Date of submission 24 04,2025

Digital Object Identifier

Mem-Finer - Enhancing System Checkpoints for Seamless Fault-Tolerant Live Virtual Machine Migration

PASINDU FERNANDO, DINUNI FERNANDO
University of Colombo School of Computing, Reid Avenue, Colombo 00700

ABSTRACT Fault-Tolerant Live Virtual Machine migration is utilized in Cloud Computing to ensure smooth and risk-free migrations, ensuring the data and operations of the running virtual machines are kept intact. To mitigate the risks of the failures that could arise during the migration periods, that could lead to loosing the virtual machine, due to the virtual machine's data and states being split among the source/destination servers and the network cables during transit, checkpoints/snapshots are utilized. Incremental checkpointing utilizes an iterative fashion where the memory content is checkpointed in several iterations to avoid multiple bulky virtual machine image level checkpoints being captured and stored. Proposing "MemFiner" model enhances incremental checkpoints by capturing finer-grain memory state without redundant memory transfers. Additionally, three lossless compression algorithms enabled models, Gzip, Lz4, and Zstd, target compressing the modified pages efficiently before sending them to the checkpoint stores. The four developed models showed mixed results where the Mem-Finer model and LZ4 showed drastic average page content reductions of **51-69%** and **69-72%** respectively, while showing similar and even lesser overhead patterns in migration times, downtimes, checkpoint replication times, performance impact and recovery times, depicting that it successfully enhances the traditional incremental checkpointing approach by reducing the network bandwidth occupied by each virtual machine, ultimately increasing the productivity and cost efficiency of cloud environments. Eventhough Gzip and Zstd models showed good page content reductions of **78-79%** and **50-52%** respectively, it was observed that Gzip showed poor performance in memory write-intensive environments, and Zstd showed poor performance in all the environments tested. The results highlight the overall better efficiency when using the Mem-Finer and Lz4 models, and the poor efficiency showed by Gzip and Zstd under the specific workloads.

INDEX TERMS Fault tolerance, Virtual Machine Migration, Checkpoints, Compression, Page Delta

I. INTRODUCTION

Virtual Machine (VM) migration is a key factor in the field of Cloud Computing, where the VMs running on physical nodes are transferred from one source node to a destination node in the instance of failures, maintenance needs, consolidation purposes, etc. Live VM migration speaks out as the most widely used migration technique where the VM is transferred from the source to the destination in a lively manner while the VM and its applications remain still usable with seamless experience from the perspective of the VM user during the period of migration [1]. During this procedure, the VMs should be preserved while the data and states of the VM should be maintained as it is.

During this live migration procedure, the running VM possesses a risk of getting corrupted or lost in the event of the failures that could occur at the source node, destination node, or even the network connecting them within the period of migration [2]. Such a loss of the VM can cause severe harm to the respective users, and the cloud service providers will be liable for such losses. To overcome this issue, VM checkpoints are taken and stored prior to and during migrations, to be used in cases of VM losses and failures to recover them, which will reduce the risk of losing the VM together with all the data and information of the running VM.

Checkpoints/Snapshots are versioned entities that contain all the required components to recover a VM to a previous state in the event of any kind of failure, ensuring the consistency and persistence of the data and execution states of the VM. As highlighted in [3], checkpoints can be taken in several levels such as virtual machine level, operating system level, hardware level, compiler level, application level, and runtime library level. However, in this paper, the focus will be the system-level checkpoints that are used in the process of live migration.

Different types of checkpointing mechanisms are used in the migration models available. In parallel to implementing safe and secure checkpointing mechanisms, those mechanisms should have a minimal impact on the application performance, migration time, downtime, and other concerns of the running VM. Additionally, the network bandwidth usage and the costs incurred for such mechanisms should be bearable. However, there are still areas that could be looked into and optimized in the field of VM checkpointing in live VM migration. In this paper, we propose an enhancement to the existing incremental checkpointing approach proposed at [2],[4],[5].

State-of-the-art literature, including [2], [4], [5], have contributed to developing checkpointing mechanisms which are resilient against all the points of failures which could occur during migration, which are, namely the source node, destination node and interconnecting network. With the incremental checkpointing introduced, the number of memory pages transmitted in a checkpoint has drastically reduced. Here, the memory content of the VM is checkpointed in several iterations, where in the first checkpoint, all the memory pages available at that instance are checkpointed, and in the successive checkpointing iterations, only the modified pages are being checkpointed. At each iteration end, the device states, including the CPU execution states, are also being captured and stored. With this mechanism, the network bandwidth is preserved as the amount of data transmitted within networks is decreased. Additionally, the overheads on the computing resources of the source VM are minimized.

However, even though the whole modified page is transferred in successive iterations, the actual modified portion of the respective memory page can be limited to a few bytes out of several kilobytes when considering a page of 4KB. In such cases, transmitting of the entire memory page with a few modified bytes and a load of unmodified bytes is in vain, which consumes additional computational resources and introduces additional network bandwidth.

Additionally, for the relevant mechanism, in addition to going to the level of page-level modifications, the authors haven't explored the possibility of compressing the checkpointing data before transmitting. The data to be transmitted can be further compressed using checkpoint compression algorithms like Gzip, LZ4, Zstd, etc. [6]–[8]. This will further reduce the overheads generated with checkpointing mechanisms. This paper proposes four models which are namely, the Mem-Finer model, which examines the modified page further and extracts the actual modified content and only sends it to the checkpoint store, Gzip model which utilizes the Gzip compression algorithm, Lz4 model which utilizes the Lz4 compression algorithm and Zstd model which uses the Zstd compression algorithm to compress the memory pages before transmitting them to the checkpoint stores respectively.

We present the implementation and the evaluation of the four models in KVM/QEMU virtualization platform[9][10]. Evaluation shows that all the models successfully reduce the average page content transferred to the checkpoint stores by a large factor. The results also emphasizes the efficiency of the Mem-Finer and Lz4 models to have similar and even lesser overhead patterns when considering the metrics migration time, downtime, performance degradation, replication time, recovery time, and the problematic situations faced by the Gzip and Zstd models when compared with the baseline PostCopyFT model[2], which uses traditional incremental checkpointing.

II. BACKGROUND

A. LIVE MIGRATION TECHNIQUES

There are several live migration techniques discussed and used in the world of Cloud Computing. The techniques differ in the order in which the memory and the CPU states are transferred from the source-node to the destination-node and the steps carried out in each of them. There are three main live VM migration techniques available in the state of the art literature which are namely,

1) Pre-copy Migration
2) Post-copy Migration
3) Hybrid Migration

The above-mentioned three techniques will be analyzed in detail in the next sections II-A1 - II-A3.

1) Pre-copy Migration Technique

The technique of pre-copy migration is one of the main approaches used in the context of live VM migration. If briefly emphasized, when a migration is needed, the source node continues to execute the VM while the memory content of the VM will be copied to the destination node in several iterations which at the end, the CPU states, device states, together with rest of the dirty pages remaining are sent to the destination and the VM is hand-overed to start it's execution at the destination [1][11]. Pre-copy is efficient in memory-read intensive environments where the iterations continue until the memory content is converged, and the transferring of the execution states of the CPU is initiated when the estimated time where the VM will be down(downtime) is becoming less than a predefined threshold value.

2) Post-copy Migration Technique

As proposed in [12], post-copy migration uses a strategic way where the CPU states are sent to the destination node first and the VM is set to resume at the destination node as the first step. Successively, the source memory pages are copied to the destination-node in an active manner. This approach faces a problem when the memory pages which are not yet been copied to the destination-node are being accessed. In such an instance, the post-copy technique utilizes *on-demand paging*, where a page fault is triggered and the respective page is being requested from the source and received. Post-copy lies ahead of pre-copy in this context as a memory page is transmitted a maximum of once through the networks in post-copy, but in pre-copy, the same memory page can be transmitted multiple times which results in a higher network traffic [12].

3) Hybrid Migration Technique

As explored in the above sections II-A1 and II-A2, both pre-copy and post-copy have some inherent pros and cons. A hybrid approach has been proposed in [13], which is a mixed approach of both pre-copy and post-copy. As highlighted in [13], in this hybrid approach, a subset of memory which is called *"working set"* that is frequently accessed by the VM is transferred in addition to the CPU execution states and device states as mentioned in post-copy migration. This will yield to produce a lesser number of page faults as already a most frequently accessed set of memory is already available in the destination VM since the initial stages of the migration.

B. CHECKPOINTING

As highlighted in [3], a large portion of the checkpointing techniques are focussed on live migration. VMware Vmotion [14] and Xen live migration presented in [11] both focus on dirty page transfer between the source node and the destination node focusing the load balancing and fast VM recovery in the aspect of migration. VNsnap [15] and CEVM [16] are some disk-based checkpointing techniques that use copy on write mechanism to create replicated images of a VM focussing minimizing the downtime. The incremental checkpointing approach [2] concerned in this research uses a strategic way where, in the successive checkpointing iterations, only the modified pages will be considered and transmitted as shown in fig 1.

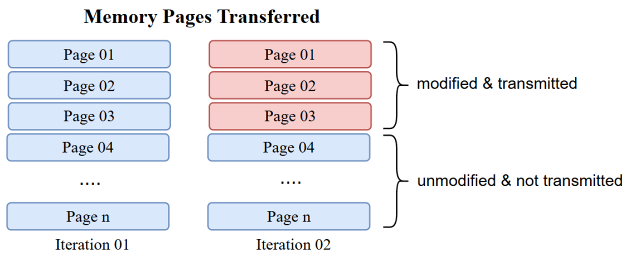


FIGURE 1. Transmitting of only the modified pages in successive checkpointing iterations in traditional incremental checkpointing introduced in [2]

III. DESIGN

The design is mainly divided into two parts to address the two major objectives of making the checkpointing data more granular by introducing a methodology to track the modified content inside a modified page, and investigating the possibility of using checkpoint compression algorithms to further reduce the data to be transmitted as checkpoints.

A. MEM-FINER MODEL

Earlier in the PostCopyFT baseline model [2], without considering the nature of the modified content, if the page was marked as modified, it was transmitted in the checkpoint. With the "Mem-Finer" optimized approach introduced, a successful way of identifying the modified content inside the modified page was successfully designed and integrated to the incremental checkpointing mechanism introduced in the PostCopyFT model. The Mem-Finer model proposed, in contrast is capable of extracting the actual modified data within a modified page, and to transmit that modified portion in an encoded format to the checkpoint store.

Algorithm 1 Mem-Finer : Fine granular memory checkpointing

```
1: while checkpointing is active do
2:     Initialize checkpointing iteration
3:     modified_pages ← ∅
4:     while scanning for modifications do
5:         Scan memory pages for modifications
6:         if modified_page found then
7:             page ← modified_page
8:             older_version ← RetrieveFromPageStorage(page)
9:             if older_version exists then
10:                delta ← XOR(page, older_version)
11:                encoded_delta ← ZeroRunLengthEncode(delta)
12:                SendToCheckpoint(encoded_delta)
13:                UpdatePageStorage(page)
14:            else
15:                UpdatePageStorage(page)
16:                SendToCheckpoint(page)
17:            end if
18:        end if
19:    end while
20:    Wait until the periodic time elapses
21:    Move to next checkpointing iteration
22: end while
```

Here this proposed methodology as explored algorithm 1, it will act in a way where, when a migration is initiated, a local in-memory data structure will be created for the purpose of storing the pages which are being sent, to be used in the future to analyze the actual content which were modified. When the checkpointing is initiated, in incremental checkpointing proposed by [2], it repeatedly checks the entire RAM of the VM to identify the modified pages and considers only the modified pages since the last iteration during the current iteration. When the modified pages are found, with the help of the page offset, the older version of that particular page stored in the local in-memory page storage is being retrieved. In the case of an older version of the page not being available, that page will be entered into the in-memory page storage structure, and it will be treated as normal and sent to the checkpoint store. If an older version is available, the new and the older versions of the pages are XORed to identify the modified bytes inside the page. Identical bytes will yield zero after the XOR operation, and the modified content will remain as it is. Then, giving priority to the modified content, the XORed version of the page is being encoded using zero run length encoding.

During zero run length encoding, the unmodified part of the page which resulted zeros is treated as a zero run, and that sequence of zero bytes is replaced by the no. of zeros in Unsigned Little Endian Base 128 (ULEB128) format. For the modified bytes or the non-zero runs, it is represented with the length of the non-zero run followed by the modified content. Finally this minimized encoded version is being sent to the checkpoint store instead of the whole 4KB page, and the in-memory page store is updated with the later version of the page for future use.

Algorithm 2 Mem-Finer : Reconstruct Memory with Fine Granular Checkpoints at Recovery

```
1:  hash ← empty page map
2:  for each checkpoint iteration do
3:      pages ← GetPagesFromCheckpoint()
4:      for all (key, encoded_page) in pages do
5:          old_page ← hash[key]
6:          if old_page exists then
7:              Decode(encoded_page, old_page)
8:              hash[key] ← old_page
9:          else
10:             page ← Copy(encoded_page)
11:             hash[key] ← page
12:         end if
13:     end for
14: end for
15: LoadPagesToMemoryFromHash(hash)
16: LoadLatestDeviceState()
```

In the event of a failure, as explored in algorithm 2 at the recovery end, when loading the memory pages, the encoded pages are utilized, and the new page is reconstructed with the help of the encoded version and the respective older page content available. The encoding/decoding is clearly elaborated in section IV-B.

B. COMPRESSION-ENABLED MODELS

To achieve the second goal of applying the compression techniques to reduce the content to be transmitted, the PostCpyFT model proposed at [2] is optimized by integrating a checkpoint compression mechanism before transmitting the pages into the checkpoint store. Here, three compression algorithms were taken into account which are namely Gzip, Lz4 and Zstd.

Before transmitting a modified page to the checkpoint store as explored in algorithm 3, it is compressed using a pure lossless compression algorithm that can be specified by the server administrator.

In the event of a failure, at the recovery end as explored in algorithm 4, when the memory pages are loaded, they are

Algorithm 3 Checkpointing with lossless compression

```
Require: compression_type ∈ {GZIP, LZ4, ZSTD}
1:  while checkpointing is active do
2:      Initialize checkpointing iteration
3:      modified_pages ← ∅
4:      while scanning for modifications do
5:          Scan memory pages for modifications
6:          if modified_page found then
7:              page ← modified_page
8:              if compression_type = GZIP then
9:                  compressed_page ← GzipCompress(page)
10:             else if compression_type = LZ4 then
11:                 compressed_page ← Lz4Compress(page)
12:             else if compression_type = ZSTD then
13:                 compressed_page ← ZstdCompress(page)
14:             end if
15:             if Size(compressed_page) greater than 4096 then
16:                 SendToCheckpoint(page)
17:             else
18:                 SendToCheckpoint(compressed_page)
19:             end if
20:         end if
21:     end while
22:     Wait until the periodic time elapses
23:     Move to next checkpointing iteration
24: end while
```

Algorithm 4 Reconstruct Memory with Compressed Checkpoints at Recovery

```
1:  hash ← empty page map
2:  for each checkpoint iteration do
3:      pages ← GetPagesFromCheckpoint()
4:      for all (key, page_data) in pages do
5:          if IsCompressed(page_data) then
6:              page ← Decompress(page_data)
7:          else
8:              page ← page_data
9:          end if
10:         hash[key] ← page
11:     end for
12: end for
13: LoadPagesToMemoryFromHash(hash)
14: LoadLatestDeviceState()
```

decompressed to the original size and loaded into the memory. Rarely in certain situations, depending on the nature of the data of the memory page, the compressed size of the page will become greater than the original size due to lack of repetitive patterns required by compression algorithms to produce a good result. In such instances, the original page is sent instead of the compressed page to reduce the transfer overheads and decompression overheads at recovery time.

IV. IMPLEMENTATION

The PostCopyFT model, which was proposed at [2] was reproduced and used as the baseline model. The implementations were done in KVM/QEMU [10],[9] based server environments. The baseline model checkpointing works in a way where the migration and checkpointing-related operations are running in different threads in the host server.

A. POSTCOPYFT MODEL IMPLEMENTATIONS

1) Migration start and the use of bitmaps

When the migration command is issued, the source node will send the CPU, I/O states, together with a minimal amount of memory required to boot up the VM at the destination end. When the destination node gets the control of the VM, from that point onwards the checkpointing will be started and several checkpointing iterations will be captured to until the migration is over. During this migration period, the source will be actively pushing the remaining pages from the source end to the destination end, and the destination will always be updated with the newer/dirty pages sent. To keep track of the pages which are being modified bitmaps are being used. There are two bitmaps used which are namely,

1) **Migration bitmap** - Keeps track of the pages which were modified and sent to the destination from the source end.
2) **Checkpointing bitmap** - Keeps track of the modified pages that should be considered in the next checkpointing iteration

The migration and the checkpointing bitmaps will be synced prior to the start of each checkpointing round. The pages which are marked as dirty in the migration bitmap will be synced and reflected in the checkpointing bitmap, and the checkpointing thread will be only considering the modified bits in the checkpointing bitmap when scanning for the modified pages. The pages modified during the period of checkpoint capturing will not be handled by the current checkpointing iteration and they will be synced to the checkpointing bitmap prior to the next immediate checkpointing iteration.

2) Checkpoint versioning

A checkpoint consists of major two parts which are namely,

1) Memory checkpoint
2) Device checkpoint

Memory checkpoint consists of a set of key value pairs, where the key represents the offset of the particular page which is used to uniquely identify the page, and the value represents the data content of the respective page. A set of key value pair respective to a certain iteration will be stored together with a group name of "memoryN", where N represents the respective checkpointing iteration. **Device Checkpoint** consists of the CPU and other device states as key value pairs and they too are grouped with a name of "deviceN", where N represents the respective checkpointing iteration.

Additionally, another key value pair is stored after the completion of the two above mentioned parts which is the last completed checkpoint representing the latest checkpoint iteration number which was successfully captured and committed to the checkpoint store. At the time of a failure, when the recovery is initiated, this value is used to determine the last checkpointing round that should be considered.

B. GRANULAR CHECKPOINTING MECHANISM

As explained in section III-A, two models were created. The first model which is Mem-Finer, integrated a byte-level granularity mechanism, capable of tracking modified content within a modified page and prioritizing only the modified content instead of transmitting the entire page.

To elaborate further with an example, the modified content of a page is identified and encoded as follows:

1) Initial Buffers

Consider two memory buffers: an old buffer and a new buffer, differing only in a few bytes.

Old-Buffer: `75 zeros 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 00 00 11 23 25 4000 zeros`

New-Buffer: `75 zeros 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 20 00 00 11 22 24 4000 zeros`

These buffers possess some smaller modifications inside the memory page of 4KB, as highlighted in the color red. Instead of transmitting the entire new buffer, we encode the differences using a run-length encoding scheme.

2) Encoding Scheme

The encoded format consists of:

- *Zero runs* – Number of consecutive zero bytes that represents an unmodified portion.
- *Non-zero runs* – Run length followed by data bytes that represents a modified portion.

Encoded Buffer (Length: 21 bytes): `4b 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 04 02 22 24`

3) Decoding Process

To reconstruct the new buffer the older buffer is loaded and while scanning the encoded version, the older buffer is updated to reflect the content of the new buffer.

- `4b` – This represents a **zero run** of length 75 bytes in ULEB128 format, indicating that the first 75 bytes in the new buffer are unchanged from the old buffer. Hence nothing needs to be done to the loaded earlier version.

- `0f` – This represents a **non-zero run** of length 15 bytes in ULEB128 format. Following this length byte, we have the actual 15 bytes of modified data (`10-1e`). These 15 bytes are updated in the older buffer.
- `04` – This represents another **zero run**, this time with a length of 4 bytes. These bytes remain unchanged in the new buffer..
- `02 22 24` – This represents a **non-zero run** of length 2 bytes followed by the modified content 22 and 24. These 2 bytes are also then updated in the loaded earlier version of the page.

The remaining 4000 bytes are implicitly unchanged, completing the reconstruction.

Once this action is completed, the initially loaded older version of the memory page will be reflecting the content of the new version of that same memory page which was never sent to the checkpoint store.

Older Buffer (updated during decoding): `75 zeros 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 20 00 00 11 22 24 4000 zeros`

Newer Buffer (never sent to checkpoint store): `75 zeros 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 20 00 00 11 22 24 4000 zeros`

C. COMPRESSION ALGORITHM INTEGRATION

The second model implemented was to integrate checkpoint compression algorithms to reduce the page content transferred as checkpoints. As emphasized in section III-B, three lossless compression algorithms namely Gzip, LZ4 and ZSTD were integrated to the model proposed by [2], to reduce the content to be transmitted. With the compression enabled, the original page content of 4096 bytes were reduced by a large factor preserving the bandwidth to transmit the memory checkpoint pages to the checkpoint store.

V. EVALUATION

A. EVALUATION METRICS USED IN THE STUDY

The improvements to the incremental checkpointing methodology from the byte-level modification tracking Mem-Finer mechanism and the integration of the compression techniques were thoroughly analyzed in terms of the **checkpointing data content reduction/network bandwidth reduction**, **checkpoint replication time reduction**, **total migration time overhead**, **total downtime overhead**, **VM application performance degradation** and **recovery time overhead**. These factors will be compared with the "PostCopyFT" baseline model proposed at [2] which uses page-level modification tracking and transfer during checkpointing. Throughout the evaluations, a VM of 8GB of memory was used. Throughout the evaluations, the term "PostCopyFT" will be utilized to refer to the baseline model.

B. WORKLOAD TYPES TESTED

To measure the metrics as mentioned in section V-A, VMs were migrated under different workloads. To test the system under diverse types of workloads running inside the VM, different behaviours like memory-intensive behaviours, network-intensive behaviours, and CPU-intensive behaviours were simulated in the VMs together with a standard idle nature. Different benchmarks like workingset, sysbench, ycsb, and quicksort were used to simulate different behaviours. The benchmarks used are explored in the next sections V-B1 - V-B4.

1) Workingset

To create memory-intensive environments, "Workingset" is a synthetic benchmark used in the context of VM migration. It is a C language-based benchmark where a defined portion of the main memory of the VM is getting continuously dirtied, simulating a memory-intensive behaviour. To simulate the usage of 25%, 50% and 75% of the memory, three working set sizes, which are 2000MB, 4000MB and 6000MB were used throughout.

2) Sysbench

To create a CPU-intensive nature, the Sysbench benchmark [17] was used where it calculates the prime numbers up to a predefined upper bound. An upper bound value of 5 million was used throughout the experiments.

3) Yahoo Cloud Services benchmark

To create a more real-life simulation with network network-intensive nature, the YCSB benchmark was used, where it simulates an environment of having a postgres database in an external server and continuously doing insert and select operations on the external database, while generating continuous incoming and outgoing network packet flow. To make the operations further aggressive 20 parallel connections were initiated with the external database from the VM and all the connections were parallely reading and writing to the database.

4) Quicksort

Quicksort, a synthetic benchmark, was used to create a CPU-intensive environment and get quantified outputs about the CPU operations done per unit time. This writes random numbers to a defined amount of memory and continuously sorts them using quicksort algorithm and outputs the no. of sorting operations successfully done at each second.

C. HOW AVERAGE PAGE CONTENT IS REDUCED IN CHECKPOINTS

When the average page content reduction is considered, instead of the traditional way of sending 4096 bytes as a whole when a page is marked as dirty, the proposed methodologies work on further reducing the page content as shown in fig 2. The models considered here are the Mem-Finer methodology

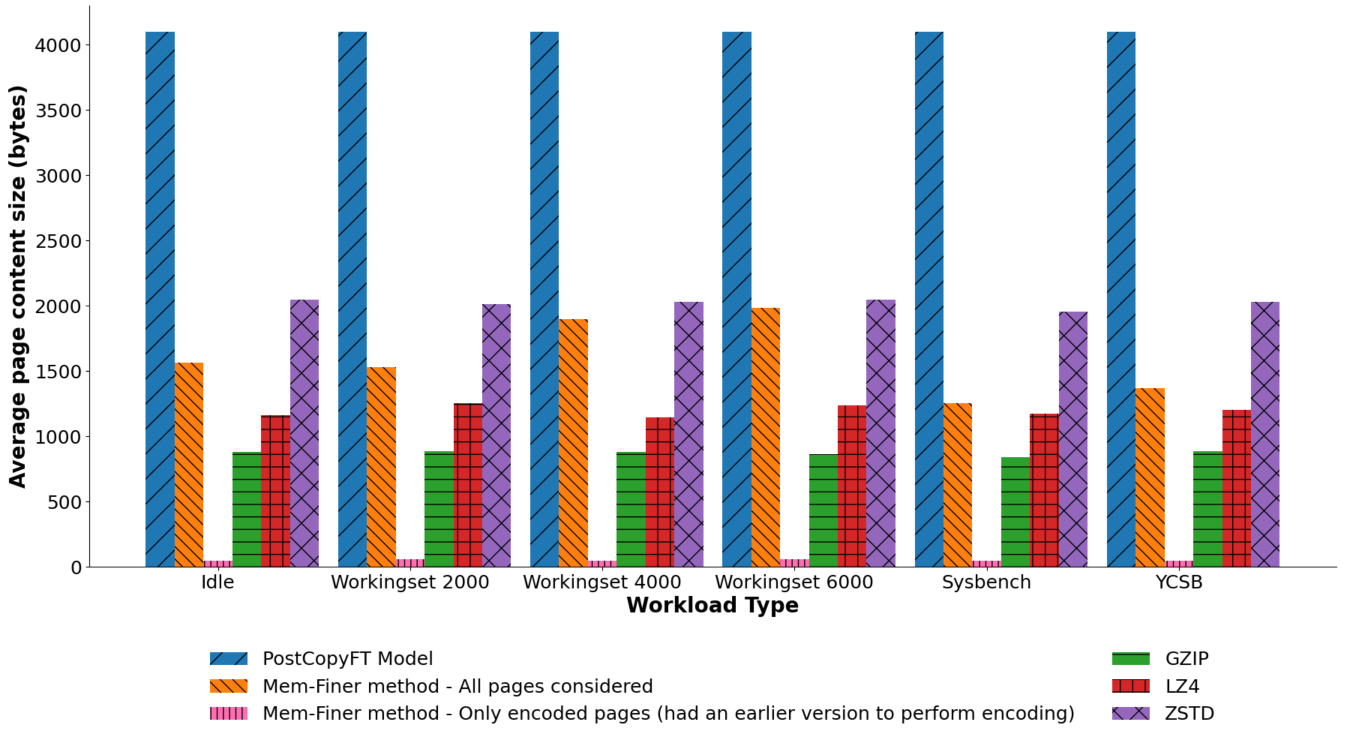


FIGURE 2. **The reductions in average page content transmitted during checkpointing across different models, under different workloads**

where the actual modified content is extracted using a combination of XOR and zero run length encoding, and the three models with lossless compression techniques enabled which are namely, GZIP, LZ4 and ZSTD.

- **Mem-Finer methodology**
  In the Mem-Finer methodology, where the actual modified content is extracted and sent, there are two major metrics to be considered. When a page is found to be dirty, the in-memory page store is searched for any earlier versions of the same page available. If an earlier version is available, the extraction can be successfully done by comparing the two different versions. In the case of an earlier version of the page not being available in the page storage, the encoding cannot be done, hence the full page has to be sent, and the page storage has to be updated with the new page content for future use. When the average page content was analyzed the page content reduction or the redundant page content transfer reduction yielded tremendous results. The experiments were done under different types of workloads, and in each of them the reduction was massive.
  With the Mem-Finer methodology enabled, and all the pages were considered to derive at the average page content, i.e the pages that had an earlier version in the page storage to perform encoding, and the pages that had to be sent as a whole due to not having any earlier version, the average page content reduction was in the range of **51% - 69%** depending on the workloads. In memory-intensive workloads where a selected portion of the memory was continuously dirtied, the reduction was the least and in all other realistic workloads, especially the YCSB workload, the reduction was greater than **65%**.
  The major thing to be highlighted here is, the above-mentioned average result is derived when both the versions of pages, where it was sent as a 4KB chunk and sent as granular encoded versions, are considered. When more focus is given towards the pages that had an earlier version in the in-memory page storage structure to perform encoding, the page content reduction was found to be massive. Across all types of workloads, the reduction showed to be closer to **98%**. This behaviour was visible due to the nature of memory getting accessed in real-life scenarios. Although a page is marked as modified, the actual modifications are really a few bytes in the bigger chunk of 4096 bytes, which represents a page unit. In such a case, this Mem-Finer methodology is capable of properly extracting the exact modified content, removing the redundant memory content transfer on a massive scale.
  Another key thing identified when evaluating this Mem-Finer methodology is that there are instances where the bitmaps represent a page as modified, however, the page is actually not modified. This is a common problem in the context of VM migration and commonly referred to as "**fake dirty pages**". With this Mem-Finer methodology, those pages are also identified and were omitted from transmitting further reducing the page content

transfer.

With the powerful results in terms of the page content reduction, it affects the lowering of the network bandwidth usage and congestion as the content transmitted is lesser than the original baseline model and it also affects in preserving the resources at the checkpoint store due to lessening of the storage space needed to persist a checkpoint.

- **Gzip**
  With the Gzip enabled model, where the modified page is identified and compressed using the Gzip lossless algorithm and sent, the page content reduction was observed to be really high. In the Gzip-enabled model, the average page content reduction was found to be in the range **78% - 79%** depending on the workloads. Gzip showed a steady compression rate regardless of the workload used and it was the best algorithm in terms of average page content reduction, surpassing the LZ4 and Zstd compressions.
- **Lz4** With the LZ4 enabled model, it showed an average page content reduction of **69% - 72%** depending on the workloads. It was observed that the LZ4 compression was the second best in terms of the average page content reduction, which was really close to Gzip, but being far more better than Zstd.
- **Zstd** With the Zstd compression technique enabled, the average page content reduction was shown to be closer to **50%- 52%** depending on the workloads. This compression technique showed the least results out of the compression techniques considered. It was observed that Zstd is not a good fit with small memory chunks, similar to 4KB chunks in this case. It fails to compress them and often results in the compressed lengths being greater than the original lengths. It was clearly visible that Zstd is not a good fit in this context.

With the results being compared with each other, the highest content reduction was possible with the Mem-Finer methodology when an earlier version was available in the page storage to perform encoding which attained a reduction percentage of **98%**. From the compression techniques, the best technique was found to be Gzip, then Lz4 and finally the least with Zstd. It is evident that these models can be used to reduce the redundant memory content transferred as checkpoints, resulting in overall network bandwidth and congestion, together with fewer resources needed at the checkpoint storage locations.

D. TOTAL MIGRATION TIME

When the total migration times were analysed as shown in fig 3, in all the models created and under all types of workloads considered, the overall migration times showed to be really close to each other. The overall migration times increase when the memory-intensive nature of the VMs increases, which is expected in post-copy migrations. However, when comparing with the baseline model, the migration times are shown to be consistent, proving that there is no additional overhead

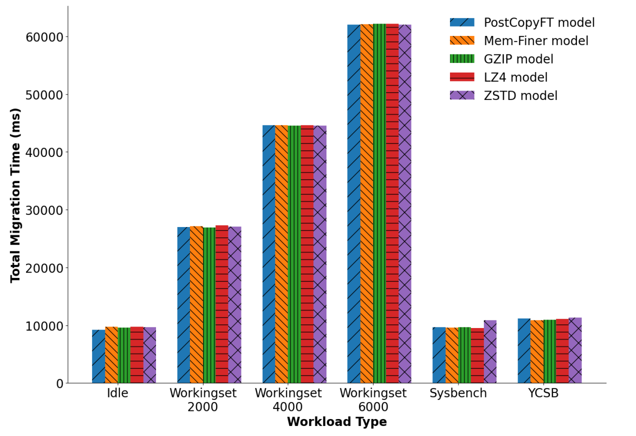


FIGURE 3. Total migration time of migrating different VMs under different workloads, with different models

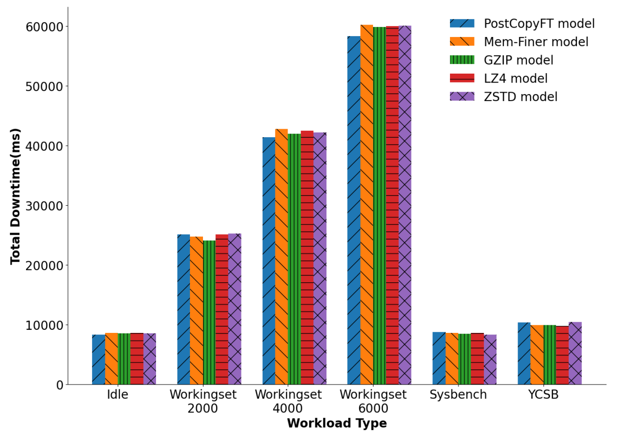


FIGURE 4. Total downtime of migrating different VMs under different workloads, with different models

for the migration times in the proposed Mem-Finer, gzip, lz4 and zstd-enabled models. This was achieved as the migration and checkpoint-related actions were executed in two different threads in an asynchronous manner.

E. DOWNTIME

When considering the downtime across the models developed and different types of workloads as shown in fig 4, the results were shown to be varying. Here, the tests were conducted under an extreme case of where a checkpointing iteration is initiated with the elapse of a periodic time of 1ms. When the overall downtime was considered, the proposed Mem-Finer, gzip, lz4, and zstd models showed little to no difference from the downtime of the traditional baseline model. It highlighted that the downtime overhead is negligible in this context.

However, this represents the total cumulative downtime. With the different models being used, the number of checkpointing iterations is different and within the period of migration, the models capture as many checkpoints as possible until the migration elapses and the checkpointing thread is terminated. Therefore, to further examine the downtime in a more granular manner, the downtimes produced at each round

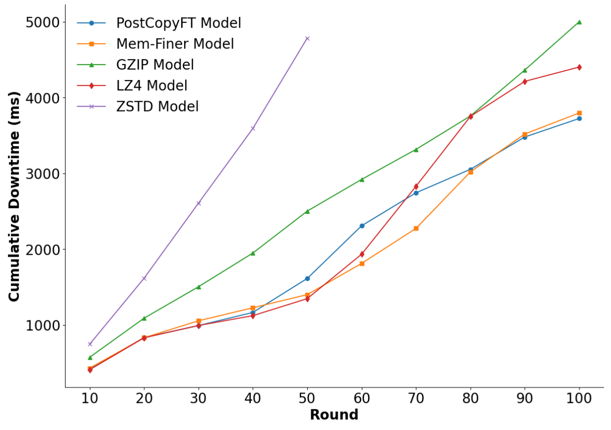


FIGURE 5. The cumulative sum of the overall downtime with the respective checkpointing round, when migrating a VM with an idle workload

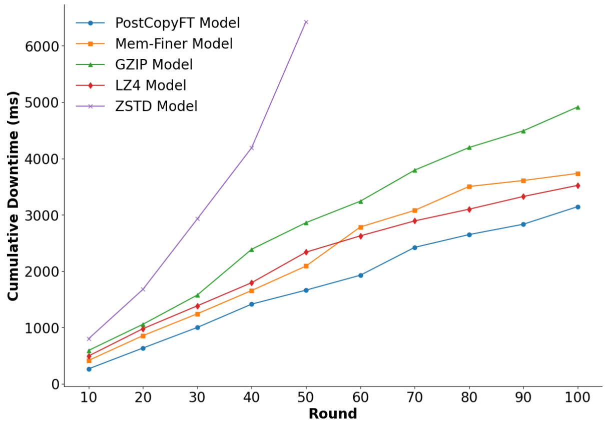


FIGURE 6. The cumulative sum of the overall downtime with the respective checkpointing round, when migrating a VM with Sysbench workload

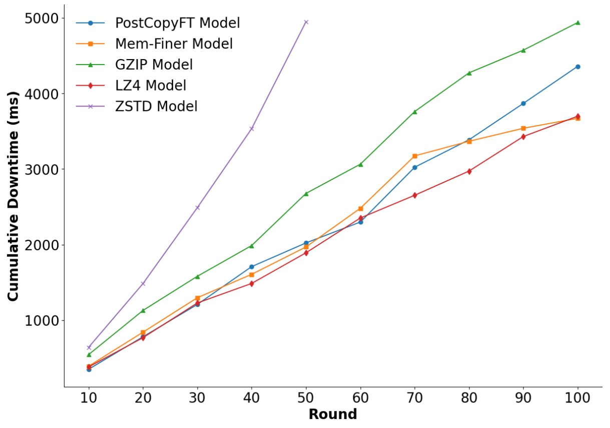


FIGURE 7. The cumulative sum of the overall downtime with the respective checkpointing round, when migrating a VM with an 2000MB workingset workload

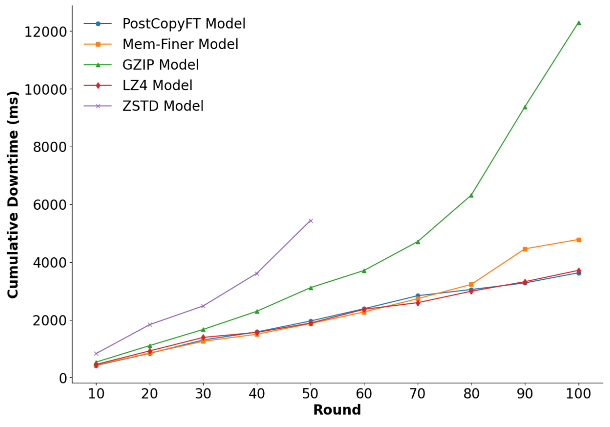


FIGURE 8. The cumulative sum of the overall downtime with the respective checkpointing round, when migrating a VM with an 4000MB workingset workload

individually were analyzed which gave a clear idea of the affect of the proposed models on the downtime of the VM.

It was observed that the zstd model is capable of only capturing around α (for these experiments, around 50-60) checkpoint iterations due to its higher downtime produced by the additional time it spends on compressing the modified pages. It was also identified that the other models which are the Mem-Finer model, gzip and lz4 were all capable of capturing more than β (for these experiments, more than 100) checkpoints due to lesser time taken in encoding and compressing the checkpoints, respectively. When considering an idle workload as shown in fig 5, the zstd model showed an exponential increment in the downtime when compared with the baseline model, which represents that it is not a good fit in this context. All the other models, which are the Mem-Finer model, gzip and lz4 models, showed a linear increment in the downtimes similar to the baseline model. Gzip model showed a higher downtime than the baseline model in the datapoints, however, the Mem-Finer and the lz4 compression models showed a downtime variation really close to the baseline model. It should be noticed that the differences of the downtimes are in a smaller range of 1-2 seconds apart from the zstd model, which grows exponentially. It can be concluded that the best-performing models in terms of downtime here are the Mem-Finer and lz4 models, and a slightly bad version is gzip, and zstd is not a good fit in this context. When considering the sysbench workload, which is CPU intensive as shown in fig 6, similar to the earlier case, Zstd showed bad results by exponentially growing the downtime. When the other models, Mem-Finer, lz4 and gzip were considered, it showed a linear increment similar to the baseline model. Mem-Finer and Lz4 models showed a downtime overhead of less than 1 second across all datapoints when compared with the baseline model, and gzip downtime overhead was slightly higher than the Mem-Finer and lz4 models. Here, it also emphasizes that the Zstd is not a good fit and the overheads of lz4 and Mem-Finer are acceptable and gzip is slightly higher than the better-performing Mem-Finer and lz4 models.

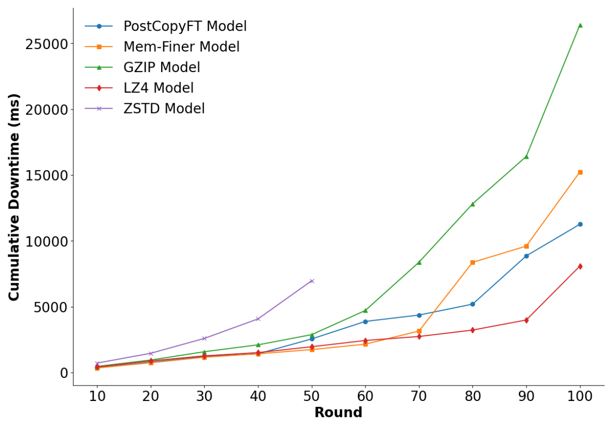


FIGURE 9. The cumulative sum of the overall downtime with the respective checkpointing round, when migrating a VM with an 6000MB workingset workload

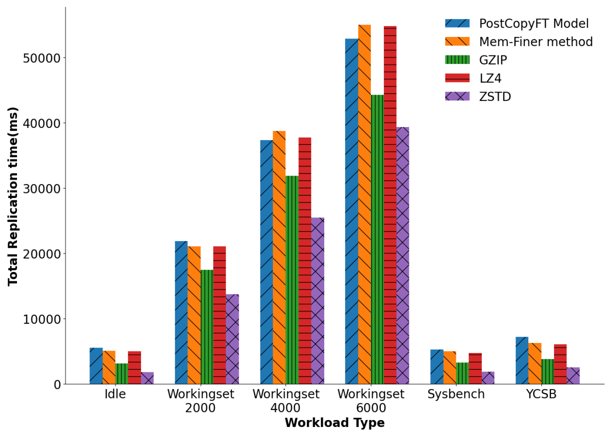


FIGURE 11. Total checkpoint replication time when migrating different VMs under different workloads, with different models

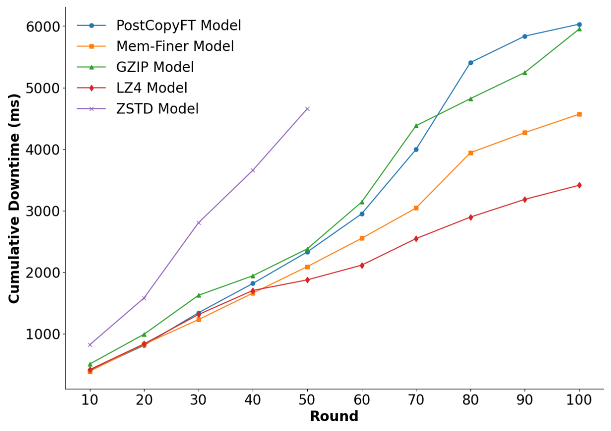


FIGURE 10. The cumulative sum of the overall downtime with the respective checkpointing round, when migrating a VM with YCSB workload

When summarizing the behaviour of the downtimes when the workload running becomes memory-intensive, as explored in fig 7-9, regardless of the working set size, zstd showed exponential increments in the downtime when compared with the baseline model. It was observed that the Mem-Finer and lz4 models showed a closer downtime to the baseline model across all three working set sizes and the downtime overhead is acceptable. However, it should be noticed that Gzip showed to struggle when the working set size is increased which speaks out as not being a good fit in memory-intensive environments. The downtime grows exponentially with the no. of rounds when the memory dirtying rate gets higher. It can be concluded that Mem-Finer and lz4 models produce little to no overhead on the downtime when compared with the baseline model and gzip and zstd is not a good fit in a memory-dirtying environment.

With the YCSB benchmark that simulated a more realistic network-intensive nature as shown in fig 10, still Zstd performance was not good. However the other three models which are namely the Mem-Finer model, gzip, and lz4 showed good results. Mem-Finer and lz4 models were showing downtimes even lesser than the baseline model, while gzip showed some acceptable overheads.

As a summary, the Mem-Finer and lz4 models showed similar downtimes to the baseline model across all workloads and even performed better at certain instances. Gzip had acceptable overheads in idle, CPU-intensive and network- intensive environments, however, it was visible that it struggles with exponential growth in downtimes, when the memory-intensive nature is increased. Across all types of workloads, zstd showed the worst results and the no. of checkpoints generated by it was the minimum. It suggests that Zstd is not a good fit and the best options when it comes to the downtime overheads are the Mem-Finer method and the lz4 compression method.

F. CHECKPOINT REPLICATION TIME

Checkpoint replication time stands for the time taken for the checkpointing data to be propagated to the checkpoint store. When the cumulative replication time is considered, the figure 11 shows a varying nature. It highlights that across all the models considered, the lowest replication time is achieved with zstd compression-enabled model. This is due the lower no. of checkpointing rounds captured due to the higher downtime resulted from zstd compression as explained in section V-E.

The next lowest replication time was achieved with Gzip compression-enabled model, where the average page content reduction was observed to be the highest, as explored in section V-C. The Mem-Finer and lz4 compression-enabled models, showed a replication time really close to the baseline model, and it exceeds the baseline model slightly at certain instances due to generating a larger no. of checkpoint rounds that collectively contribute to the overall replication time. Similar to the analysis of the downtime, the replication times were also examined deeper to identify the patterns at each checkpointing round.

When an idle workload was considered as explored in fig 12, the replication times showed a similar pattern to the

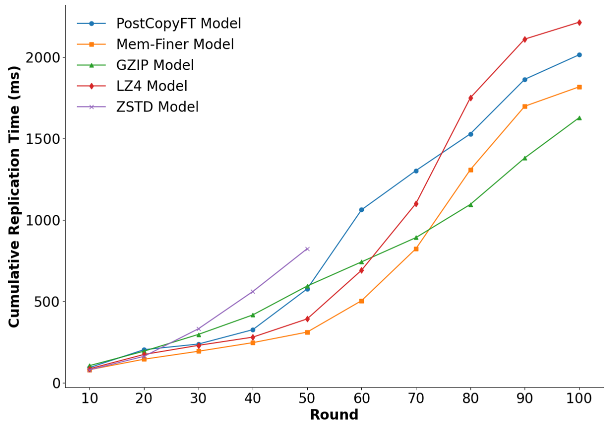


FIGURE 12. The cumulative sum of the overall checkpoint replication time with the respective checkpointing round, when migrating a VM with an idle workload

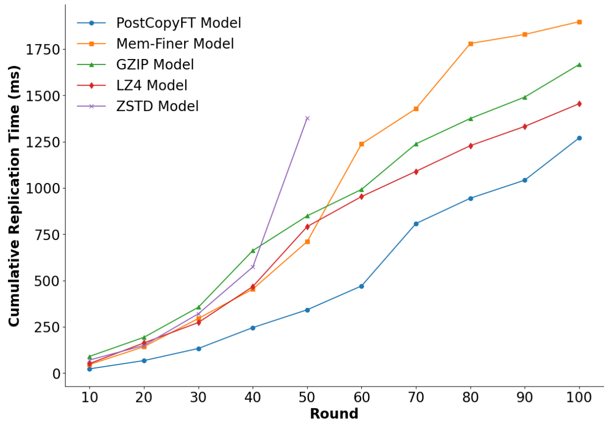


FIGURE 13. The cumulative sum of the overall checkpoint replication time with the respective checkpointing round, when migrating a VM with Sysbench workload

baseline model. Zstd-enabled model shows a replication time greater than the baseline model, and the other three models which are namely the Mem-Finer model, gzip model and the lz4 model, showed a replication time closer to the baseline model and even lesser than the baseline model at most of the rounds.

When the CPU-intensive benchmark sysbench was utilised as explored in fig 13, although all the models showed a pattern similar to the baseline model, apart from the zstd model, which increments exponentially. The Mem-Finer, gzip, and lz4 models followed that pattern of the baseline model with a slight overhead of less than a second at all instances, which is acceptable.

When the memory-intensive workload "workingset" was used, similar to the section V-E, zstd and gzip showed negative results of incrementing exponentially as shown in fig 14=16. It is evident that gzip struggles in memory write-intensive environments resulting in an exponential increment in replication times as well. However, the Mem-Finer and lz4 models goes really close to the baseline model even becoming

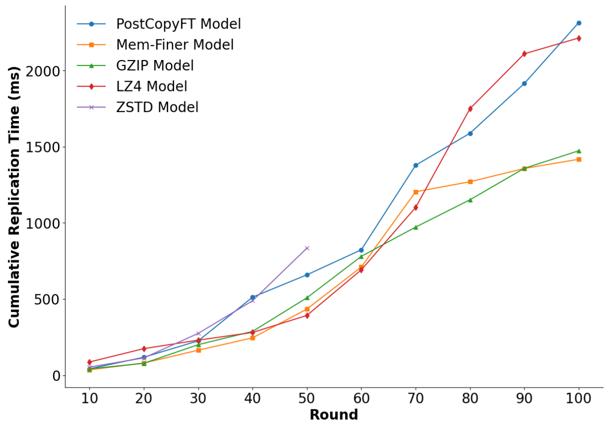


FIGURE 14. The cumulative sum of the overall checkpoint replication time with the respective checkpointing round, when migrating a VM with 2000MB workingset workload

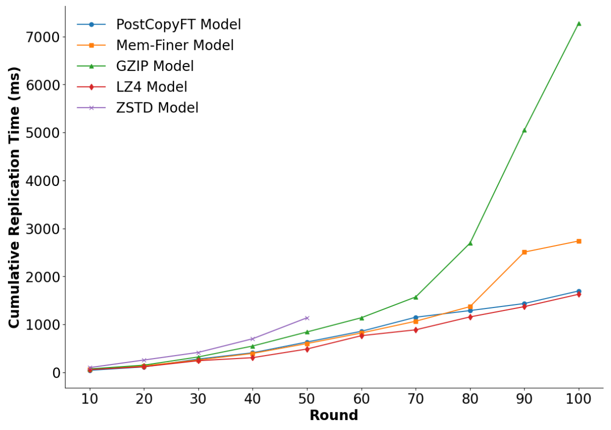


FIGURE 15. The cumulative sum of the overall checkpoint replication time with the respective checkpointing round, when migrating a VM with 4000MB workingset workload

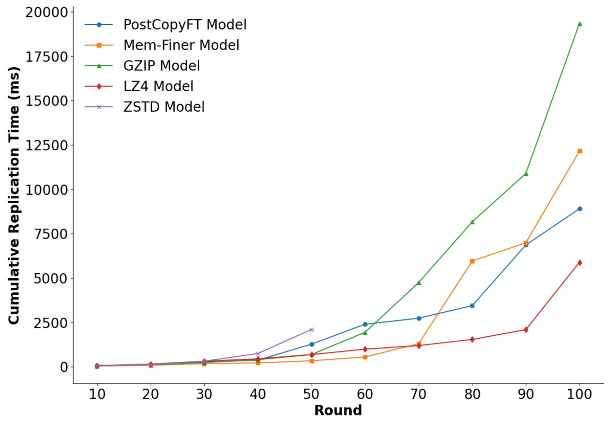


FIGURE 16. The cumulative sum of the overall checkpoint replication time with the respective checkpointing round, when migrating a VM with 6000MB workingset workload

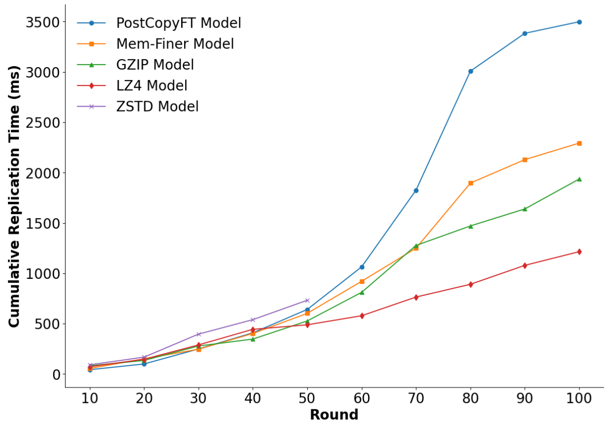


FIGURE 17. The cumulative sum of the overall checkpoint replication time with the respective checkpointing round, when migrating a VM with YCSB workload

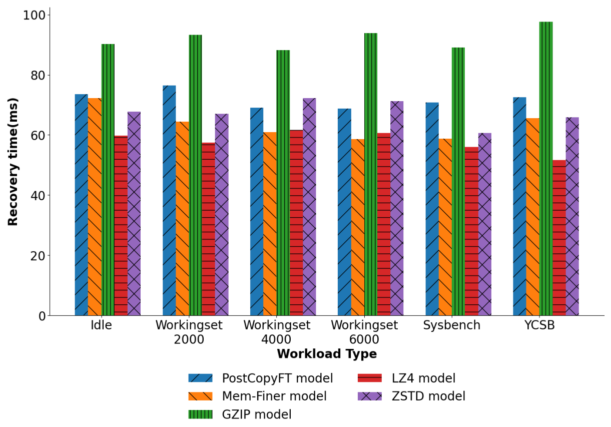


FIGURE 18. The time taken to recover a VM using checkpoints in different models with varying workloads

lesser than the baseline version at some instances.

When a network-intensive workload YCSB was used as shown in fig 17, apart from the zstd model that gave the worst results throughout, all other models which are namely the Mem-Finer model, gzip model and the lz4 model shows a replication time lesser than the baseline model at all instances.

When summarizing the overall replication time statistics across all models, zstd showed the worst results, always resulting in higher replication times, which are even greater than the baseline model. The replication times of the Mem-Finer and lz4 models were following the pattern of the baseline model and even becoming lesser than the baseline model in many instances. Gzip however resulted acceptable patterns apart from the memory-write intensive environments, where it showed signs of struggling. To conclude it can be said that in terms of the replication times, the best performing models are the Mem-Finer and lz4, which give acceptable and even better statistics than the baseline model in most instances. Gzip even is a great fit, given that the workloads are not memory-write intensive. Zstd performs poorly no matter the type of workload running.

G. RECOVERY TIME

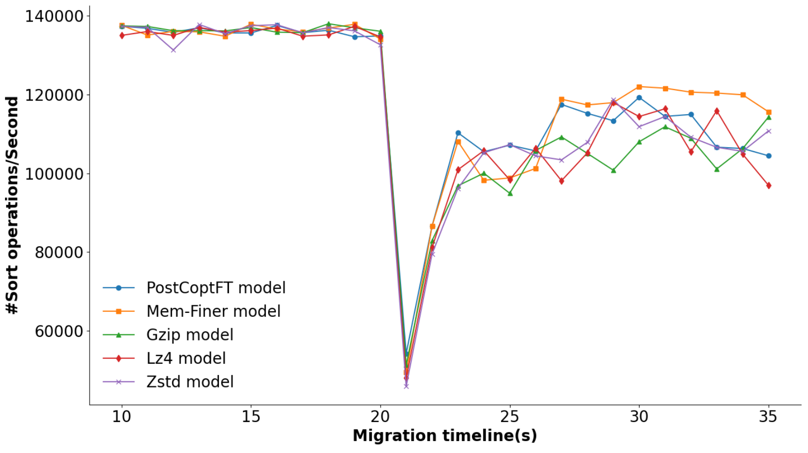
The time to recover the VMs in the instances of failures were also examined as how fast a VM could be recovered and go live in the instance of a failure is a crucial factor to be considered in a real-world Cloud Computing environment. To trigger recoveries, the methodology followed was to initiate the migration and killing the destination when a predefined threshold of pages were committed to the checkpoint store. At the recovery end, when the destination is killed, the source detects it and initiates recovery. The pages retrieved from the checkpoint store are decoded/decompressed in to the original size and loaded into the memory.

When examining the overall recovery times as explored in fig 18, the results observed across different workloads were of varying nature. It was observed that Gzip enabled model showed the highest recovery time, exceeding the baseline model, which represents that the decompression time of gzip is far greater than the other models.It showed 21-36% increment in the recovery time. It was observed that the Mem-Finer and lz4-enabled models showed a lesser recovery time than the baseline model with a reduction percentage of 1-16% and 10-28% respectively, throughout different workloads, which suggests that the decoding/decompression algorithms of the two respective models are really efficient. Finally, Zstd too shows lesser recovery times than the baseline model in most cases, with 7-14% reduction percentage and an increment percentage of 3-4% when the memory write intensive nature is increased. However the performance of Mem-Finer and lz4 at the recovery end, surpasses the performance of Zstd at all workloads.

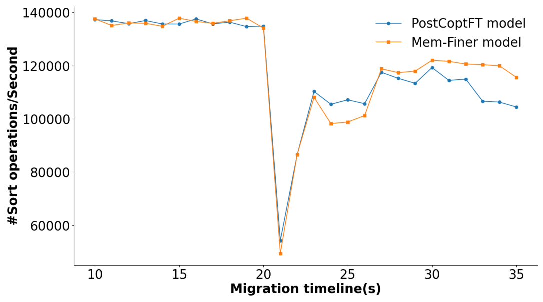
H. APPLICATION PERFORMANCE IMPACT

The impact of the Mem-Finer and compression enabled checkpointing when compared with the baseline model was evaluated using the quicksort benchmark. With this benchmark, a set of random numbers will be generated and assigned to a defined area in the VM to be migrated. Then the numbers will be continuously sorted using the quicksort algorithm and the no. of sorting operations completed at each second will be printed. The evaluation was done by letting the benchmark run for some time prior to the migration and observing the patterns of the no. of operation counts printed at each second, which depicts the performance of the running VM. The benchmark was set to run for 20 seconds and the migration was initiated at the 20th second from the source.

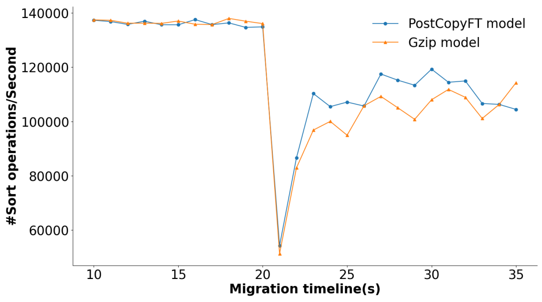
As explored in fig 19, the patterns observed with the Mem-Finer approach and the Gzip, LZ4 and Zstd compression enabled models showed a similarity to the baseline model, depicting that there was little to no additional performance impact incurred from the operations resulting from the improved models. In all the improved models as well as the baseline model, a significant drop was observed at the 20th second which corresponds to the downtime where the VM execution states were transferred from the source to the destination.



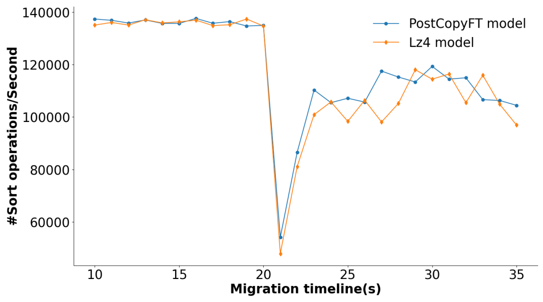
(a) All models: Performance impact analysis on Quicksort benchmark



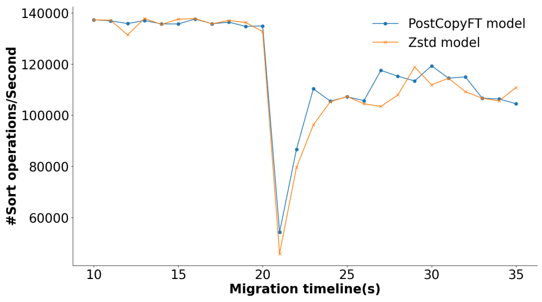
(b) PostCopyFT vs Mem-Finer



(c) PostCopyFT vs Gzip



(d) PostCopyFT vs LZ4



(e) PostCopyFT vs Zstd

FIGURE 19. **Performance impact comparisons between PostCopyFT and other models on the Quicksort benchmark**

Afterwards, the sorting operations counts were observed to be gradually increased, and to remain varying in the range from 100000 to 120000 sorts continuously. The variations of each model when compared with the baseline model will be explored in the below diagrams.

VI. RELATED WORK

This section explores the related literature on VM checkpointing mechanisms, optimization techniques, which utilizes numerous ways to reduce the overheads produced and problems associated with checkpointing in numerous ways.

A. TRADITIONAL CHECKPOINTING APPROACHES

High-Frequency OS Image-Level Checkpointing[18] targets the high availability of the VMs. Here, the approach used was to have fine-grained OS image-level checkpoints taken at really high frequencies and asynchronously replicating them in a failover node. The target was to maintain the VM live to the user, as the checkpoints are asynchornously replicated in the failover node in the background. As mentioned in [18], this methodology requires highly architectural specific environments and the high-frequency checkpointing required considerable computational power and related application logic. **Log-Based Checkpointing**: In contrast to VM-level checkpoints, [19] introduced a traditional **log based checkpointing** mechanism, where the full system trace is logged, and with the help of synchronization algorithms, they are replayed at a secondary node. This reduced the overheads related to bandwidth, down times and migration times, but it possesses strict assumptions such as log transfer and log replay rates to be greater than log accumulation rate, and was limited only to single processor environments.

B. OPTIMIZED CHECKPOINTING INFRASTRUCTURES

Multi-Level Checkpointing: Several studies were carried out to optimize the checkpointing related infrastructure as well. [20] introduced a **multi-level checkpointing** mechanism where several checkpoints are taken at once with varying costs and resiliency levels. Then, according to the resiliency levels, they are stored in different locations such as RAM, disks or parallel file systems, which are accessed according to the level of failure the VMs encounter [21]. **Scalable Checkpoint Restart Mechanisms** proposed at [21], made multi level checkpointing mechanism implementable, which supported checkpoint caching and having the most recent checkpoints updated in the cache and transmitting them to external file systems only when some period elapses. **Efficient Checkpointing with AUFS** proposed at [22], introduced a smarter infrastructure called **AUFS**, which was capable of separating the read-only portions of the checkpoints from read-write parts so that only the read-write parts are needed to be transmitted across the networks.

C. INNOVATIVE CHECKPOINT STORAGE STRATEGIES

Node-Local Non-Volatile Memory (NVM) Checkpointing proposed at [23] introduced a way of using **node local NVM**, as storage units for the checkpoints. Here checkpoints taken at a higher frequency are stored at local NVM, while checkpoints taken at a lower frequency with respect to a periodic time were transmitted to remote locations. **Recycling Checkpoints in Revisited Nodes** proposed by [24] identified that the nodes where the VMs travel are following a pattern. He introduced a method where VM checkpoints are stored in every node where a VM travels and in the event of the same VM being migrated to a previous node that it has visited, the older checkpoint is used in recycling some parts of the VM so that a minimal amount of data needs to be sent again to construct a consistent checkpoint. This was a ground breaking finding which drastically reduced the overheads related with checkpointing. **Micro-Level Checkpoints for Lightweight Migration** : [25] researched with pure software based mechanisms that developed **micro-level checkpoints** which were capable of reducing the weight and overhead of the checkpoints. This methodology targeted the COW mechanism with dirty page prediction and used the node local memory to store the checkpoints for the purpose of in-place recovery of the crashed VMs. This research also paved the way to analyze the checkpoint corruption possibilities and related recovery mechanisms. [25] proposed that the detection latency of the corrupted checkpoints is first to be decided using artificial error injection, and then to minimise the risk of checkpoint corruption, the checkpointing interval was given a value which is greater than the detection latency. **Dual Checkpointing for Corruption Resilience**: The authors of [25] and [26] analyzed the use of **dual checkpointing schemas** to address the problem of checkpoint corruption, where two checkpoints are kept at a single instance. In case of one checkpoint gets corrupted the VM is rolled back to the secondary checkpoint to mitigate the error.

D. COMPRESSION TECHNIQUES FOR CHECKPOINT DATA

Studies were also done regarding the ways to compress the checkpointing data that are transmitted via the networks. [27] analysed the usage of checkpointing techniques like gzip and delta, and concluded that none of the compression techniques are efficient in all kinds of situations. It highlighted that gzip is not suited for CPU-intensive workloads while delta is not efficient with memory intensive workloads. **Gzip** is a widely used compression algorithm that is a combined approach of Lempel-Ziv coding (LZ77) with Huffman coding [6]. This combination allows Gzip to have a relatively high compression ratio, highlighting the effective use of it with data with repetitive patterns which aligns well with memory compression scenarios. [6] also highlighted that this good compression ratio comes with a price of having a relatively high CPU overhead. **LZ4** is another fast compression algorithm with low latency which makes it particularly suitable for real-time data processing applications like live VM migration. LZ4 is a lossless data compression algorithm that is focused on real-time compression and decompression speed over compression ratio and it is based on LZ77 algorithm. LZ4 works by simply spotting and compressing any repetitive data patterns with a much high speed compression and decompression rates according to [7]. LZ4's fast compression can significantly help in minimizing service downtime and improving resource utilization. **ZSTD** or "Z-standard" is another lossless compression technique discussed in the literature. Developed by Facebook, ZSTD uses a combination of dictionary-based compression and Huffman coding, allowing it to achieve higher compression ratios, maintaining competitive speed [8].

Efficient Memory Page Differencing Technique: There have also been studies on how to identify the **page delta** or the memory page difference and efficiently transmit them to the destination during VM migrations. The combined use of XOR with run length encoding as highlighted in [28] is a good finding that shows the possibility of digging deeper to explore the modified content in the a page rather than transmitting the entire page as a whole.

E. INCREMENTAL CHECKPOINTING FOR MIGRATION OPTIMIZATION

Standard Incremental Checkpointing

[4] proposed the approach of having **incremental checkpoints** to reduce the eviction time of the migration. The incremental checkpoints are taken in a way,

- First, all the memory pages(entire pages) are transmitted as a whole.

- In the successive iterations, only the modified pages(entire pages) since the last iteration are transmitted.

Here, the checkpoints are incrementally transmitted to the destination or a third staging node. When the time arrives to migrate the VM, only the remaining memory since the last snapshot is needed to be transmitted which reduced the eviction time drastically. Here the checkpoints are not complete VM images, but portions of the VM memory that will finally contribute to have a consistent memory state at the end with the final memory transmitted at during the eviction time.

Reverse Incremental Checkpointing (RIC) The same authors [2] introduced **reverse incremental checkpointing** which solved a major issue. During postcopy migration, as the execution states are resumed at the destination when the migration starts, if a network or destination node failure occurs, the VM becomes unrecoverable as the execution state is split between the source and the destination. Here soon as the destination starts its execution, incremental checkpoints are generated and stored in a third node or the source node itself, which are utilized in cases of network or destination failures.

Bidirectional Incremental Checkpointing (FIC + RIC) Combining this technology, the same authors [5] introduced a combination of fic and ric, where the checkpoints are taken in both directions i.e. the source VM checkpoints are stored in destination or a third staging node while the destination VM checkpoints are stored in the source or a third staging node. With this mechanism, the VM migrations can be resilient against source, network and the destination failures, which are the only three points of failure present during migration.

VII. CONCLUSION

This study evaluates various methods to optimise checkpointing during live virtual machine (VM) migration, focusing on reducing memory redundancy and improving performance. Among the approaches tested Mem-Finer, Gzip, LZ4, and Zstd, Mem-Finer and LZ4 emerged as the most effective. Mem-Finer significantly reduced redundant memory transfers (up to 98% in ideal cases) and consistently outperformed or matched baseline methods in migration time, downtime, application performance, and recovery. Its ability to identify false dirty pages makes it well-suited for large-scale, failure-prone environments.

LZ4 compression also showed strong results, achieving 69–72% memory reduction with minimal system overhead, making it ideal for general-purpose VM migrations. While Gzip offered higher memory reduction (up to 79%), it suffered under memory write-intensive workloads, and also led to increased recovery times. Zstd performed poorly across key metrics, particularly in latency-sensitive scenarios, suggesting limited applicability in real-time systems. Overall, Mem-Finer and LZ4 present scalable, efficient solutions for enhancing live VM migration in cloud environments.

VIII. ACKNOWLEDGEMENT

The authors gratefully acknowledge the support of the School of Computing, University of Colombo, for providing the computational infrastructure and research facilities essential to this work. We extend our sincere appreciation to WSO2 LLC for their generous donation of cluster-grade hardware, which enabled the experimental validation of our approach.

References

[1] A. Agarwal and S. Raina, "Live migration of virtual machines in cloud," *International Journal of Scientific and Research Publications*, vol. 2, no. 6, pp. 1–5, 2012.

[2] 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 *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 343–351.

[3] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, "Checkpointing virtual machines against transient errors," in *2010 IEEE 16th International On-Line Testing Symposium*, IEEE, 2010, pp. 97–102.

[4] D. Fernando, H. Bagdi, Y. Hu, *et al.*, "Quick eviction of virtual machines through proactive live snapshots," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, 2016, pp. 99–107.

[5] D. Fernando, J. Terner, P. Yang, and K. Gopalan, "V-recover: Virtual machine recovery when live migration fails," *IEEE Transactions on Cloud Computing*, 2023.

[6] S. Sardashti and D. A. Wood, "Could compression be of general use? evaluating memory compression across domains," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 1–24, 2017.

[7] M. Bartík, S. Ubik, and P. Kubalik, "Lz4 compression algorithm on fpga," in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, IEEE, 2015, pp. 179–182.

[8] Y. Collet, *Rfc 8878: Zstandard compression and the'application/zstd'media type*, 2021.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: The linux virtual machine monitor," in *Proceedings of the Linux symposium*, Dttawa, Dntorio, Canada, vol. 1, 2007, pp. 225–230.

[10] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, California, USA, vol. 41, 2005, pp. 10–5555.

[11] C. Clark, K. Fraser, S. Hand, *et al.*, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 2005, pp. 273–286.

[12] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS operating systems review*, vol. 43, no. 3, pp. 14–26, 2009.

[13] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," in *2012 IEEE international conference on cloud computing in emerging markets (CCEM)*, IEEE, 2012, pp. 1–5.

[14] M. Nelson, B.-H. Lim, G. Hutchins, *et al.*, "Fast transparent migration for virtual machines.," in *USENIX Annual technical conference, general track*, 2005, pp. 391–394.

[15] A. Kangarlou, P. Eugster, and D. Xu, "Vnsnap: Taking snapshots of virtual networked environments with minimal downtime," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, IEEE, 2009, pp. 524–533.

[16] K Chanchio, C Leangsuksun, H Ong, V Ratanasamoot, and A Shafi, "An efficient virtual machine checkpointing mechanism for hypervisor-based hpc systems," in *High Availability and Performance Computing Workshop*, 2008.

[17] A. Pokharana and R. Gupta, "Using sysbench, analyze the performance of various guest virtual machines on a virtual box hypervisor," in *2023 2nd International Conference for Innovation in Technology (INOCON)*, IEEE, 2023, pp. 1–5.

[18] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX symposium on networked systems design and implementation*, San Francisco, 2008, pp. 161–174.

[19] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, 2009, pp. 101–110.

[20] E. Gelenbe, "A model of roll-back recovery with multiple checkpoints," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 251–255.

[21] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2010, pp. 1–11.

[22] Í. Goiri, F. Julia, J. Guitart, and J. Torres, "Checkpoint-based fault-tolerant infrastructure for virtualized service providers," in *2010 IEEE network operations and management symposium-NOMS 2010*, IEEE, 2010, pp. 455–462.

[23] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IEEE, 2013, pp. 29–40.

[24] T. Knauth and C. Fetzer, "Vecycle: Recycling vm checkpoints for faster migrations," in *Proceedings of the 16th Annual Middleware Conference*, 2015, pp. 210–221.

[25] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, "Vm-μcheckpoint: Design, modeling, and assessment of lightweight in-memory vm checkpointing," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 2, pp. 243–255, 2014.

[26] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2015, pp. 141–152.

[27] K.-Y. Hou, K. G. Shin, Y. Turner, and S. Singhal, "Tradeoffs in compressing virtual machine checkpoints," in *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, 2013, pp. 41–48.

[28] A. Bhardwaj and C Rama Krishna, "Improving the performance of pre-copy virtual machine migration technique," in *Proceedings of 2nd International Conference on Communication, Computing and Networking: ICCCN 2018, NITTTR Chandigarh, India*, Springer, 2019, pp. 1021–1032.

PASINDU FERNANDO is a final year undergraduate for the B.Sc. (Hons.) degree in Computer Science from the University of Colombo School of Computing, Sri Lanka. He is currently a researcher at the Cloudnet Research Group, UCSC, specializing in cloud computing and virtual machine migration. His research interests include fault-tolerant systems, virtualization, operating systems and cloud infrastructure optimization.

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