



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

H. K. P. S. Fernando
Index number: 20000512

Supervisor: Dr. Dinuni Fernando

April 2025

Submitted in partial fulfillment of the requirements of the
B.Sc in Computer Science Final Year Project (SCS4224)



Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Student Name : H. K. P. S. Fernando

Registration Number : 2020/CS/051

Index Number : 20000512

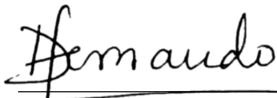


22-04-2025

Signature & Date

This is to certify that this dissertation is based on the work of Mr. H. K. P. S. Fernando under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor Name : Dr. Dinuni K. Fernando



22-4-2025

Signature & Date

Abstract

Fault-Tolerant Live Virtual Machine migration is utilised in Cloud Computing (CC) to ensure smooth and risk-free migrations, ensuring the data and operations of the running VMs are kept intact. To mitigate the risks of the failures that could arise during the migration periods, that could lead to losing the VM, due to the VM'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 VM image-level checkpoints being captured and stored. It captures the main memory content in a series of iterations, where the first checkpointing iteration captures all the memory pages available at that instance, and the successive checkpointing rounds capture only the modified pages when compared with the earlier iterations. This research optimises the incremental checkpointing approach by making the memory capturing more fine-granular. The Mem-Finer model developed, uses a granular methodology to successfully extract the actual modified content within a modified page without repeatedly sending the whole 4096-byte page redundantly. 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 models developed 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 baseline incremental checkpointing approach by reducing the network bandwidth occupied by each VM, 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.

Acknowledgement

I would like to express my deepest appreciation to my supervisor, Dr. Dinuni Fernando, for her exceptional guidance and unwavering support throughout this research. Her expertise, encouragement, and insightful feedback have been pivotal in shaping the direction and success of this work.

I am also immensely grateful to WSO2 for providing the necessary resources and to Dr. Dinuni Fernando for facilitating this collaboration, which has been instrumental in the completion of this project.

A special thank you goes to the staff at the Network Operations Center (NOC) at UCSC for their technical assistance in setting up the servers and providing the infrastructure necessary to carry out this research. Their dedication and expertise ensured the smooth operation of the project from start to finish.

I would also like to acknowledge the members of the research project panel for their valuable time, constructive feedback, and insightful suggestions. Their feedback has greatly enriched the quality of this work.

Finally, I am forever grateful to my family, friends, and colleagues for their continued support, patience, and encouragement. Their unwavering belief in me has been a steady source of strength, and I wouldn't have reached this milestone without their support.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Need for VM migration and checkpoints/snapshots	1
1.1.2	Live VM migration vs Non-live migration	1
1.1.3	Live Migration Techniques	2
1.1.4	Pre-copy Migration Technique	2
1.1.5	Post-copy Migration Technique	3
1.1.6	Hybrid Migration Technique	3
1.1.7	Checkpointing in VM migration	5
1.2	Motivation	5
1.3	Research Gap	6
1.4	Research Questions	7
1.5	Aims and Objectives	7
1.5.1	Aim	7
1.5.2	Objectives	7
1.6	Scope	8
1.6.1	In Scope	8
1.7	Significance of the Research	9
1.8	Outline of the Dissertation	10
2	Literature Review	11
2.1	Overview	11
2.2	Introduction to Checkpointing Techniques	11
2.3	Traditional Checkpointing Approaches	11
2.3.1	High-Frequency OS Image-Level Checkpointing	11
2.3.2	Log-Based Checkpointing	11
2.4	Optimized Checkpointing Infrastructures	12
2.4.1	Multi-Level Checkpointing	12
2.4.2	Scalable Checkpoint Restart Mechanisms	12
2.4.3	Efficient Checkpointing with AUFS	12
2.5	Innovative Checkpoint Storage Strategies	12
2.5.1	Node-Local Non-Volatile Memory (NVM) Checkpointing	12
2.5.2	Recycling Checkpoints in Revisited Nodes	12
2.6	Micro-Level and Fault-Tolerant Checkpointing	13
2.6.1	Micro-Level Checkpoints for Lightweight Migration	13
2.6.2	Dual Checkpointing for Corruption Resilience	13
2.7	Compression Techniques for Checkpoint Data	13
2.7.1	General Checkpoint Compression Techniques	13
2.7.2	Specific Memory Compression Algorithms	13
2.8	Efficient Memory Page Differencing Technique	14
2.9	Incremental Checkpointing for Migration Optimisation	14
2.9.1	Standard Incremental Checkpointing	14
2.9.2	Reverse Incremental Checkpointing (RIC)	14

2.9.3	Bidirectional Incremental Checkpointing (FIC + RIC)	15
3	Design and Methodology	16
3.1	Experimental Testbed	16
3.2	Research Design	17
3.2.1	Identifying the modified page content instead of transmitting the entire page	17
3.2.2	Ability to use checkpoint compression techniques to reduce the content to be transmitted	20
4	Implementation	22
4.1	Baseline model implementation	22
4.1.1	Migration start and the use of bitmaps	22
4.1.2	Checkpoint versioning	22
4.2	Mem-Finer Mechanism	23
4.3	Compression Algorithm Integration	26
5	Evaluation	26
5.1	Evaluation of the accuracy of the implemented methodologies	26
5.2	Evaluation Metrics used for evaluation in the study	26
5.3	Workload types tested	27
5.3.1	Workingset	27
5.3.2	Sysbench	27
5.3.3	Yahoo Cloud Services benchmark	27
5.3.4	Quicksort	27
5.4	Evaluation Criteria	28
5.4.1	How average page content is reduced in checkpoints	28
5.4.2	Total Migration time	31
5.4.3	Downtime	32
5.4.4	Checkpoint replication time	38
5.4.5	Application performance impact	44
5.4.6	Recovery time	47
6	Discussion and Conclusion	49
6.0.1	Different models considered	49
6.0.2	Insights and conclusion	50
7	Limitations and Future Directions	51

List of Figures

1	Pre-copy migration stages (Agarwal & Raina 2012)	3
2	Hybrid migration phases (Choudhary et al. 2017)	4
3	Incremental checkpointing(Fernando et al. 2019)	6
4	Proposed improvement(Fernando et al. 2019)	9
5	TestBed Setup	16
6	Mem-Finer Approach	18
7	Average page content reduction across different models, with dif- ferent workloads	28
8	Total migration time of different models across different workloads .	31
9	Total downtime of different models across different workloads	32
10	Cumulative downtime vs Checkpointing round with an idle workload	33
11	Cumulative downtime vs Checkpointing round with sysbench work- load	34
12	Cumulative downtime vs Checkpointing round with 2000MB work- ingset workload	35
13	Cumulative downtime vs Checkpointing round with 4000MB work- ingset workload	35
14	Cumulative downtime vs Checkpointing round with 6000MB work- ingset workload	36
15	Cumulative downtime vs Checkpointing round with YCSB workload	37
16	Total checkpoint replication time of different models across different workloads	38
17	Cumulative checkpoint replication time vs Checkpointing round with an idle workload	39
18	Cumulative checkpoint replication time vs Checkpointing round with sysbench workload	40
19	Cumulative checkpoint replication time vs Checkpointing round with 2000MB workingset workload	41
20	Cumulative checkpoint replication time vs Checkpointing round with 4000MB workingset workload	41
21	Cumulative checkpoint replication time vs Checkpointing round with 6000MB workingset workload	42
22	Cumulative checkpoint replication time vs Checkpointing round with YCSB workload	43
23	Performance impact across all models	44
24	Baseline vs Mem-Finer performance impact	45
25	Baseline vs Gzip performance impact	45
26	Baseline vs LZ4 performance impact	46
27	Baseline vs ZSTD performance impact	46
28	Total recovery time of different models across different workloads .	47

List of Tables

1	Percentage of Content Reduced Across Different Categories	28
2	Recovery Time Reduction Percentages across Different Workloads .	47

Acronyms

AUFS Another Union File System. 12

CC Cloud Computing. 2, 6, 9, 46

CDC Cloud Data Center. 5–7, 9

COW Copy On Write. 5, 13

FIC Forward Incremental Checkpointing. 15

IaaS Infrastructure as a Service. 5

NVM Non Volatile Memory. 12

OS Operating System. 16

PaaS Platform as a Service. 5

RIC Reverse Incremental Checkpointing. 15

SaaS Software as a Service. 5

ULEB128 Unsigned Little Endian Base 128. 24

VM Virtual Machine. ii, 1, 2, 5–7, 9–16, 19, 22, 26, 27, 29, 31, 32, 43, 46–49

YCSB Yahoo Cloud Services Benchmark. 10, 27, 37, 42

1 Introduction

1.1 Background

1.1.1 Need for VM migration and checkpoints/snapshots

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 (Agarwal & Raina 2012). During this procedure, the VMs should be preserved while 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 (Fernando et al. 2019). 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 and during migrations, to be used in cases of VM losses and failures to recover them which will reduce risk of loosing the VM together with all the data and information of the running VM. 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. Different checkpointing mechanisms have been studied in the state of the art literature throughout the years addressing different aspects and problems associated with checkpointing which are discussed in section 2.2. However, there are still areas that could be looked into and optimized in the field of VM checkpointing in live VM migration.

1.1.2 Live VM migration vs Non-live migration

VM migration refers to the transferring of the VM from one physical host to another seamlessly safeguarding the data, states of execution, device states, etc. of the residing VM. When exploring the concept of migration, there are two main domains identified (Ahmad et al. 2015). They are

- Live migration
- Non-live migration

The process of *non-live migration* as discussed in Milošević et al. (2000), is the process of migrating the VM from the source node to the destination node in an

inactive state. That is when a migration needs to be done, the source node is suspended and the migration takes place. Once the migration is completed and the destination node is ready to go online, the VM execution is fully terminated from the source and resumes at the desired destination. Within the period of migration, the users won't be able to interact with the VM to obtain the services.

In contrast to non-live migration, during *live migration* the VM is continuously in a powered-up state within the whole period of migration (Clark et al. 2005). In parallel to the migration happening, the VM remains still usable from the perspective of the user. Clark et al. (2005) highlights that for a successful proper live migration, the user won't feel any noticeable delays or glitches during the execution of the tasks of the VM. This facilitates the supply of uninterrupted services to the users which is an utmost requirement in the field of CC.

1.1.3 Live Migration Techniques

There are several live migration techniques discussed and used in the world of CC. 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 analysed in detail in the next sections 1.1.4 - 1.1.6.

1.1.4 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 emphasised, 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 the rest of the dirty pages remaining, are sent to the destination and the VM is handed over to start its execution at the destination (Fernando et al. 2019). Here the iteration continues until the memory content is converged, and the transferring of the execution states of the CPU is initiated when the estimated time when the VM will be down(downtime) is becoming less than a predefined threshold value.

The pre-copy migration is structured and broken down into six steps as shown in figure 1 (Agarwal & Raina 2012), (Clark et al. 2005). A possible concern is that how to decide the point at which the page transmission is converged and the *stop_and_copy* phase is to be instantiated. As mentioned in Fernando et al. (2019),

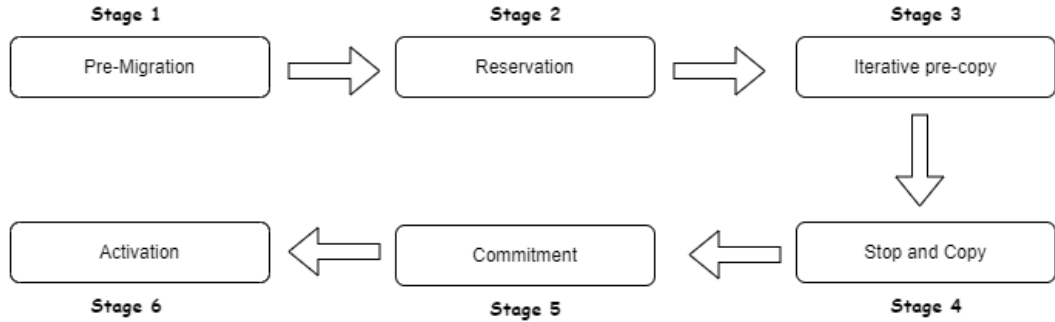


Figure 1: Pre-copy migration stages (Agarwal & Raina 2012)

pre-copy migration is ideal for memory read-intensive tasks rather than write-intensive tasks. If it is a write-intensive task, the transferring of the memory pages has a tendency to never get converged as memory pages are being continuously dirtied, paving the way to not reaching a level below the pre-defined threshold. The post-copy migration that will be discussed next can tackle this problem, and it is discussed in section 1.1.5.

1.1.5 Post-copy Migration Technique

The approach that post-copy uses is different from the approach used in pre-copy. As proposed in Hines et al. (2009), 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 being 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 utilises *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 (Hines et al. 2009). However, post-copy suffers in the context of read-intensive workloads as the no. of page faults will get increased with the increased number of read operations, which will result in VM performance degradation due to the latencies generated for the demand-paging operations.

1.1.6 Hybrid Migration Technique

As explored in the above sections 1.1.4 and 1.1.5, both pre-copy and post-copy have some inherent pros and cons. A hybrid approach has been proposed in Sahni & Varma (2012), which is a mixed approach of both pre-copy and post-copy. As highlighted in Sahni & Varma (2012), 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.

As shown in figure 2, hybrid migration can be mainly structured into five phases as discussed in Choudhary et al. (2017) and Damania et al. (2018). As emphasised

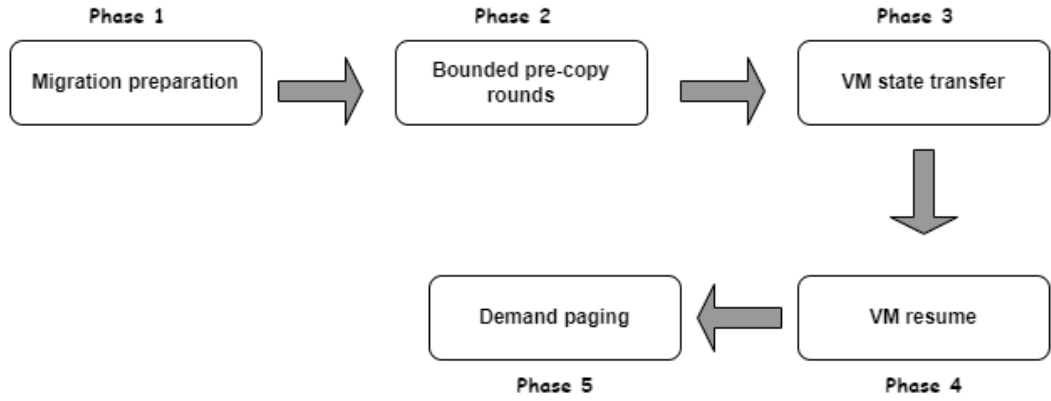


Figure 2: Hybrid migration phases (Choudhary et al. 2017)

in the diagram, the hybrid approach is a combined effort of both pre-copy and post-copy to achieve the target of ultimately reducing migration and downtimes without having noticeable overheads in the application performance.

1.1.7 Checkpointing in VM migration

The term “checkpoints” also referred to as snapshots, comes into play in VM migration failure scenarios. Checkpoints 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 Wang et al. (2010), 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 research, the focus will be on the system-level checkpoints that are used in the process of live migration.

As highlighted in Wang et al. (2010), a large portion of the checkpointing techniques are focussed on live migration. VMware Vmotion (Nelson et al. 2005) and Xen live migration presented in Clark et al. (2005) 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 Kangarlou et al. (2009) and CEVM Chanchio et al. (2008) are some disk-based checkpointing techniques that use Copy On Write (COW) mechanism to create replicated images of a VM, focusing on minimising the downtime.

Depending on the frequency or the interval the checkpoints are being taken, checkpoints are broadly categorised into two sections as mentioned in Fernando et al. (2019).

1. **Periodic checkpointing** which takes checkpoints of the VMs according to a predefined time interval.
2. **Event-based checkpointing**, which does not follow strict time constraints to obtain checkpoints, but follows a method of creating checkpoints with the capturing of several predefined sets of events. With the events fired, a create checkpoint operation is triggered to capture the state of the VM.

1.2 Motivation

With the technology evolving day by day, a major portion of the things related to computing are moving towards the domain of cloud, which falls under one of the three categories SaaS, IaaS or PaaS. The major names resonating when it comes to cloud service providers are Microsoft Azure Microsoft (2024), Google Cloud Platform GCP (2024), Amazon Web Services AWS (2024), Alibaba Cloud Alibaba Cloud (2024) etc., which are giants that offer services to a huge portion of the businesses and industries distributed all across the world. With this huge responsibility at hand, these Cloud Data Center (CDC)s are liable to each customer and the critical operations running within their VMs. In such an environment, with the rapid growth and the demand for Cloud Services, the operations running within them should be optimized, and the resources should be utilised wisely. In the context of VM migration, in parallel to everything running in the VM residing in the nodes, the checkpointing-related operations are utilising the resources

of the same node itself. Another major fact is that the checkpoint sizes increase when the respective VM size increases, making it difficult and time-consuming to replicate the checkpoints via the networks. The checkpoint sizes should be made as small as possible to mitigate this problem. Additionally the migration time and downtime overheads, network bandwidth overheads, application performance degradation overheads, etc., introduced with these checkpointing mechanisms embedded in the migration techniques still have room to be improved. With successful improvements and optimizations, the operations in the CDCs can be made much more productive and cost effective. Such a contribution will be of good value to the domain of Cloud Computing (CC), which is a rapidly growing industry in today's tech world.

1.3 Research Gap

State of the art literature, including Fernando et al. (2016), Fernando et al. (2019), Fernando et al. (2023) have contributed to develop 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 the interconnecting network. With the incremental checkpointing introduced, the amount of memory pages transmitted in a checkpoint have drastically reduced due to the sending of only the modified memory pages since the last iteration.

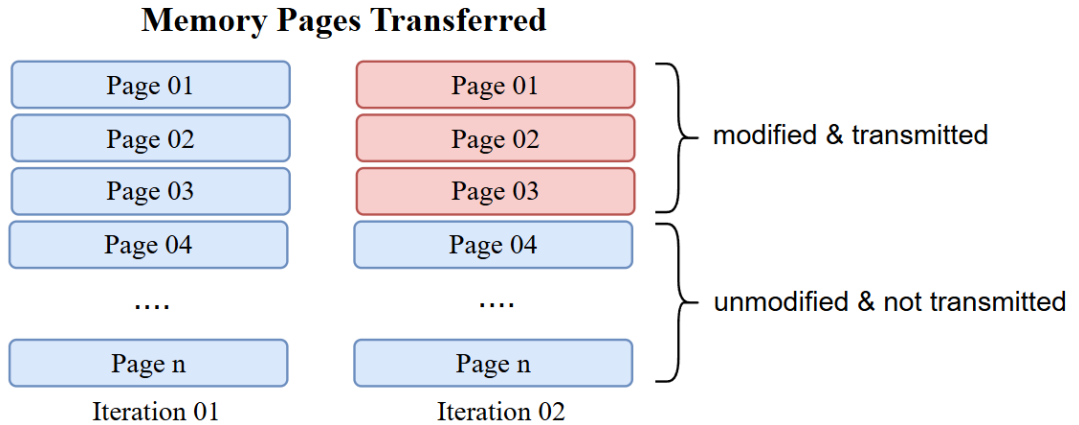


Figure 3: Incremental checkpointing(Fernando et al. 2019)

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 minimised.

However, even though the whole modified page is transferred as shown in figure 3 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 (Sardashti & Wood 2017, Bartík et al. 2015, Collet 2021). This will further reduce the overheads generated by checkpointing mechanisms.

1.4 Research Questions

1. How can the incremental checkpoints be further improved to reduce overheads incurred on VM migration ?
2. How can the state of the art compression techniques be incorporated in reducing checkpoint size during state transfers ?

1.5 Aims and Objectives

1.5.1 Aim

The main aim of this research is to improve and enhance the productivity and cost-effectiveness of incremental checkpointing in VM migration in CDCs by making the checkpointing operations more granular and to analyse the improvements in related overheads with the integration of checkpoint compression techniques.

1.5.2 Objectives

- Design and develop a strategy to make the checkpoint data transmitted more granular for the purpose of reducing the overheads associated with current checkpointing operations.
- Analyse the improvements in the overheads incurred by the new granular methodology with respect to the existing methodologies.
- Integrate checkpoint compression techniques and analyse the improvements made in terms of the overheads introduced.

1.6 Scope

1.6.1 In Scope

- **Baseline model implementation** - The baseline incremental checkpointing model in Fernando et al. (2019) will be reproduced in QEMU/KVM.
- **Byte/Bit-level granular model implementation** - A byte/bit level fine granular checkpointing method will be introduced that goes beyond page level modification tracking and transfer in checkpointing iterations.
- **Implementation of a working prototype** - A working prototype will be implemented with the granular byte/bit-level mechanism.
- **Integration of checkpoint compression techniques** - State of the art checkpoint compression techniques will be integrated to further reduce the checkpoint data transfer overheads.
- **Evaluation** - The results and overhead reductions will be evaluated with approved realistic industrial benchmarks.
- **Scope limited to KVM-QEMU hypervisor** - All implementations will be restricted to the KVM-QEMU hypervisor environment.
- **Single VM failure scenarios** - The focus will be on handling single VM failures, with the potential for future extension to multiple VM failure scenarios.
- **Linux-based Ubuntu OS** - The implementations will be developed and tested exclusively on Linux-based Ubuntu operating systems.

1.7 Significance of the Research

With the CC domain evolving day by day, the need for enhancements and improvements in virtualisation and VM migrations in CDCs is vital. So far in the literature, the incremental checkpointing techniques in VM migration highlighted in Fernando et al. (2016), Fernando et al. (2019) and Fernando et al. (2023) etc. follow an order where the first checkpoint captures all memory pages and the successive checkpointing rounds capture and transmit only the modified pages.

However, when a page of 4 KB is considered, although the page is modified, the actual modifications can be limited to several bytes/bits out of 4 KB. In such a case, transmitting the whole 4 KB worth of data via the networks is an invain of time and resources, which incurs several overheads like network bandwidth, higher migration and downtimes, application performance degradation in running VMs, etc. This mechanism can further be improved to make the data transmitted via the networks more granular. Instead if transmitting the entire page where modifications were made, the page can be examined deeper to identify the respective bytes/bits which were modified, and only the modified bytes/bits can be transmitted without transmitting the entire memory page as shown in figure 4. This will result in further reducing the network bandwidth usage to transmit the checkpointing data to the checkpoint stores, reduce the costs associated with checkpoint stores and significantly reduce the checkpoint replication time to and from the checkpoint stores to the nodes and vice versa. The overheads on the total migration time and downtime, application performance degradation etc., will also be acceptable. This methodology will also reduce the failure window of the migration and the recovery time window.

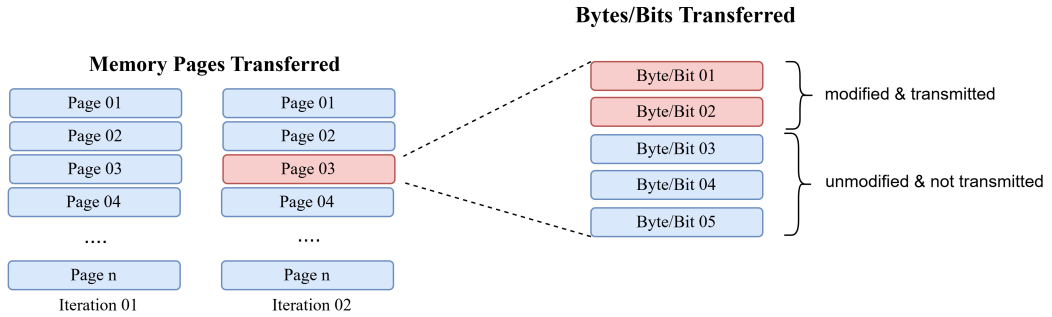


Figure 4: Proposed improvement(Fernando et al. 2019)

With this byte/bit level granularity introduced, only the modified bytes/bits out of the 4KB memory page is transmitted which will be drastically reducing the load of checkpointing data that needs to be transmitted. When a VM bearing around 8GB of memory is considered, the savings resulted through this optimization technique is massive. With this improvement, the VM migration domain will be much more improved, efficient and cost-effective at the same time that leads to a good contribution to the domain of Cloud Computing.

1.8 Outline of the Dissertation

The rest of the dissertation is organised as follows. Section 2 will be exploring the related work to the concept of checkpointing in VM migration and related work for the methodologies used in the research approach. Section 3 will be exploring the experimental testbed setup, research design, which explores the two main branches which are,

1. Granular methodology, which successfully extracts the actual modified content from a page that is marked as dirty.
2. Compression enabled methodology which utilises lossless compression algorithms to compress the modified pages, prior to sending them to the checkpointing stores.

Section 4 will be exploring the implementation details of the proposed models and the algorithms used. Section 5 will analyse the evaluation criteria, comparing the proposed models with the baseline model of Fernando et al. (2019). The analysis will be done under all types of workloads, which are memory-intensive, network-intensive and CPU-intensive environments. Industrial and synthetic benchmarks which are namely workingset, sysbench, YCSB, and quicksort, will be utilised to create the required workloads and environments. The evaluation criteria will be covering how the page content is reduced with the proposed models, how the migration time is affected, how the overall and per-round downtime is observed, how the overall and per-round checkpoint replication time is observed, how the proposed models affect the application performance of the running VMs, and finally how the recovery time is affected with the proposed models.

Section 6 will be summarizing the overall findings of the evaluation and the dissertation will be concluded with the limitations and future directions in section 7.

2 Literature Review

2.1 Overview

For the purpose of exploring the possible gaps in the relevant domain, a comprehensive literature review was done under the topic “**System Checkpoints in Fault-Tolerant Live VM Migration**”. Throughout the state of the art literature, there were numerous approaches taken to optimize the checkpointing mechanisms and to minimise the impact of the additional overheads introduced to VM migration with the checkpointing-related operations. Common issues related with the available checkpointing mechanisms include the application performance degradation of the running VM, difficulty to deal with larger checkpointing file sizes when VMs have large memory sizes, additional cost and time (migration and downtimes) incurred for the checkpointing operations, network bandwidth usage overheads, checkpoint corruption risk and complexity in dealing with secure consistent checkpoints in complex systems with distributed data.

2.2 Introduction to Checkpointing Techniques

Checkpointing is a fundamental mechanism in virtual machine (VM) migration, ensuring high availability and fault tolerance. By periodically saving VM states, checkpointing enables rapid recovery in case of failures. Over the years, various approaches have been proposed to optimise checkpointing efficiency while minimising overhead.

2.3 Traditional Checkpointing Approaches

2.3.1 High-Frequency OS Image-Level Checkpointing

Cully et al. (2008) proposed a VM checkpointing mechanism targeting 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 asynchronously replicated in the failover node in the background. As mentioned in Cully et al. (2008), this methodology requires highly architectural specific environments and the high-frequency checkpointing required a considerable computational power and related application logic.

2.3.2 Log-Based Checkpointing

In contrast to VM level checkpoints, Liu et al. (2009) introduced a traditional **log based checkpointing** mechanism, where the full system trace is logged, and with the help of synchronisation 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.

2.4 Optimized Checkpointing Infrastructures

2.4.1 Multi-Level Checkpointing

Several studies were carried out to optimise the checkpointing-related infrastructure as well. Gelenbe (1976) 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 (Moody et al. 2010).

2.4.2 Scalable Checkpoint Restart Mechanisms

Moody et al. (2010) introduced a **scalable checkpoint restart library**, which 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.

2.4.3 Efficient Checkpointing with AUFS

Goiri et al. (2010) 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.

2.5 Innovative Checkpoint Storage Strategies

2.5.1 Node-Local Non-Volatile Memory (NVM) Checkpointing

Kannan et al. (2013) 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.

2.5.2 Recycling Checkpoints in Revisited Nodes

Knauth & Fetzer (2015) 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 groundbreaking finding which drastically reduced the overheads related with checkpointing.

2.6 Micro-Level and Fault-Tolerant Checkpointing

2.6.1 Micro-Level Checkpoints for Lightweight Migration

Wang et al. (2014) 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 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. Wang et al. (2014) 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.

2.6.2 Dual Checkpointing for Corruption Resilience

The authors of Wang et al. (2014) and Li et al. (2015) analysed 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.

2.7 Compression Techniques for Checkpoint Data

2.7.1 General Checkpoint Compression Techniques

Studies were also done regarding the ways to compress the checkpointing data that are transmitted via the networks. Hou et al. (2013) analysed the usage of compression techniques such as gzip and delta and concluded that none of the compression techniques are efficient in all kinds of situations. He explored that gzip is not suited for CPU-intensive workloads, while delta is not efficient with memory-intensive workloads.

2.7.2 Specific Memory Compression Algorithms

There have been numerous studies on memory compression algorithms that are efficient and lossless. **Gzip** is a widely used compression algorithm that is a combined approach of Lempel-Ziv coding (LZ77) with Huffman coding (Sardashti & Wood 2017). This combination allows Gzip to have a relatively high compression ratio, highlighting the effective use of it with the data with repetitive patterns, which aligns well with memory compression scenarios. Sardashti & Wood (2017) 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 the LZ77 algorithm. LZ4 works by simply spotting

and compressing any repetitive data patterns with a much high speed compression and decompression rates according to Bartík et al. (2015). LZ4’s fast compression can significantly help in minimising service downtime and improving resource utilisation. **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 (Collet 2021).

2.8 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 Bhardwaj & Rama Krishna (2019), is a good finding that shows the possibility of digging deeper to explore the modified content in the page rather than transmitting the entire page as a whole.

2.9 Incremental Checkpointing for Migration Optimisation

2.9.1 Standard Incremental Checkpointing

Fernando et al. (2016) 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.

2.9.2 Reverse Incremental Checkpointing (RIC)

The same authors Fernando et al. (2019) 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.

2.9.3 Bidirectional Incremental Checkpointing (FIC + RIC)

Combining this technology, the same authors Fernando et al. (2023) introduced a combination of Forward Incremental Checkpointing (FIC) and Reverse Incremental Checkpointing (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.

3 Design and Methodology

3.1 Experimental Testbed

An experimental testbed setup with servers configured in a single cluster was used. It consists of two physical servers with Intel Xeon E5-2697 v2 (48) @ 3.500GHz CPUS and each with 314GB of memory. The host OS chosen is Ubuntu Server with QEMU-KVM Bellard (2005) utilised as the virtualisation platform and hypervisor. The VMs running in the servers are configured with Linux-based OSs. The servers are interconnected via Gigabit Ethernet connections with a bandwidth of 1000 Mbps. Additionally, a third server is used as a staging server for checkpointing operations. An Intel Xeon E5503 (4) @ 1.596GHz server with 8GB of memory was used as the staging server for the checkpoint store. It was configured to have a Redis key-value database to act as the checkpoint store. The figure 5 shows the testbed setup.

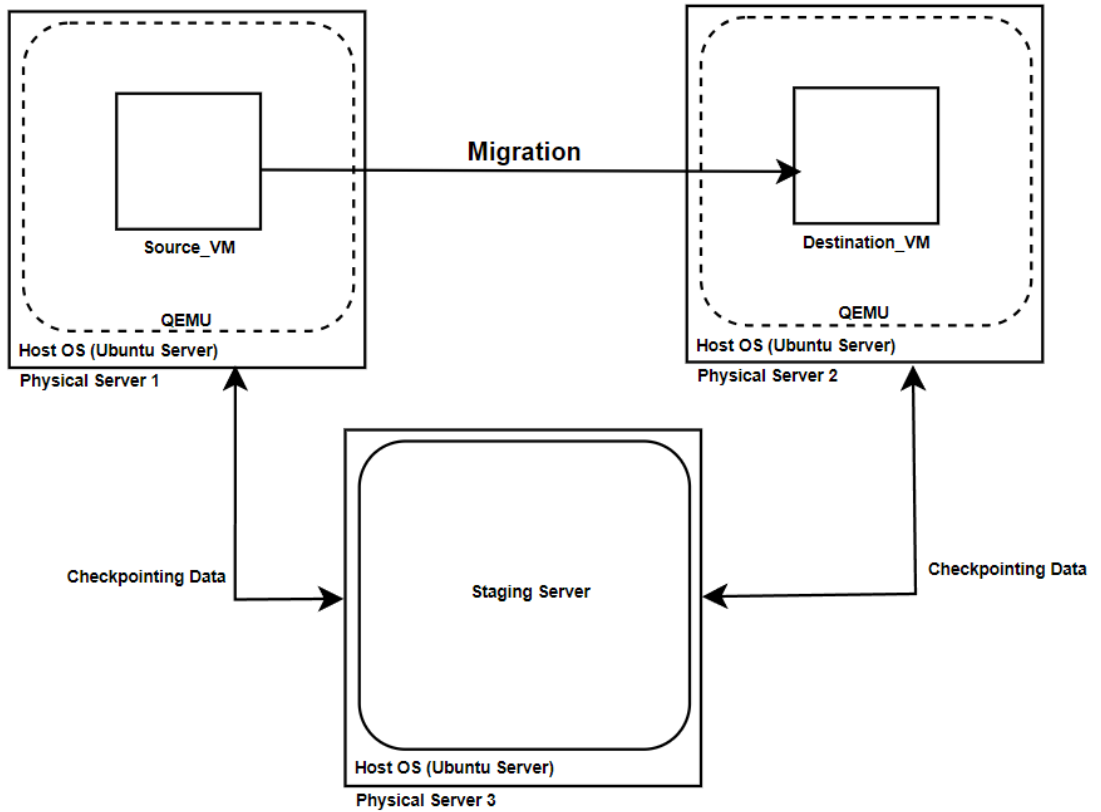


Figure 5: TestBed Setup

3.2 Research Design

The design of the research 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.

3.2.1 Identifying the modified page content instead of transmitting the entire page

As in Fernando et al. (2019), in successive checkpointing rounds, the modified pages since the last completed checkpointing iteration are only considered. Earlier in the baseline model of Fernando et al. (2019), without considering the nature of the modified content, if the page was marked as modified it was transmitted in the checkpoint. With the optimised 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 Fernando et al. (2019). The architecture of the newly proposed Mem-Finer methodology is shown in the below diagram 6 and explored in the below algorithm 1.

Algorithm 1 Mem-Finer : Fine granular memory checkpointing

```
1: while checkpointing is active do
2:   Initialize checkpointing iteration
3:   modified_pages  $\leftarrow \emptyset$ 
4:   while scanning for modifications do
5:     Scan memory pages for modifications
6:     if modified_page found then
7:       page  $\leftarrow$  modified_page
8:       older_version  $\leftarrow$  RetrieveFromPageStorage(page)
9:       if older_version exists then
10:        delta  $\leftarrow$  XOR(page, older_version)
11:        encoded_delta  $\leftarrow$  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
```

As proposed in the figure 6, and explored in the algorithm, the model will be capable of tracking the modified content inside the modified pages, and only transmit

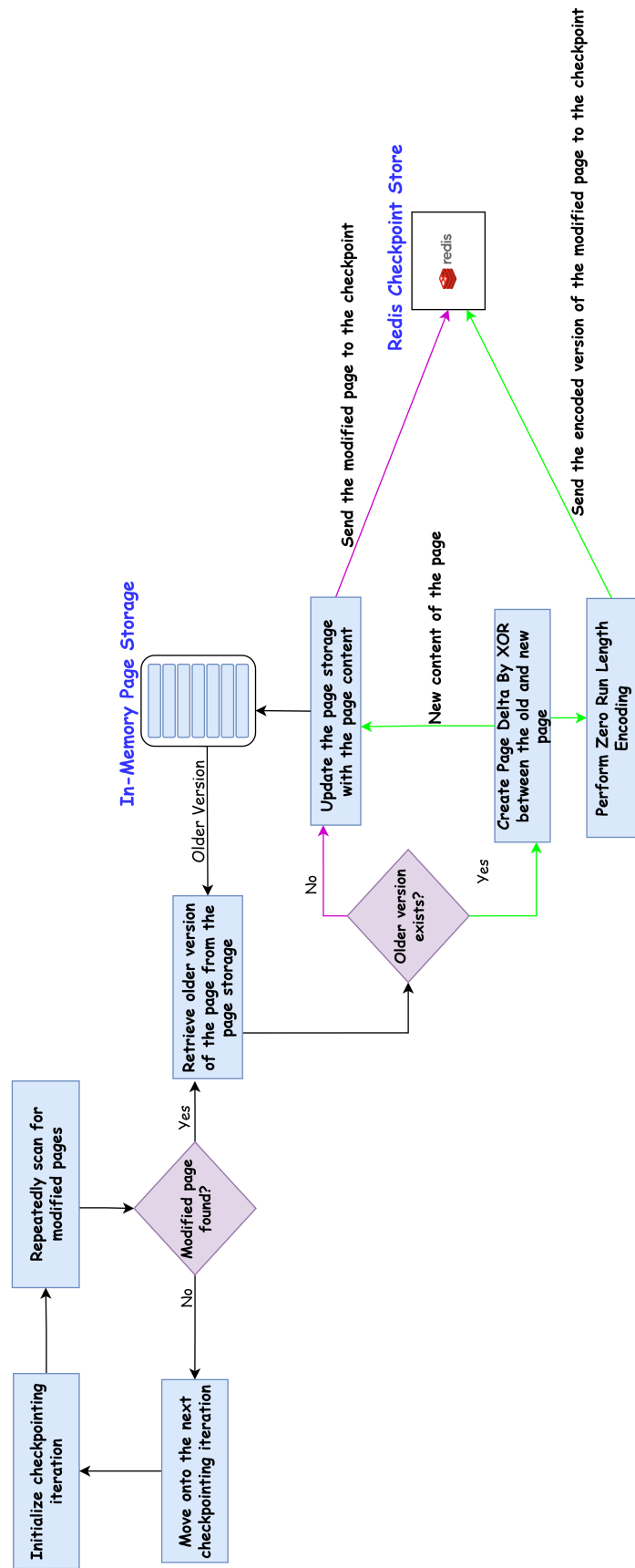


Figure 6: Mem-Finer Approach

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

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

them during checkpointing, instead of repeatedly transmitting entire page. Here this proposed methodology 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 Fernando et al. (2019), 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 being not available, that page will be entered to 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. 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 4.2.

3.2.2 Ability to use checkpoint compression techniques to reduce the content to be transmitted

To achieve the second goal of applying the compression techniques to reduce the content to be transmitted, the model proposed at Fernando et al. (2019) 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, which were discussed in section 2.2. As explored in algorithm 3, before transmitting a modified page to the checkpoint store, it is compressed using a pure lossless compression algorithm that can be specified by the user. In the event of a failure, as explored in algorithm 4, at the recovery end, when the memory pages are loaded, they are decompressed to the original size and loaded into the memory.

The algorithm with the lossless compression techniques integrated is defined in section 3. 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 a 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. The choice of the compression technique to be used can be configured by the server administrator at QEMU level.

Algorithm 3 Checkpointing with lossless compression

Require: `compression_type` $\in \{\text{GZIP}, \text{LZ4}, \text{ZSTD}\}$

```
1: while checkpointing is active do
2:   Initialize checkpointing iteration
3:   modified_pages  $\leftarrow \emptyset$ 
4:   while scanning for modifications do
5:     Scan memory pages for modifications
6:     if modified_page found then
7:       page  $\leftarrow$  modified_page
8:       if compression_type = GZIP then
9:         compressed_page  $\leftarrow$  GzipCompress(page)
10:      else if compression_type = LZ4 then
11:        compressed_page  $\leftarrow$  Lz4Compress(page)
12:      else if compression_type = ZSTD then
13:        compressed_page  $\leftarrow$  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  $\leftarrow$  empty page map
2: for each checkpoint iteration do
3:   pages  $\leftarrow$  GetPagesFromCheckpoint()
4:   for all (key, page_data) in pages do
5:     if IsCompressed(page_data) then
6:       page  $\leftarrow$  Decompress(page_data)
7:     else
8:       page  $\leftarrow$  page_data
9:     end if
10:    hash[key]  $\leftarrow$  page
11:  end for
12: end for
13: LoadPagesToMemoryFromHash(hash)
14: LoadLatestDeviceState()
```

4 Implementation

4.1 Baseline model implementation

4.1.1 Migration start and the use of bitmaps

The PostCopyFT model, which was proposed at Fernando et al. (2019), was reproduced and used as the baseline model. The implementations were done in KVM/QEMU (Bellard 2005, Kivity et al. 2007) based server environments. The baseline model checkpointing works in a way where the migration and checkpointing-related operations are running in different threads on the host server.

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.

4.1.2 Checkpoint versioning

A checkpoint consists of two major 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 pairs 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.

4.2 Mem-Finer Mechanism

As explained in section 3.2, 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:

Initial Buffers

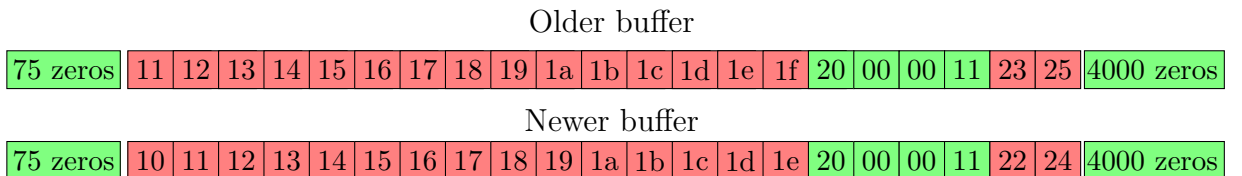
Consider an old memory page buffer and a new memory page buffer that differ only in a few bytes. The buffers are as follows:

- **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
```



In this example, the only differences between the **old** and **new** buffers are a few bytes in the middle of the buffer as shown. The green segments represent the unmodified portions and the red segments represent the modified portions. Instead of sending the entire new buffer during checkpointing, this mechanism allows us to encode just the differences, thereby reducing the amount of data transferred.

Encoded Buffer

Using this mechanism, the data is compressed into a series of **runs**. Each run is either:

- A *zero run*, which specifies the number of consecutive zero bytes, or
- A *non-zero run*, which specifies the length of the run followed by the actual data bytes.

The encoded buffer for this example is as follows:

Encoded Length: 21

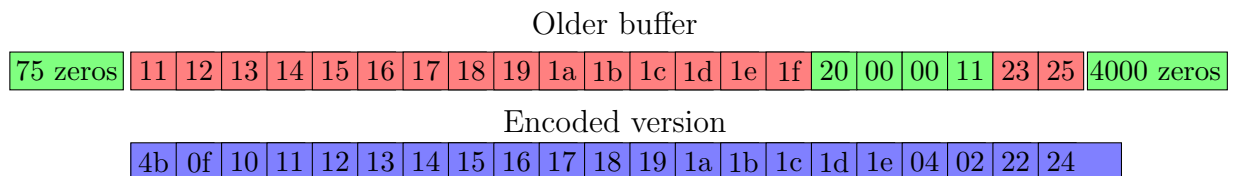
4b 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 04 02 22 24

Breaking down each component of this encoded buffer:

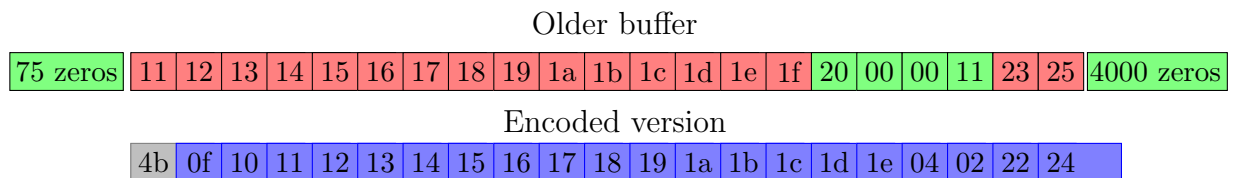
Decoding the Encoded Buffer

At the decoding end, the older version of the page is loaded to memory and it is updated using the encoded version of the page difference to produce the newer version of the page as explained below.

Initial versions – At the time of loading the older version of the page and the encoded version of the page is as follows.



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 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e

Older buffer

75 zeros	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	20	00	00	11	23	25	4000 zeros
----------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	------------

Encoded version

4b	0f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	04	02	22	24
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

These bytes are the updates in the new buffer which are then updated in the loaded older version of the page, which is shown in a darker shade of red.

04 – This represents another **zero run**, this time with a length of 4 bytes. These bytes remain unchanged in the new buffer.

Older buffer

75 zeros	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	20	00	00	11	23	25	4000 zeros
----------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	------------

Encoded version

4b	0f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	04	02	22	24
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

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 as shown in a darker shade of red.

Older 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
----------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	------------

Encoded version

4b	0f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	04	02	22	24
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

The last 4000 bytes in the older buffer are kept as it is, because the encoded page scanning is now finished depicting that the rest of the page is not needed to be modified depicting that it stands for an unmodified portion.

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 not sent to the checkpoint store. Only the encoded version which represented the actual modified content was sent to the checkpoint store.

Older buffer which was updated while scanning the encoded page

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 which was never sent to the 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
----------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	------------

Discussion

Using this byte-level granularity “Mem-Finer” mechanism, we can significantly minimize the amount of data transferred by encoding only the differences between the `old` and `new` buffers. For this example, the encoded buffer length is just 21 bytes, which is much smaller than the full memory page which is 4KB, thereby reducing the bandwidth required for transmitting checkpoints.

4.3 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 3.2.2, three lossless compression algorithms, namely Gzip, LZ4 and ZSTD were integrated to the model proposed by Fernando et al. (2019), to reduce the content to be transmitted. With the compression enabled, the original page content of 4096 bytes was reduced by a large factor, preserving the bandwidth to transmit the memory checkpoint pages to the checkpoint store.

5 Evaluation

5.1 Evaluation of the accuracy of the implemented methodologies

When Mem-Finer methodology as well as compression techniques are integrated to the working reproduced version of the Fernando et al. (2019) model, with each change added, VMs are migrated and recovered to check the usability of the checkpoints. With the new techniques integrated, QEMU is rebuilt and installed and VMs are migrated and failures will be manually triggered to initiate the recovery process. With the completion of the recovery process, if the VM is still running smoothly, it implies that the captured memory and CPU checkpoints are accurate and the methods are working efficiently.

5.2 Evaluation Metrics used for evaluation 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 will be 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 Fernando et al. (2019), 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.

5.3 Workload types tested

To measure the metrics as mentioned in section 5.2, 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.

5.3.1 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.

5.3.2 Sysbench

To create a CPU-intensive nature, the Sysbench benchmark (Pokharana & Gupta 2023) 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.

5.3.3 Yahoo Cloud Services benchmark

To create a more real-life simulation with network-intensive nature, YCSB benchmark was used, where it simulates an environment of having a Postgresql 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 parallelly reading and writing to the database.

5.3.4 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 the quicksort algorithm and outputs the no. of sorting operations successfully done at each second.

5.4 Evaluation Criteria

5.4.1 How average page content is reduced in checkpoints

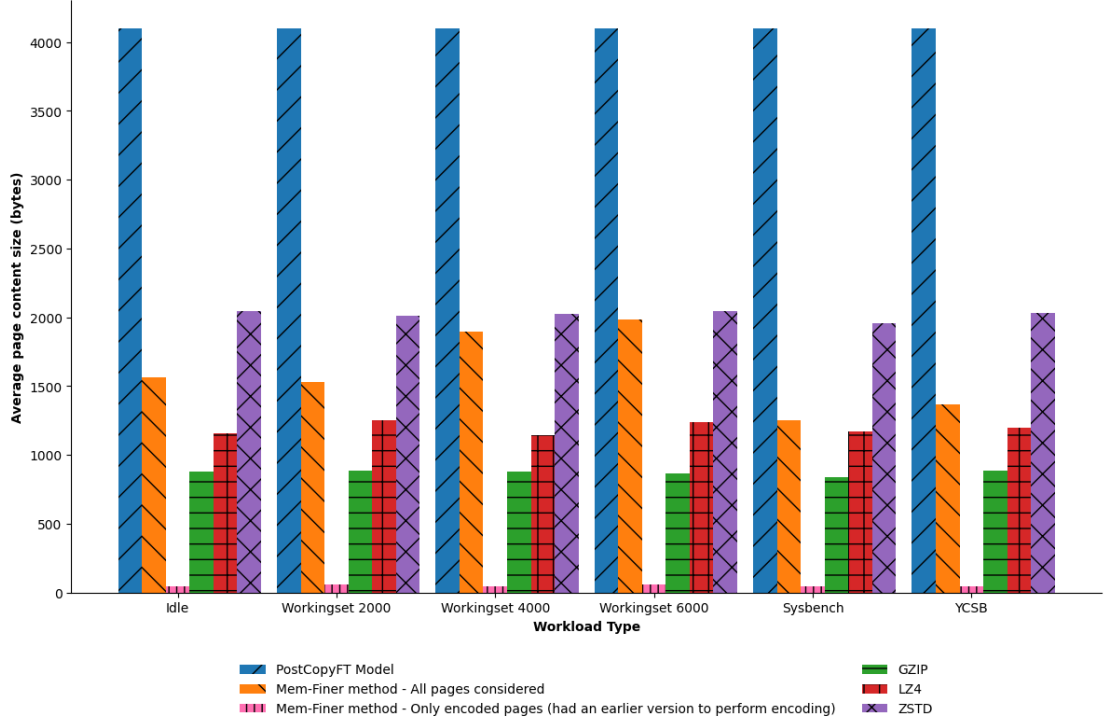


Figure 7: Average page content reduction across different models, with different workloads

Technique	Idle (%)	Workingset 2000 (%)	Workingset 4000 (%)	Workingset 6000 (%)	Sysbench (%)	YCSB (%)
Mem-Finer - All Pages	61.84	62.65	53.71	51.61	69.48	66.60
Mem-Finer - Only Encoded Pages	98.85	98.66	98.85	98.66	98.93	98.88
Gzip	78.59	78.37	78.52	78.96	79.49	78.44
LZ4	71.73	69.46	72.17	69.85	71.44	70.75
ZSTD	50.12	50.98	50.56	50.12	52.27	50.49

Table 1: Percentage of Content Reduced Across Different Categories

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 figure 7. The models considered here are the Mem-Finer methodology, 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. The average page content reduction percentages are given in table 1.

- **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 are emitted from transmission, 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.

5.4.2 Total Migration time

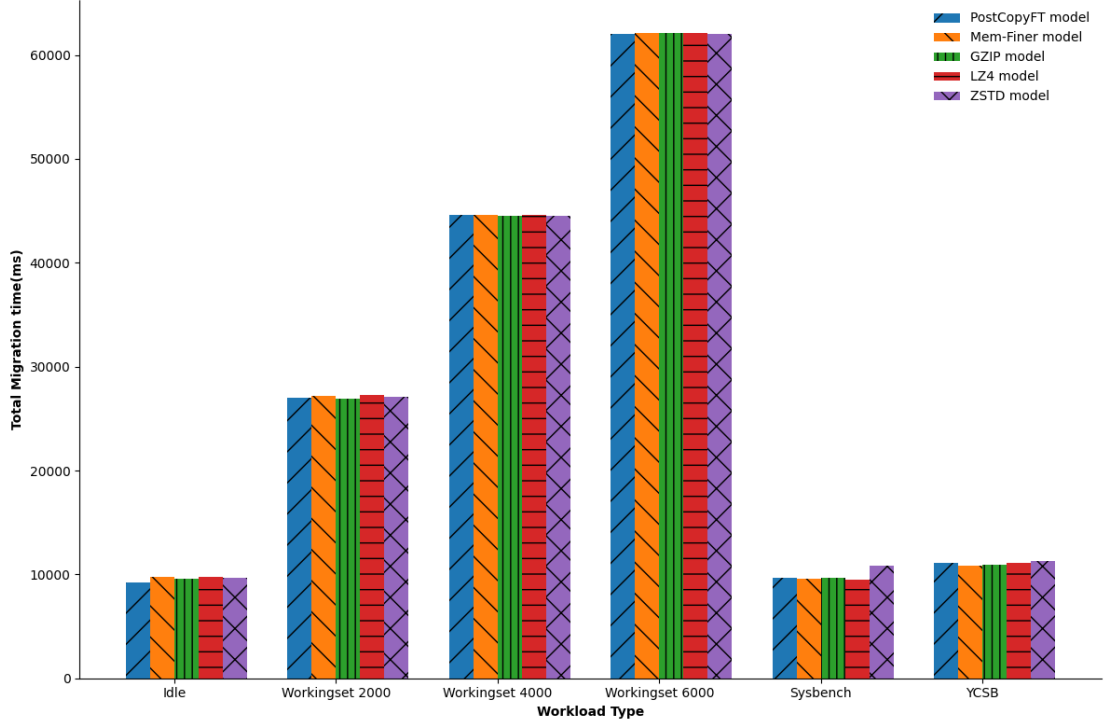


Figure 8: Total migration time of different models across different workloads

When the total migration times were analyzed, in all the models created and under all types of workloads considered, the overall migration times showed to be really close to each other, as shown in figure 8. 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 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.

5.4.3 Downtime

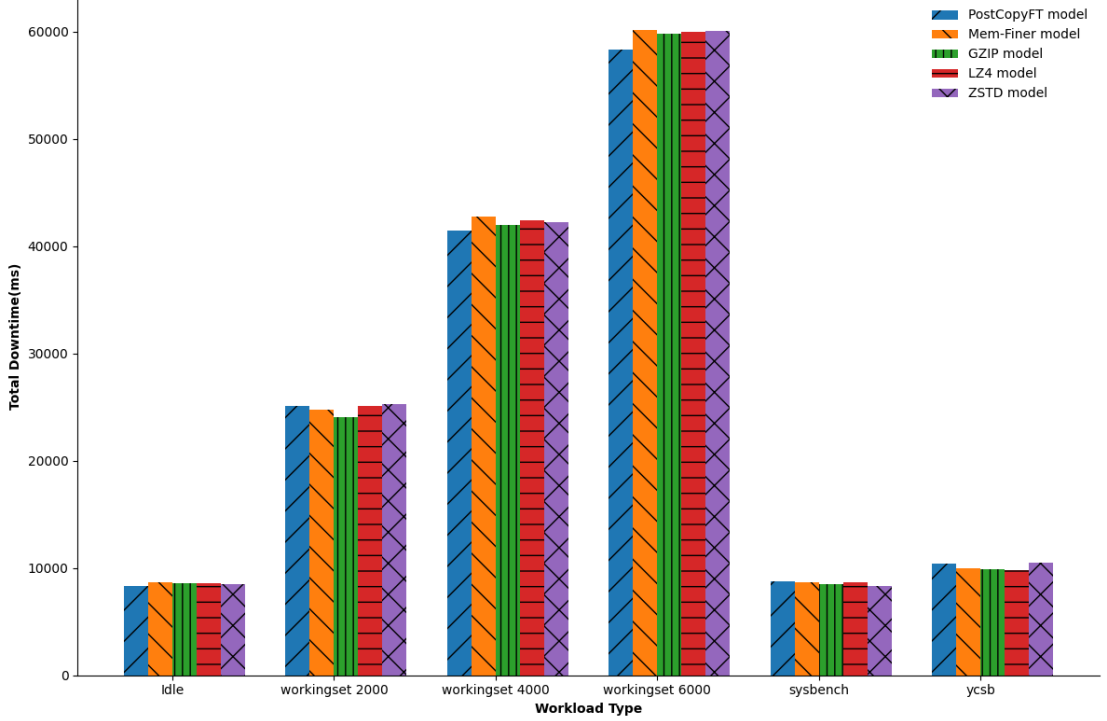


Figure 9: Total downtime of different models across different workloads

When considering the downtime across the models developed and different types of workloads, the results were showing 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, as shown in figure 9. It highlighted that the downtime overhead is negligible in this context.

However, this represents the cumulative downtime. With the different models being used, the no. 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 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 50 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 100 checkpoints due to the lesser time taken in encoding and compressing the checkpoints, respectively.

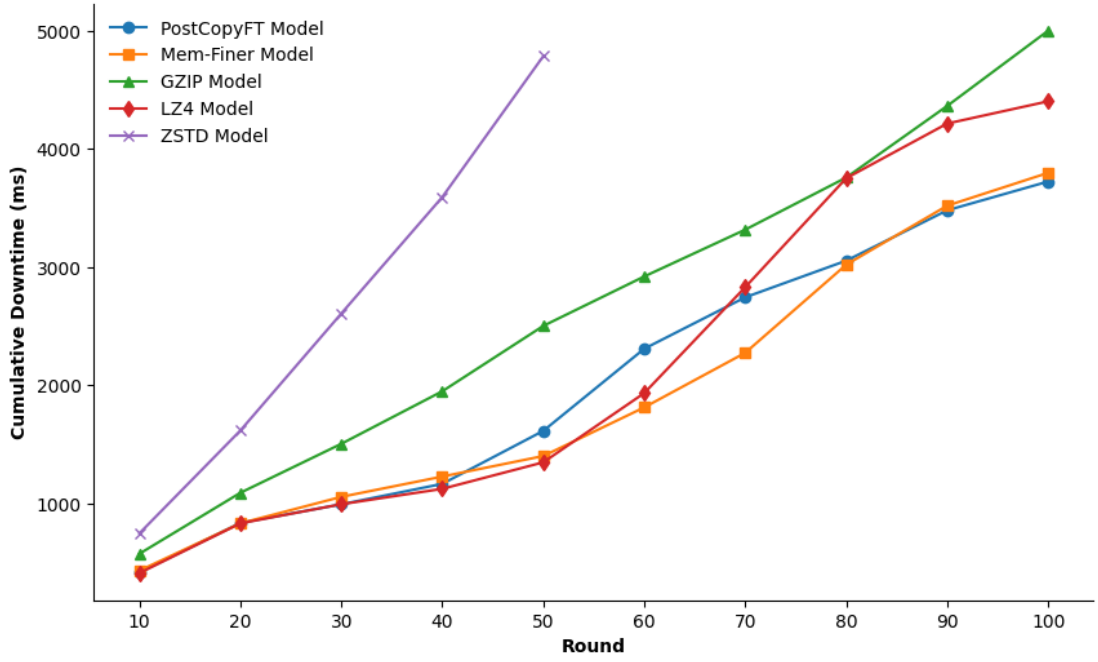


Figure 10: Cumulative downtime vs Checkpointing round with an idle workload

When considering an idle workload as shown in figure 10, 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.

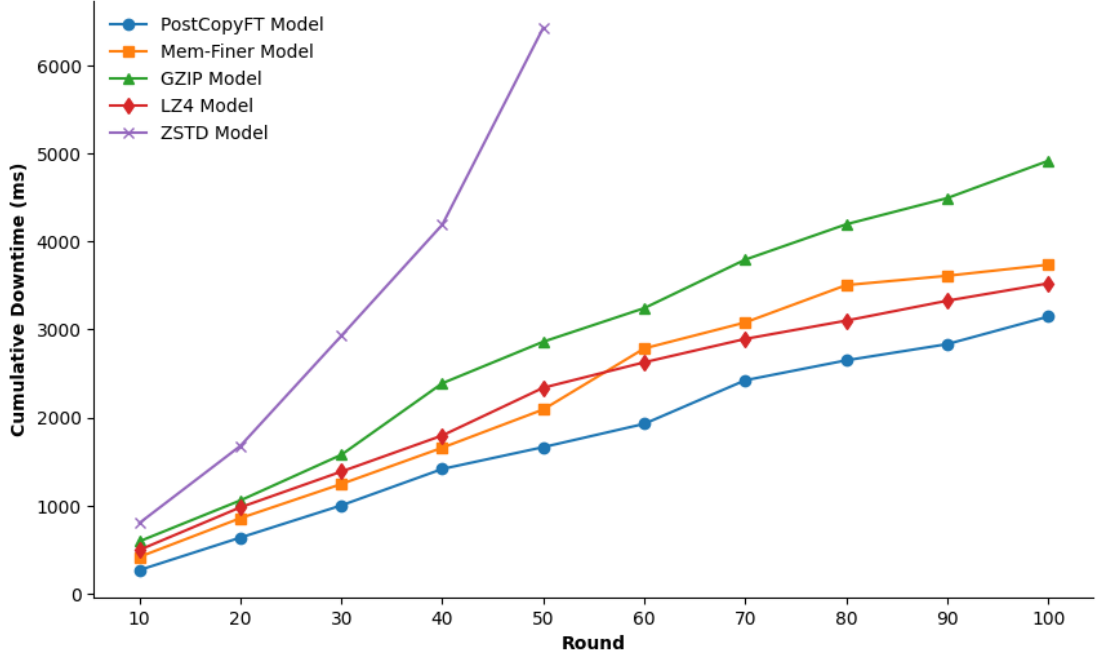


Figure 11: Cumulative downtime vs Checkpointing round with sysbench workload

When considering the sysbench workload, which is CPU intensive as shown in figure 11, 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 data-points when compared with the baseline model, and gzip downtime overhead was slightly higher than the Mem-Finer and lz4 models. Here also it 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.

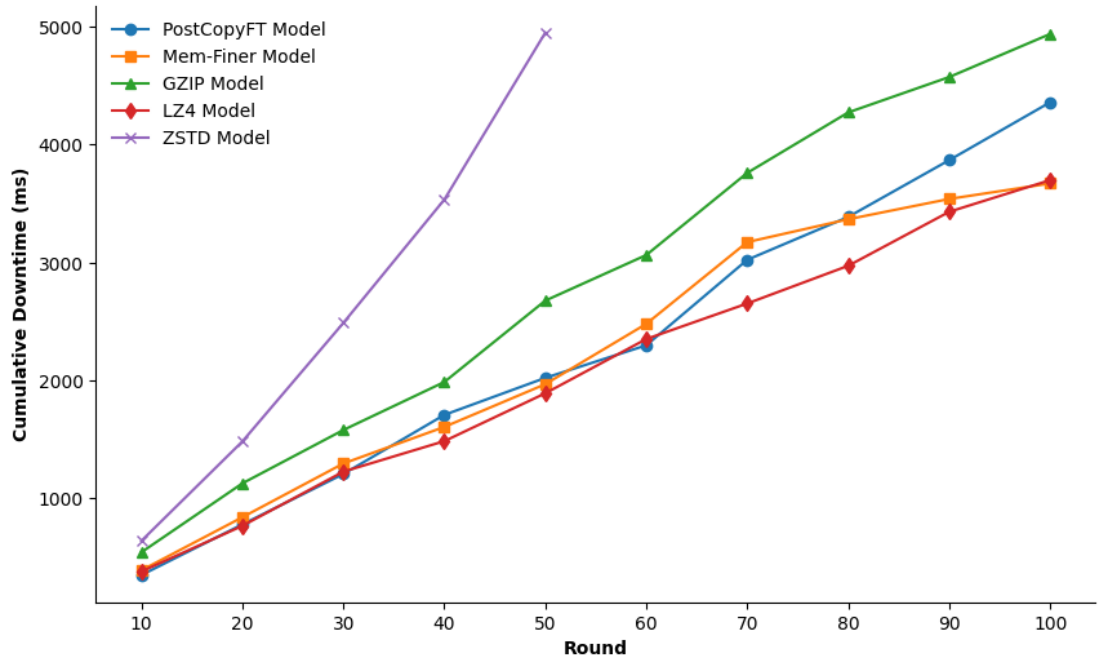


Figure 12: Cumulative downtime vs Checkpointing round with 2000MB workingset workload

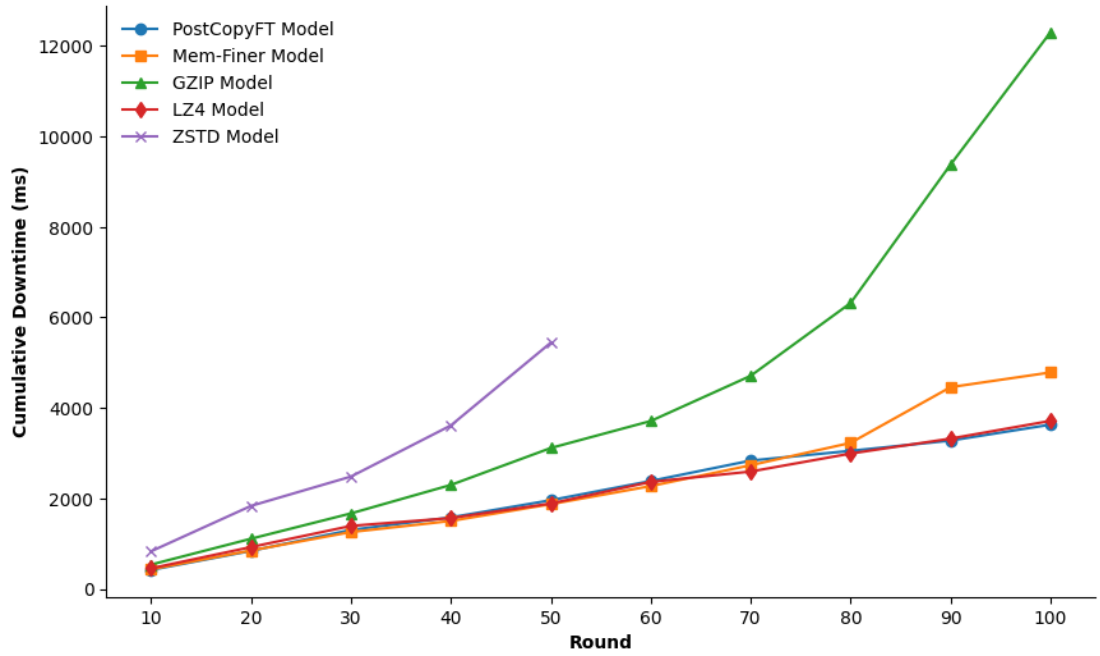


Figure 13: Cumulative downtime vs Checkpointing round with 4000MB workingset workload

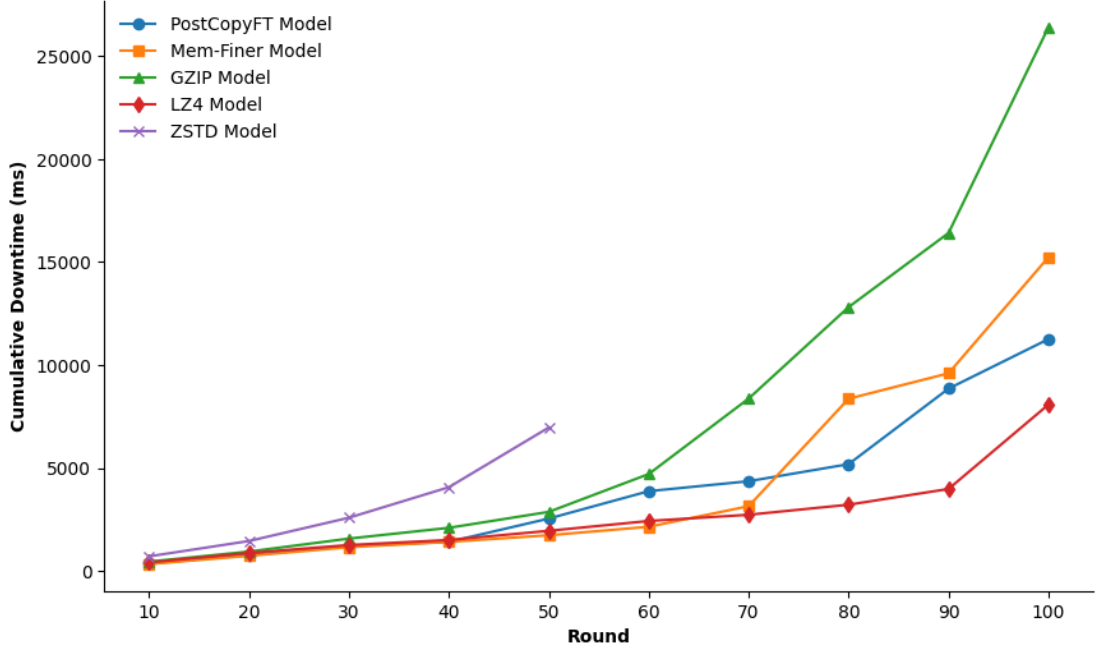


Figure 14: Cumulative downtime vs Checkpointing round with 6000MB workingset workload

When summarizing the behaviour of the downtimes when the workload running becomes memory intensive, regardless of the working set size, zstd showed exponential increments in the downtime when compared with the baseline model as shown in figures 12-13. 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.

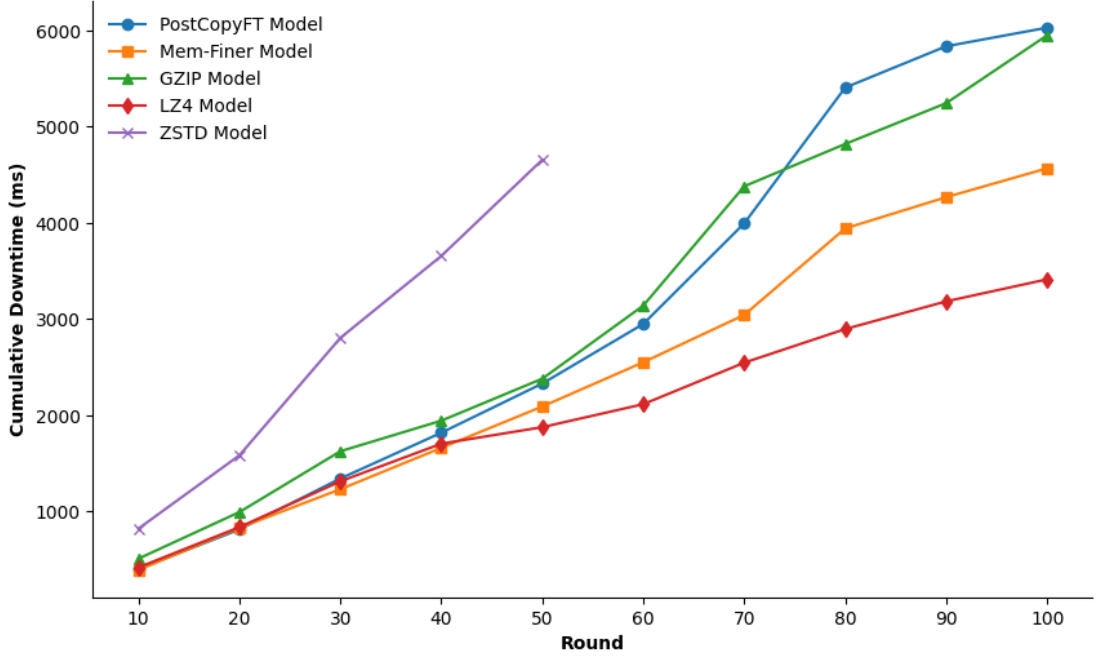


Figure 15: Cumulative downtime vs Checkpointing round with YCSB workload

With the YCSB benchmark that simulated a more realistic network-intensive nature, still Zstd performance was not good as shown in figure 15. 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 growths in downtimes, when the memory-intensive nature is incremented. 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.

5.4.4 Checkpoint replication time

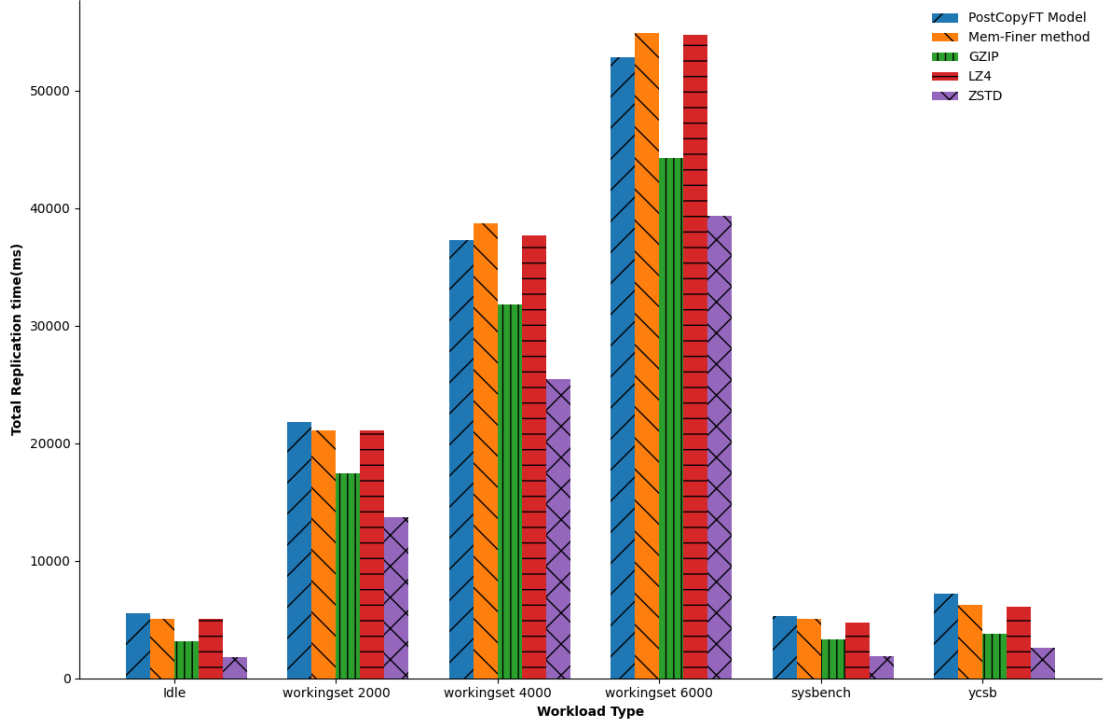


Figure 16: Total checkpoint replication time of different models across different workloads

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 16 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 5.4.3.

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 5.4.1. 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.

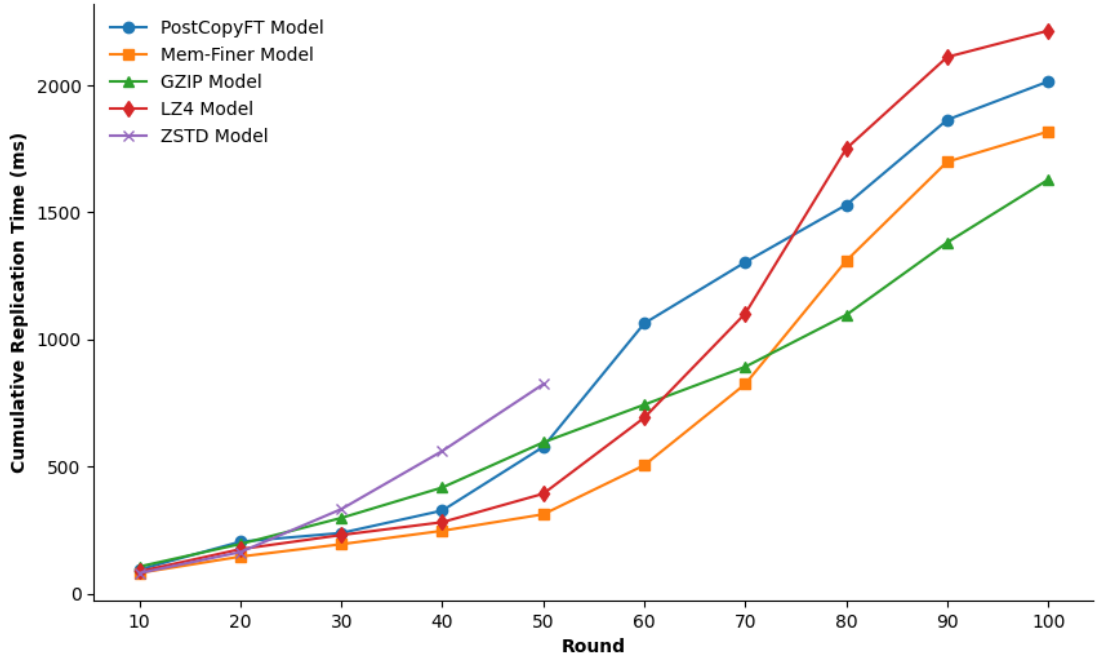


Figure 17: Cumulative checkpoint replication time vs Checkpointing round with an idle workload

When an idle workload was considered as shown in figure 17, the replication times showed a similar pattern to the 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.

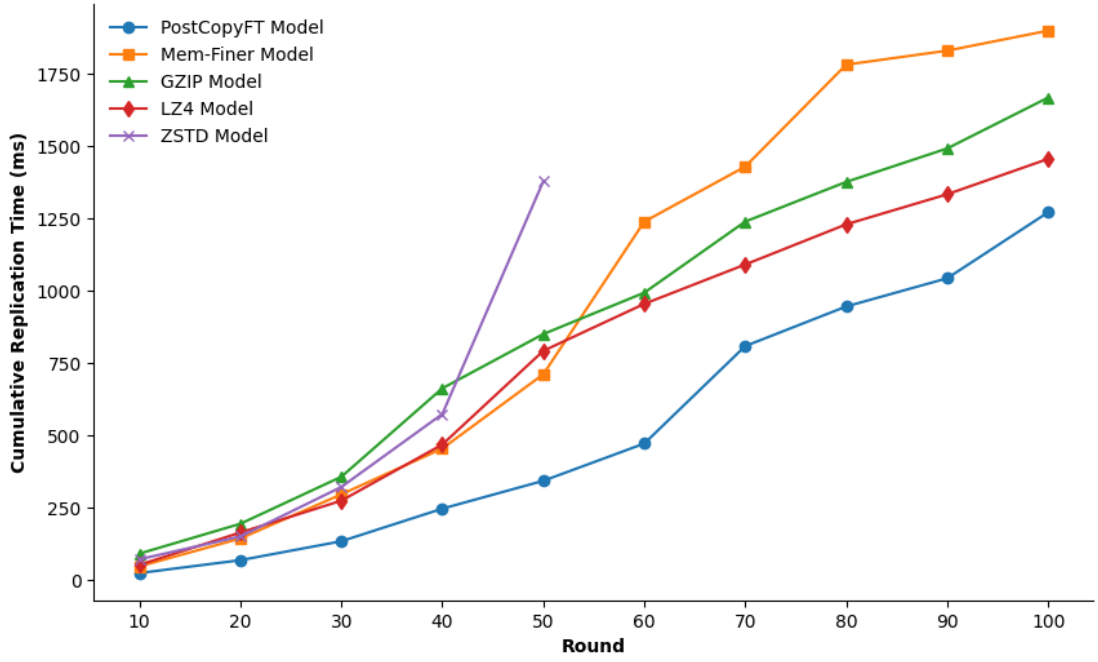


Figure 18: Cumulative checkpoint replication time vs Checkpointing round with sysbench workload

When the cpu-intensive benchmark sysbench was utilized as shown in figure 18, 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.

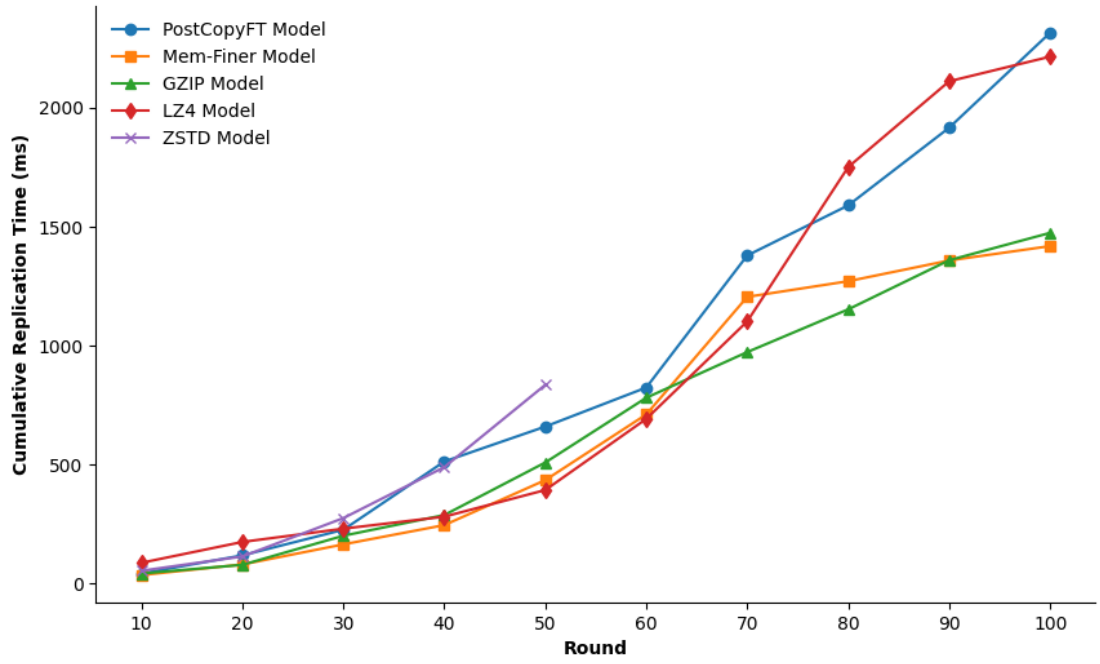


Figure 19: Cumulative checkpoint replication time vs Checkpointing round with 2000MB workingset workload

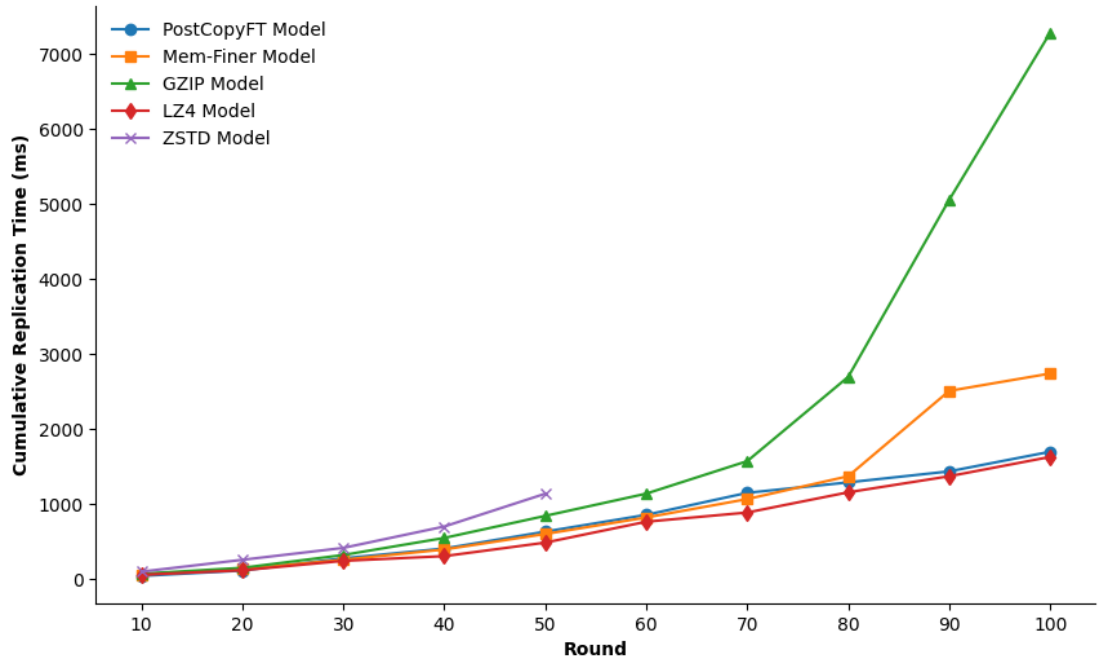


Figure 20: Cumulative checkpoint replication time vs Checkpointing round with 4000MB workingset workload

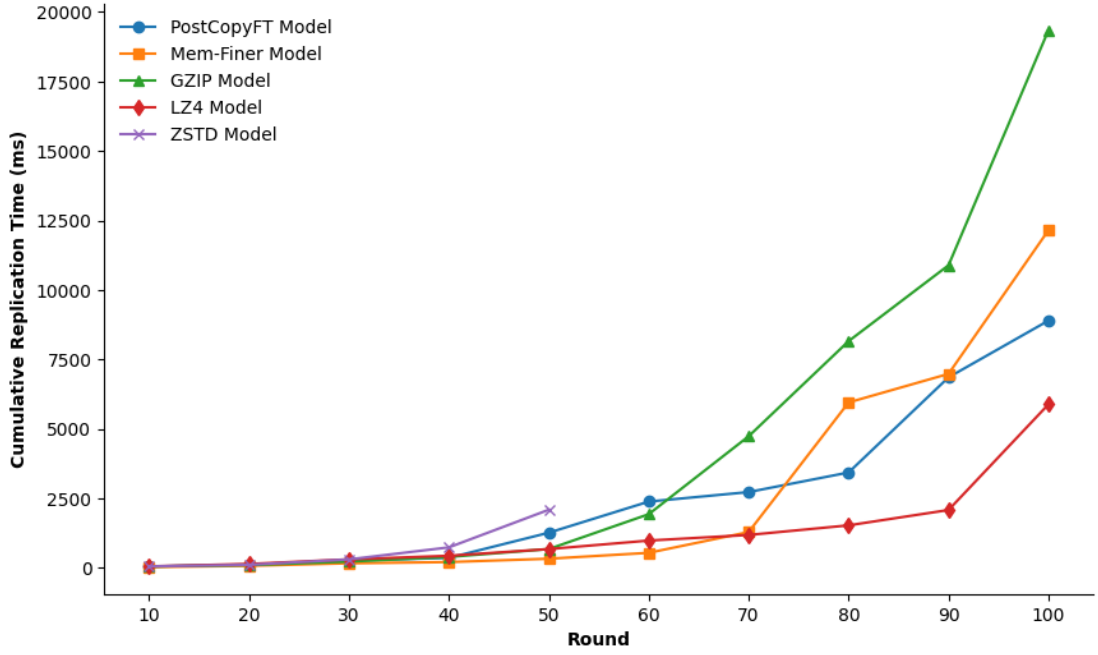


Figure 21: Cumulative checkpoint replication time vs Checkpointing round with 6000MB workingset workload

When the memory-intensive workload “workingset” was used, similar to the section 5.4.3, zstd and gzip showed negative results of incrementing exponentially as shown in figures 19-21. 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 lesser than the baseline version at some instances.

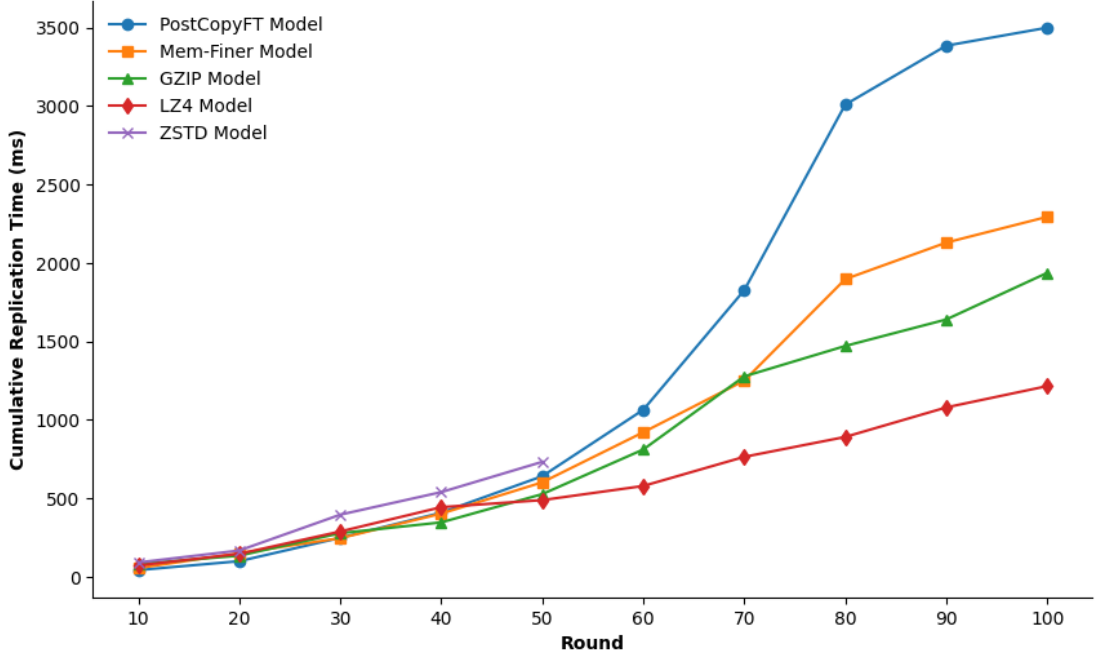


Figure 22: Cumulative checkpoint replication time vs Checkpointing round with YCSB workload

When a network-intensive workload YCSB was used as shown in figure 22, 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 summarising 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 gives acceptable and even better statistics than the baseline model at 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.

5.4.5 Application performance impact

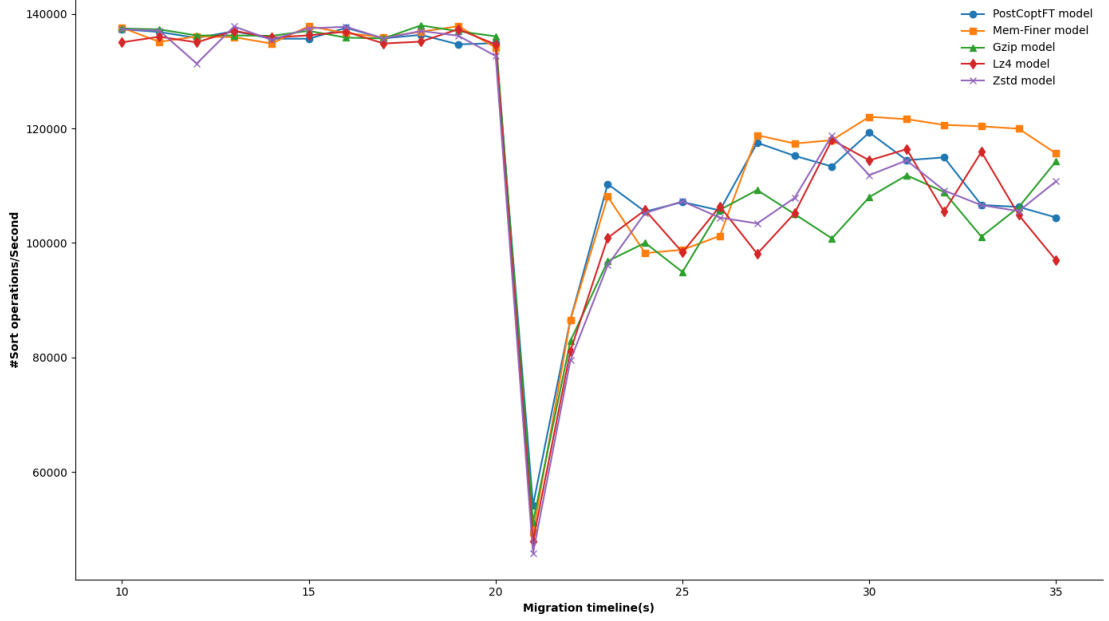


Figure 23: Performance impact across all models

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 observed in figure 23 and separately elaborated in figures 24-27, Mem-Finer approach and the Gzip, LZ4 and Zstd compression enabled models showed a pattern similar 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. 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 diagrams below.

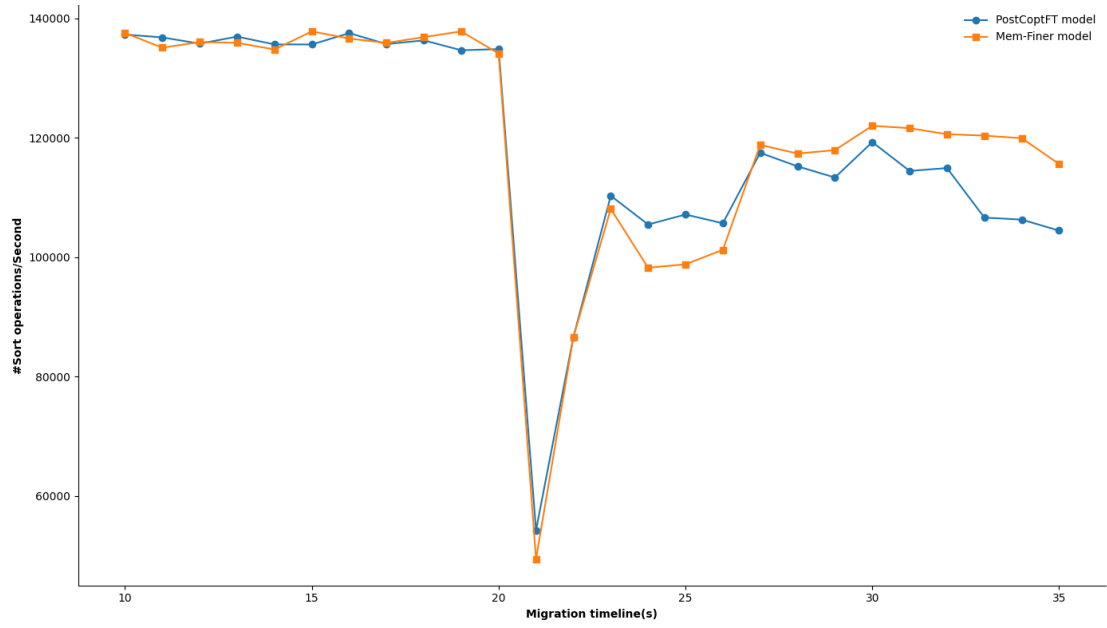


Figure 24: Baseline vs Mem-Finer performance impact

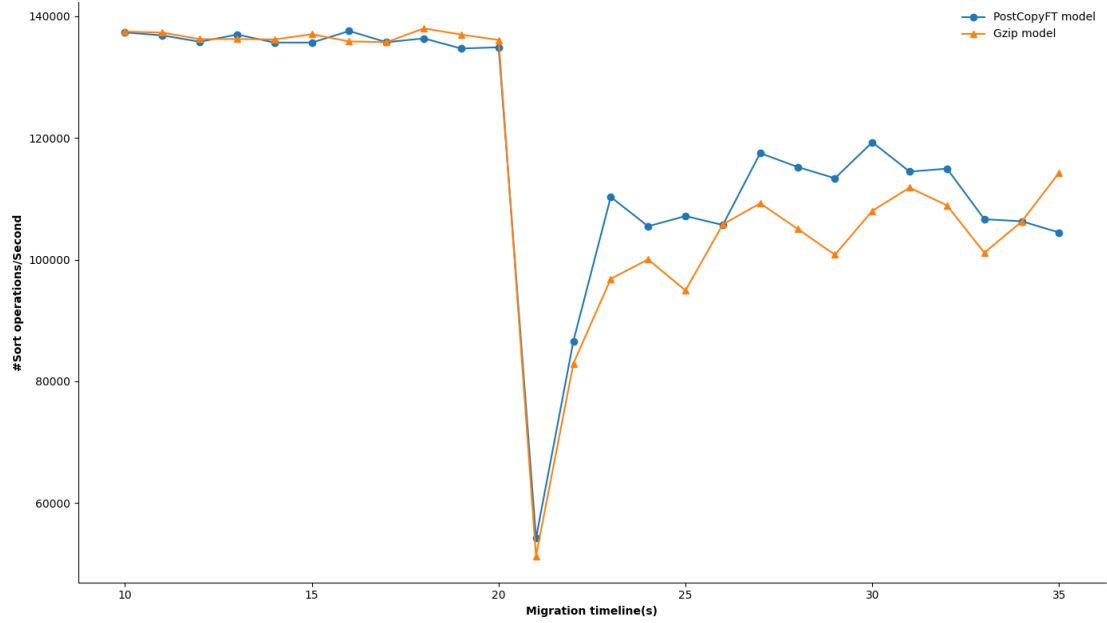


Figure 25: Baseline vs Gzip performance impact

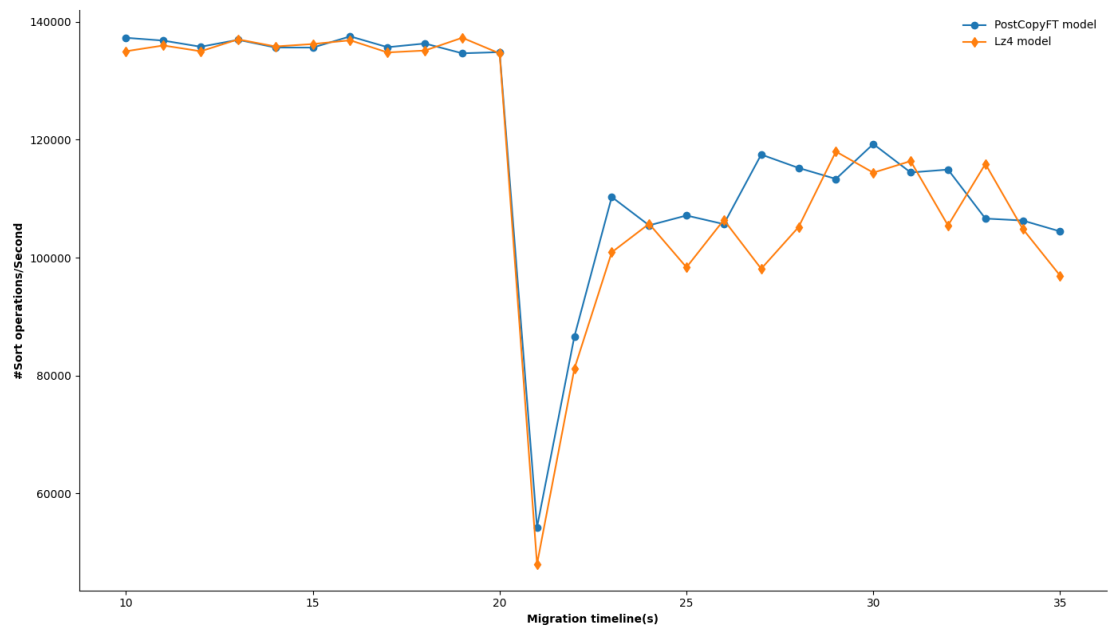


Figure 26: Baseline vs LZ4 performance impact

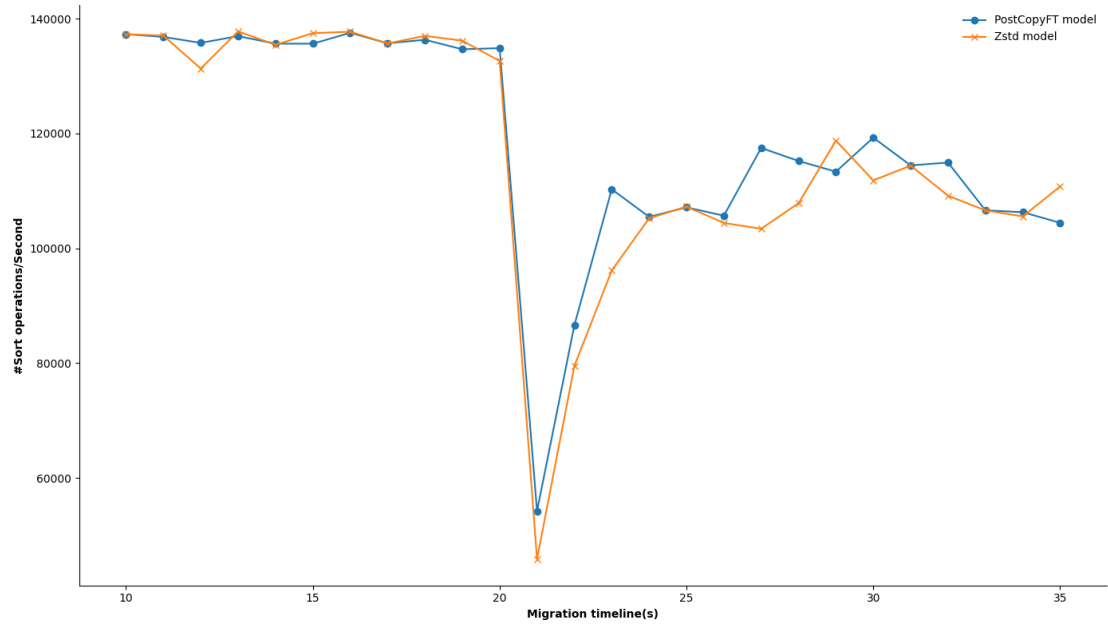


Figure 27: Baseline vs ZSTD performance impact

5.4.6 Recovery time

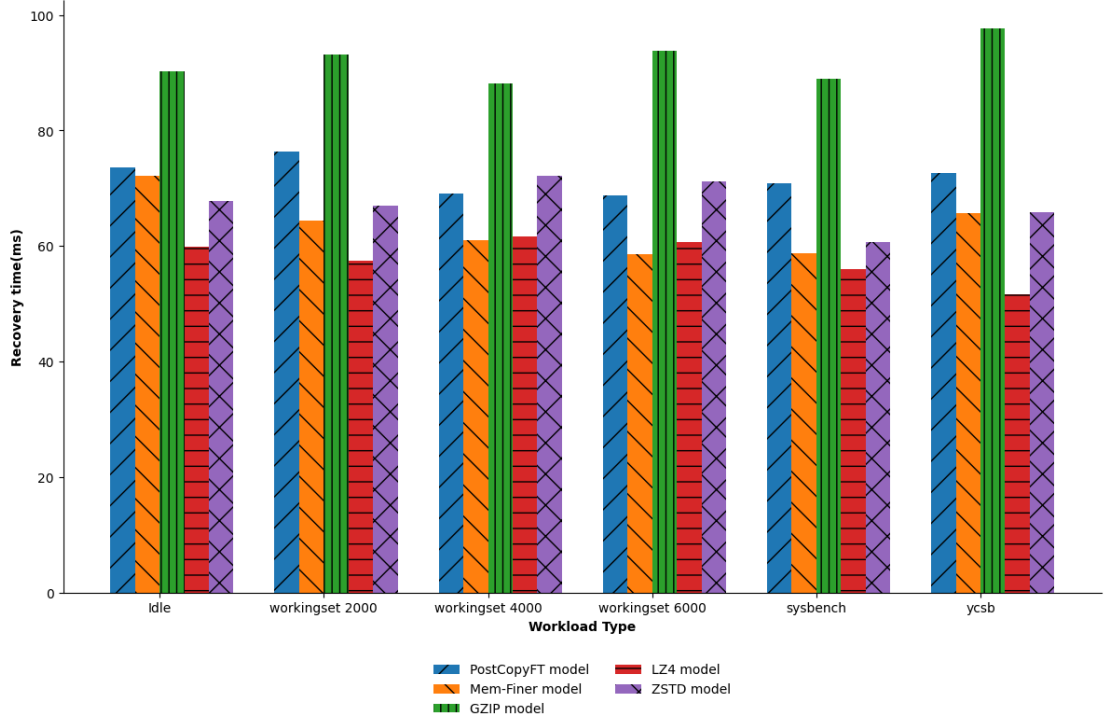


Figure 28: Total recovery time of different models across different workloads

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 CC 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, 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.

Table 2: Recovery Time Reduction Percentages across Different Workloads

Workload type	Mem-Finer Model	Gzip Model	Lz4 Model	Zstd Model
Idle	1.90	-22.55	18.75	7.88
Workingset 2000	15.71	-21.99	24.87	12.30
Workingset 4000	11.59	-27.83	10.72	-4.64
Workingset 6000	14.83	-36.34	11.92	-3.49
Sysbench	16.95	-25.71	20.90	14.41
Ycsb	9.64	-34.44	28.93	9.37

When examining the overall recovery times as observed in table 2 and showed in figure 28, 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.

6 Discussion and Conclusion

6.0.1 Different models considered

When considering all the models developed, which are namely the Mem-Finer model that was capable of extracting the actual modified content in the dirtied page, the Gzip compression-enabled model, the LZ4 compression-enabled model and the Zstd compression-enabled model, varying results were observed with respect to different evaluation criteria.

1. Mem-Finer model

The Mem-Finer approach was able to get the average page content get reduced by **51-69%** depending on the different workloads considered. It was observed that when only the pages that had an earlier version of the page available in the in-memory page storage to perform encoding, it showed a massive page content reduction of **98%**. It was also observed that this method was capable of identifying the fake dirty pages, avoiding those redundant replicated data entirely. This suggests that using the Mem-Finer methodology will reduce the network bandwidth overheads incurred by checkpointing operations by a larger factor which also results in reducing the costs of storage infrastructure due to the reduced checkpointing content. Furthermore, it resulted a similar migration time, similar downtime, similar and even lesser cumulative downtime patterns than the baseline model at certain workloads when compared with the baseline model.

It also showed similar replication times and cumulative replication time patterns even surpassing the baseline model performance at certain workloads. It showed an application performance degradation pattern similar to the baseline model which suggests that the performance degradations are acceptable, and it resulted in a lesser recovery time with a reduction percentage of **1-16%** than that baseline model under different workloads. All these criteria suggest that the Mem-Finer methodology is a good performing choice which can be utilized to make the checkpointing operations much enhanced, especially in the cases where large number of VMs are simultaneously migrated to destinations in the events of failures.

2. GZIP model

The gzip compression enabled model showed a massive page content reduction of **78-79%** across different workloads, and it was the best performing algorithm in terms of the page content reduction. It showed similar migration times when compared with the baseline model, but when considering the downtimes, it was observed that Gzip compression struggles when the memory write-intensive behaviour is gradually increased. It can be concluded that Gzip model is not a good fit in memory write-intensive environments. The replication time patterns too showed negative behaviours due to increasing the downtimes during memory write-intensive environments that

results in collecting more dirty pages to be sent in successive iterations, which will ultimately increase the checkpoint replication time as well. The performance impact pattern was found to be similar to the baseline model, but Gzip struggled in the recovery time as well. It was observed that Gzip has the highest recovery time with an increment percentage of **21-36%** under different workloads, which suggests that the decompression algorithm of Gzip compression takes a considerable time.

3. LZ4 model

LZ4 model showed good page content reductions which spanned between **69-72%** across different workloads that suggest that LZ4 is capable of reducing the redundant page content transfer during checkpointing by a massive scale. It showed migration times, downtimes and cumulative downtime patterns similar to the baseline model, even having better cumulative downtimes than the baseline model at certain workloads. The replication time patterns were identical to the baseline model even surpassing the baseline model performance at certain workloads. The application performance degradation analysis patterns were shown to be similar to the baseline model and LZ4 showed lesser recovery times with a reduction percentage of **10-28%** than the baseline model during all workloads. The overall picture depicts that LZ4 model is a good approach that could be used under any workload to make the checkpointing operations more efficient.

4. ZSTD model

ZSTD model showed negative results throughout the experiments which suggests that Zstd is not a good fit in the context of checkpointing during VM migrations. It showed a page content reduction of **50-52%** and similar migration times. However, it was seen that the downtimes and replication times are growing exponentially when ZSTD is used, which is worse than using the basic baseline model. It showed similar application performance degradations like the other models and a lesser recovery time in most cases with a reduction percentage of **7-14%** and having **3-4%** increment when the memory write intensive nature changes, but it was capable of generating 50-60 checkpointing iterations while all the other models generated more than 100 checkpoints during the period of migration. It suggests that ZSTD should be avoided in latency-sensitive real-time systems which deal with minute portions of memory.

6.0.2 Insights and conclusion

When considering all the factors, it can be concluded that the Mem-Finer methodology and LZ4 compression-enabled models are the ideal fits to be used in the context of VM migrations. They showed good performances across all workloads and benchmarks while remaining identical and even lesser in terms of the overheads incurred, when compared with the baseline model. Mem-Finer methodology opens the branches for different areas, where it suggests that it is a good methodology

to find false negatives like fake dirty pages in the areas of system research.

Gzip compression enabled model is a fairly good approach, but it should not be used in memory write-intensive environments where it struggles and also it comes with an additional overhead of increased recovery time than the traditional baseline model. It is a fairly good approach, giving the highest page content reductions, if used in CPU or network-intensive environments. Finally, Zstd should be avoided in real-time systems where a minimal latency is a must. It struggles when dealing with minute portions of memory taking an enormous amount of time for compressions when compared with the other models.

7 Limitations and Future Directions

The experiments conducted were done for single VM migrations and failure scenarios. The overall efficiency improvements may be further tested in bigger server clusters with multiple VM failures and simultaneous migrations. Additionally, the experiments were conducted in KVM/QEMU-based environments, which suggests that there's potential future research to incorporate these models to other technologies like VMWare, Oracle VirtualBox, Xen etc.

References

- Agarwal, A. & Raina, S. (2012), ‘Live migration of virtual machines in cloud’, *International Journal of Scientific and Research Publications* **2**(6), 1–5.
- Ahmad, R. W., Gani, A., Ab. Hamid, S. H., Shiraz, M., Xia, F. & Madani, S. A. (2015), ‘Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues’, *The Journal of Supercomputing* **71**, 2473–2515.
- Alibaba Cloud (2024), ‘Alibaba cloud’.
URL: https://in.alibabacloud.com/?utm_key=se_1007710386&utm_content=se_1007710386&gclid=Cj0KCQiA8a0eBhCWARIsANRFrQFZhZk5PnatY5otgaHs5v4Z0-vFwNnjqDHoWZRgn3uXSJOW_BnYIWQaAik3EALw_wcB
- AWS (2024), ‘Amazon web services’.
URL: <https://aws.amazon.com/>
- Bartík, M., Ubik, S. & Kubalik, P. (2015), Lz4 compression algorithm on fpga, *in* ‘2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)’, IEEE, pp. 179–182.
- Bellard, F. (2005), Qemu, a fast and portable dynamic translator., *in* ‘USENIX annual technical conference, FREENIX Track’, Vol. 41, California, USA, pp. 10–5555.
- Bhardwaj, A. & Rama Krishna, C. (2019), 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, pp. 1021–1032.
- Chanchio, K., Leangsuksun, C., Ong, H., Ratanasamoot, V. & Shafi, A. (2008), An efficient virtual machine checkpointing mechanism for hypervisor-based hpc systems, *in* ‘High Availability and Performance Computing Workshop’.
- Choudhary, A., Govil, M. C., Singh, G., Awasthi, L. K., Pilli, E. S. & Kapil, D. (2017), ‘A critical survey of live virtual machine migration techniques’, *Journal of Cloud Computing* **6**(1), 1–41.
- Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. & Warfield, A. (2005), Live migration of virtual machines, *in* ‘Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2’, pp. 273–286.
- Collet, Y. (2021), ‘Rfc 8878: Zstandard compression and the application/zstd media type’.
- Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. & Warfield, A. (2008), Remus: High availability via asynchronous virtual machine replication, *in* ‘Proceedings of the 5th USENIX symposium on networked systems design and implementation’, San Francisco, pp. 161–174.

- Damania, K., Holmukhe, S., Singhai, V. & Bhavathankar, P. (2018), An overview of vm live migration strategies and technologies, *in* ‘2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)’, pp. 1185–1190.
- Fernando, D., Bagdi, H., Hu, Y., Yang, P., Gopalan, K., Kamhoua, C. & Kwiat, K. (2016), Quick eviction of virtual machines through proactive live snapshots, *in* ‘Proceedings of the 9th International Conference on Utility and Cloud Computing’, pp. 99–107.
- Fernando, D., Turner, J., Gopalan, K. & Yang, P. (2019), 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, pp. 343–351.
- Fernando, D., Turner, J., Yang, P. & Gopalan, K. (2023), ‘V-recover: Virtual machine recovery when live migration fails’, *IEEE Transactions on Cloud Computing*.
- GCP (2024), ‘Google cloud platform’.
URL: <https://cloud.google.com/>
- Gelenbe, E. (1976), A model of roll-back recovery with multiple checkpoints, *in* ‘Proceedings of the 2nd international conference on Software engineering’, pp. 251–255.
- Goiri, Í., Julia, F., Guitart, J. & Torres, J. (2010), Checkpoint-based fault-tolerant infrastructure for virtualized service providers, *in* ‘2010 IEEE network operations and management symposium-NOMS 2010’, IEEE, pp. 455–462.
- Hines, M. R., Deshpande, U. & Gopalan, K. (2009), ‘Post-copy live migration of virtual machines’, *ACM SIGOPS operating systems review* **43**(3), 14–26.
- Hou, K.-Y., Shin, K. G., Turner, Y. & Singhal, S. (2013), Tradeoffs in compressing virtual machine checkpoints, *in* ‘Proceedings of the 7th international workshop on Virtualization technologies in distributed computing’, pp. 41–48.
- Kangarlou, A., Eugster, P. & Xu, D. (2009), Vnsnap: Taking snapshots of virtual networked environments with minimal downtime, *in* ‘2009 IEEE/IFIP International Conference on Dependable Systems & Networks’, IEEE, pp. 524–533.
- Kannan, S., Gavrilovska, A., Schwan, K. & Milojevic, D. (2013), Optimizing checkpoints using nvm as virtual memory, *in* ‘2013 IEEE 27th International Symposium on Parallel and Distributed Processing’, IEEE, pp. 29–40.
- Kivity, A., Kamay, Y., Laor, D., Lublin, U. & Liguori, A. (2007), kvm: the linux virtual machine monitor, *in* ‘Proceedings of the Linux symposium’, Vol. 1, Dttawa, Dntorio, Canada, pp. 225–230.
- Knauth, T. & Fetzer, C. (2015), Vecycle: Recycling vm checkpoints for faster mi-

- grations, *in* ‘Proceedings of the 16th Annual Middleware Conference’, pp. 210–221.
- Li, G., Pattabiraman, K., Cher, C.-Y. & Bose, P. (2015), Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption, *in* ‘2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)’, IEEE, pp. 141–152.
- Liu, H., Jin, H., Liao, X., Hu, L. & Yu, C. (2009), 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’, pp. 101–110.
- Microsoft (2024), ‘Microsoft azure’.
URL: <https://azure.microsoft.com/en-us>
- Milojčić, D. S., Douglass, F., Paindaveine, Y., Wheeler, R. & Zhou, S. (2000), ‘Process migration’, *ACM Computing Surveys (CSUR)* **32**(3), 241–299.
- Moody, A., Bronevetsky, G., Mohror, K. & De Supinski, B. R. (2010), 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, pp. 1–11.
- Nelson, M., Lim, B.-H., Hutchins, G. et al. (2005), Fast transparent migration for virtual machines., *in* ‘USENIX Annual technical conference, general track’, pp. 391–394.
- Pokharana, A. & Gupta, R. (2023), 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, pp. 1–5.
- Sahni, S. & Varma, V. (2012), A hybrid approach to live migration of virtual machines, *in* ‘2012 IEEE international conference on cloud computing in emerging markets (CCEM)’, IEEE, pp. 1–5.
- Sardashti, S. & Wood, D. A. (2017), ‘Could compression be of general use? evaluating memory compression across domains’, *ACM Transactions on Architecture and Code Optimization (TACO)* **14**(4), 1–24.
- Wang, L., Kalbarczyk, Z., Iyer, R. K. & Iyengar, A. (2010), Checkpointing virtual machines against transient errors, *in* ‘2010 IEEE 16th International On-Line Testing Symposium’, IEEE, pp. 97–102.
- Wang, L., Kalbarczyk, Z., Iyer, R. K. & Iyengar, A. (2014), ‘Vm- μ checkpoint: Design, modeling, and assessment of lightweight in-memory vm checkpointing’, *IEEE Transactions on Dependable and Secure Computing* **12**(2), 243–255.