Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

8-1-2024 12:00 PM

Container Migration: A Perfomance Evaluation Between MIGrror AND Pre-copy

Xinwen Liang, *The University of Western Ontario*

Supervisor: Hanan, Lutfiyya, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science
© Xinwen Liang 2024

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Other Computer Sciences Commons

Recommended Citation
Liang, Xinwen, "Container Migration: A Perfomance Evaluation Between MIGrror AND Pre-copy" (2024). *Electronic Thesis and Dissertation Repository*. 10358.
https://ir.lib.uwo.ca/etd/10358

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

The concept of migration and checkpoint/restore has been a very important topic in research for many types of applications including any distributed systems/applications or single massive systems/applications; and low latency vehicular use cases, augmented reality(AR) and virtual reality(VR) applications. Migrating a service requires that the state of the service is preserved. This requires checkpointing the state and restoring it on a different server in multiple rounds to avoid a total loss of all data in case of a failure, fault or error. There are many different types of migration techniques utilized such as cold migration, Pre-copy migration, post-copy migration.

Compared with the above migrations, MiGrror migration needs to consider rounds of memory changes for when it does the migration. The utilization of rounds of memory changes means that it is more precise to migrate instead of migrating on a unsuitable time interval. In this thesis, we will implement a testbed for the MiGrror and Precopy technique (as there are currently none that are not simulations) and evaluate their performance.

Keywords: Checkpoint/Restore, migration, CRIU, container

Summary for Lay Audience

In the recent years, there have been a massive push to use containers on the cloud rather than virtual machines. This increase has warranted new and more powerful resources in order to transfer a massive throughput through a very small amount of time that may be used on the cloud. Migration is the process of migrating information from one device to another.

The process of migration is part of checkpoint and restore where the goal is to checkpoint application states and then restore them after so it is ready to be used later. Migration is mainly done in different types of techniques that differ by when/how the memory is transferred and the amount transferred. The main technique most containers use are a time gated one but may not be efficient enough for more low latency dependent applications.

Therefore, we implemented a testbed for a new proposed migration technique compared with the usual migration technique used by many containers. The implementation we proposed utilizes podman containers on two servers. We evaluated the metrics for both migration techniques migrating from one container to another. The results show that the proposed migration technique has a lower downtime for the migration compared with the main migration technique used.

// Acknowledgements

Doing research is a lonely yet profoundly exciting and engaging experience to have. On this journey, I could not have reached my goal without the assistance of many people. I would like to express my sincere gratitude to my supervisor Dr. Hanan Lutifyya. I really appreciated the opportunity you offered to me. Your continuous support, encouragement, and guidance during my study allowed me to face any challenges during the research. It was my great honor to be your student and a member of the Western Distributed Systems Management Laboratory team.

I am thankful for the guidance from Duff Jones and Arshin Rezazadeh, your recommendations and detail explanation of concepts provided an important foundational knowledge for this project.

Furthermore, I wish to extend my gratitude to all the faculty members, mentors and administration staff in the Department of Computer Science, University of Western Ontario. The five years of undergraduate studies here have laid a solid foundation for my computer knowledge, and the two years of graduate level study have given me sufficient conditions to explore in-depth fields.

At last, but not least, I sincerely thank my parents for their love and support. Without love, writing a thesis would have been significantly harder.

Contents

Abstract **ii**

Summary of Lay Audience **iii**

Acknowledgments **iv**

List of Figures **vii**

List of Tables **viii**

List of Listings **ix**

1 Introduction **1**
1.1 Thesis Contribution 1
1.2 Thesis Structure 2

2 Background **3**
2.1 Linux Environment and Processes in Linux 3
2.2 CRIU: Checkpoint Restore in Userspace 4
2.2.1 Comparison with other C/R (Checkpoint/Restore) tools 7
2.3 Migration Techniques 7
2.3.1 Cold Migration 8
2.3.2 Pre-copy Migration 9
2.3.3 MiGrror Migration 10
2.3.4 Post-copy Migration 11

3 Related Work **12**
3.1 Checkpoint Restore and CRIU 12
3.2 Migration Techniques 13
3.3 Algorithms or Applications of C/R and Migration 13

4 Architecture and Algorithm **16**
4.1 Comparison of MiGrror and Pre-copy 16
4.2 Algorithm 17
4.3 Architecture 19

5 Implementation **20**

5.1 Building blocks 20
5.1.1 Pagemap utilization for MiGrror 20
5.2 Podman commands 21
5.3 Code on the source 22
5.3.1 MiGrror 22
5.3.2 Pre-copy 22
5.4 Code on the destination 23

6 Evaluation **24**
6.1 Metrics and Experimental Outline 24
6.2 Experimental Results of the Metrics 25
6.2.1 Application Downtime 25
6.2.2 Total Network Usage 27
6.2.3 Total Migration Time 29

7 Conclusion and Future Work **31**
7.1 Conclusions and Lessons Learnt 31
7.2 Future Work 32

Bibliography **33**

Curriculum Vitae **36**

List of Figures

2.1 Cold Migration steps[32] 8
2.2 Pre-copy Migration steps[32] 9
2.3 Distinction between MiGrror and Pre-copy[35] 10
2.4 Post-copy Migration steps[32] 11

4.1 Overall Architecture utilized for the Algorithm 19

6.1 Application Downtime comparison of fixed memory dirty rate 25
6.2 Application Downtime comparison of variable memory dirty rate 26
6.3 Total network usage comparison for fixed memory dirty rate 27
6.4 Total network usage comparison for variable memory dirty rate 27
6.5 Total migration time comparison for fixed memory dirty rate 29
6.6 Total migration time comparison for variable memory dirty rate 29

List of Tables

2.1 Table of process specific entries utilized in the thesis [27] 3

4.1 Table comparing MiGrror and Pre-copy techniques 16

Listings

5.1 Pagemap command line 20
5.2 Podman commands 21

Chapter 1

Introduction

1.1 Thesis Contribution

The concept of migration and checkpoint/restore has been an important topic in research. Early research on migration primarily focused on cloud computing where latency is relatively low. However with emerging applications (e.g., augmented reality, virtual reality applications) there is a need for migration in geographically dispersed locations which typically have high latency between locations.

Migrating a service requires that the state of the service is preserved. This requires checkpointing the state and restoring it on a different server in multiple rounds to avoid a total loss of all data in case of a failure. There are many different types of migration techniques utilized such as cold migration, pre-copy migration, post-copy migration which will be explained in Chapter 2.

In the field of migration, new migration techniques and approaches are emerging, accompanied by new simulation results, such as the works by Arshin et. al [35]. Simulations provide only theoretical results; therefore there is a need of evaluation for these simulations in more realistically implemented environments to address their viability in production. The research gap to be addressed by this masters thesis is: What is the performance evaluation on containers between different types of migration techniques?

The research in this thesis addresses this research gap in two parts throughout this thesis: We create a shell script implemented environment for the migration technique proposed by Arshin et al.[35] and the standard Pre-copy migration technique widely used. An evaluation is conducted on the two techniques to observe the results on different metrics formulated. A comparison of the evaluations of both of the techniques reveals the disadvantages and advantages of each, addressing the viability of using the new migration technique.

1.2 Thesis Structure

Chapter 2 provides the background knowledge for this thesis, addressing different migration techniques and the checkpoint/restore tool CRIU. Chapter 3 discusses the related works of this thesis, including an expansion of the knowledge from Chapter 2 and the algorithms or applications of checkpoint/restore and migration. Chapter 4 presents the architecture and algorithm of the thesis, including the algorithm of the MiGrror technique, the main architecture utilized, and a comparison of the two main migration techniques in a tabular format. Chapter 5 details the main implementation of the thesis, explaining the building blocks, Podman commands, and source code for the source and destination nodes. Chapter 6 addresses the evaluation of the thesis, including the application downtime, total network usage, and total migration time. The results from the evaluation are compared between the two migration techniques to justify the viability of the MiGrror migration technique. Finally, Chapter 7 concludes the thesis and discusses future work.

Chapter 2

Background

The main goal of this thesis is to implement MiGrror technique of running containers and then do a comparison with Pre-copy(iterative) migration. It builds on migration techniques and its implementation in the CRIU project. This chapter provides a detailed introduction to these concepts as they are necessary to understand the contributions we present later on.

2.1 Linux Environment and Processes in Linux

This thesis uses the Linux kernel [40]. Processes in the kernel is used later on in the thesis for the checkpointing and restoring of processes(programs).

This work requires understanding of the processes (/proc) filesystem/directory [27].The /proc directory contains (among other things) one subdirectory for each process running on the system, which is named after the process identifier (PID). The following is a table of the process specific entries in /proc that are utilized for checkpoint and restoring.

Entry	Meaning
/proc/$pid/pagemap	Page table
/proc/$pid/fd	Directory, which contains all file descriptors
/proc/$pid/fdinfo	Information about opened file
/proc/$pid/maps	Memory maps to executables and library files
/proc/$pid/map_files	Information about memory mapped files
/proc/$pid/smaps	Showing memory consumption of each mapping and flags associated with maps
/proc/$pid/clear_refs	Clears page referenced bits shown in smaps output
/proc/$pid/task	Directory, which contains tasks
/proc/$pid/task/$tid/children	Information about task children

Table 2.1: Table of process specific entries utilized in the thesis [27]

2.2 CRIU: Checkpoint Restore in Userspace

Checkpoint/Restore in Userspace (CRIU) is an open-source C/R tool [20, 16] introduced in 2011, with a distinctive feature in that mainly implemented in userspace, rather than in the kernel.

The main goal of CRIU is to perform a snapshot of the current process' tree state, which represents a tree of all the subprocesses and superprocesses related to the current process, to a set of image files, so that it can be later restored at that exact point in time, without reproducing the steps that led to it.

Tracking Memory Changes

Tracking memory changes is essential for CRIU in order to determine the memory difference from different memory image dumps for restoring. This is also useful later in the implementation when implementing the MiGrror algorithm to detect any memory changes. To track the memory changes for different migration techniques:

1. Request that the kernel keeps track of memory changes by writing "4" into /proc/$pid/clear_refs file to clear the soft-dirty bit (a bit on the page table entry that tracks which pages a task writes to, for tracking the changes and then checkpointing/restoring) for each $pid we are interested in [21].
2. Retrieve the list of modified pages of a process by reading its /proc/$pid/pagemap file and looking at so called soft-dirty bit in the pagemap entries [21].

Checkpoint

The checkpoint procedure highly relies on the /proc linux root file system including:

- Files descriptors information (a list of all opened files and descriptors used by the process via /proc/$pid/fd and /proc/$pid/fdinfo).
- Pipe parameters (the parameters that explain the read and write end of the pipe, used for interprocess communication).
- A memory map represents where the process is mapped in the memory via /proc/$pid/maps and /proc/$pid/map_files/.

Checkpointing starts with retrieving the process identifier(PID) of the first process in the process tree(a tree of all the sub-processes, which are the processes launched by the parent process, that are used for a certain process) provided by the user through the command line –tree option [17]. Before starting to checkpoint the process, it is necessary that the process will not change their state. The state not only includes opening new files, sockets, changing session(group of processes) and others, but also producing new child processes [12]. To achieve

this transparently, instead of sending a stop signal (which could affect the process' state) CRIU freezes processes using ptrace's PTRACE_SEIZE command [25] or utilizing the freezer cgroup (which obtains a consistent image of the processes by attempting to force the tasks in a cgroup into a paused state where later can be gathered for the information then restarted) [29]. In order to find all active children processes of the first process in the process tree, the $pid dumper iterates through each /proc/$pid/task/ entry, recursively gathering threads and their children from /proc/$pid/task/$tid/children.

When all processes are frozen (the processes are only sleeping and not being altered, but not entirely stopped and deleted), CRIU collects all the relevant information about the process' task resources into image files. Virtual memory areas are parsed from /proc/$pid/smaps and mapped files are read from /proc/$pid/map_files links. File descriptors and registers are read, dumped via ptrace interface and parsed through /proc/$pid/fd and /proc/$pid/stat respectively. There is a novel technique in order to dump contents of memory and credentials which is referred to as **parasite code**.

Parasite code is a binary blob of code built in PIE(Position-independent executable) format based on machine code, for execution inside another process address space. The main and only purpose of the parasite code is to execute CRIU service routines inside dumpee tasks address space [22]. All architecture independent code calling for parasite service routines is sitting in parasite-syscall.c file. To run parasite code inside some dumpee process, CRIU carries out these steps:

1. Move task into the specified parent process identifier with ptrace(PTRACE_SEIZE, ...) which allows the parent process to inspect the task. The task is stopped without any signal triggered hence the state remains unaltered.

2. Inject and execute mmap syscall, which creates a new mapping in the virtual address space of the calling process, inside dumpee address space with help of the ptrace system call. When this stage is reached, there is a need to allocate a shared memory area which will be used for the stack and parameters exchange between CRIU and dumpee.

3. A local copy of shared memory space from /proc/$PID/map_files/ is created, where $PID is the process identificator of a dumpee.

4. The original dumper process retrieves information of the dumpee's address space through the parasite code either utilizing trap mode (one command at a time) or daemon mode (parasite behaves like UNIX socket).

5. With the data for the virtual memory areas and the flags from /proc/$pid/smaps and /proc/$pid/pagemap respectively, the parasite code then transfer the actual content through pipes, which in turn translates them into image files.

Lastly, the original process to checkpoint is cleaned of the parasite code by the ptrace facility again which drops the all the parasite code and restoring original code. CRIU then detaches from the processes and they continue to operate.

Restore

During the restore process, CRIU enters into the process it tries to restore. Initially, CRIU reads in image files (created from the checkpoint step) and finds out which processes share which resources(file descriptors, pipes). Later shared resources are restored by one process and all the others either inherit one on the 2nd stage (like sessions) or obtained in some other way. The latter is, for example, shared files which are sent with SCM_CREDS, representing UNIX credentials for authentication, messages via Unix sockets, or shared memory areas that are restoring via memfd file descriptor.

Next, since a checkpoint processes trees of processes rather than single processes, CRIU must fork itself many times on the processes, in the process tree to be restored, since every process tree is recursive with multiple processes within. CRIU requires that the restored tasks have the same PID they had before the dump. With clone3() system call vs original clone(), it becomes now possible to clone a process and specify the desired PID for it [9].

CRIU opens files, prepares namespaces, maps (and fills with data) private memory areas, creates sockets, then calls chdir() and chroot(). However, it restores memory mappings, timers, credentials and threads later.

In order to restore memory areas in-place, before exiting CRIU would have to unmap itself and map the application code. To overcome this issue, a similar approach to the parasite code one is followed, by the restorer blob. It is a position-independent executable that does what is stated above and then allows to restore the process successfully.

Live migration with CRIU

CRIU operates by design on a single system. Supports for live migration requires further additions [19]. In particular, it is up to the user to ensure that the dump files are on the remote host upon restore. Furthermore, IP addresses used by the application in the original host, must also be available in the new restored host. CRIU developers have implemented go-criu [14] in go which is a continuation of the original p.haul project(no longer maintained) in python for migration.

One approach to live migration is to use the iterative approach like Pre-copy, where after a certain time interval there is a transfer of execution state and memory pages pushed from the source to the destination, as we will cover in 2.2. However, support for lazy migration, where the memory pages are pulled from the destination to the source when there is a difference of the pages between the source and destination, and a page server is also available [18]. A major drawback with iterative migration is that, as explained before, CRIU freezes the process while the snapshot takes place. This means that some memory pages may have dirty memory during the snapshot.

2.2.1 Comparison with other C/R (Checkpoint/Restore) tools

The main differences between C/R tools are the way they interact with the kernel and the application use cases. CRIU is implemented completely in userspace, and as a consequence relies heavily on existing kernel interfaces, otherwise execution fails. Additionally, CRIU's target application are containers.

Other open-source tools that implement C/R are DMTCP [36], BLCR [1] and FTI [4]. They all focus on high performance computing.

DMTCP

Distributed Multi-Threaded Checkpointing (DMTCP)[2] is an active project lead by Prof. Cooperman at Northeastern University that implements C/R on a library level. This means that if a user wants to checkpoint an application, this must be dynamically linked from the very beginning and executed with custom wrappers (which decreases transparency). It works under Linux, with no modifications to the Linux kernel nor to the application binaries. It can be used by unprivileged users. DMTCP intercepts all system calls instead of assuming existing kernel interfaces, as CRIU does, and is, as a consequence, more robust and reliable. It is very popular in HPC environments.

BLCR

Berkeley Lab Checkpoint/Restart (BLCR) is a system-level checkpointing tool aimed also at High Performance Computing jobs. It requires loading an additional kernel module and is currently not maintained (last supported kernel version is 3.7).

A detailed table comparing the software is presented here, and some other solutions, is maintained by the CRIU foundation [15].

FTI

FTI stands for Fault Tolerance Interface and is a library that aims to give computational scientists the means to perform fast and efficient multilevel checkpointing in large scale supercomputers. FTI leverages local storage plus data replication and erasure codes to provide several levels of reliability and performance. FTI is application-level checkpointing and allows users to select which datasets needs to be protected, in order to improve efficiency and avoid wasting space, time and energy. In addition, it offers a direct data interface so that users do not need to deal with files and/or directory names [5].

2.3 Migration Techniques

Container migrations include either stateless or stateful migrations. The former is based on stateless containers where it does not retain persistent data such as the container state. Therefore, once on the destination node, the container restarts from scratch. Stateless migration

consists of two steps: (i) Instantiation of a new container on the destination node; and (ii) The deletion of the old container on the source node. With stateful migration the volatile and persistent states of the container are made available at destination once migration is completed. This section focuses on stateful container migration. More specifically, this section provides an overview of the many techniques that may be adopted distinguishing between cold(the first technique) and live migration techniques(the remaining three techniques).

2.3.1 Cold Migration

The steps for cold migration is depicted in Figure 2.1. (i) the source device container is stopped so that the state is not modifiable, (ii) the state is dumped (fully checkpointed on the current device) and transferred to the other device, and finally (iii) the container is finally resumed on the destination device only when the state is available [32]. As a result of this, there is a long downtime, namely the time where the container is not up and running. This indirectly also affects the total migration time, the total time to migrate the container from one device to another. However, we highlight that the memory pages are only transferred one time which should reduce the total migration time and the overall amount of data transferred during migration.

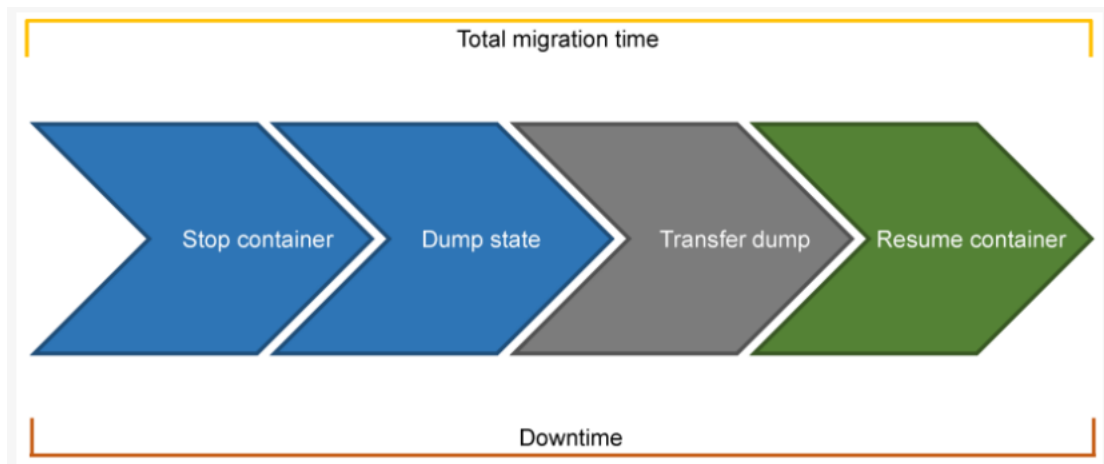

Figure 2.1: Cold Migration steps[32]

2.3.2 Pre-copy Migration

Pre-copy migration transfers most of the state prior to freezing the container for a final dump and state transfer, after which the container runs on the destination node. It is also known as iterative migration, since it may perform the Pre-copy phase through multiple iterations such that each iteration only dumps and retransmits those memory pages that were modified during the previous iteration without stopping the source. The modified memory pages are referred to as dirty pages. After a number of iterations the container is then suspended on the source node in order to capture the last dirty pages along with the modifications in the execution state and copy them at destination without the container modifying the state again. Finally, the container resumes on the destination node with its up-to-date state [32]. Figure 2.2 shows the technique with only one step Pre-copy step but maybe be expanded for multiple Pre-copy steps too.

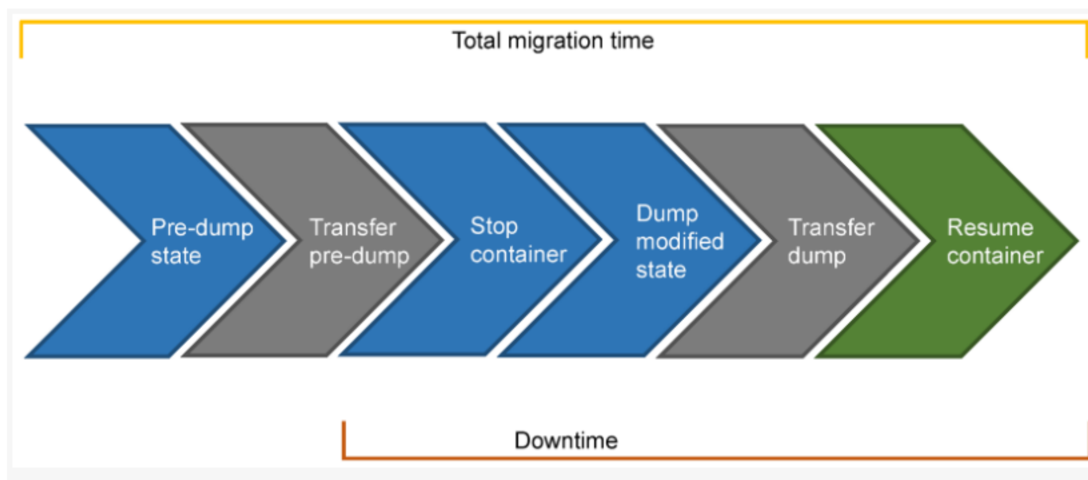

Figure 2.2: Pre-copy Migration steps[32]

The main difference between cold and Pre-copy migrations lies in the nature of their dumps. The dump in cold migration represents the whole container state (as the pre-dump in Pre-copy migration) and thus always includes all the memory pages and the execution state. The dump in Pre-copy migration includes those memory pages that were modified during the Pre-copy phase, together with the changes in the execution state. As such, downtime for Pre-copy migration should be in general shorter than that for cold migration since less data is transferred while the container is stopped. However, downtime for Pre-copy migration is not deterministic, as it significantly depends on the number of dirty pages. Therefore, we expect Pre-copy migration to be affected by the two factors that may increase the number of dirty pages: 1) the page dirtying rate(how frequently are pages being changed) at which the service modifies memory pages 2) amount of data that are transferred during the Pre-copy phase.

2.3.3 MiGrror Migration

MiGrror migration is migrating and mirroring put together. It is similar to Pre-copy. Figure 2.3 depicts the distinction between the MiGrror and Pre-copy approaches. Assume a user equipment(UE) is moving from one node to another. As illustrated in Figure 2.3, Pre-copy transmits dirty memory at the end of a round representing a predefined amount of time. With MiGrror, the goal is to reduce the amount of data that must be transferred during downtime in order to achieve higher performance; therefore, there was a focus on reducing transfer. The distinction between the two methods is that MiGrror, as shown at the bottom of Figure 2.3, uses events to synchronize (sync) the source and destination as events occur, rather than waiting for the end of a round as Pre-copy does. Each memory change at the source causes an event to be generated, indicating that the source and destination must be synced. MiGrror does not need to wait for a period of time to elapse. Instead, MiGrror allows the possibility of multiple synchronizations of the source and destination during the period of time that corresponds to Pre-copy's round in order to mirror the current VM/container available at the destination. These number of MiGrror sync events(variable n) and number of rounds of Pre-copy(variable m) are depicted in Figure 2.3. In most cases, n is expected to be larger than m since MiGrror syncs as soon as a memory change occurs and sends memory differences(image transfer) as soon as they become available. A small amount of dirty memory remains when hand-off is triggered. After the hand-off trigger, this data is the last memory difference that the source sends to the destination for synchronization. This is less memory than the last round in Pre-copy since other memory differences have already been transmitted. As a result, the image transfer is reduced in MiGrror when compared to Pre-copy. Consequently, as shown in Figure 2.3, downtime is reduced compared to Pre-copy. The diagram's right- most section represents the resumption time required to restart the VM/container at the destination [35].

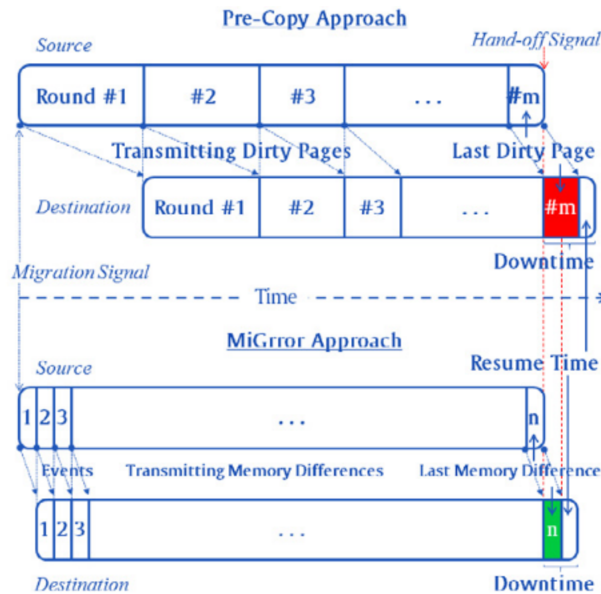

Figure 2.3: Distinction between MiGrror and Pre-copy[35]

2.3.4 Post-copy Migration

Post-copy migration suspends the container on the source and copies the execution state to the destination so that the container can resume its execution there. It copies all the remaining state, namely all the memory pages. Three variants of post-copy migration [7], which differ from one another on how they perform this second step. We will only describe the post-copy migration with demand paging variant, better known as lazy migration (see Figure 2.4), which is the only one that may be currently implemented using the functionalities provided by CRIU (see https://criu.org/Lazy_migration), e.g., the -lazy-pages and -page-server options. With lazy migration, the resumed container tries to access memory pages at destination, but, since it does not find them, it generates page faults. The outcome is that the lazy pages daemon at destination contacts the page server on the source node. This server then "lazily" (i.e., only upon request) forwards the faulted pages to the destination [32].

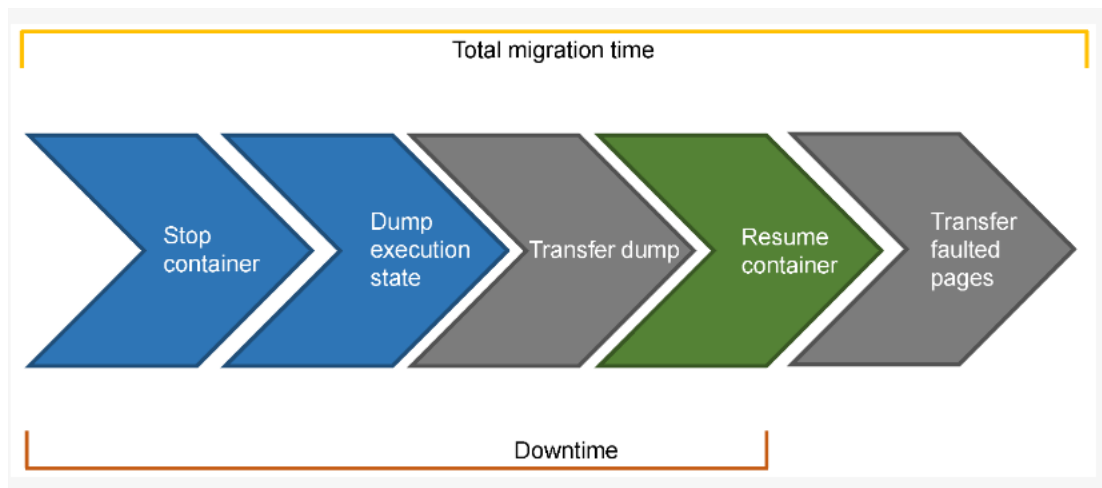

Figure 2.4: Post-copy Migration steps[32]

The main benefits of post-copy migration is that memory pages are copied only once. Therefore, it should transfer a data volume that is comparable with that of cold migration and with that of the Pre-copy phase of Pre-copy migration. For post-copy migration, downtime is irrespective of the page dirtying rate. There are however two main drawbacks. Firstly, page faults degrade service performances, as memory pages are not immediately available at destination once the container resumes. This is not acceptable for infrastructure that requires ultra-low latency. Secondly, the migration process distributes the overall up-to-date state of the container between source and destination compared to the whole up-to-date state for cold migration and Pre-copy migration. Therefore if there is a failure on the destination node, it may not be capable to recover the most up-to-date state for the post-copied container [32].

Chapter 3

Related Work

In this chapter we introduce the most relevant pieces of related work addressed in this thesis, together with similar approaches to tackle live migration of processes, containers or other applications. We also include, given the educational nature of this work, references on the bibliography we have based our claims on, together with the materials used throughout our learning process as we understand it is relevant in the frame of a Master's thesis.

3.1 Checkpoint Restore and CRIU

First, we have a starting point definition from the Encyclopedia of Parallel computing [37]. Within this chapter, it first briefly explains what is checkpointing and then explains the difference between system and application checkpointing. We also leveraged a set of slides by Brandon Barker from Cornell University [3]. There, the author does a non-scientific introduction and motivation for C/R and goes on to cover the different available tools. Program memory, PIDS, shared memory segments etc. are saved. For the distributed processes, coordination of checkpointing across all processes is needed as well. The applications of C/R include recovery/fault tolerance, saving scientific interactive sessions, reducing long initialization times, debugging, migrating processes and interacting and analysis results of in progress CPU-intensive processes. There are a variety of C/R tools being used. Barker [3] focuses on high performance computing, where DMTCP [36] is the software of choice. The paper does a great job explaining differences of the different tools being used. Some of the origins of ways for rollback recovery strategies for fault tolerant systems are highlighted by the work of Elnozahy et al. from 2002 [11]. Some of these strategies gained further traction with VM migration, a topic outlined by Clark's survey [8].

For our project, we used CRIU as the base C/R tool as it was the most suitable one for containers and was already used by many major container engines and runtimes. Checkpoint-Restore in Userspace [20] is an open-source community-driven project. Therefore, it has a very actively maintained wiki covering all related topics. Adrian Reber is a maintainer of the project in charge of, among others, part of the integration with runC and podman checkpoint/restore, and has a set of very interesting an easy-to-follow articles on CRIU. One of the most delicate parts of process restore is how to handle old, new, and dependent process identifiers (PIDs).

Adrian also has an article describing how this is done in CRIU [34] before the introduction of new clone function [9].

When comparing different C/R tools, and in addition to the previously mentioned work by Barker [3], CRIU's developers themselves maintain a comparative table where they list the pros and cons of each different tool (namely CRIU, DMTCP, BLCR) [15]. In spite of the natural biases they may have, the resource has plenty of detail and is of great use. The main alternative to CRIU for C/R is the Distributed-MultiThreaded Checkpointing project [36] (DMTCP). Developed under the supervision of professor Gene Cooperman from Northeastern University, the project has a long-standing record of successes in the high performance computing domain, being the tool of choice by several national laboratories in the US. Additionally, the Berkeley Lab's Checkpoint-Restart [1] (BLCR) is also an HPC-focused tool, although it has lost some traction during the last years as it is not maintained. Another HPC-focused tool on the application level checkpointing is FTI [4]. Developed under supervision of Dr. Leonardo Bautista-Gomez, it is a library that aims to give computational scientists the means to perform fast and efficient multilevel checkpointing in large scale supercomputers.

A stretch goal for this project would be to implement live migration or MiGrror migration of distributed container deployments, for which distributed checkpointing algorithms would be crucial. Even though we have not had time to address the implementation of such a concept, we have used several well-established resources for documentation purposes. We would like to highlight the works by Raynal [33] and Kshemkalyani [26].

3.2 Migration Techniques

Migration is one of the main reasons for utilizing checkpoint restore as stated before in section 3.2. There are many methods of migration that are utilized to accomplish migration from one container to another.

For most of the techniques, we refer to the paper written by Puliafito et al.[32]. In this paper they first split up migration techniques into stateful and stateless migrations. They then explain in detail the different types of stateful migration techniques used such as cold migration, pre-copy migration and post-copy migration. For the main migration technique we are approaching in this project for implementation, we use the paper by Rezazadeha et al. [35]. This is the main paper we use to understand how their technique is different from the usual de-facto pre-copy migration used by many container engines and runtimes.

3.3 Algorithms or Applications of C/R and Migration

Even though C/R and live migration are a relatively mature topic of research, scientific contributions covering particular applications are way more scarce. This was, among others, one of our initial motivations for this work. Most of these related works are complementary to our work utilizing instead the MiGrror algorithm for the migration parts of the applications.

On the topic of checkpointing, we would like to highlight the article by Fernando [13]. It proposes PostCopyFT, that superimposes a reverse incremental checkpointing mechanism over the forward transfer of VM states on recovering from post-copy migration failures. It utilizes heartbeats to monitor liveness and reachability. Another topic on checkpointing abit different relating to checkpointing on applications, we would like to highlight the work by Huang [24] that adds a checkpoint restore feature into docker swarm by utilizing CRIU to further make it fault tolerant for orchestration. They propose to use direct or incremental ways to do the checkpointing.

For application-oriented projects leveraging CRIU we would like to highlight the work by Venkatesh et al. [41]. In particular, the authors present an optimization to the file-based image procedure using the new (as of 2019) kernel support for multiple independent virtual addresses space (MVAS). We can not leverage the findings in our project as it only focuses on single-machine C/R.

A contribution to CRIU which stemmed from an application use case and which we could leverage in the project was presented by Stoyanov et al. [39]. The author optimizes down-time during container live migration by utilizing CRIU's newly added feature: the image cache/proxy.

For comparison of container and vm migration, lies a piece of by Bhardwaj et al.[6]. This work is mainly focused on the comparison of containers vs vm migration by contributing a testbed of the container based migration. They experiment on different types of applications to compare the metrics between container and vms.

In the HPC domain, but focused on container migration, lie the piece of work by Sindi [38]. It showcases different applications of CRIU's migration capabilities in HPC. In particular, the authors present a migration of an MPI application.

A contribution that is still very closely related to our work is displayed by Htet et al. [23] presents a job migration function using CRIU on UPC(User-PC computing) systems(It uses idling resources of personal computers (PCs) for daily usage by users as the workers, to run the requested jobs or application programs that may need various environments on Docker containers). This is quite useful to our work because we utilize podman to try to test migration except that we are trying to use a diferent type of migration process while they are utilizing cold migration.

Lastly, the contribution that most closely relates to our goal of providing an efficient, transparent, easy-to-use migration library for running containers is the go-criu project [14]. Initiated by the same CRIU developers, the work attempts to wrap all the technical details behind efficient live migration and deliver it as a solution to the end user. This replaced the original P.Haul project that was implemented in python. The Podman commands used are based off this project so that we are able to implement the Pre-copy and MiGrror algorithms.

The literature has been predominantly orchestration and management applications on the

existing Pre-copy migration technique. This limits exploration of other migration techniques on mobility applications or other applications. This gap will be addressed by utilizing MiGrror migration technique on some simple programs and evaluating its future viability for varieties of applications including mobility applications(smart tours).

Chapter 4

Architecture and Algorithm

This chapter describes the architecture and the algorithms used for migration.

4.1 Comparison of MiGrror and Pre-copy

This section describes a comparison of MiGrror and Pre-copy

Pre-copy	**MiGrror**
Utilizes iterative transfer of state	Utilizes iterative transfer of state
Checkpoint full image of state and transfer image to the destination	Checkpoint full image of state and transfer image to the destination
Iterative transfer of state based on time interval changes	Iterative transfer of state based on memory changes
A set amount of time before migration	Migration dependent on number of memory changes

Table 4.1: Table comparing MiGrror and Pre-copy techniques

The MiGrror migration technique is a memory change based iterative migration technique. In order for MiGrror to work, there is a need to count the memory changes that occurs to determine the next transfer of the image. This requires a counter that keeps track of memory changes that have occurred from the last transfer of the image . The number of changes before a migration can be one or more. The flexibility of a varied number of changes before migration is useful for a better control of the I/O. The control of the efficiency on the I/O is mainly dependent on how much time the I/O is not idling and doing useful work. For higher memory change frequencies (shorter time before a memory change) it would be more ideal to have a higher number of changes before a migration in order to not overload the I/O with too frequent image transfers, while lower memory change frequencies (longer time before a memory change) would be more ideal to have lower number of changes before migration to allow for the I/O to not be idling and efficiently image transferring. A variable is needed for the total number of memory changes before an image transfer occurs. As the total number of memory changes increases before an image transfer occurs, the greater the difference between two memory states which incur higher downtime for calculating and applying the memory state

differences.

The Pre-copy migration technique is based on time-interval changes. This allows the MiGrror technique to have one main advantage over the Pre-copy technique: fewer chances of missed data transfer when some memory changes occur between the time intervals of the Pre-copy technique. Iterative transfer of state is done with pre-checkpoints(the difference in memory state is stored in an image file instead of the whole memory state and then sent to the destination). As the number of total image transfers needed before hand-off increases, the higher the downtime for the destination node to restoring the pre-checkpoints but the lower the downtime needed for the final checkpoint restore on the destination after the source hand-off.

This means that both types of migration techniques are flexible. These techniques can tune their variables for different types of migration use cases so that they are more suitable for those specific use cases such as when there is better throughput and I/O use lower values for variable tuning while slower throughput and I/O use high values for variable tuning. Examples of the higher throughput and I/O include virtual reality application or tour car applications. Examples of slower throughput and I/O include bank applications or school applications.

4.2 Algorithm

This section describes the algorithm for MiGrror.

Algorithm 1 Source algorithm

```
1: trigger migrationRequest;
2: handoffSignalReceived = False
3: totaltransfers = 3
4: totalchanges = 2
5: currenttransfers = 0
6: currentchanges = 0
7: while handoffSignalReceived == False do
8:     if currenttransfers != totaltransfers then
9:         if Memory Change then
10:            if currentchanges<totalchanges then
11:                currentchanges+=1
12:            else
13:                pre_checkpoint()
14:                send_pre_checkpoint()
15:                currentchanges = 0
16:                currenttransfers+=1
17:    else
18:        handoffSignalReceived = True
19:        checkpoint_and_stop_container()
20:        send_checkpoint()
21: wait for t seconds then release VM/Container
```

Algorithm 2 Destination algorithm

```
1: trigger migrationRequest;
2: handoffSignalReceived = False
3: while handoffSignalReceived == False do
4:     case Memory Change
5:         receive pre_checkpoint;
6:         calculate and apply memoryDifference;
7:         restore VM/container
8:     case handoffRequest
9:         handoffSignalReceived = True
10:        receive checkpoint
11:        restore VM/container
12: communicate from this Node onwards;
```

The source node is the node that is currently providing services to end-user applications. After these steps are completed, the destination node will provide the service. The specifics of both the **source and destination** steps are described in the rest of this section.

S1: Source Node memory changes, pre-checkpoint, pre-checkpoint transfer

The source node triggers a migration request signal to start the migration from source node to the destination node(algorithm 1, line 1 and algorithm 2, line 1). The hand-off signal is set to false on both the source and destination node(algorithm 1 line 2 and algorithm 2 line 2). The total number of image transfers before hand-off(the total transfers variable) is initialized to 3

as an example, the total number of memory changes before an image transfer(the total changes variable) is initialized to 2 as an example respectively(algorithm 1 line 3 and 4). The *currenttransfers* variable for image transfers completed and *currentchanges* variable for the memory changes occurred are both initialized to 0(algorithm 1 line 5 and 6). While the hand-off signal has not been received yet(algorithm 1 line 7), the *currenttransfers* variable is compared with the totaltransfers variable. If the two values are not equal(algorithm 1 line 8), on a memory change and the *currentchanges* variable is less than *totalchanges* variable(algorithm 1 line 9 to 10), the algorithm increments the current number of memory changes(algorithm 1 line 11). If the *currentchanges* variable reaches the *totalchanges* variable, the algorithm will create a pre-checkpoint image of the container's memory state and send the pre-checkpoint image to the destination(algorithm 1 line 12 to 14). The algorithm restores the current number of memory changes to 0 to prepare for another image transfer and increment the current amount of image transfers by 1(algorithm 1 line 15 to 16).

S2: Calculate and Apply Memory Difference then Restore VM/Container in the Destination Node

The pre-checkpointed image from the source node is received by the destination, where the destination node container is paused in order to calculate and apply the memory difference changes between the the received image on the destination and its current image(algorithm 2, line 3 to 6). It then restores and resumes the VM/container with the new memory difference changes(algorithm 2, line 7).

S3: Stop Source VM/Container and Hand-off, Service Running on the Destination Node

The source node checkpoints and stops the current running container then sends the most recent memory state image to the destination after the handoffSignalReceived variable is set to True(algorithm 1 line 17 to line 20). The destination now has all of the fully updated data from the source(algorithm 2 line 8 to line 11). Control is handed off to the destination node after the hand-off is triggered (algorithm 2, line 12). Simultaneously, the destination node will proceed to step S1 in order to prepare for future possible migrations.

S4: Clean Up the Source Node

Removing everything from the existing source container/VM will result in a cleaned source node. Algorithm 1, line 21 represents the waiting time before the migrated VM/container is released. The benefit of the waiting period is that if the end-user moves back or it may serve as a back-up if the next node's connection is lost.

4.3 Architecture

This section describes the architecture related to the algorithm described in 4.2. This section describes the architecture: the operating system(OS) used, the container engine chosen and the containerfile/image used. Below is the overall architecture utilized.

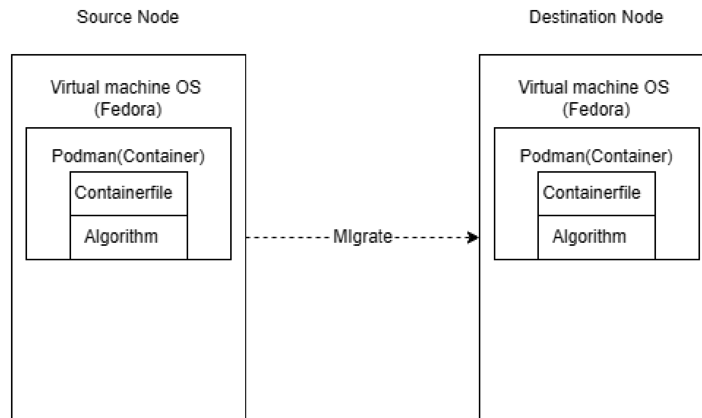

Figure 4.1: Overall Architecture utilized for the Algorithm

In the top level architecture, Fedora OS was used instead of the more commonly used Ubuntu OS. The reason for utilizing Fedora was that there were some missing features in Ubuntu OS that did not allow for the utilization of checkpoint and restore in the kernel. This feature was the core principle of the thesis to conduct performance evaluation with checkpoint and restoring.

In the next level, *podman* was selected as the container engine. Although Docker is the more favored container engine in most use cases, the main issue was no updates for the checkpoint restore functionalities in Docker. The functionality was part of the experimental branch of Docker but had bugs of not allowing for the checkpoint and restore of containers. On the other hand, podman allowed for the utilization of those features seamlessly on the Fedora OS so that was utilized instead.

In the next level, the Containerfile is utilized for the configuration of the podman containers. In the beginning of the Containerfile, the busybox image from Docker is utilized. The reason for that is the minimal overhead and bloat from the image when starting it up as a container.

Chapter 5

Implementation

This chapter explains about the implementation relating to the two migration techniques.

5.1 Building blocks

This section explains about the program that is utilized in the implementation for memory changes detection for the MiGrror algorithm.

5.1.1 Pagemap utilization for MiGrror [1]

The functionality of the pagemap program is a userspace tool to map virtual page addresses to physical addresses, based on the /proc/pid/pagemap interface described in kernel docs (pagemap.txt)[2] which is used as an implementation for the memory change detection in algorithm 1 on line 9 and in algorithm 2 on line 4 in chapter 4. This tool is used to find the virtual page addresses that are changed by the soft-dirty bit flag utilized by the kernel in the softdirty.txt[28] documentation in order to find the memory changes of a program(process).

To utilize the pagemap code [10], the following command line code is used to save and compile as an executable file pagemap2 to detect changes in the memory pages of processes as:

```
./pagemap2 $$ | grep 'soft-dirty 1'
```

Listing 5.1: Pagemap command line

where $$ represents the container PID. The goal of the command is to retrieve all the pages of the container PID first, then pipe it into the grep Linux tool and retrieve pages that have a ”soft-dirty 1” bit present which means that there are changes in the relevant page.

[1] https://github.com/liang995/thesis/blob/main/pagemap2.c
[2] https://www.kernel.org/doc/Documentation/vm/pagemap.txt

5.2 Podman commands

This section gives a breakdown on some of the podman commands utilized for the implementation and where they are used.

The goal of the project is attempting to accomplish container migrations on podman containers. Below will be a list of the podman checkpoint[30], restore[31] commands and other commands that are utilize to build the implementation for the two migration techniques:

```
1 podman container checkpoint -P -e pre-checkpoint.tar -l
2 podman container checkpoint -R --compress=none --export=checkpoint.tar
3 podman container checkpoint srcimage --compress=none --export=checkpoint.
    tar
4 podman container restore --import-previous pre-checkpoint.tar --import
    checkpoint.tar
5 podman container restore --import checkpoint.tar
6 podman run -d --name=srcimage testimage
7 podman rm srcimage
```

Listing 5.2: Podman commands

The first command is for pre-checkpointing the container(checkpoint only changes to the container memory pages).
The second command is for checkpointing which doesn't stop(for the iterative part).
The third command is for full checkpointing that stops the container.
The fourth command is for restoring from a pre-checkpoint.
The fifth command is for restoring from a full checkpoint.
The sixth command is for starting the container from image name srcimage with name testimage.
The last command is for removing the srcimage image

5.3 Code on the source

In the next two sections the implementation of the two migration techniques on the source and destination node are explained. The link to the source code for both of the techniques are attached as footnotes in the sections.

Due to the Docker's checkpoint restore feature not working properly and no existing implementations of MiGrror, the source code for both technique was programmed from scratch in shell script.

5.3.1 MiGrror[3]

The implementation for MiGrror utilizes a counter attached with the pagemap program searching through the current process memory pages and detecting memory changes in the process and triggering migrations in regards with the counter.

In this implementation, the original steps of the pseudocode algorithm for sending the base image is done beforehand. It then starts the container from the image and gets the pid of the process that starts printing numbers as dirty values. The program then creates a checkpoint image of the container without stopping the container from running(as by default the checkpoint implementation stops the running container)then send the checkpoint to the other server(algorithm 1 before line 1). The program then waits for a set amount of time before starting the migration process(algorithm 1 line 1). The program initializes variables similar to the algorithm 1 line 2 to 6 and then initializes prefix names and suffix names to uniquely identify every pre-checkpoint that will be sent to the destination. The pagemap program is used to check for any changes and if the current number of changes is less than maximum allowable changes before migrating and the current number of migrations is less than the total migrations before hand-off, the current number of changes is incremented by 1(algorithm 1 line 7 to 11). Otherwise if the current number of changes reaches the total changes before migration but the current migration number has not reached the maximum allowable image migrations before hand-off, the source node will create and send uniquely identified pre-checkpoint images with names previously initialized for their the prefix and suffix, to the destination node(algorithm 1 line 13 to 14). The current number of changes is reset to 0 and current number of migrations is incremented by 1(algorithm 1 line 15 to 16). When the current number of migrations is the total migrations before hand-off, the final full checkpoint image is created to stop the program and send the checkpoint to the destination (algorithm 1 line 18 to 20). The program waits for some time before the container is released(algorithm 1 line 21).

5.3.2 Pre-copy[4]

In this implementation most of the steps previously described are reused from the MiGrror implementation with some differences. As the difference in memory changes in not required for the Pre-copy algorithm, the step for getting the pid for the writing number dirty memory program and the variables that are related to finding number of memory changes before a

[3]https://github.com/liang995/thesis/blob/main/src.sh
[4]https://github.com/liang995/thesis/blob/main/pre_src.sh

migration are removed. In the actual migration steps, instead of utilizing pagemap for finding a memory change and incrementing the changes, the program only needs to wait a set amount of time before a image migration is finished.

5.4 Code on the destination[5]

Regardless of the migration technique, the same implementation on the destination is utilized. As both of the migrations utilize iterative pre-checkpoint image transfers, the main difference is only on their source implementations' timing for when the image transfer is done.

The image is already received on the destination before the start of the program. The program starts the VM container, receives the pre-dump from the source, applies the pre-dump and then restores the operation of the destination container(algorithm 2 before line 1). There is a set amount of time to wait before starting the migration process(algorithm 2 line 1). If the signal for hand-off is not sent to the destination, as soon as a different uniquely named pre-checkpoint images file is received, the program calculates and applies the difference from the source then restores them on the destination to lower the downtime of the full restore(algorithm 2 line 4 to 7). When the hand-off signal from the source is triggered, a full restore with the last checkpoint sent over by the source to hand-off is finished(algorithm 2 line 8 to 11). This destination node now becomes the new source node for migrations(algorithm 2 line 12).

[5]https://github.com/liang995/thesis/blob/main/dst.sh

Chapter 6

Evaluation

This chapter explains about the evaluation of the thesis' migration techniques between Pre-copy and MiGrror migration on different metrics such as migration downtime, total data transferred and total migration time.

6.1 Metrics and Experimental Outline

In this section, we study the impact of different dirty memory frequency rates(how often to dirty the memory) and dirty memory sizes(values) for the two types of migration techniques on the metrics of: application downtime, total network usage and total migration time.

As we will compare MiGrror and pre-copy migration, we establish parameters for: the number of pre-checkpoint migrations before we fully migrate and stop(handoff), and the number of changes(MiGrror) or time intervals(pre-copy) before every pre-checkpoint migration. The values selected for the parameters increase proportionally between the MiGrror and pre-copy techniques. For example 1 memory change and 2 memory changes before transfer for pre-copy versus 100ms and 200ms before transfer for MiGrror so that both techniques checkpoint on the source, and transfer to the destination the same number of times before handoff. We use the same types of memory dirty rates and memory dirty sizes(values) for every different type of metric.

In order to test the above metrics, we set up the following experiments running them 5 times. We deploy two podman containers on both VMs, the source and the destination, with the same podman image while running. We use a bash script. In each run, we use dirty memory values: ”20” represents *input20*, ”2020” represents *input2020* and ”20202020” represents *input20202020* in the graphs in the next section for the metrics. For each run, we have a fixed memory dirty frequency rate and a variable memory dirty frequency rate. For the fixed memory dirty frequency rate, a dirty memory value is created every 100, 200 or 300ms. In particular for the variable memory dirty frequency rate, we have:

test1: variable dirty frequency rate where first time a dirty memory is created 50ms after, second time 100ms after and third time 150ms after, then repeated.(50/100/150ms frequencies)

test2: variable dirty frequency rate where first time a dirty memory is created 150ms after, second time 200ms after and third time 250ms after, then repeated.(150/200/250ms frequencies)

test3: variable dirty frequency rate where first time a dirty memory is created 250ms after, second time 300ms after and third time 350ms after, then repeated.(250/300/350ms frequencies)

6.2 Experimental Results of the Metrics

The main application for the experiments is a container that outputs in varying times, the given memory values from the experimental outline in the previous section. The results of the metrics from the previous section are explained in the following subsections.

6.2.1 Application Downtime

In this subsection, we study the application downtime metric for the two types of migration techniques.
We present our results in Figures 6.1 and 6.2.

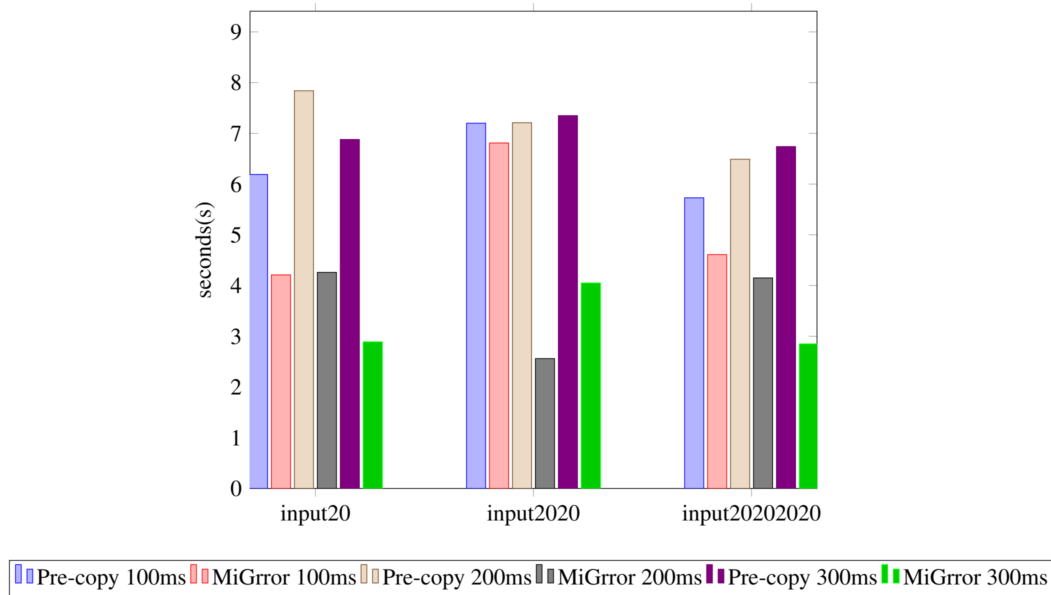

Figure 6.1: Application Downtime comparison of fixed memory dirty rate

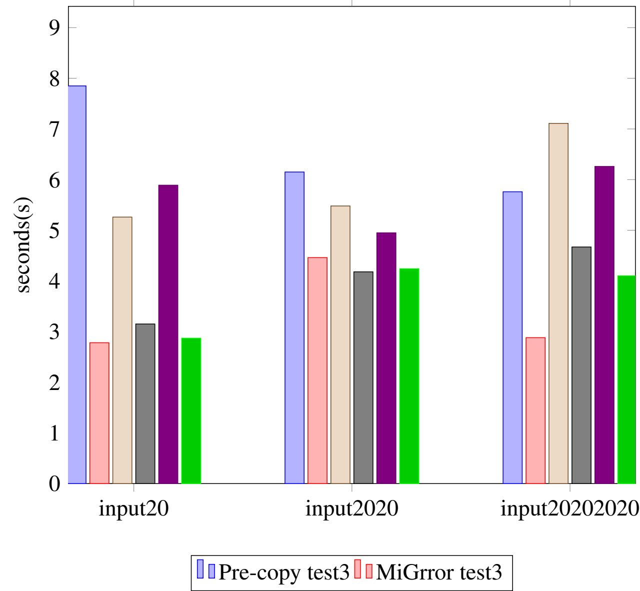

Figure 6.2: Application Downtime comparison of variable memory dirty rate

We observe that, for our particular setting, regardless of the type of input dirty memory size, the MiGrror migrated container had a lower application downtime. Moreover it is also interesting to see that, the comparison of the application downtime for the all the MiGrror migrated containers did not always decrease with higher dirty memory rate. The results are specific to our setting with two virtual machines and limited memory, but a similar benchmark could be reproduced in production on larger applications too.

6.2.2 Total Network Usage

In this subsection, we study the total network usage metric for the two types of migration techniques.
We present our results in Figure 6.3 and 6.4.

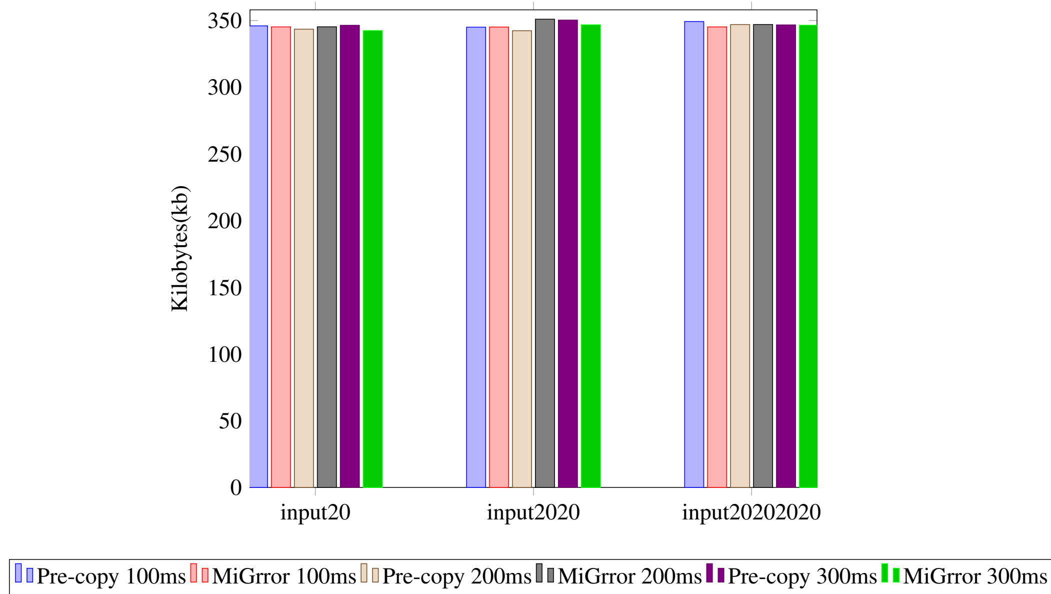

Figure 6.3: Total network usage comparison for fixed memory dirty rate

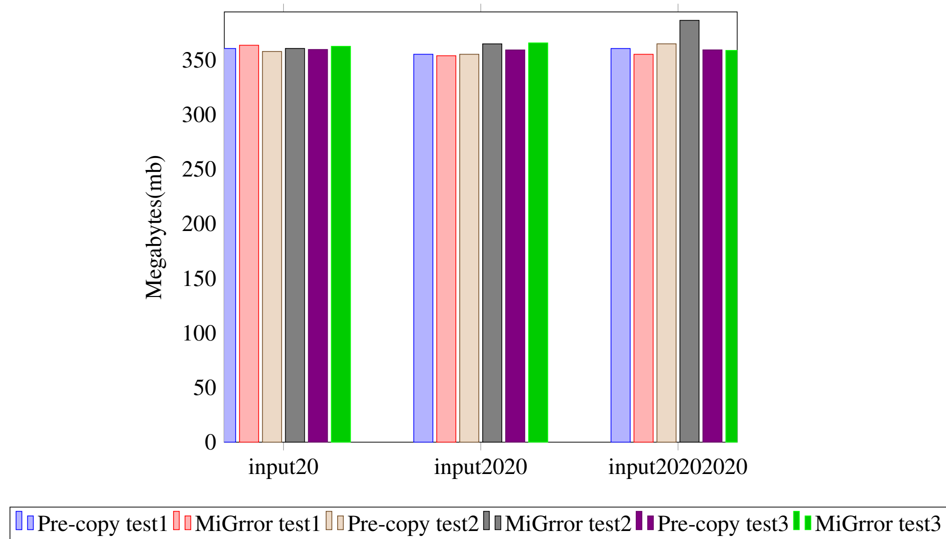

Figure 6.4: Total network usage comparison for variable memory dirty rate

From our results, we are able to extract different conclusions. First of all, the MiGrror results resulted in a little bit more of data transfer compared to pre-copy. This is as expected because more transfers may result in the MiGrror containers compared to the pre-copy which

means more overhead. Lastly, we notice that most of the network transfer really do not vary a whole lot between the pre-copy and MiGrror containers, mostly hovering around 350 mbs. This is expected because the changes are not huge differences so the checkpoints would not usually grow exponentially with size.

6.2.3 Total Migration Time

In this section, we study the total migration time metric for the two types of migration techniques.
We present our results in Figure 6.5 and 6.6.

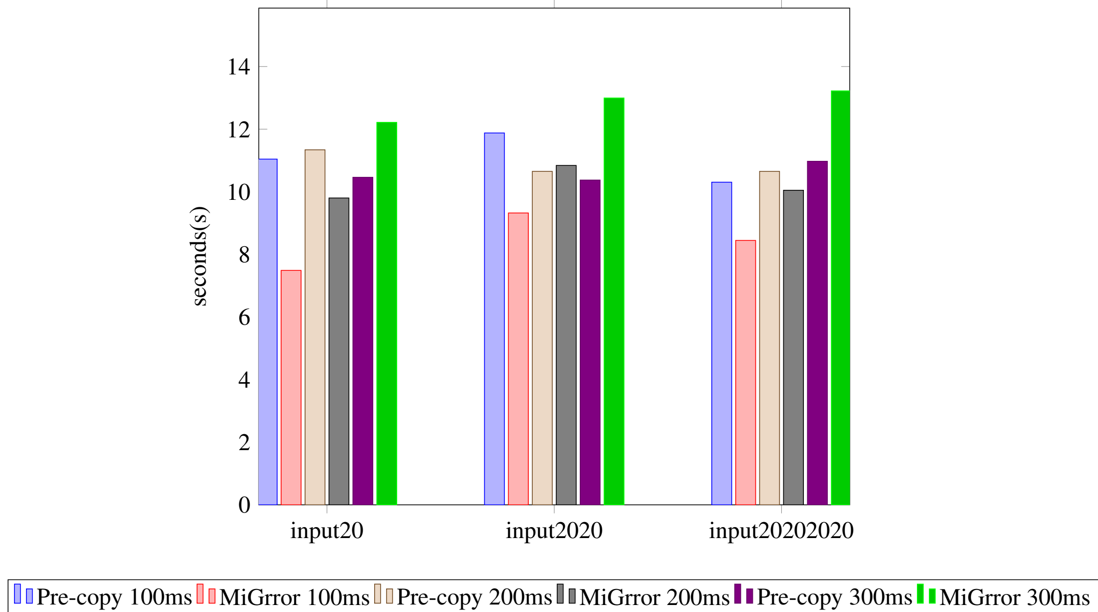

Figure 6.5: Total migration time comparison for fixed memory dirty rate

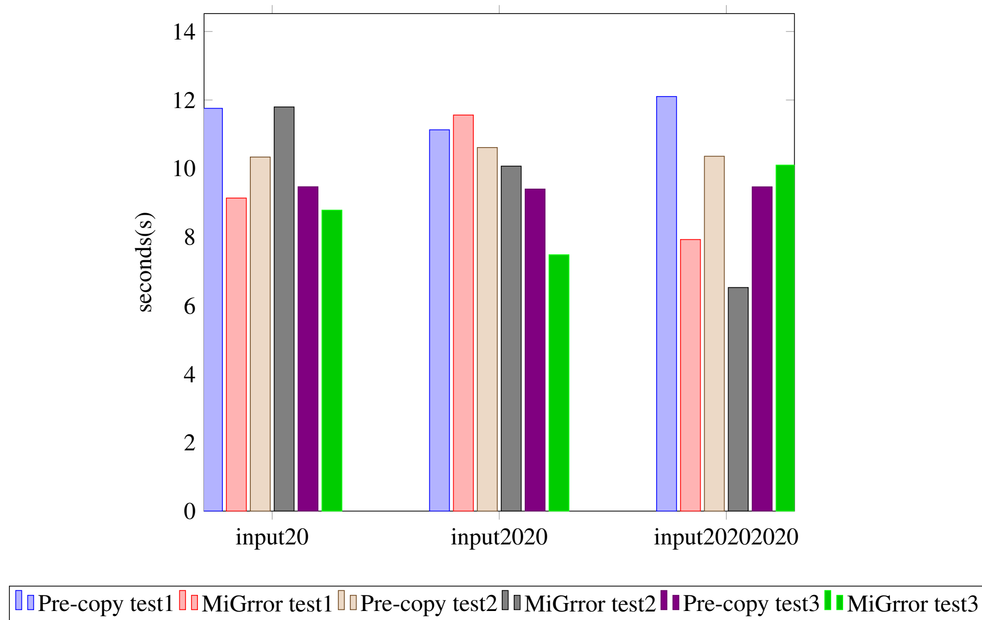

Figure 6.6: Total migration time comparison for variable memory dirty rate

After running this experiment 5 times, we are able to extract an interesting conclusion. For

most of the inputs and dirty rates, the MiGrror results have lower total migration time than Pre-copy results. This is as expected as we do not have as long of a time delay for checkpointing and transferring on the MiGrror compared to Pre-copy. The possible reason for some of the MiGrror values being longer total time compared with the Pre-copy is from the creation of the last checkpoint, not the pre-checkpoints to transfer before the full handoff is triggered. This checkpoint is created by CRIU which we are not able to control as easily as a user. This means there is a possible variance of the time it takes for the checkpoint to be created.

Chapter 7

Conclusion and Future Work

In this chapter we summarize the work presented, and critically assess whether our contributions match the objectives we initially planned. In section 7.1, we provide an objective overview of the results presented in order to establish if the techniques we use are possibly usable for more scalable applications. Lastly, in section 7.2 we cover things we would have liked to include and accomplish in this thesis but have not been able to, and the research lines we believe this initial approach heads us to.

7.1 Conclusions and Lessons Learnt

Our initial goal was to assess the two techniques, MiGrror and Pre-copy, on a more realistic testbed compared to just simulations. It was surprising to find that, the mainstream tool for container related experiments, Docker, had abandoned an apparently useful tool (checkpoint and restore) from its experimental branch. CRIU is an incredibly complex tool, with a very helpful community, but whose intricate relation with the kernel makes it hard to debug whenever things don't go as expected. Luckily, the integration with other container engines (other than Docker), is way more maintained, resourceful, documented, and tested. This meant that other container engines was the way to implement the techniques at the time.

Implementing MiGrror and Pre-copy turned out to be a very complicated task. Before even implementing the techniques, setting up the cloud virtual machines took a lengthy amount of time. This was due to different OS mismatching with CRIU or podman and requiring proxying multiple times. During the implementation process for the MiGrror and Pre-copy techniques, synchronizing when to do a migration and detecting if checkpoints were migrated took many trial and errors before everything worked out. This included different numbers suitable for how often to transfer the migration, such as every two seconds or two memory changes and also using proper data structure for handling the checkpoints on the destination virtual machine.

Our experimental results presented in Chapter 6, validate our implementations. Firstly, the application downtime for MiGrror migration technique for all types of variable inputs is drastically reduced compared to Pre-copy migration technique. Secondly, the total data transferred is in most cases a bit more for the MiGrror compared to the Pre-copy but not by a lot (in general only hovering around 350KB transferred only) even though more transfers occurred. Lastly, the total migration time for MiGrror is lower than Pre-copy but sometimes with more delays.

This is expected as some of the total migration time included the time for CRIU to create the final checkpoint before hand-off that may take a bit longer than usual creating extra time.

As a result, we believe that the MiGrror migration technique is a good starting point to use as a varied way to do migration instead of Pre-copy, at least for smaller containers and data flow. We are also confident that this technique can be utilized in more mobility focused applications if more research is done to find ideal amounts of changes before doing a migration in the coming years.

7.2 Future Work

Unfortunately, and as it tends to be the case, there has been much work we would have liked to include in the present work but we have not been able to. Either due to a lack of time or a lack of expertise and experience, there are some areas of this research that we would like to polish, and some ones which we would like to push forward in the future.

From a technical standpoint, there are some implementation and evaluation details we would like to complete. Firstly, the application we utilize for the experiments in this thesis is only a relatively simple application of different constant memory values. Given more time, support for trying the migration techniques on more sophisticated programs would be ideal. These may include web(text), music and photo heavy programs. Secondly, we did not have any evaluations relating to the data loss. This is an important factor to determine what migration techniques are better in order to avoid losing data. With a limited amount of time, it was hard to find a great way to measure the data that was lost.

On a broader scope, the over-arching goal of this project was to support different migration techniques for distributed container deployments and possibly WAN or mobility migration. We believe the work here presented is a necessary first step towards achieving it, but there's still much work to be done. From an algorithmic standpoint, distributed checkpointing and coordination algorithms need to be implemented. From an infrastructure standpoint, distributed container deployments are managed through an orchestrator. We would also need to pair up with some hardware from an engineering perspective to simulate a tour car with migration on a fog based network. The integration of CRIU and podman with such a tool is, to the best of our knowledge, unexplored territory and something we look forward to doing in the future.

Bibliography

[1] Angelos-se. Berkeley lab checkpoint/restart (blcr) for linux. `https://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/`, 2013.

[2] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[3] B. Barker. Autosave for research: Where to start with checkpoint/restart., 2014.

[4] L. Bautista-Gomez. Fault tolerance interface. `https://github.com/leobago/fti`, 2023.

[5] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[6] Aditya Bhardwaj and C. Rama Krishna. A container-based technique to improve virtual machine migration in cloud computing. *IETE Journal of Research*, 68(1):401–416, 2022.

[7] A. Choudhary, M. C. Govil, G. Singh, L. K. Awasthi, E. S. Pilli, and D. Kapil. A critical survey of live virtual machine migration techniques. *Journal of Cloud Computing*, 6(23), 2017.

[8] Christopher Clark, Keir Fraser, Steven Hand, Hansen Gorm, Jacob, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, volume 2, pages 273–286. USENIX Association, 2005.

[9] Jonathon Corbet. clone3(), fchmodat4(), and fsinfo(). `https://lwn.net/Articles/792628/`, 2019.

[10] dwks. pagemap. `https://github.com/dwks/pagemap`, 2019.

[11] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, sep 2002.

[12] P. Emelyanov. Freezing the tree. `https://criu.org/Freezing_the_tree`, 2023.

[13] Dinuni Fernando, Jonathan Terner, Kartik Gopalan, and Ping Yang. Live migration ate my vm: Recovering a virtual machine after failure of post-copy live migration. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 343–351, 2019.

[14] CRIU Foundation. checkpoint-restore/go-criu. `https://github.com/checkpoint-restore/go-criu`, 2023.

[15] CRIU Foundation. Comparison to other cr projects. `https://criu.org/Comparison_to_other_CR_projects`, 2023.

[16] CRIU Foundation. Criu - checkpoint restore. `https://github.com/checkpoint-restore/criu`, 2023.

[17] CRIU Foundation. Criu - checkpoint restore. `https://criu.org/Checkpoint/Restore`, 2023.

[18] CRIU Foundation. Criu - lazy migration. `https://criu.org/Lazy_migration`, 2023.

[19] CRIU Foundation. Criu - live migration. `https://criu.org/Live_migration`, 2023.

[20] CRIU Foundation. Criu - main page. `https://criu.org/Main_Page`, 2023.

[21] CRIU Foundation. Memory changes tracking. `https://criu.org/Memory_changes_tracking`, 2023.

[22] CRIU Foundation. Parasite code. `https://criu.org/Parasite_code`, 2023.

[23] Hein Htet, Nobuo Funabiki, Ariel Kamoyedji, Xudong Zhou, and Minoru Kuribayashi. An implementation of job migration function using criu and podman in docker-based user-pc computing system. In *Proceedings of the 9th International Conference on Computer and Communications Management*, ICCCM '21, page 92–97, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Cheng-Hao Huang and Che-Rung Lee. Enhancing the availability of docker swarm using checkpoint-and-restore. In *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*, pages 357–362, 2017.

[25] M. Kerrisk. ptrace. `https://man7.org/linux/man-pages/man2/ptrace.2.html`, 2021.

[26] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, USA, 1 edition, 2008.

[27] P. Menage. The /proc filesystem. `https://docs.kernel.org/filesystems/proc.html`, 2009.

[28] P. Menage. Soft-dirty ptes. `https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt`, 2013.

[29] P. Menage. cgroup freezer. https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt, 2019.

[30] podman. podman-container-checkpoint. https://docs.podman.io/en/latest/markdown/podman-container-checkpoint.1.html, 2019.

[31] podman. podman-container-restore. https://docs.podman.io/en/latest/markdown/podman-container-restore.1.html, 2019.

[32] Carlo Puliafito, Carlo Vallati, Enzo Mingozzi, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Container migration in the fog: A performance evaluation. *Sensors*, 19(7), 2019.

[33] M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Publishing Company, Incorporated, 2013.

[34] A. Reber. Criu and the pid dance. https://linuxplumbersconf.org/event/4/contributions/472/attachments/224/397/2019-criu-and-the-pid-dance.pdf, 2019.

[35] Arshin Rezazadeh, Davood Abednezhad, and Hanan Lutfiyya. Migrror: Mitigating downtime in mobile edge computing, an extension to live migration. *Procedia Computer science*, 203(41-50), 2022.

[36] M. Rieker. Distributed multithreaded checkpointing. http://dmtcp.sourceforge.net/, 2019.

[37] M. Schulz. *Checkpointing*, pages 264–273. Springer, Boston, MA, 2011.

[38] Mohamad Sindi and John R. Williams. Using container migration for hpc workloads resilience. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10, 2019.

[39] R. Stoyanov and M. J. K. *Efficient live migration of linux containers*, pages 184–193. Springer, Boston, MA, 2018.

[40] Linus Torvalds. linux. https://github.com/torvalds/linux.

[41] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 53–65, New York, NY, USA, 2019. Association for Computing Machinery.

Curriculum Vitae

Name:	Xinwen Liang
Post-Secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada 2017 - 2022 B.Sc.
	The University of Western Ontario London, Ontario, Canada 2022 - 2024 M.Sc.
Honours and Awards:	Western Graduate Research Scholarship 2022-2023
Related Work Experience	Teaching Assistant(CS1026 Computer Science Fundamentals I), Teaching Assistant(CS2208b Introduction to Computer Organization and Architecture) The University of Western Ontario 2022-2023

Publications: