-MTD` command from the coordinator. After receiving the command, the MTD enters the “stop-and-copy” state.

One of the key features of the MTD algorithm’s design is its cooperation with the global ending condition mechanism of the coordinator, which handles the imbalance of precopy durations among active instances. According to the FSM diagram in Figure 3, the MTD algorithm running in the hypervisor of each active instance will stop operations and immediately transition to the “stop-and-copy” state upon receiving the `END-MTD` command from the coordinator. This can happen at any of the “copy-initial-set”, “copy-dirty-set”, or “wait-for-END-MTD” states. If the command is received while the MTD is operating in the “copy-initial-set” or “copy-dirty-set” state, the initial set or the dirty set may not be empty when the MTD enters the “stop-and-copy” state. Alternatively, if the command is received while the MTD is in the “wait-for-END-MTD” state, only the dirty set may not be empty when the MTD performs the “stop-and-copy” operation.

At the “stop-and-copy” state, the MTD algorithm suspends the active instance as a result of the rendezvous time mechanism (described in step 4 of Section 2.2) Then, it retrieves the memory pages that have been updated by the VM from the memory tracking module and moves from the clean set to the dirty set. Note that these pages are those that have been modified since the last time the movement from the clean set to the dirty set took place. Next, the algorithm transfers the contents of the remaining memory pages in the initial and dirty sets to the shadow instance. These memory pages will be moved from either set to the clean set. Eventually, all memory pages will be in the clean set, and the initial and dirty sets will be empty. The MTD will then transfer all the device states to the shadow instance and finish.

2.4. Global ending condition

The global ending condition is our invention to address the issue of imbalanced durations of the time MTD required to empty the initial set. This imbalance can occur due to various factors such as host workload, VM workload, and I/O contention. As a result, the coordinator is unable to initiate the step 4 of the checkpoint protocol (in Section 2.2) until the longest transferring initial set has been completed. This is undesirable in a large VC. Therefore, the global ending condition was created to solve this problem.

In our design, the coordinator would perform the following algorithm to evaluate the global ending condition. Let n represents the total number of VMs in a VC, and c represents the number of starter active instance that sent the `MTD-EMPTY` message to the coordinator. Initially, $c = 0$.

– Based on the MTD algorithm shown in Figure 3, the active instance that holds an empty initial set will send the `MTD-EMPTY` message to the coordinator. These are the *starter active instances* introduced in Section 2.2. Upon receiving this message, the coordinator performs $c = c + 1$.
– The coordinator evaluates the (1).

$$c \geq \lfloor n/2 \rfloor + 1 \tag{1}$$

If the (1) is true, the global ending condition is satisfied. The coordinator will proceed to perform the next step. Otherwise, the coordinator will perform the previous step, waiting for additional `MTD-EMPTY` messages from the starter instances.

– Finally, the coordinator broadcasts the `END-MTD` message to all active instances within the VC. Note that we call the active instances that did not send the `MTD-EMPTY` as the *non-starter active instances* (mentioned earlier in Section 2.2). Therefore, based on the (1), the majority of the active instances would be the starter instance while the rest are the non-starter.

In Figure 2, it can be observed that the coordinator identifies the global ending condition as true upon receiving the `MTD-EMPTY` message from two starter active instances (the active instance 1 and the active instance 2). Note that after sending the message, the starter instances enter a "wait-for-END-MTD" state (or the extended precopy stage), denoted by the "ext" labels in the figure. Conversely, the non-starter instance (the active instance 3) continues performing the precopy stage (labeled "pre") of its MTD operation until the `END-MTD` message is received.

It is important to note that the global ending condition equation can be generalized to diversify the behavior of the VC checkpoint operation. For instance, if we change the (1) to $c = n$, the checkpoint protocol would wait until the initial set of every active instance is emptied. On the other hand, if $c = 0$, the checkpoint protocol would send the `END-MTD` message together with the `START-MTD` message, causing the MTD algorithm to execute the "stop-and-copy" stage without performing the precopy operation.

2.5. Synchronization using rendezvous time

The rendezvous time is a predetermined time in the future on a physical host at which the hypervisor will execute an operation. This synchronization mechanism was first introduced in [21]. However, the calculation of the rendezvous time in Mekha is different. In Mekha, the coordinator calculates the rendezvous time using (2):

$$rendezvous = now + nwd + ovh \qquad (2)$$

In (2), the rendezvous time is the sum of the current time on the coordinator when the rendezvous time is calculated (now), the VC-wide network delay (nwd) or the management network in Figure 1, and miscellaneous overheads (ovh). To determine the nwd, the coordinator broadcasts the `CHECK-STATUS` message to the hypervisors of all active instances, which in turn replies with their status back to the coordinator. The nwd is the duration from the starting time of the broadcasting of the `CHECK-STATUS` message to the time when the coordinator receives the last reply back from the hypervisors.

We estimate the ovh using the equation $ovh = 4\sigma$, where σ represents the standard deviation of the nwd values collected by the coordinator offline prior to the checkpoint operation. The multiplier 4 in the equation is based on RFC 6298 [29], which specifies that it should be used as the multiplier for the standard deviation in the transmission control protocol (TCP) retransmission timeout calculation. In our experiments, we calculate σ from 50 offline broadcasts of the `CHECK-STATUS` message to determine the nwd and set the ovh value to 20 milliseconds.

2.6. Global consistency

Figure 4 illustrates the space-time diagram of three VMs and the different types of transmissions of layer-2 Ethernet frames among them during a VC checkpoint operation. The annotations "r", "u", "v", "w", "x", "y", and "z" are used to indicate the starting and ending points of time of each phase. These annotations correspond to the same annotations in Figure 2.

The *preparation phase* starts at "r", where the coordinator instructs the hypervisor of every active instance to spawn a shadow instance, and ends at "u", where the coordinator instructs the hypervisor of every active instance to perform the MTD algorithm. The *precopy phase* starts at "u" and ends at "v", when the algorithm suspends the active instance and enters the "stop-and-copy" state for the first time. Next, the *brownout phase* begins when the first active instance is suspended at "v" and ends when the last one is suspended at "w". The *blackout phase* begins at "w" when all active instances are suspended and ends at "x" when the first active instance starts resuming VM computation. The *whiteout phase* starts at "x" and ends at "y" when all active instances are resumed and running. The final phase, the *post-checkpoint phase*, begins at "y" and ends at "z" when the coordinator receives termination notification from all shadow instances.

In Figure 4, we define the downtime periods (the dotted lines) of every active instance to collectively represent a wide *cut* line that separates events of every VM in the VC into those before and after the cut. The cut covers the area from the "S1" line to the "R1" line in the space-time diagram of every VM. It has the following properties: i) According to the MTD algorithm, the suspension of each active instance always occurs before the resumption of that instance during a VC checkpoint operation; ii) All the suspension events of every active instance in a VC happened before all the resumption events of the active instances. This is because all the suspension events (represented by the cross-cut line "S1" in Figure 4) are generated from the rendezvous time barrier synchronization in step 4 of the checkpoint protocol, while all the resumption events (represented by the cross-cut line “R1”) are generated by the rendezvous time barrier synchronization in step 6.

We define the sets of events that occur before the cut as the “PAST” set and the set of events occurring after the cut as the “FUTURE” set, respectively. Based on these definitions, four types of layer-2 Ethernet frames can occur during communication during a checkpoint operation: type “a” frames are not involved in the checkpoint operation, type “b” frames are those sent from send events in the “PAST” set to receive events in the “FUTURE” set, type “c” frames are those sent from the “PAST” events to suspended VMs in the cut, and type “d” frames are those sent from the “FUTURE” events to suspended VMs in the cut. It is important to note that the type “c” and “d” frames are lost as the destination VM is suspended.

– **Theorem 1.** *The checkpoint protocol of Mekha always generates a set of globally consistent checkpoint files.*
– **Proof of Theorem 1.** According to the checkpoint protocol of Mekha, a VC checkpoint operation creates a set of checkpoint files, which contain the state information of each active instance in the VC. The protocol ensures global consistency by preventing a send event in the “FUTURE” set from sending a frame to a receive event in the “PAST” set in Figure 4. Based on the global consistency principle [30], a set of checkpoints is globally consistent if and only if the inconsistent cut does not exist. According to the checkpoint protocol in Figure 2, two barrier synchronizations are used to ensure the following: The first barrier synchronization at Step 4 ensures that every event in the “PAST” set has occurred before *all active instances* are suspended to perform MTD's stop-and-copy stage. The second barrier synchronization at Step 4 of the protocol ensures that the stop-and-copy stage on every active instance must complete before any event in the “FUTURE” set occurs. As a result, there cannot be a receive event in the “PAST” that receives a frame sent from the “FUTURE”. Therefore, every event in the “PAST” set always happens before every event in the “FUTURE” set, making an inconsistent cut impossible.

2.7. Time virtualization and message retransmission

In Mekha's design, we apply the concept of time virtualization to every VM in the VC. Under this concept, the clock of a VM progress when the computation of the VM progress. If the VM is suspended the clock of the VM is stopped. Time virtualization allows us to leverage the TCP retransmission mechanism to retransmit any TCP packet that may be lost during a VC checkpoint operation. When Mekha's checkpoint protocol suspends every active instance, their clocks are also paused. During that time, some TCP packets may be lost. However, the senders of these packets have their copies in the TCP send buffer. When the checkpoint protocol resumes the computation of the active instance, the clock on every active instance will continue and the TCP retransmission mechanism will retransmit the lost packet. Since the clock on the sender VMs resumed from the time when they were suspended, TCP can retransmit the packet immediately without suffering retransmission delays caused by TCP exponential backoff or TCP connection termination due to long idle time.

– **Theorem 2.** *For any Ethernet frame transmitted under a data retransmission mechanism in the network stack of a VM, if the retransmission of a data frame is valid before the suspension of the VM and after the resumption, the frames sent during a VC checkpoint operation will eventually be received and the retransmission mechanism will eventually finish.*
– **Proof of Theorem 2.** In a network stack that employs a data retransmission mechanism for Ethernet frames, the frames transmitted by the sender are subject to potential retransmissions until they are successfully received by their intended recipients. Based on the TCP retransmission mechanism [31], the sender would continue to retransmit data frames until their acknowledgments (ACKs) are received. Furthermore, we define the retransmission to be valid as long as the sender does not exceed the maximum number of retransmissions allowed by the protocol.

  During the VC checkpoint operation, VMs undergo suspension. This means that their normal execution and the progression of virtual time within the VM are temporarily halted. The VM’s network stack is also frozen, preventing any further transmission or reception. Once the VM resumes, the network stack becomes active again, and the transmission or reception of Ethernet frames continues from the point it left off prior to suspension. Consequently, the transmission state of the frames, including those awaiting retransmission, remains unaffected during the suspension period. Therefore, any frames that required retransmission before the suspension will still be eligible for retransmission after the VM resumes.

  According to Mekha’s checkpoint protocol, we consider the sending of a data frame in two cases. The former, namely Case 1, occurs before the VC-wide VM suspension (line “S1” in Figure 4) caused by the barrier synchronization at step 4 of the checkpoint protocol. The latter, namely Case 2, happens after the VC-wide VM resumption (line “R1” in Figure 4) caused by the barrier synchronization at step 6 of the protocol.

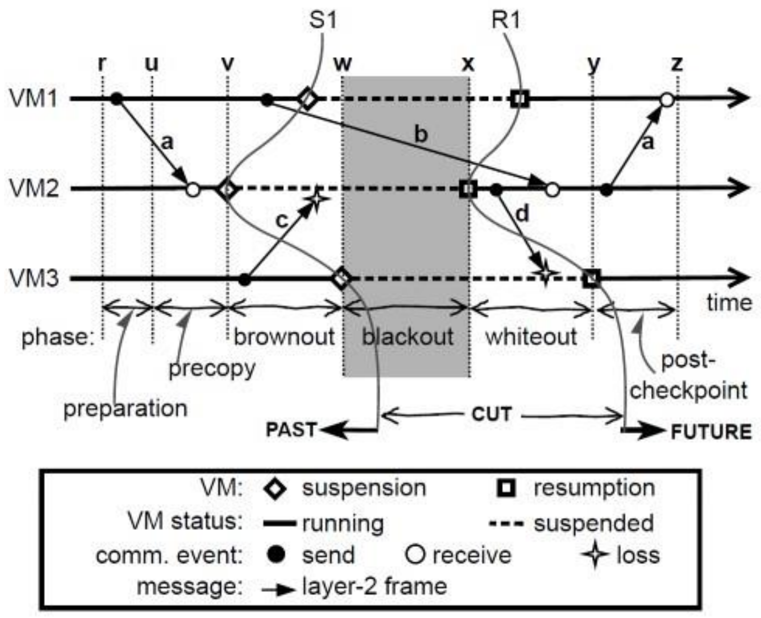


Figure 4. Phases and transmissions of the layer-2 frames during a Mekha's checkpoint operation

Case 1: There are three scenarios in this case.

- If a transmitted frame arrives at its destination before both the sender and receiver are suspended at line S1, the frame is of type "a" frame. The sender will continue retransmitting the frame until it receives the ACK for that frame, after which it will continue retransmission under Case 2.
- If the sender VM transmits a data frame that is lost because the receiver is already suspended, the frame becomes the type "c" frame in the figure. In this case, the sender VM will keep transmitting the frame until it is suspended. After the VM resumes, the retransmission will continue under Case 2.
- If the sender transmits a data frame but the frame arrives at the destination after the destination VM resumes its computation (beyond the R1 line), the frame becomes the type "b" frame. Under this scenario, the retransmission mechanism will continue until the sender VM is suspended at line S1. After the resumption, the retransmission mechanism will operate under Case 2. However, if the sender VM receives the ACK of the type "b" frame while the retransmission under Case 2 is operating, it will immediately terminate the retransmission mechanism of that frame.

Case 2: After the sender VM resumes, it may transmit brand new or retransmitted data frames left over from before the suspension. In any case, if the resumed VM successfully sends a data frame to its destination, the frame becomes a type "a" frame. Once the sender receives the ACK for that frame, the retransmission will finish. On the other hand, if the resumed VM cannot transmit the frame because the destination VM is not yet resumed, the frame becomes a type "d" frame. However, since the resumption of the sender and the destination VM is performed under the same barrier synchronization at Step 6 of the checkpoint protocol (the "R1" line), the retransmission will eventually succeed and terminate.

3. IMPLEMENTATION

In the implementation of Mekha, we used Java to develop the checkpoint coordinator and checkpoint agent. We employed an asynchronous and event-based approach for handling requests and responses. Our implementation utilizes RabbitMQ as a middleware for efficient message routing and broadcasting between the coordinator and agents. The communication between the coordinator and agents is carried out through asynchronous requests and responses. The coordinator sends commands as request messages, which are then broadcasted to the agents. The agents receive these commands and translate them into the QEMU machine protocol (QMP) format before forwarding them to the instances.

We have implemented the MTD algorithm by modifying the precopy live migration mechanism of QEMU version 4.1. We have implemented two new QMP functions, *stop(rendezvous)* and *cont(rendezvous)*, in the QEMU hypervisor to suspend and resume a VM at the rendezvous time. To implement time virtualization, we leveraged the real-time clock (RTC) of the QEMU instance using the $clock = vm$ option.

4. EVALUATION

We have conducted several experiments to assess the checkpoint performance of Mekha. These experiments aim to achieve two primary objectives. First, we aim to evaluate checkpoint performance by employing various checkpoint mechanisms across different types of checkpoint storage. Second, we intend to measure the impact of differences in precopy durations on the performance of VC checkpoint mechanisms. These experiments are vital for gaining a comprehensive understanding of Mekha's checkpoint capabilities and their adaptability to various scenarios.

The experiments were conducted using three physical hosts, i.e., Host1, Host2, and Host3, as shown in Figure 5. Host1 and Host2 are used for running a VC while Host3 is used for running the coordinator. Host1 and Host2 are Dell PowerEdge R720 servers. Each server has two hyper-threading 10 cores Xeon E5-2600v2 2.8GHz CPUs and 192GB DDR3 RAM. The local SSD storage is a 500GB Samsung 970 EVO Plus NVMe card connected to the PCIe 3.0 of the server. Both hosts are connected to a 36TB Dell PS4110E EqualLogic SAN storage via a 10Gbps network and use iSCI protocol to access the storage. Each server has two ethernet network interface cards (NICs); the first NIC is connected to a 10Gbps Ethernet network (the data network in the figure) and the second NIC is connected to a 1Gbps network (the management network). Host3 is a server that has a 1Gbps Ethernet NIC connected to the management network. The network time protocol (NTP) is used to synchronize time among these hosts. Figure 5 shows the VC architecture used in our experiments. All VMs in a VC are connected to a virtual network created over the 10Gbps Ethernet network. We use Open vSwitch [32] to create virtual switches on Host1 and Host2 and link them together using a GRE tunnel over the 10 Gbps Ethernet network.

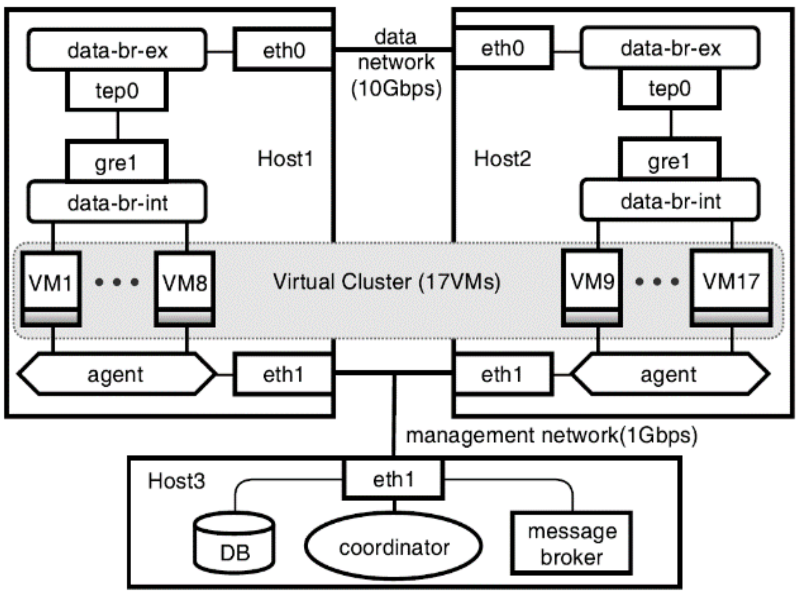


Figure 5. The virtual cluster architecture used in our experiments

In our experiments, we created two VCs of different sizes to evaluate the performance of three different VC checkpoints and restart mechanisms.

– The H1 VC consists of 5 VMs, including 1 master VM and 4 worker VMs. The master VM launches MPI applications while the worker VMs perform computation. In the H1 VC, we run 2 and 3 VMs on Host1 and Host2, respectively. Each VM has 16 vCPU cores, 16 GB vRAM, 40 GB of storage, a 1 Gbps virtio-net virtual network interface (vNIC), and a 10 Gbps vNIC.
– The H2 VC consists of 17 VMs, including 1 master VM and 16 worker VMs. We assign 8 VMs and 9 VMs to run on Host1 and Host2, respectively. Each VM has 4 vCPU cores, 4 GB vRAM, 20 GB of storage, a 1 Gbps vNIC, and a 10 Gbps vNIC.

In our experiments, we use both a local NVMe solid-state drive (SSD) and a SAN for checkpoint storages. Since we have noticed that I/O often hangs when we simultaneously save state of all instances in the H2 VC to the SAN, we have implemented a scheduling algorithm in which the state of at most two instances on a host will be saved to storage simultaneously. To test Mekha on computation and communication-intensive applications, we run the MPI version of the Block Tri-diagonal solver (BT) class D from the NAS Parallel Benchmark version 3.3.1 [26] in the VCs. The BT has execution times long enough to create multiple checkpoints and yet not too long to perform multiple runs. Additionally, the problem sizes are suitable for the available resources.

4.1. Other checkpoint mechanisms

We have developed two additional VC checkpoint mechanisms for performance comparisons with Mekha. They are the virtual cluster checkpoint-restart (VCCR) based on the work of [33] and the In- memory virtual cluster checkpoint-restart (IMVCCR) based on the work of [34]. Both mechanisms use a checkpoint coordinator to control the checkpoint operation and employ time virtualization and guest OS-level packet retransmission mechanisms similar to that of Mekha for maintaining reliable communication during a VC checkpoint operation.

The *VCCR* protocol is a coordinated checkpointing method that uses a two-phase commit protocol to achieve barrier synchronization and maintain global consistency. The protocol begins by suspending all VMs after applying a barrier synchronization before proceeding to save the state of each VM to designated checkpoint storage. Once all VMs have completed their state savings, the coordinator initiates another barrier synchronization to resume the execution of all VMs. We have mapped the operations of the VCCR checkpoint protocol into phases, which partially resemble those of the Mekha protocol, as illustrated in Figure 4. Like Mekha, the duration of each phase is measured by the coordinator. The preparation phase is the duration from the initiation of the checkpoint operation to the first suspension of a VM. The brownout phase is defined as the duration from the end of the preparation phase to the suspension of all VMs in the VC. The blackout phase begins at the point of all VM suspension and ends at the point of the first VM resumption. The whiteout phase is the duration from the end of the blackout phase to the resumption of all VMs. The precopy and post-checkpoint phases do not exist in VCCR, as it does not implement the precopy algorithm.

IMVCCR is a VC checkpoint mechanism that uses a modified version of the precopy algorithm for VM live migration in order to reduce the downtime of VM checkpointing. Similar to Mekha, IMVCCR uses shadow VMs to temporarily save checkpoints in memory before eventually saving them to persistent storage. The checkpoint protocol of IMVCCR operates in five phases, which closely resemble those of Mekha. In the preparation phase, the coordinator creates a shadow VM for each active instance of the VC. Once all shadow VMs have been created, the coordinator initiates the precopy phase, instructing every active instance to begin the modified VM live migration mechanism, namely the *extended precopy algorithm*. This algorithm employs traditional precopy live migration techniques to transfer memory pages to the shadow instance until the estimated downtime of transferring dirty pages in a single iteration is less than the *maximum acceptable downtime*. The active instance then sends a `PRECOPY-CONVERGED` message to notify the coordinator. After this, the active instance continues to transfer dirty pages. Upon receipt of the `PRECOPY-CONVERGED` message from every VM, the coordinator broadcasts an `END-PRECOPY` message with a rendezvous time to the hypervisors of each active instance. Upon receipt of this message, the hypervisor suspends the VM at the rendezvous time, ending the precopy stage and initiating the stop-and-copy stage of live migration. The remaining operations in the brownout, blackout, whiteout, and post-checkpointing phases are identical to those of Mekha. IMVCCR also uses time virtualization and rendezvous time synchronization similar to Mekha. However, unlike Mekha, IMVCCR does not have mechanisms to handle the imbalance of precopy durations. We will discuss their differences in Section 4.6.

4.2. Experimental parameters and configurations

For Mekha, we assume that the number of the initial pages of the MTD algorithm is greater than the number of dirty pages. Therefore, we set the initial-set timeout to 3000 milliseconds, the dirty-set timeout to 750 milliseconds, and the ovh value (see Section 2.5) to 20 milliseconds. These values serve as the default parameters for all experiments. For IMVCCR, we have to define different values for the maximum acceptable downtime parameter used in the execution of the BT benchmark on the H1 and H2 VCs. The default value for this parameter in traditional QEMU is 300 milliseconds; however, when running the BT on the H1 cluster, the precopy operation does not stop until the benchmark finishes. Therefore, we are required to manually set the parameter value to 3500 milliseconds to avoid failure of the experiments. In contrast, the default value is sufficient for running the BT benchmark on the H2 VC.

In both Mekha and IMVCCR, we set the maximum data transfer bandwidth for sending VM state from active to shadow instances to its maximum Int 32 value (approximately 2.14 Gigabytes per second or 17.12 Gbps). This is done to limit interference with other VMs running on the same host. During the execution of the BT benchmark on H1 and H2 VCs, the checkpoint interval was set to 8 minutes; this means that the coordinator will initiate a checkpoint operation every 8 minutes until the BT benchmark is completed. We evaluated three checkpoint mechanisms (VCCR, IMVCCR, and Mekha) and two storage types (SSD and SAN). Since both Mekha and IMVCCR use shadow instances as transient checkpoint storage, we defined a *SHADOW* storage as a baseline measurement of storage performance; using this storage the protocols terminate the shadow instance after the precopy algorithm finishes without saving state into persistent storage. Thus, giving us 8 cases of a combination of checkpoint mechanisms/storage types for experiments on H1 and H2 VCs. Every data point presented here is an average of 10 independent runs.

4.3. VC checkpoint overheads

The overheads of checkpoint operations are the extra amount of time that the application runs with checkpoint capability enabled compared to its normal execution time. The average checkpoint overheads per operation (avg.overheads) is referred to the overheads that are created from a single checkpoint operation. Since Mekha and IMVCCR perform live migration to replicate the state of active instances to main memory before lazily saving them to persistent storage, they are able to effectively hide latency when writing VM state to storage, particularly on SAN storage, which has low write bandwidth. Thus, there are three main factors that affect the checkpoint overheads of Mekha and IMVCCR: i) the interference from precopy, suspension, and resumption operations; ii) the checkpoint downtime duration; and iii) the interference caused by saving the state of the shadow VMs to checkpoint storage. In contrast, VCCR pauses VMs during the process of saving VM state to storage, and as a result, the checkpoint overheads are dominated by the downtime of each VM. Therefore, the checkpoint overheads of VCCR are substantially higher than that of Mekha and IMVCCR.

As shown in Figure 6(a), the avg.overheads of the Mekha and IMVCCR on SAN storage are significantly lower than that of the VCCR. The avg.overheads of Mekha on SAN storage on the H1 and H2 VCs are 91.6% and 83.7% lower, respectively than those of VCCR on SAN storage. However, the overheads of VCCR improved significantly when we perform checkpoint operation on the SSD storage that has high write bandwidth. In Figure 6(a), the avg.overheads of Mekha on the H1 and H2 VCs are 58.53% and 26.88% lower than those of VCCR when using SSD instead of SAN. Finally, we observe that the overheads of Mekha and IMVCCR are slightly different from one another since both mechanisms use precopy algorithms. We will discuss the impact of the precopy mechanisms in Section 4.5.1.

4.4. VC checkpoint latency

The latency of a VC checkpoint operation is the duration from the time the VC checkpoint mechanism starts to the time it ends. The average checkpoint latency (avg.latency) is referred to the latency that is created from a single checkpoint operation. From Figure 6(b), we can see that avg.latencies for each mechanism are dependent on the write IOPS and write bandwidth of checkpoint storage. Our experiments have revealed that most of the checkpoint latency is spent on saving VC state to storage, suggesting that latency is highly influenced by storage's write IOPS and write bandwidth. As a result, avg.latencies for SSD storage are significantly lower than those for SAN storage due to the superior performance of SSD; for example, Mekha's avg.latencies using SSD storage are 85.6% and 88.5% lower than those using SAN storage on H1 and H2 VCs respectively.

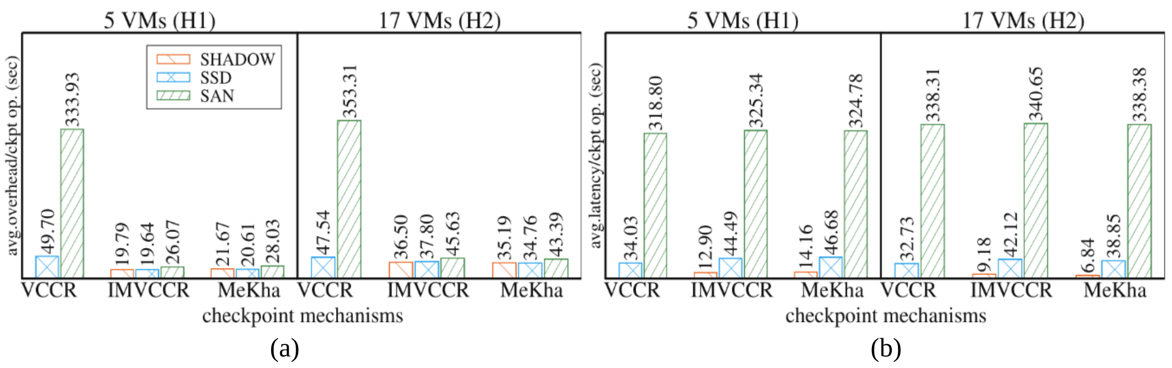


Figure 6. Comparing checkpoint performance metrics for different checkpoint mechanisms on the H1 and H2 VCs in (a) average checkpoint overheads per a checkpoint operation (sec) and (b) average checkpoint latency per a checkpoint operation (sec)

4.5. Analysis of the checkpoint phases

The avg.latencies presented in Figure 6(b) can be further dissected into the various checkpoint phases, as shown in Figure 7. We analyze the duration of each phase in the following sections. Note that the coordinator uses time on its host to record the starting and ending times of each phase. In the beginning, the duration of the preparation phase refers to the time that the coordinator uses to prepare the checkpoint mechanism. It is generally negligible, as shown in Figure 7(a). The average duration of the preparation phase (avg.preparation) of Mekha and IMVCCR is longer than that of VCCR due to the spawning of the shadow instance.

4.5.1. Performance of memory precopy mechanism

According to the MTD algorithm, the length of its precopy duration depends on i) the total amount of memory pages used by the VM, ii) the data transfer bandwidth, and iii) the dirty page generation rate. Figure 7(b) reveals that the average duration of the precopy phase (avg.precopy) for Mekha in H1 VC is longer than that of H2 VC due to the following reasons: i) The amount of memory usage for active instances in H1 VC when running the BT benchmark is higher than that of the H2 VC. In our experiment, both H1 and H2 VCs ran the same class of BT benchmark consisting of 64 MPI tasks; however, as the H1 cluster has 4 compute node VMs each VM hosted 16 MPI tasks. On the other hand, each VM in the H2 VC ran 4 MPI tasks as there were 16 compute node VMs. Our measurements show that compute node VMs in H1 and H2 VCs use the memory of approximately 7.99 GB and 2.15 GB, respectively; and ii) Dirty page generation rates of compute node VMs in H1 VC are higher than those in H2 VC as H1 VMs run 4 times more MPI processes than those on H2 VMs.

The precopy durations of Mekha and IMVCCR are different since they use different precopy algorithms. We will discuss their differences in two cases: i) On H2 VC, avg.precopy durations of Mekha are substantially lower than those of IMVCCR because Mekha's global ending condition can handle load imbalance better than that of IMVCCR. For instance, avg.precopy duration using SAN storage is 55.38% lesser than that of IMVCCR in the Figure 7(b). We will discuss the difference of the precopy duration of Mekha and IMVCCR again in Section 4.6; and ii) On the other hand, the precopy durations of IMVCCR are slightly lower than those of Mekha in the H1 VC because we had to manually set the maximum acceptable downtime value to 3,500 milliseconds (instead of using the default value) in order to achieve desired precopy performance as shown in Figure 7(b). Using lower values would result in checkpoint failure since the extended precopy algorithm will continue to transfer memory pages from active to shadow instances until the BT benchmark completes execution. It is important to note that this problem does not exist in Mekha because the MTD algorithm does not rely on the maximum acceptable downtime value to terminate.

4.5.2. VC suspension and resumption performance

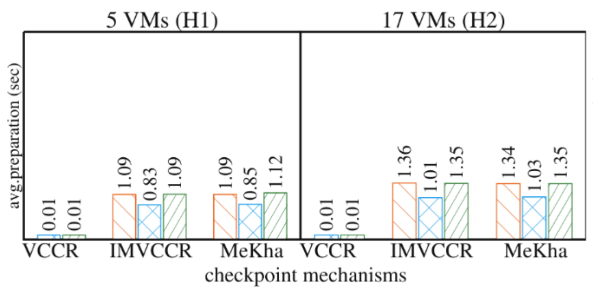
The experimental results confirm the advantages of the rendezvous time synchronization in Mekha and IMVCCR over the simple, two-phase commit synchronization mechanism used in VCCR. As shown in Figure 7(c) and Figure 7(d), the average duration of the brownout phase (avg.brownout) and the average duration of the whiteout phase (avg.whiteout) for Mekha and IMVCCR are significantly shorter than that of VCCR. Note that the avg.whiteout of all three checkpoint mechanisms is higher than their avg.brownout because the resume VM operation depends on the individual workload of the VM and requires a longer duration to complete.

4.5.3. VC checkpoint downtime

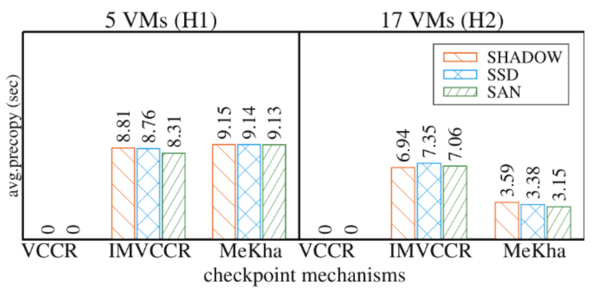
From experiments, it can be observed that the average duration of the blackout phase (avg.blackout) of Mekha and IMVCCR is substantially shorter than that of VCCR for two reasons. First, Mekha and IMVCCR transfer state of active instances to shadow instances while VCCR saves state of the VM directly to persistence storages. Second, Mekha and IMVCCR use memory precopy mechanisms to minimize the number of memory pages transferred from active to shadow instances during the blackout phase. From Figure 7(e), avg.blackout of Mekha and IMVCCR is much shorter than avg.blackout of VCCR in both H1 and H2 clusters.

4.5.4. VC state saving latency (VSL)

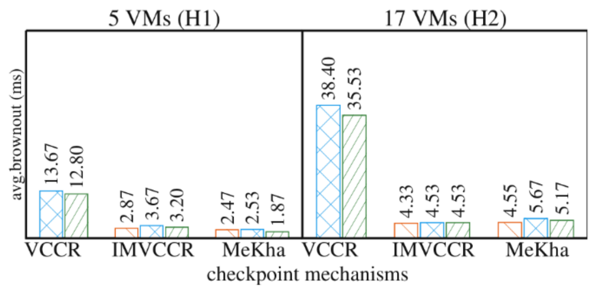
For VCCR, the VSL is equivalent to the duration of the blackout phase; however, for Mekha and IMVCCR, it is the average duration of the post-checkpoint phase (avg.post-checkpoint). By comparing the VSL of each mechanism reported in Figure 7(e) and Figure 7(f) against the checkpoint latencies in Figure 6(b), it was determined that VM state saving latencies are a dominant factor in overall latency in each case. For example, the VSL of Mekha using SSD and SAN storage on H1 VC in Figure 7(f) account for approximately 96.55% and 95.56% of corresponding avg.latencies of Mekha in Figure 6(b).



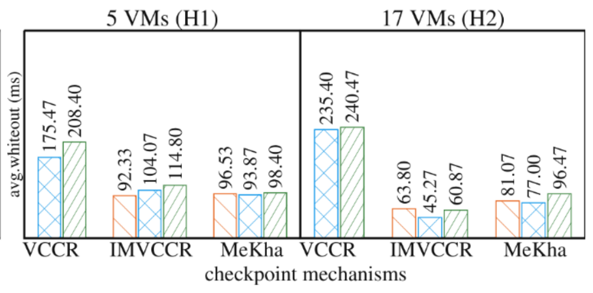
(a)



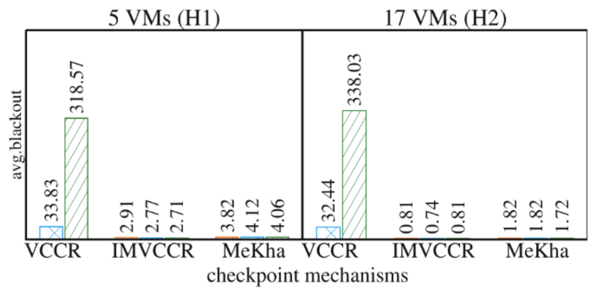
(b)



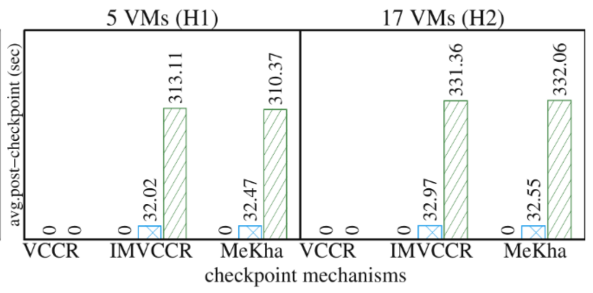
(c)



(d)



(e)



(f)

Figure 7. Comparing the duration of each checkpoint phase for each checkpoint mechanism when performing the checkpointing operation on the VC of 5 and 17 VMs with respect to: (a) average duration of the preparation phase (sec), (b) average duration of the precopy phase (sec), (c) average duration of the brownout phase (ms), (d) average duration of the whiteout phase (ms), (e) average duration of the blackout phase (sec), and (f) average duration of the post-checkpoint phase (sec)

4.6. Impacts of the imbalance of durations of VM memory page transfers

This section aims to investigate the effect of an imbalance in the memory page transfer duration on the checkpoint performance of Mekha and IMVCCR. In our study, we chose to analyze the experimental results of two representative checkpoint operations, which were conducted on the H2 cluster. Figure 8 presents the trimmed checkpoint phase diagrams of IMVCCR and Mekha for the selected runs, with timelines adjusted to begin at the same point in time and have a consistent timescale.

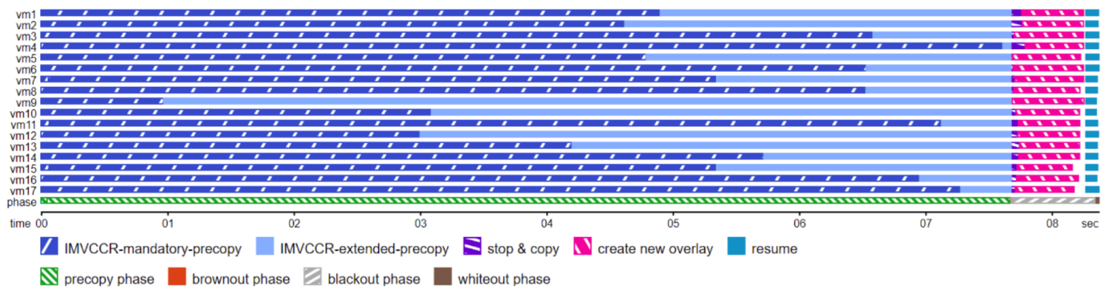
From Figure 8(a), it can be observed that the length of the precopy phase of IMVCCR is equivalent to the longest "IMVCCR-mandatory-precopy" duration, the dark blue lines in the figure. On each VM, this duration starts from the beginning of the precopy operation and ends when it sends a `PRECOPY-CONVERGED` message to the coordinator. The active instance of VM4 shows the longest "IMVCCR-mandatory-precopy" period, which ends when the extended precopy mechanism of IMVCCR estimates the downtime to be lower than 300 milliseconds (the default maximum acceptable downtime).

After a VM sent the `PRECOPY-CONVERGED` message, it will enter an "IMVCCR-extended-precopy" period as illustrated by light blue lines in Figure 8(a). During this period, the hypervisor will continue to transfer dirty pages until they receive an `END-PRECOPY` message from the coordinator. According to global ending conditions of IMVCCR, the coordinator will broadcast an `END-PRECOPY` message if and only if it received the `PRECOPY-CONVERGED` message from every active instance.

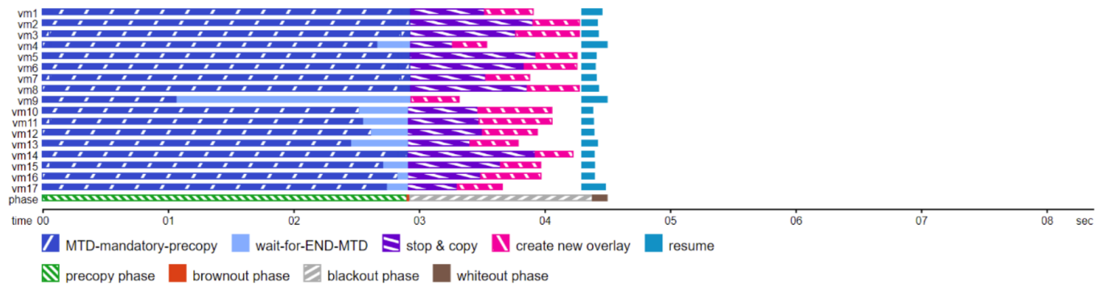
Therefore, VC precopy phase will be prolonged until the completion of the longest "IMVCCR-mandatory-precopy" duration.

On the other hand, the length of the VC precopy phase of Mekha depends on the majority of the "MTD-mandatory-precopy" durations (the dark blue lines in Figure 8(b)) of the active instances. On each VM, this duration starts from the beginning of the MTD precopy operation and ends either when the VM sends an `MTD-EMPTY` message to the coordinator or when it receives an `END-MTD` message from the coordinator. As discussed earlier in Section 2.4, the VC consists of the starter and non-starter active instances. In Figure 8(b), the MTD mechanism of the starter active instances will enter the "wait-for-END-MTD" state (the light blue lines in the figure) after they send the `MTD-EMPTY` message. During this period, the mechanism continues to transfer dirty pages to shadow instances. Since the global ending condition of Mekha is true when the coordinator receives `MTD-EMPTY` messages from the majority of active instances, the length of the VC precopy phase will be as long as either the median value (if the number of VMs is odd) or the highest of the two middle values (if the number of VMs is even) of the "MTD-mandatory-precopy" durations of every active instance. From our analysis, we find the following.

– The global ending condition of Mekha can effectively handle the imbalance of precopy durations among the active instance and reduce VC precopy phase duration substantially. By comparing the duration of the H2 VC precopy phase of Mekha and IMVCCR in Figure 7(b), it can be seen that Mekha's precopy phase using SAN storage is 55.38% shorter than IMVCCR's; this difference can also be observed in visualizations in Figure 8(a) and Figure 8(b).
– While Mekha reduces the imbalance of the precopy durations among the active instances, the blackout duration of Mekha is higher than those of IMVCCR. Since the active instances of Mekha have shorter precopy duration than those of IMVCCR, Mekha has more memory pages left to be transferred from the active to shadow instances during the blackout phase. The experimental results in Figure 7(e) as well as the visualization in Figure 8 support this observation.



(a)



(b)

Figure 8. The trimmed checkpoint phase diagram shows impact of imbalance of the durations of VM memory pages transfer of (a) IMVCCR and (b) Mekha on the H2 cluster

5. RELATED WORK

The Emulab system [21], allows users to roll back and replay applications in a testbed environment. The checkpoint mechanism in Emulab implements a coordinated checkpointing mechanism, in which all virtual machines in the testbed environment are suspended simultaneously before the checkpoint operation begins, and then resumed simultaneously after the operation completes. The Emulab system also introduces the concept of a temporal firewall, which is used to isolate the time inside the guest operating system from that of the physical host. The time virtualization concept of Mekha is similar to Emulab's temporal firewall.

Additionally, while the implementation of the temporal firewall in Emulab requires modifications to the guest operating system's kernel, Mekha's time virtualization leverages the existing timing options of the QEMU software. Emulab also employs time synchronization on all hosts to reduce the effects of clock skew caused by the suspension and resumption of virtual machines. Mekha's rendezvous time synchronization is inspired by Emulab, however, the rendezvous time and detailed implementation are different. In Emulab, every in-flight layer-2 Ethernet frame is saved in Emulab's delay nodes to avoid network data loss, whereas no modifications to the networking components are required in Mekha.

VCCR [33] is another coordinated checkpoint-restart mechanism that uses a barrier synchronization to prevent the occurrence of an inconsistent global state during a virtual machine checkpoint operation. Like Mekha, VCCR also employs virtual time inside the virtual machines. We have described VCCR and used it for performance comparison in Section 4.1 However, a major drawback of VCCR is that it may cause high downtime during a checkpoint operation.

VNsnap [19] is a coordinated checkpointing system that pioneers the use of precopy live migration mechanisms for virtual machine checkpoint-restart operations. At a virtual machine checkpoint event, VNsnap instructs each VM to perform the traditional precopy live migration mechanism, transferring the virtual machine state to a “VNsnap-memory” daemon process while the virtual machine continues running. This daemon operates as an in-memory storage, similar to the shadow virtual machine of Mekha. However, since the traditional precopy mechanism relies on the maximum acceptable downtime parameter, the precopy duration of each virtual machine may differ. To prevent the global inconsistency problem, VNsnap implements a specialized virtual network infrastructure, the VIOLIN network, which uses Mattern's algorithm [35] to maintain global consistency by dropping some layer-2 Ethernet frames. However, this can cause the TCP backoff problem between virtual machines, resulting in long delays in MPI communication during and after virtual machine checkpoint operations. To mitigate this issue, VNsnap stores and re-injects the layer-2 Ethernet frames that do not create an inconsistent global state during a cluster-wide checkpoint operation in buffers provided by the virtual network infrastructure. However, in some cases, storing and re-injection of the layer-2 Ethernet frames are ineffective, such as re-injecting the frames after the retransmission mechanism has begun. On the other hand, Mekha does not control the Ethernet frames in the virtual network and lets the TCP handle the occurrence of lost frames. Since VNsnap uses the message coloring algorithm to create a consistent checkpoint, the hypervisor needs to check all inbound layer-2 Ethernet frames during a virtual machine checkpoint event. Therefore, it creates significant overhead, especially for a communication-intensive application.

SyncSnap [24] attempts to alleviate the layer-2 Ethernet frames retransmission problems by suspending all virtual machines at the same time to minimize the TCP backoff problems. To do this, it needs to adjust the transfer speed of the precopy algorithm of each virtual machine in the cluster. However, finding an appropriate transfer speed for arbitrary applications is not a trivial task. Adjusting the precopy speed may reduce the overall performance of the virtual machine checkpoint operation and prolong the checkpoint latency. In contrast, Mekha synchronizes the suspension and resumption of virtual machines using the rendezvous time and does not reduce the precopy speed.

IMVCCR is a coordinated checkpoint mechanism that is improved from VCCR. It uses a precopy algorithm to reduce the number of memory pages that need to be transferred after suspending the virtual machine. It uses main memory as transient storage to reduce checkpoint downtime. IMVCCR has two drawbacks: i) the performance of the checkpoint mechanism is poor when the VC checkpoint operation runs into a workload imbalance situation; and ii) Unless the precopy algorithm is manually configured, the precopy algorithm does not stop iterating memory page transfers when virtual machines run computation-intensive or memory-intensive workloads, thus the checkpoint operation will never complete until the workload of virtual machines decreases.

6. DISCUSSION

In this section, we will discuss the costs associated with using Mekha to provide transparent checkpointing capabilities in a cloud data center. We will then argue that these expenses are reasonable. Regarding expenses, Mekha necessitates the allocation of the following additional resources: i) Using main memory as transient checkpoint storage may cause insufficient memory space problems on physical hosts. Mekha requires at least twice the current size of running VMs during a checkpoint operation. As the VMs in a VC are distributed across multiple hosts, the available memory space on some hosts may be insufficient; and ii) In addition, extra CPU cores are required to run the shadow VMs and perform the precopy operation. Before the checkpoint operation, the coordinator must run a shadow VM for each active instance. The shadow VM requires a single thread of operation to receive memory pages from the active instance and save them to the checkpoint storage. During the precopy phase, the hypervisor of each active instance has to spawn a migration thread to perform the MTD precopy algorithm. Most of these tasks operate while the VM is running, which may slow down the VM's execution.

We contend that the cloud management system in the data center can address these issues. Possible solutions include the followings. First, the cloud management system can pause or relocate other VMs that are not part of the VC to other hosts to free up memory space. Second, the cloud management system may move active instances to other hosts that have sufficient memory. Third, the checkpoint coordinator can spawn shadow instances on different hosts. Finally, in the worst case where the remaining memory space is insufficient, the hypervisor may switch to other hypervisor-level checkpoint mechanisms such as VCCR. However, these solutions require further investigation in future work. Additionally, although Mekha requires extra resources to perform a VC checkpoint operation, these resources are only utilized during a VC checkpoint event. They are not constantly being used throughout the application's execution like in the HA solution such as Remus [36]. We believe that if the VC checkpoint operation is performed at an appropriate interval or under a practical operational policy, the costs of additional resources such as those required to run the shadow VM would be acceptable to users. Once again, more research is necessary to explore these issues.

7. CONCLUSION AND FUTURE WORKS

In this paper, we introduce Mekha, a novel hypervisor-level checkpoint-restart mechanism. Mekha is specifically designed to efficiently handle checkpoint and restart operations for clusters of virtual machines or virtual clusters in cloud computing environments. Mekha operates transparently with the guest operating system and applications, allowing users to seamlessly deploy their applications without modification. There are three key design principles that make Mekha a practical and efficient mechanism. First, Mekha employs a novel memory transfer precopy live migration, namely the MTD, mechanism to save the state of each VM in a VC within a memory-bound duration. We also introduce the global ending condition to handle the imbalance of the MTD precopy duration among the VMs. Second, Mekha utilizes barrier synchronizations to provide a globally consistent state for the checkpoints of every VM. The proposed rendezvous time synchronization allows for each VM to perform a VC-wide operation almost simultaneously. Finally, Mekha leverages the existing data retransmission mechanisms of the guest operating system, under the time virtualization principle, to handle message loss during a VC checkpoint operation. Our approach eliminates the need for complex network subsystems or modifications to the message passing interface (MPI) library.

We believe that Mekha is a promising software system for provisioning a checkpoint-restart mechanism in a cloud computing environment. By using Mekha, users can run their MPI applications on a cluster of VMs without modifying the applications of guest operating system since all checkpoint and restart mechanisms are performed at the hypervisor level. In terms of performance, our experimental evaluations have shown that Mekha generates low per-checkpoint overhead and latency compared to other hypervisor-level mechanisms. In future work, we plan to evaluate Mekha under real-world distributed applications and examine the performance of the memory transfer mechanism on VMs running other memory-intensive applications.

REFERENCES

[1] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, "HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–29, Jan. 2018, doi: 10.1145/3150224.

[2] F. Petrini, K. Davis, and J. C. Sancho, "System-level fault-tolerance in large-scale parallel machines with buffered coscheduling," in *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2004 (Abstracts and CD-ROM)*, 2004, vol. 18, pp. 2903–2910, doi: 10.1109/ipdps.2004.1303239.

[3] R. T. Liu and Z. N. Chen, "A large-scale study of failures on petascale supercomputers," *Journal of Computer Science and Technology*, vol. 33, no. 1, pp. 24–41, Jan. 2018, doi: 10.1007/s11390-018-1806-7.

[4] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, Oct. 2010, doi: 10.1109/TDSC.2009.4.

[5] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, Feb. 2013, doi: 10.1007/s11227-013-0884-0.

[6] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, "CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 501–514, Mar. 2019, doi: 10.1109/TPDS.2018.2866794.

[7] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi, "Fault tolerance of MPI applications in exascale systems: The ULFM solution," *Future Generation Computer Systems*, vol. 106, pp. 467–481, May 2020, doi: 10.1016/j.future.2020.01.026.

[8] I. Laguna *et al.*, "Evaluating and extending user-level fault tolerance in MPI applications," *International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 305–319, Jul. 2016, doi: 10.1177/1094342015623623.

[9] Y. Zhang, D. Wong, and W. Zheng, "User-level checkpoint and recovery for LAM/MPI," *Operating Systems Review (ACM)*, vol. 39, no. 3, pp. 72–81, Jul. 2005, doi: 10.1145/1075395.1075402.

[10] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy, 2009, pp. 1-12, doi: 10.1109/IPDPS.2009.5161063.

[11] R. Garg, G. Price, and G. Cooperman, "MANA for MPI: MPI-agnostic network-agnostic transparent checkpointing," in *HPDC 2019- Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, Jun. 2019, pp. 49–60, doi: 10.1145/3307681.3325962.

[12] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," *USENIX 2007 - 2007 USENIX Annual Technical Conference*. pp. 323–326, 2007.

[13] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, pp. 494–499, Sep. 2006, doi: 10.1088/1742-6596/46/1/067.

[14] S. Sankaran *et al.*, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, Nov. 2005, doi: 10.1177/1094342005056139.

[15] T. Naughton, H. Ong, S. L. Scott, O. Ridge, C. Science, and M. Division, "Checkpoint/restart of virtual machines based on xen," *Conference: High Availability and Performance Computing Workshop 2006*, Santa Fe, NM, USA, 2006.

[16] P. Barham *et al.*, "Xen and the art of virtualization," *Operating Systems Review (ACM)*, vol. 37, no. 5, pp. 164–177, Oct. 2003, doi: 10.1145/1165389.945462.

[17] QEMU, "Disk images-QEMU documentation," QEMU Project. https://qemu-project.gitlab.io/qemu/system/images.html (accessed Oct. 11, 2023).

[18] R. Garg, K. Sodha, Z. Jin, and G. Cooperman, "Checkpoint-restart for a network of virtual machines," 2013 *IEEE International Conference on Cluster Computing (CLUSTER)*, Indianapolis, IN, USA, 2013, pp. 1-8, doi: 10.1109/CLUSTER.2013.6702626.

[19] A. Kangarlou, 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, doi: 10.1109/TSC.2011.29.

[20] H. Ong, N. Sarago, K. Chanchio, and C. Leangsuksun, "VCCP: A transparent, coordinated checkpointing system for virtualization-based cluster computing," *2009 IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, USA, 2009, pp. 1-10, doi: 10.1109/CLUSTR.2009.5289183.

[21] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau, "Transparent checkpoints of closed distributed systems in Emulab," in *Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys'09*, Apr. 2009, pp. 173–186, doi: 10.1145/1519065.1519084.

[22] L. Cui, B. Li, Y. Zhang, and J. Li, *HotSnap: A hot distributed snapshot system for virtual machine cluster*. USENIX Association, 2013. Accessed: Oct. 17, 2023. [Online]. Available: https://www.usenix.org/conference/lisa13/technical-sessions/papers/cui

[23] M. Zhang, H. Jin, X. Shi, and S. Wu, "VirtCFT: A transparent VM-level fault-tolerant system for virtual clusters," in *Proceedings of the International Conference on Parallel and Distributed Systems-ICPADS*, Dec. 2010, pp. 147–154, doi: 10.1109/ICPADS.2010.125.

[24] B. Shi, B. Li, L. Cui, J. Zhao, and J. Li, "SyncSnap: Synchronized live memory snapshots of virtual machine networks," in *Proceedings - 16th IEEE International Conference on High Performance Computing and Communications, HPCC 2014, 11th IEEE International Conference on Embedded Software and Systems, ICESS 2014 and 6th International Symposium on Cyberspace Safety and Security, CSS 2014*, Aug. 2014, pp. 490–497, doi: 10.1109/HPCC.2014.82.

[25] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: challenges, techniques, and open issues," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 2, pp. 1206–1243, 2018, doi: 10.1109/COMST.2018.2794881.

[26] D. H. Bailey *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, Sep. 1991, doi: 10.1177/109434209100500306.

[27] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman, "Optimized pre-copy live migration for memory intensive applications," *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WA, USA, 2011, pp. 1-11, doi: 10.1145/2063384.2063437.

[28] F. Bellard, "QEMU, a fast and portable dynamic translator," *USENIX 2005 Annual Technical Conference*. pp. 41–46, 2005.

[29] M. Allman and V. Paxson, "Computing TCP's retransmission timer," *Network Working Group-RFC 2988*, pp. 1--8, Jun. 2000, doi: 10.17487/RFC6298.

[30] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, Feb. 1985, doi: 10.1145/214451.214456.

[31] W. Goralski, "Transmission control protocol," *The Illustrated Network*, pp. 279–300, 2009, doi: 10.1016/b978-0-12-374541-5.50017-1.

[32] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A.Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015.

[33] C. Pechwises and K. Chanchio, "A transparent hypervisor-level checkpoint-restart mechanism for a cluster of virtual machines," *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, Nakhonpathom, Thailand, 2018, pp. 1-6, doi: 10.1109/JCSSE.2018.8457176.

[34] J. Yaothanee and K. Chanchio, "An in-memory checkpoint-restart mechanism for a cluster of virtual machines," in *JCSSE 2019 - 16th International Joint Conference on Computer Science and Software Engineering: Knowledge Evolution Towards Singularity of Man-Machine Intelligence*, Jul. 2019, pp. 131–136, doi: 10.1109/JCSSE.2019.8864198.

[35] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Journal of Parallel and Distributed Computing*, vol. 18, no. 4, pp. 423–434, Aug. 1993, doi: 10.1006/jpdc.1993.1075.

[36] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," *5th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2008*. pp. 161–174, 2008.

BIOGRAPHIES OF AUTHORS

Jumpol Yaothanee is a distinguished senior software engineer with a robust educational foundation and a wealth of professional experience. He earned his Bachelor of Engineering degree in computer engineering from Mae Fah Luang University in 2008, where he demonstrated exceptional aptitude in software development and problem-solving. Building upon his solid undergraduate education, he pursued his passion for computer science by obtaining a master of science degree from Thammasat University in 2011. His commitment to excellence and continuous learning has consistently propelled him to the forefront of the software engineering field, making him an invaluable contributor to cutting-edge technology and innovation. He can be contacted at email: lopmuj@gmail.com.

Kasidit Chanchio is a lecturer at the Department of Computer Science, Faculty of Science and Technology, Thammasat University, Thailand. His research interests are in the areas of live migration, checkpoint-restart, fault-tolerance, cluster computing, computer and network virtualization, cloud computing, and parallel and distributed computing. Before joining Thammasat University, Dr. Chanchio is a research staff member at the Computer Science and Mathematics Division at the Oak Ridge National Laboratory, USA from 2001 to 2004. He received PhD in computer science from the Louisiana State University, USA in 2000. He can be contacted at email: kasiditchanchio@gmail.com.