SHARING-AWARE LIVE MIGRATION OF VIRTUAL MACHINES

BY

ROJA ESWARAN

BE, Mepco Schlenk Engineering College, Anna University, 2018
MS, Binghamton University, 2020

DISSERTATION

Submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2024

© Copyright by Roja Eswaran 2024

All Rights Reserved

Accepted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2024

June 6, 2024

Prof. Kartik Gopalan, Chair and Advisor
Department of Computer Science, Binghamton University

Prof. Kanad Ghose, Committee Member
Department of Computer Science, Binghamton University

Prof. Yifan Zhang, Committee Member
Department of Computer Science, Binghamton University

Prof. Yu Chen, Outside Examiner
Department of Electrical and Computer Engineering,
Binghamton University

Abstract

One of the key challenges of cloud/edge computing is working with a limited amount of resources available, especially memory and bandwidth. Virtual machines (VMs) can ensure both isolation and efficient resource utilization. Live migration is a crucial technique to transfer running VMs from one physical node to another. This can occur either between different hosts (inter-host) for load balancing, infrastructure maintenance, and meeting service-level agreements or within the same host (intra-host) for updates to VM management processes, bug fixes, and VM introspection.

Unfortunately, current live migration techniques are unaware of pre-existing memory sharing between VMs that are being migrated to the same destination. As a result, shared pages are transferred and replicated multiple times at the destination, as if they were separate pages, resulting in an expanded memory footprint at the destination. The duplication of previously shared pages also results in longer migration times and increased network traffic, potentially affecting the performance of other network-bound workloads in the cluster. Further, current intra-host live migration unnecessarily copies the pages within the same host, leading to memory spikes, instead of simply transferring the ownership of pages.

In this dissertation, we describe efficient ways to incorporate sharing-awareness in live migration of multiple VMs while avoiding memory and network resource contention. For inter-host migration, our techniques rely on existing *copy-on-write* (COW)

sharing between VMs maintained by the host/hypervisor. This enables the transfer of a copy of the page only once and preserves existing COW sharing by remapping pages at the destination. For intra-host migration, our technique implements a mechanism to identify shared pages and transfer their ownership instead of copying them. We prototype and evaluate our techniques in the KVM/QEMU virtualization platform. We show that, besides preventing memory footprint expansion during migration, our techniques also result in a shorter total migration time and less network traffic.

Dedicated to Dad for also taking on the role of Mom and for all his other sacrifices.

Acknowledgements

I would like to sincerely thank my advisor, Prof. Kartik Gopalan, for his invaluable support throughout this journey. He is a brilliant researcher and an excellent mentor who cares about the physical and mental well-being of his students. I am truly grateful for this opportunity with him. I would like to thank my committee members Prof. Kanad Ghose, Prof. Yifan Zhang, Prof. Yu Chen, for their valuable time and suggestions to improve my dissertation.

I would like to express my gratitude to Dr. Pavan Balaji for providing me with the opportunity for an internship at Argonne National Laboratory. This experience helped me comprehend the practical implications of user-level vs. kernel-level threads which was very helpful during the implementation of the prototypes. I would also like to express my gratitude to ZEDEDA, the company that manages and orchestrates distributed edge solutions, for offering me an internship opportunity to explore the real-world implications of edge computing. This experience provided me with valuable insights for my publications. I would like to thank Kevin Cheng and Yongheng Li for their contributions to my work Template-aware Live Migration of Virtual Machines. I would also like to thank my fellow OSNET Lab member, Mingjie Yan for all his collaborations with my work.

I wish to express my gratitude to my fellow OSNET Lab members Prathamesh Patil, Atharva Ranade, and especially my friend Dr. Spoorti Doddamani, for helping me get started with the basics of VMs and Live Migration. My Ph.D. wouldn't be possible without these people who offered me love and support: Kirupaa Thanani, Dr. Achutha Lakshmi, Rohini, Sujith Anna, Saravanan Anna, Mrs. Denise Peterson, Yassine Mansour, Dr. Debaruti Roy, Dr. Meenakshy Jyothis, Gautham Pandiyan. I would also like to

thank my adorable black short-haired domestic cat, Serene Eswaran, for her invaluable emotional support. Additionally, I would like to extend my thanks to the Binghamton University Counseling Center for helping me understand myself better, with special appreciation for Ms. Nancy Lamberty and Ms. Portia Johnson.

I would like to express my gratitude to Prof. Suenghee Shin for accepting me into his laboratory during my master's, sparking my interest in systems research. I would also like to thank Prof. Leslie Lander for his assistance with administrative procedures.

I would also like to thank my fiancé, Karthik Suresh Arulalan, for always being there for me, anywhere, anytime. Finally, I would like to thank my grandmother, Mrs. Rajamani, and Thanthai Periyar for being my inspirations in the educational journey. I would love to dedicate this dissertation to my dad, Mr. Eswaran, who not only provided me with the undergraduate college education he was denied but also so much more.

Contents

List of Table **xiv**

List of Figures **xix**

List of Abbreviations **xix**

1 Introduction **1**

1.1 Problem Statement: Lack of Sharing-awareness in Live Migration: . . . 3

1.2 Contributions 5

1.3 Outline 7

2 Background **8**

2.1 KVM/QEMU and Live Migration 8

2.2 Performance Metrics 9

2.3 Migration Streams 10

2.4 Live Migration Techniques 11

2.4.1 Pre-copy Live Migration 11

2.4.2 Post-copy Live Migration 12

2.4.3 Hybrid Live Migration 13

2.5 Existing COW optimizations 14

2.5.1 VM Templating 14

2.5.2 Kernel Samepage Merging (KSM) 16
2.5.3 *mmap* and COW Protection 18
2.6 Transferring Ownership of Pages 18
2.7 Userfaultfd 19

3 Inter-host Template-aware Live Migration **22**
3.1 Problem Statement 23
3.2 Contributions 24
3.3 Design 25
3.3.1 Seamless Templating 28
3.3.2 Chained Templating 28
3.3.3 Snapshot Overhead 29
3.3.4 Memory-layout-aware and Parallel Snapshot 29
3.4 Implementation 30
3.5 Evaluation 32
3.6 Related Work 37
3.7 Chapter Summary 38

4 Intra-host Template-aware Live Migration **39**
4.1 Problem Statement 40
4.2 Contributions 41
4.3 Problem Demonstration: Memory Spikes During Intra-host Live Migration 42
4.4 Design of Intra-host TLM 43
4.5 Implementation 47

4.5.1 Saving delta of Source VMs 47
4.5.2 Transferring ownership of delta from Source VMs 49
4.6 Evaluation 50
4.6.1 Reduced Memory Footprint 51
4.6.2 Improvement in Total Migration Time 53
4.6.3 Reduction in Pages Transferred 55
4.6.4 Effect on Downtime 57
4.7 Related Work 58
4.8 Chapter Summary 58

5 Sharing-aware Live Migration **60**
5.1 Problem Statement 60
5.2 Contributions 61
5.3 Problem Demonstration 62
5.4 SLM Design 64
5.4.1 Identifying Page Type at Source 66
5.4.2 Preserving COW Sharing at Destination 68
5.5 Implementation 70
5.5.1 Retrieval and Tracking of PFN 70
5.5.2 In-Memory Backend File 71
5.5.3 Synchronization Across Multiple VMs 71
5.6 Evaluation 72
5.6.1 Live Migration of Single VM 73
5.6.2 Memory Footprint of VMs After Migration 77
5.6.3 TMT, Downtime, and Network Traffic Reduction 77

5.6.4 Network Bandwidth Using iPerf 79
5.6.5 Redis Cluster Benchmark 82
5.6.6 LAMP/ApacheBench Response Time 84
5.6.7 Performance of SLM on templated VMs 86
5.7 Related Work 89
5.8 Chapter Summary 91

6 Conclusions and Future Directions 92

6.1 Inter-host Template-aware Live Migration of Virtual Machines 92
6.2 Intra-host Template-aware Live Migration of Virtual Machines 93
6.3 Sharing-aware Live Migration of Virtual Machines 93

List of Tables

5.1 Determining page type using PFN and VPN 67

List of Figures

1.1 Inter-host live migration transfers a VM across two different hosts. Intra-host live migration transfers a host within a node, but from one VMM to another. 2

2.1 VM Templating: Multiple VMs can be started from a common shared template to reduce their startup times and initial memory footprint. Memory pages that are dirtied by a VM (represented by deltas *dk*) are not shared. 15

2.2 (a) Without KSM, each virtual page has its own physical page in RAM. (b) With KSM, duplicated pages are COW-mapped to single physical page in RAM. 17

2.3 (a) Without the bypass-memory flag, the memory of the VM is copied again. (b) The bypass-memory flag skips copying the memory resulting in the transfer of only the I/O, VCPU, and Device states. 20

3.1 Template-aware migration works by migrating only the *delta* pages during migration. The shared VM template is available to the destination either over a networked storage or transferred ahead of time before migration begins. 27

3.2 Time taken to take the memory snapshot of VMs varying in size. 28

3.3 Memory usage of the memory template of varying VM size. 29
3.4 Snapshot copy time linearly increases with the working set size using memcpy. 30
3.5 Lseek-copy performs better than both `cp` in terms of downtime. 31
3.6 Lseek-copy performs better than both `cp` and `memcpy` in terms of memory usage 31
3.7 Memory footprint increase at destination when multiple templated VMs are migrated using generic pre-copy 33
3.8 Memory footprint expansion problem in regular non-templated VMs at destination after live migration using generic pre-copy. 33
3.9 Memory footprint of templated VMs at the source before migration and destination after migration using Generic and TLM pre-copy. 34
3.10 Total migration time of multiple VMs started from the same template and migrated concurrently using Generic and TLM pre-copy. 35
3.11 Downtime of multiple templated VMs migrated using Generic and TLM Pre-copy 36
3.12 Total Pages Transferred of multiple VMs started from the same template and migrated concurrently using Generic and TLM pre-copy. 37

4.1 Memory footprint of templated VMs migrated within the same host using Generic TLM. 43
4.2 High-level overview of Intra-host TLM using userfaultfd. 44
4.3 Illustration of how Intra-host TLM transfers the ownership of delta pages using userfaultfd mechanism. 45

4.4 Memory footprint of templated VMs at the source before migration and destination after migration using Intra-host TLM 49
4.5 Total migration time of multiple VMs started from the same template and migrated concurrently using Intra-host TLM 50
4.6 Memory footprint of templated VMs running write-intensive benchmarks migration using (a) Generic TLM (b) Intra-host TLM. 52
4.7 Total Migration Time (TMT) of idle multiple templated VMs migrated concurrently using Generic TLM and Intra-host TLM. 53
4.8 Total Migration Time (TMT) of multiple templated VMs running write-intensive workload of varying size in MB migrated concurrently using Generic TLM and Intra-host TLM. 54
4.9 DownTime (DT) of multiple idle templated VMs migrated concurrently using Generic TLM and Intra-host TLM. 54
4.10 DownTime (DT) of multiple templated VMs running write-intensive workload migrated concurrently using Generic TLM and Intra-host TLM. 55
4.11 Total 4KB pages transferred of multiple templated VMs running write-intensive workload of varying size. 56
4.12 Idle Templated VMs 56
4.13 Busy Templated VMs 57

5.1 Memory footprint of VMs expands at destination after both pre-copy and post-copy live migration, because pages shared among VMs at the source are replicated for each VM at the destination. 63
5.2 SLM Algorithm 64

5.3 SLM classifies pages of VMs at the source as Unique, Shared, and Dirty. Shared pages are not re-transmitted; instead Destination COW-maps them into a common in-memory backend file. 67
5.4 Memory footprint of virtual machine of varying size migrated using Generic and SLM pre-copy and post-copy. 73
5.5 Memory footprint of virtual machine of varying size migrated using Generic and SLM post-copy. 74
5.6 Total migration time of single VM of varying size migrated using generic, SLM pre-copy and post-copy. 74
5.7 Total pages transferred of single VM migrated using generic, SLM pre-copy and post-copy 75
5.8 Comparison of memory footprint at source vs. destination when multiple VMs are migrated concurrently using generic vs. SLM (a) pre-copy and (b) post-copy. We observe a significant increase in memory footprint for generic pre-copy and generic post-copy vs. no significant increase for SLM pre-copy and SLM post-copy. 76
5.9 Total migration time of multiple VMs concurrently migrated using generic and SLM pre-copy and post-copy. 78
5.10 Pages transferred during concurrent migration of multiple VMs using generic and SLM pre-copy and post-copy. 78
5.11 iPerf bandwidth for generic and SLM pre-copy. 80
5.12 iPerf bandwidth for generic and SLM post-copy. 80
5.13 Redis-cluster read throughput when migrating 3 VMs using generic and SLM versions of pre-copy. 82

5.14 Redis-cluster read throughput when migrating 3 VMs using generic and SLM versions of post-copy. 83

5.15 ApacheBench response times for generic and SLM pre-copy 85

5.16 ApacheBench response times for generic and SLM post-copy 85

5.17 Memory footprint of templated VMs at the source before migration and destination after migration using generic, TLM and SLM pre-copy. . . 87

5.18 Total migration time of multiple VMs started from the same template and migrated concurrently using generic, TLM and SLM pre-copy. . . 88

5.19 Total pages transferred of multiple VMs started from the same template and migrated concurrently using generic, TLM and SLM pre-copy. . . 89

1 Introduction

In this chapter, we begin by presenting various scenarios of live migration techniques and their respective use cases. We then discuss the limitations of current live migration techniques, particularly their lack of awareness of existing Copy-On-Write (COW) shared pages, and outline our contributions to address these limitations, followed by an overview of the remaining sections of the dissertation.

Live migration, a critical technology within cloud computing infrastructure, facilitates the seamless transfer of active virtual machines (VMs) from one physical node to another. While the naive way for relocating a VM from one node to another involves employing a *stop-and-copy* mechanism, this approach extends the application's downtime, as the VM remains unavailable until the pages are copied to a new VM at the destination and finally resuming the VM.

Live migration of VMs can occur within a data center's local area network (LAN), or between different data centers connected by a wide area network (WAN). In the former case of migration within a data center, memory, virtual CPU (VCPU), and I/O states of the VMs are transferred but, since disk images can be stored in a shared network file system, there is no need to transfer the disk contents. In the latter case of migration across data centers, in addition to transferring the memory, CPU, and I/O states, the disk image must also be normally transferred.

In this dissertation, we focus on live migration within a data center without the need

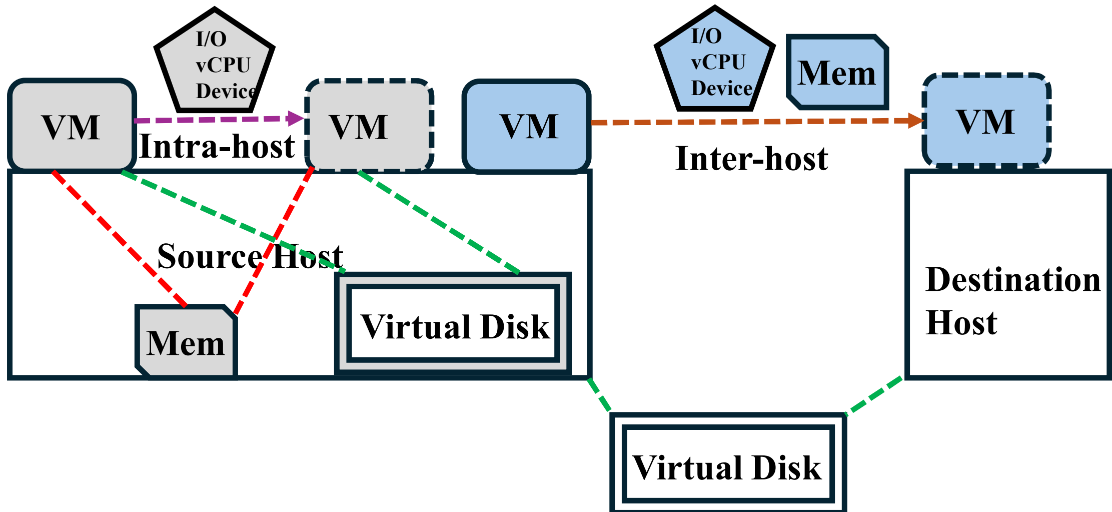


Figure 1.1: Inter-host live migration transfers a VM across two different hosts. Intra-host live migration transfers a host within a node, but from one VMM to another.

to migrate storage contents. In the discussion below, we note that each VM is typically managed by an external process, which we call a *VM Monitor* (VMM), that coordinates the lifecycle of the VM from it's creation and execution to termination. Within a data center, we consider two scenarios for live migration as shown in Figure 1.1:

1. **Inter-host live migration:** This involves relocating a VM from one host (called the source host) to another (called the destination host) within the same data center. Inter-host live migration is widely used for a variety of purposes, such as load balancing [2, 70, 20], infrastructure maintenance, meeting service level agreements [56], security-related updates, energy savings [77], hardware failure and seamless maintenance of physical edge nodes. Inter-host migration typically involves establishing a TCP connection between VMMs at the source and destination hosts and migrating the VCPU, memory, and I/O states of the VM. However, virtual disk image remains available to the VM over the network and may not necessarily be transferred.

   In the context of this dissertation, multiple co-located VMs may often need to be

migrated together to the same destination host for various reasons. For example, multiple tightly-coupled VMs that run different components of a multi-tier application [31] may require to be migrated together to the same destination machine to maintain low inter-VM communication latency [80, 42, 68, 37, 47] or to meet other performance targets [84]. Additionally, physical server availability, hardware availability, and multi-tenancy limitations may necessitate the migration of co-located VMs to the same destination machine.

2. **Intra-host live migration:** This type of migration involves moving a VM within the same host. To understand why one might want to migrate a VM within the same host, consider situations where the VMM needs to be replaced, such as due to bugs, software upgrades or, less commonly, for VM introspection [29, 14, 82]. In such cases, the VM can be live migrated from the old VMM to a new VMM within the same host, or what we call intra-host live migration. While intra-host migration can also be accomplished using the standard techniques used in inter-host migration, the fact that VM is being migration within the same host lends itself to certain useful optimizations. For example, as we propose later in this dissertation, the VM memory pages can be transferred from one VMM to another by transferring the ownership of the pages, instead of copying the pages.

1.1 Problem Statement: Lack of Sharing-awareness in Live Migration:

COW page sharing (both within a VM and across co-located VMs) is often used by the hypervisor's memory management system to reduce the collective memory footprint by sharing identical pages whenever doing so is feasible and safe.

An example where inter-VM page sharing is employed is in a technique called VM Templating. Instantiating a VM from scratch typically takes a long time because it involves initialization of software, guest OS, and virtual hardware, including time to load the corresponding contents to memory from the disk. VM templating allows multiple new VM instances to be quickly instantiated from a single pre-checkpointed VM image (or template). Figure 2.1 shows the steps involved in instantiating VMs from a template image. First step is to save a custom VM template as a snapshot of a pre-booted VM that is pre-initialized with necessary software. The template image can be pre-loaded into memory to reduce disk access latency. Next step is to quickly instantiate multiple VM instances from the common template image by mapping the base template image COW into the memory of each new VM instance. This is essentially a variant of a typical checkpoint-restore operation. As VMs execute and write to (dirty) different pages of their memory, those pages are copied for the respective VM, and their memory footprints diverge over time.

Another example of COW page sharing among co-located VMs is a technique called Kernel Samepage Merging (KSM) [1], which is a Linux kernel feature that identifies identical memory pages among VMs (that run the same guest OS or similar applications) and maps them to the same physical page through COW page sharing. As KSM gradually identifies and merges identical pages across different VMs, the collective memory footprint of co-located VMs progressively decreases.

Unfortunately, current live VM migration techniques fail to consider pre-existing COW page sharing within and across multiple templated and regular non-templated VMs that are being migrated to the same destination. As a result, shared pages are transferred and replicated multiple times at the destination, as if they were separate

pages, resulting in an expanded memory footprint at the destination. This expansion can lead to migration failures when the destination lacks sufficient memory to accommodate the additional footprint of the VMs that were comfortably co-located at the source. The duplication of COW-shared pages also results in longer migration times and increased network traffic, potentially affecting the performance of other network-bound workloads including distributed multi-tier applications.

In the context of intra-host live migration, while live migration techniques have been extensively studied for migration between different hosts, there has been limited exploration of these techniques in intra-host environments. Current live migration techniques for both regular non-templated and templated VMs are inefficient when migrating VMs within the same host as they involve unnecessary copying of pages causing spikes in memory usage during migration. Besides memory contention, they also increase the total migration time, thereby prolonging the spikes in resource usage, which can adversely impact application performance in nodes with limited memory.

1.2 Contributions

Inter-host Template-aware Live Migration: Traditional pre-copy live migration is unaware of the underlying COW page sharing among templated VMs. Hence it ends up transferring the shared base pages of the template multiple times. We propose Inter-host *Templating-aware Live Migration* (TLM) which addresses this shortcoming of pre-copy migration by ensuring that multiple templated VM instances maintain their COW page sharing with the base template even at the destination node and transfers only the *delta* pages that differ among various VM instances. TLM first tracks delta (or dirty) pages for VM instances before migration. Then, during live migration, only

the delta pages are transferred for each instance. We implement a prototype of Inter-host TLM in the KVM/QEMU [38] virtualization platform and evaluate it using several benchmarks. Besides maintaining pre-existing page sharings among templated VMs at the destination machine, TLM reduces the total migration time and network traffic compared to normal migration, which transfers each VM instance individually.

Intra-host Template-aware Live Migration: We also propose Intra-host Template-aware Live Migration of Virtual Machines or Intra-host TLM a new way to handle migration of templated VMs within the same host. The fundamental idea behind Intra-host TLM is to transfer ownership of delta pages, rather than duplicating them during migration. Intra-host TLM utilizes a mechanism called *userfaultfd* [35, 45] to monitor delta pages and save them in a backend-file. When live migration is started, the source only sends the corresponding backend-file offset for the delta pages so that the destination VM can claim ownership of those pages once the migration is complete. This technique completely removes spike in memory usage during migration by eliminating copying of memory pages.

Sharing-aware Live Migration: Finally, we present a more general *Sharing-aware Live Migration* (SLM) of both templated and regular non-templated VMs for both pre-copy and post-copy live migration techniques. SLM identifies and preserves all types of pre-existing page sharings during live migration and works with any existing memory sharing mechanism such as KSM, VM templating, or others. We implement and evaluate SLM in the KVM/QEMU [38] virtualization platform using several workloads and microbenchmarks. Besides preserving pre-existing page sharings at the destination machine, SLM also reduces the total migration time.

1.3 Outline

The remainder of the dissertation is structured as follows: In Chapter 2, we delve into the background of live VM migration, different COW sharing optimizations, and the *userfaultfd* mechanism. Chapter 3 introduces Inter-host Template-aware Live Migration (TLM), a technique tailored for the live migration of templated VMs that involves transferring only the delta. Chapter 4 outlines Intra-host Template-aware Live Migration (Intra-host TLM), a live migration technique that is aware of co-located memory and transfers the ownership of pages without duplicating them due to copying. In Chapter 5, we introduce Sharing-aware Live Migration (SLM), a more comprehensive approach that considers all COW optimizations, including KSM and VM templating for both templated and regular non-templated VMs. Finally, Chapter 6 summarizes the key findings and outlines potential directions for future research.

2 Background

This chapter starts with key performance metrics during live migration of multiple VMs, followed by an explanation of various live migration techniques such as pre-copy, post-copy, and hybrid using different migration streams. It then discusses existing COW optimizations done by the host/hypervisor, including VM Templating and Kernel Samepage Merging (KSM). Finally, the chapter concludes with background on the *userfaultfd* mechanism for transferring page ownership.

2.1 KVM/QEMU and Live Migration

Here we provide a brief background about the KVM/QEMU [38] virtualization platform which is used in this work for prototyping and evaluation of sharing-aware live migration techniques. In KVM/QEMU, each VM is managed by a userspace VMM process, called QEMU, which performs device emulation and various management functions, including creation, execution, live migration, checkpointing, and termination. A kernel module, called KVM, implements the core hypervisor functionality. KVM uses hardware virtualization features and coordinates with QEMU to execute the VM in guest mode (or non-root mode).

To perform live migration, the QEMU process (which manages the VM being migrated) at the source machine establishes a TCP connection with another QEMU process at the destination machine. Then the memory, VCPU, I/O, and optionally storage,

states of the VM are transferred from the source QEMU to the destination QEMU over this TCP connection. Depending on the mode of live migration, the VM either continues to execute at the source machine (pre-copy migration), or at the destination machine (post-copy) during live migration.

2.2 Performance Metrics

Our key performance metrics for evaluation are as follows:

1. **Memory Usage:** The collective memory footprint of the VMs at the source (before migration) and destination (after migration). This is measured using the `free` command and includes the memory used by both the QEMU processes and the VMs. Recording the `use` column value from the `free` command before and after spawning a process gives the memory usage of that process.

2. **Total Migration Time (TMT):** The total migration time (TMT) refers to the time from the start to the end of the entire migration process. For single VM migration, in pre-copy, TMT is measured from the start of migration on the source machine to when the VM resumes on the destination machine. In post-copy, it is measured from the initiation of migration on the source to the release of the VM's resources at the source after all pages have been transferred. For multiple VM migration, in pre-copy, TMT is calculated from the beginning of the first VM's migration on the source to the resumption of the last VM on the destination. In post-copy, it is calculated from the start of the first VM's migration to the release of the last VM's resources at the source.

3. **Downtime:** Downtime refers to the period during which a VM's execution is

fully suspended during live migration. In pre-copy, downtime is used to transfer the VM's remaining Dirty pages, I/O device states, and VCPU states to the destination. In post-copy, the processor state and the essential execution state necessary to start the VM on the destination are transferred during downtime.

4. **Network Traffic Reduction:** The reduction in the total number of pages transferred during live migration by eliminating the transfer of COW-shared pages.

5. **Application performance degradation:** The extent to which live migration slows down the performance of an application running inside VMs during migration.

In order to accurately measure TMT and downtime in QEMU, we employ a more precise method instead of solely relying on the source QEMU's measurement (which we found to be inaccurate). Specifically, we send UDP packets to a third, separate measurement node, at the start of migration and at the end of migration. For downtime measurement and pre-copy TMT measurement, the start message is sent from source node and the end message is sent from the destination node, whereas for post-copy TMT measurement, both messages are sent from the source node. The measurement node observes the arrival times of these packets using the *tcpdump* tool, and the difference in these arrival times represents TMT or downtime. This method provides more accurate timings, as the dedicated measurement node has a better view of the end-to-end live migration timeline than the source node alone.

2.3 Migration Streams

QEMU uses a separate migration thread for transferring the pages. The migration thread typically utilizes a byte stream format, allowing for transmission over various

transport mechanisms [64].

1. **TCP Migration:** Relying on TCP sockets for data transfer, this method is particularly well-suited for migrations between different hosts.

2. **Unix Migration:** It uses Unix sockets for migration.

3. **exec Migration:** This migration method utilizes standard input (stdin) and standard output (stdout) for data transfer, facilitated by a dedicated process. It's particularly useful for migrating VMs within the same physical host.

4. **fd Migration:** QEMU leverages a file descriptor for migration irrespective of the method used for opening the descriptor.

5. **File Migration:** QEMU supports file-based migration by accepting a designated file path. Furthermore, a file offset feature enables management applications to include their own metadata at the beginning of the file, ensuring it remains separate from QEMU's migration data.

2.4 Live Migration Techniques

2.4.1 Pre-copy Live Migration

The pre-copy live migration [8, 57] is the most common technique to migrate VMs from a source machine to a destination machine. It works by first transferring the VM's memory pages to the destination, even as the VM continues running at the source, and then transfers the CPU execution state at the end; hence the name *pre*-copy which means to transfer memory before CPU state. However, as the VM's memory is being transferred, its virtual CPUs (VCPUs) may write to previously transferred pages, thus *dirtying* them again and requiring their retransmission.

For the traditional pre-copy technique, the VM's memory is transferred over multiple rounds. The first pre-copy round is the longest since it transfers all pages of the VM. The second round transfers only the pages dirtied in the first round; the third stage transfers only pages dirtied from the second round, and so forth. When the number of remaining dirtied pages is small enough, the migration process switches to the final *downtime* stage in which all the VCPUs of the VM are paused at the source, and the remaining dirtied pages, VCPU states, and I/O device states are sent to the destination. Finally, the VCPUs are resumed at the destination, and the VM starts execution where it left off at the source.

Pre-copy live migration uses TCP migration to transfer memory pages to the destination VM running on a different node. Regardless of whether the page is dirty (modified) or clean (unchanged), pre-copy employs the same mechanism. The source VM writes the page content to the socket, and the pre-allocated memory on the destination uses the guest physical address as a unique identifier while receiving pages. This allows the destination to directly receive the page content into the correct memory location, preventing memory corruption.

2.4.2 Post-copy Live Migration

Post-copy live migration [36, 28, 20] is another technique that first transfers a VM's VCPUs to the destination, resumes them there, and then transfers the VM's memory pages from the source. The VM's pages are transferred by two concurrent mechanisms: (a) *active-push* of the pages from the source to the destination, with preference to pages in the VM's working set, and (b) remote *demand-paging* by the destination from the source when a VM's VCPU faults on a page that has not yet been transferred from the source. Post-copy aims to reduce the number of remote page faults by pushing pages

before the VM accesses them at the destination. Since the VCPU state is transferred and already running on the destination, neither active push nor demand paging will encounter any dirty pages.

Userfaultfd mechanism is employed to resolve the faulted addresses using TCP migration stream [35]: Once the pages are received from the network socket at the destination, the migration thread copies the data from TCP socket into a temporary buffer, unlike pre-copy where the pages are directly received into the guest physical address. The fault-handler thread (userfaultfd) is used to map the temporary page to the guest physical address for both active-push or demand-paging. During demand-paging, a fault-handler thread actively polls for userfaultfd events. A page fault is generated whenever the guest tries to access a page. Now, the fault-handler thread receives the faulted address and generates `-EAGAIN` in case the page fault was already resolved using active push. Else, the fault-handler thread writes the faulted address to the network socket, and the VCPU thread is paused until the fault-handler receives the page, maps it to the faulted guest address, and finally resumes the VCPU thread.

2.4.3 Hybrid Live Migration

The hybrid live migration [71] involves combining both pre-copy and post-copy techniques to harness the benefits of both approaches. Depending on the nature of the application's working set size, either pre-copy or post-copy is chosen to optimize the total migration time, total pages transferred, and application performance. During the first few fixed number of rounds, hybrid migration utilizes the pre-copy feature, where the source sends pages over multiple rounds using dirty page tracking. Subsequently, it switches to post-copy to transfer any remaining pages. This approach reduces retransmission of dirty pages during pre-copy and also reduces network faults during

post-copy.

2.5 Existing COW optimizations

Copy-on-Write (COW) is a memory management optimization technique employed by operating systems. It prioritizes efficiency by initially avoiding duplicate data creation. Instead, multiple processes share a single reference to the data. When a process attempts to modify the shared data, COW springs into action. It creates a separate copy of the data for the modifying process, effectively ”cutting off” the shared reference. This ensures that the original data remains untouched, and any further changes occur on the newly allocated copy. Various COW optimizations performed by the host/hypervisor are described below.

2.5.1 VM Templating

Instantiating a VM from scratch typically takes a long time because it involves initialization of software, guest OS, and virtual hardware, including time to load the corresponding contents to memory from the disk. VM templating allows multiple new VM instances to be quickly instantiated from a single pre-checkpointed VM image (or template) [60]. Figure 2.1 shows the steps involved in instantiating VMs from a template image. First step is to save a custom VM template as a snapshot of a pre-booted VM that is pre-initialized with necessary software. The template image can be pre-loaded into memory to reduce disk access latency. Next step is to quickly instantiate multiple VM instances from the common template image by mapping the base template image COW into the memory of each new VM instance. This is essentially a variant of a typical checkpoint-restore operation, where we checkpoint the base image once and restore it multiple times for concurrent VM instances. While COW allows efficient deploy-

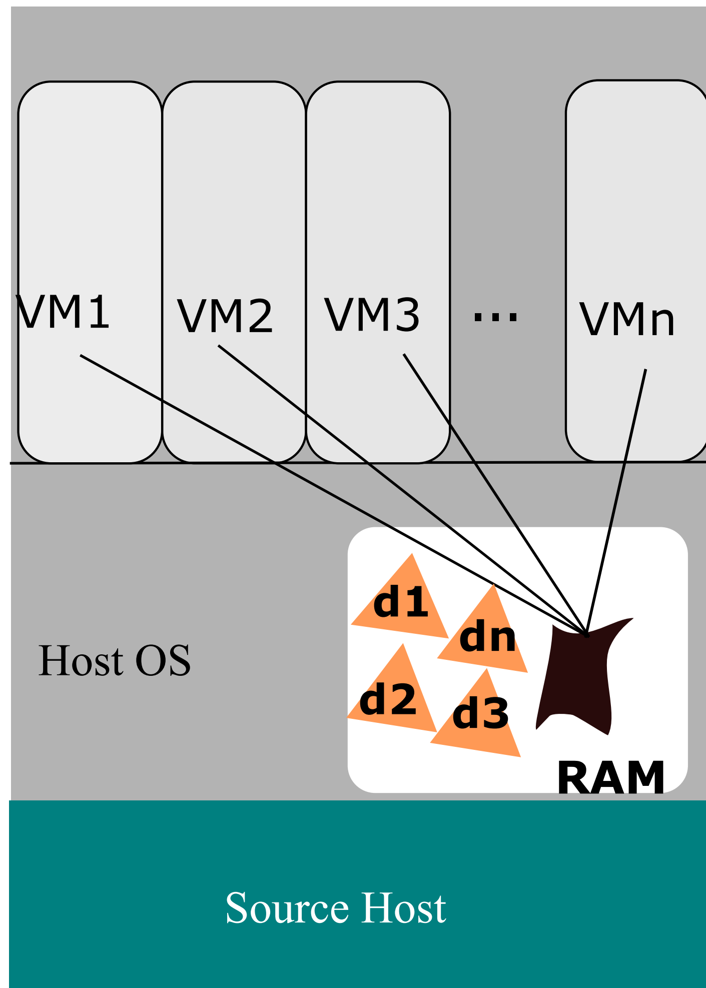


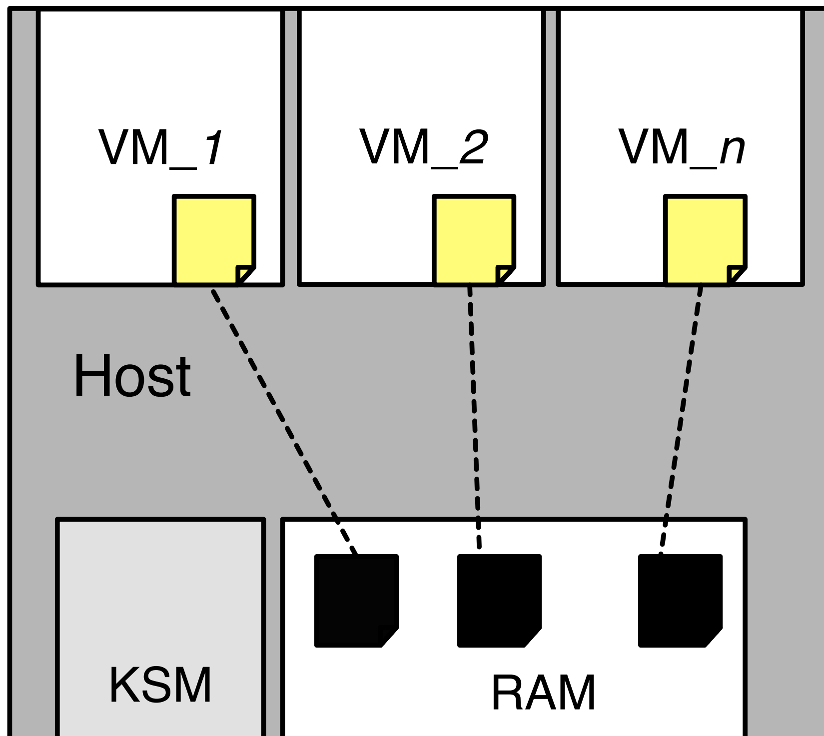
Figure 2.1: VM Templating: Multiple VMs can be started from a common shared template to reduce their startup times and initial memory footprint. Memory pages that are dirtied by a VM (represented by deltas *dk*) are not shared.

ment of VMs from a template, modifications (deltas) by individual VMs cause them to diverge from the template over time. This reduces the memory efficiency advantage of templating.

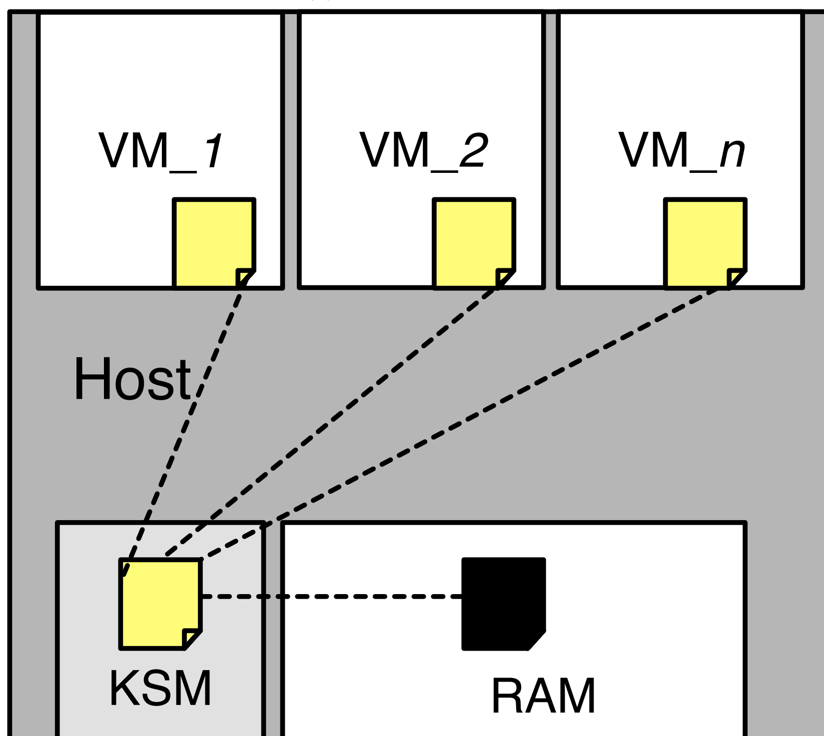
2.5.2 Kernel Samepage Merging (KSM)

Kernel Samepage Merging (KSM) [16] is a technique which performs memory deduplication among co-located VMs to fit more VMs into physical memory. Many duplicated pages exist when the same guest OS and applications run in different co-located VMs. Without KSM, as shown in Figure 2.2(a), multiple identical virtual pages of VMs are mapped to their own physical pages resulting in increased memory usage. In contrast, KSM regularly scans the memory of all VMs and identifies identical pages using red-black trees. When identical pages are found, KSM replaces them with a single COW-shared page, as shown in Figure 2.2(b).

The *ksmd* daemon [16], also known as the KSM daemon, conducts periodic scans on designated regions of user memory with the primary aim of identifying pages containing identical content and replacing them with a single write-protected page. KSM only operates on those areas of address space which an application has advised to be likely candidates for merging, by using the `madvise` system call. The number of pages that KSM scans in a single pass and the time between the passes are configurable; this enables the administrator to adjust the aggressiveness with which KSM uses CPU resources to identify identical pages. More aggressive settings allow KSM to converge to a smaller memory footprint faster, but at the expense of potentially affecting VMs' performance. It's important to note that KSM exclusively consolidates anonymous (private) pages and doesn't merge page cache (file) pages.



(a) KSM Turned-off



(b) KSM Turned-on

Figure 2.2: (a) Without KSM, each virtual page has its own physical page in RAM. (b) With KSM, duplicated pages are COW-mapped to single physical page in RAM.

2.5.3 *mmap* and COW Protection

In this subsection, we focus on manually establishing COW protection for shared pages. Memory mapping using `mmap` system call allows us to map a file (or a shared memory object) to a calling process' virtual address space. In this system call, the address specifies the base address for the mapping, and `offset` specifies the starting byte location in the backend file into which the process's virtual address is mapped. Once mapped, the process can access the mapped region as if it's accessing its local memory [46, 65].

If `MAP_PRIVATE` option is used, the region of the backend file is mapped to the virtual address with COW protection. When the process tries to write to such a virtual page, the OS allocates a new page to write to; the corresponding changes are private to the process and not committed to the backend file. If the file is mapped to the virtual address using the `MAP_SHARED` option, it is written back to the backend file whenever the process dirties the virtual page. By default, if the `addr` parameter is set to NULL, the operating system assigns a virtual address to the newly mapped region. To specify a fixed address for the mapped region, use the `MAP_FIXED` flag.

2.6 Transferring Ownership of Pages

The intra-host VM migration is especially relevant when dealing with a VM that requires maintenance, upgrades, or recovery from failures, necessitating the migration of its hosted processes. In contrast to inter-host migration, it's possible to avoid transferring the memory state by using the memory backend-file and bypass-memory flag as shown in Figure 2.3 (b). Here is the process for VM migration within the same node without any copying of pages:

1. Start the source VM with the memory backend-file [22].

2. Launch the destination QEMU in incoming mode, utilizing the same memory backend-file as the source. Since the destination is in incoming mode, it refrains from writing anything to the backend-file, preventing memory corruption.

3. Enable the `bypass-memory` flag before migration. This prompts QEMU to skip the memory transfer during pre-copy migration.

4. Initiate the migration. With the `bypass-memory` flag, pre-copy completely skips the initial and iterative pre-copy rounds, proceeding directly to the downtime phase, where it solely transfers the device and VCPU state.

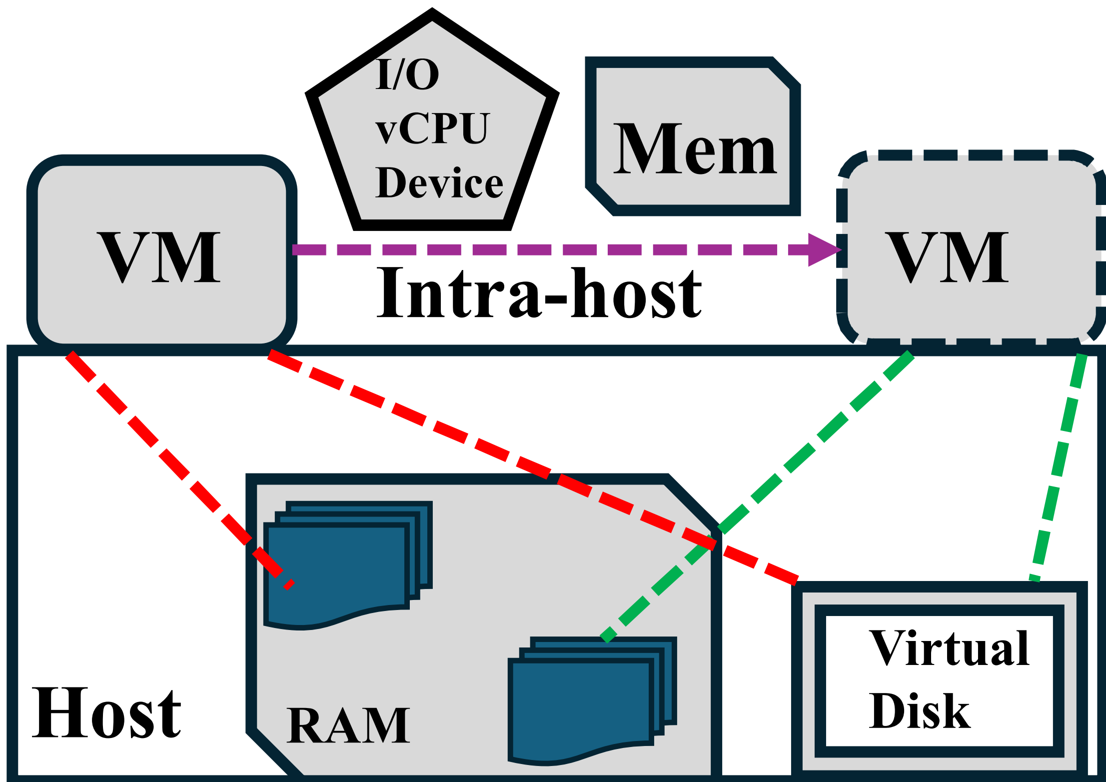
2.7 Userfaultfd

The conventional approach for detecting page faults involves the use of kernel modules. However, the userfaultfd [35, 45] offers user processes the ability to capture page faults and instruct the kernel on how to handle them through the `ioctl` interface. To capture these faults, the following steps should be followed:

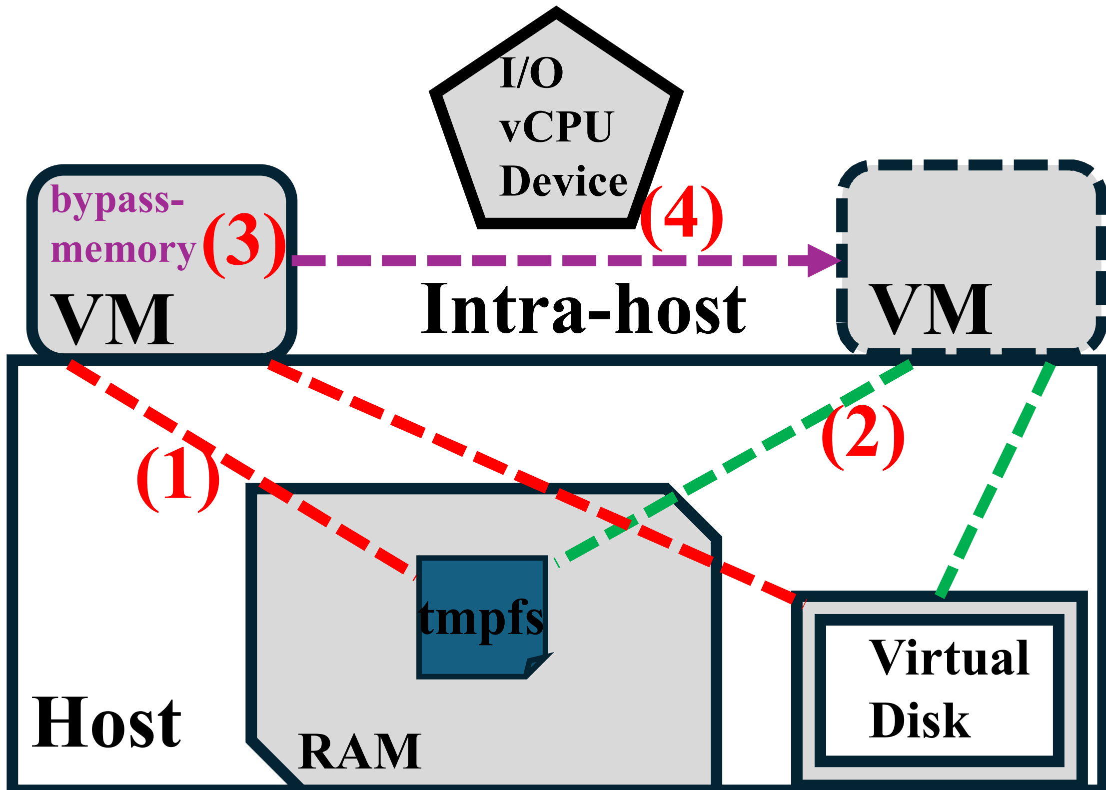
1. Register the specific address space region for which page faults should be triggered. The following modes should be used depending on the type of the pages. In QEMU, this memory registration for userfaultfd happens during virtual machine initialization. At this stage, the VCPU threads are not yet running.

    - To provoke page faults for *unallocated pages*, use `UFFDIO_REGISTER_MODE_MISSING`.

    - To provoke page faults when attempting to write *write-protected pages*, use `UFFDIO_REGISTER_MODE_WP`.



(a) Disabling bypass-memory



(b) Enabling bypass-memory

Figure 2.3: (a) Without the bypass-memory flag, the memory of the VM is copied again. (b) The bypass-memory flag skips copying the memory resulting in the transfer of only the I/O, VCPU, and Device states.

- To induce page faults for already *populated pages*, opt for UFFDIO_REGISTER_MODE_MINOR.

2. Once a fault is triggered, the faulting thread can be either suspended or woken up again based on the above *mode* flags. To wake up a suspended faulting thread, an `ioctl` request must be made with the appropriate parameters. Depending on the registered page type, the parameter passed with `ioctl` request may vary:

   - For unallocated pages, `ioctl` must be passed with `UFFDIO_COPY`.
   - For write-protected pages, `ioctl` must be passed with `UFFDIO_WRITEPROTECT`.
   - For populated pages, `ioctl` must be passed with `UFFDIO_CONTINUE`.

3 Inter-host Template-aware Live Migration

The widespread use of smart devices in fields like healthcare, transportation, and social media is creating a large amount of data every day. Since these devices have limitations in their resources, they send complex tasks to be processed in the cloud. However, when tasks need to be completed quickly, using mobile cloud computing isn't the best choice. The arrival of 5G technology is pushing phone companies to improve user experiences, leading to the growth of Multi-access Edge Computing environments. Multi-access edge computing nodes offer an attractive option for executing tasks where users require low latency and high bandwidth [43, 53]. Virtual machines (VMs) can ensure both isolation and efficient resource utilization at the edge computing infrastructure [52, 10, 87]. Live migration [8, 28] is a key technology in an edge computing infrastructure that transfers running VMs from one physical node to another. It is widely used for a variety of purposes, such as load balancing [2, 70, 20], meeting service level agreements [56], energy savings [77], and seamless maintenance of physical edge nodes.

Multiple VMs that run different components of a distributed application may be colocated on the same physical node to reduce their inter-VM communication costs [80, 51]. Typically, it can take a long time to boot up VMs from scratch from traditional virtual disks because it involves OS and software initialization, including loading the necessary content to memory. VM templates allow multiple new VM instances to be

quickly started up from a single pre-checkpointed VM image as described earlier in Chapter 2 and shown in Figure 2.1. First one must save a custom VM template as a snapshot of a pre-booted VM that is initialized with necessary software. Next one can quickly restore multiple VM instances from the common template image instead of the virtual disk, where the template image is mapped COW into the memory of each VM instance. VM templating [40, 81, 60, 59] is one approach to quickly instantiate multiple lightweight VMs from a common read-only image, called a template, which is shared COW memory among the VM instances [1, 74]. Templated VM instances may often need to be migrated together to the same destination node for various reasons. For example, templated VM instances that run different components of a distributed application may require migration to the same destination node to maintain low inter-VM communication latency or to meet other QoS targets. Additionally, physical node availability, hardware availability, and multi-tenancy limitations may necessitate the migration of templated VM instances to the same destination node.

3.1 Problem Statement

Unfortunately, current live VM migration techniques do not consider page sharing among templated VM instances that are being migrated to the same destination. As a result, shared pages are transferred and replicated multiple times, as if they were separate pages, resulting in an increase in memory pressure at the destination node. This can also lead to migration failures when the edge destination lacks sufficient memory to accommodate the expanded memory footprint of the VMs that were comfortably co-located at the source. Replication of previously shared pages among VMs also results in longer migration time and increased network traffic.

Previous approaches to reduce the transfer of duplicate pages during live migration (such as [12, 24, 52, 10, 87] among others) use content-based hashing to detect identical pages and avoid their retransmission. However, they do not identify or maintain preexisting COW mappings among VMs when pages are transferred to the destination. Additionally, while hashing may be used to identify identical pages that are not COW-shared, it is unnecessary and computationally expensive for COW-shared pages.

3.2 Contributions

In this chapter, we address this problem of memory footprint expansion of templated VMs instances as they are live migrated together to a common destination node. The contributions of this work are as follows:

1. We identify and demonstrate the problems caused by live migration being unaware of underlying page sharing among templated VM instances, leading to a larger memory footprint at the destination, longer total migration time, and higher network traffic.

2. We present *Templating-aware Live Migration* (TLM) which migrates templated VM instances to a common destination while maintaining COW memory sharing with the base template. We also discuss potential ways to account for other forms of page sharing during live migration besides those due to templating.

3. We implement prototype of TLM in KVM/QEMU [38] virtualization platform and evaluate it using several benchmarks. Besides maintaining preexisting page sharings among templated VMs at the destination machine, TLM reduces the total migration time by up to 95.37% and network traffic by up to 92.15%.

In the rest of this chapter, we first present the design, implementation, and evaluation of TLM. The chapter concludes with a discussion of related work and summary of results.

3.3 Design

Traditional pre-copy live migration is unaware of the underlying COW page sharing among templated VMs. Hence it ends up transferring the shared base pages of the template multiple times. We propose Template-aware Live Migration (TLM) which addresses this shortcoming of pre-copy migration by ensuring that multiple templated VM instances maintain their COW page sharing with the base template even at the destination node and transfers only the *delta* pages that differ among various VM instances. TLM requires that we first track delta (or dirty) pages for VM instances before migration. Then, during TLM, only the delta pages are transferred for each instance. We describe these steps below in greater detail.

Delta Tracking before Migration: As mentioned earlier, the common template image is COW-mapped into each VM's memory. The template image could also be pre-loaded into the host's memory (such as into `tmpfs` [69]) to minimize access latency. When a VM tries to write to a COW-mapped page, a write fault is triggered and the hypervisor allocates a new private page into which the original page is copied and the VM can write to. We call these new pages *delta* pages, which are stored separately from the shared template.

TLM uses a dirty page tracking mechanism [19] to keep track of the *delta* pages of the templated VMs before the migration begins [20]. Templated VMs have COW access to the base pages of the template, so a write by a VM instance to a COW-shared

page triggers a trap to the hypervisor (a KVM kernel module in QEMU/KVM), which updates a dirty bitmap, and finally grants write permission on the trapped page. Any subsequent writes to the same page by the VM are no longer trapped until the migration starts.

Live Migration: We assume that the base template image is already accessible at the destination over network storage; if not, it could be transferred to the destination once before live migration begins. Initially, at the destination, TLM maps the base template image COW into the memory of each new VM instance. As illustrated in Figure 3.1, TLM then exclusively transfers the delta pages from the source. These delta pages replace the corresponding COW-mapped pages at the destination, resulting in new memory allocations for the delta pages.

In each round of TLM, a user-space VMM, called QEMU, fetches the latest dirty bitmap from KVM, similar to the conventional pre-copy approach. The delta pages in each round are subsequently marked as read-only and sent to the destination. Eventually, downtime is initiated when a minimal number of delta pages remain. At this point, the VCPUs are paused, the remaining VM states are transferred, and the templated VM instances are then resumed at the destination. Multiple VMs that started from the same base template are migrated concurrently.

With TLM, these VMs preserve the same collective memory footprint at the destination as they did the source. When compared to traditional pre-copy migration, TLM has a shorter total migration time because, besides the one-time transfer of base template image, only the *delta* pages need to be transferred to the destination during live migration.

TLM also supports live migration of hotplugged devices. The pages of these hot-

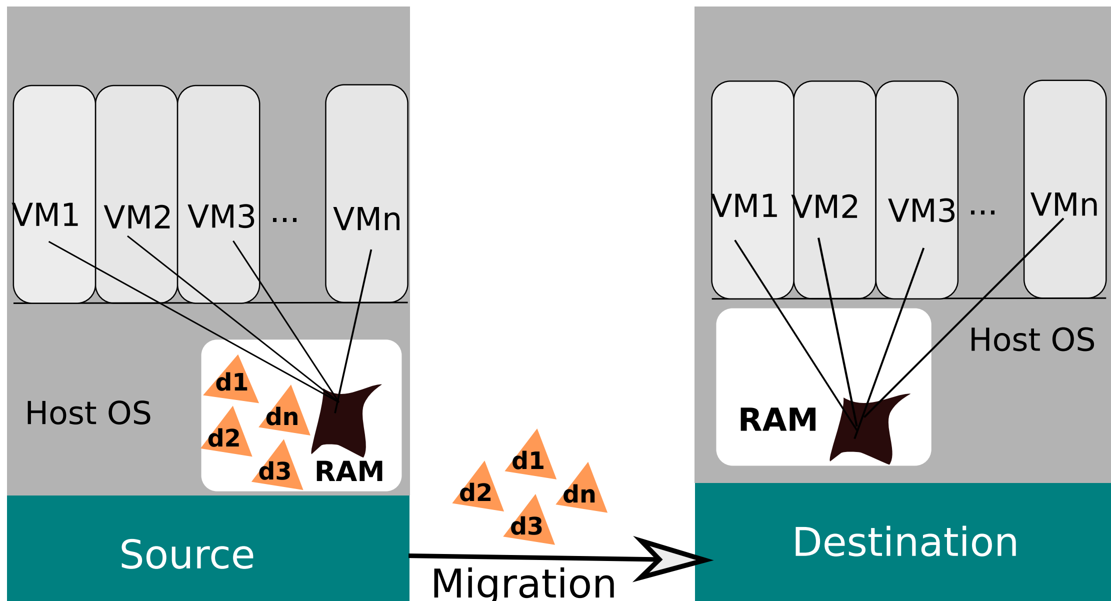


Figure 3.1: Template-aware migration works by migrating only the *delta* pages during migration. The shared VM template is available to the destination either over a networked storage or transferred ahead of time before migration begins.

plugged devices are write-protected so there is a trap to the hypervisor whenever the VM access the pages of the hot-plugged device. With hotplugging [48], during the first round of TLM, we transfer all the dirtied pages of hotplugged devices. The subsequent rounds transfer the additional dirty pages from the hotplugged devices. Multiple VMs that started from the same base template are migrated concurrently.

The current VM templating has following limitations:

1. In the existing VM templating mechanism in KVM/QEMU, once a template image is created from a VM, the VM is terminated. This prevents the user from creating multiple templates at different stages of the VM's execution. We investigated a seamless VM templating feature using which multiple templates can be created during a VM's execution.

2. Existing VM templating mechanism has a limitation in that a VM initialized from an existing template cannot be templated again due to its existing dependencies on

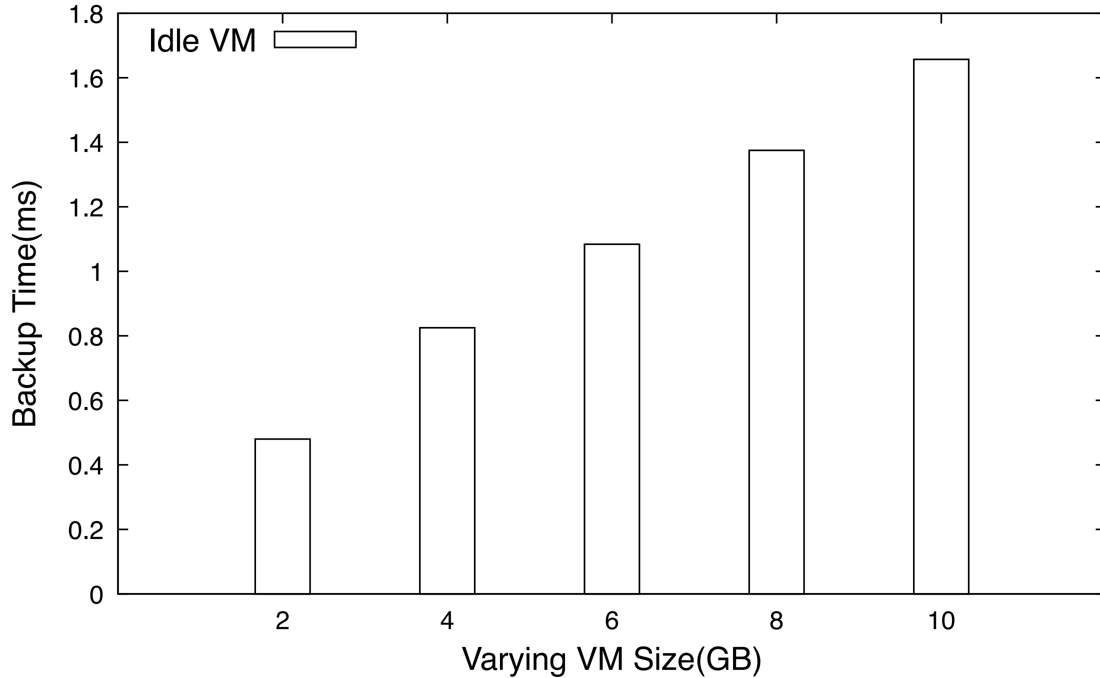


Figure 3.2: Time taken to take the memory snapshot of VMs varying in size.

the original template. We investigated chained templating feature that identifies the root causes of this limitation and ways to address the limitation.

3.3.1 Seamless Templating

Once the snapshot is created, the QEMU/KVM pauses the VCPU which prevents the templated VM to create multiple templates. Resuming the VCPU automatically after a snapshot is created enables the seamless templating feature.

3.3.2 Chained Templating

To create a snapshot from a VM, we need the memory backend and state files. The VMs booted from the existing template only have read access to the backend file, so the guest writes are not reflected on the underlying memory backend file, preventing them from creating a new template. By giving write access to the memory backend file, we were able to create new templates from the templated VMs.

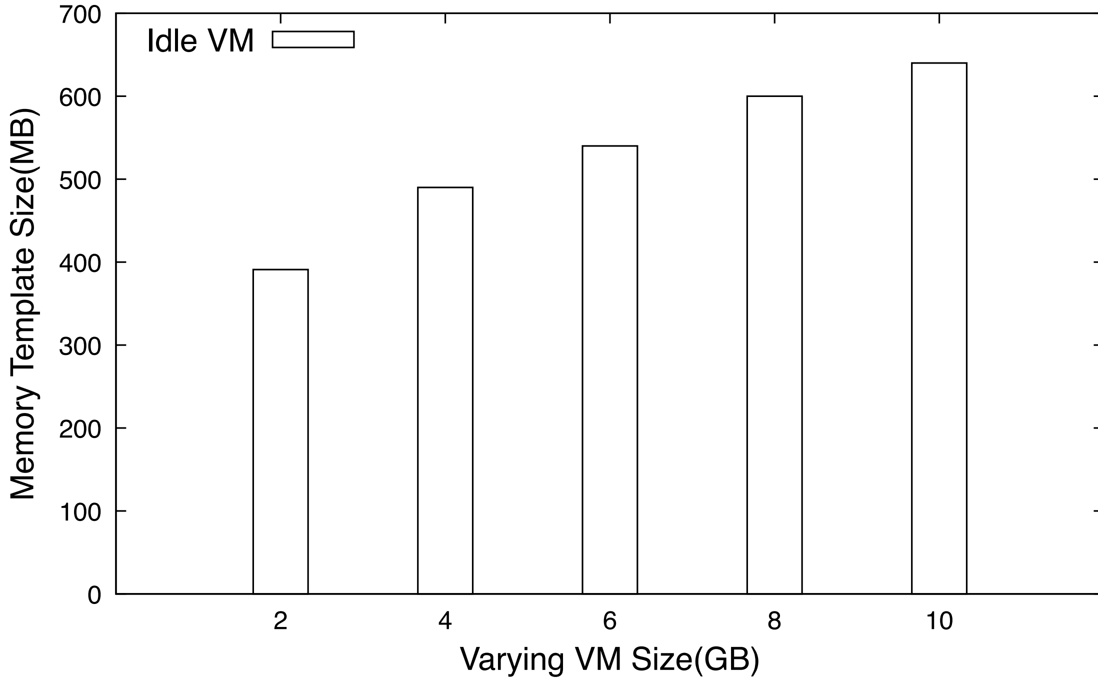


Figure 3.3: Memory usage of the memory template of varying VM size.

3.3.3 Snapshot Overhead

We allocated dedicated memory backend files and the VM state files for each template to maintain consistency and robustness. Initially, we used the cp command with bash to take the subsequent snapshots, which caused a substantial downtime, as shown in Figure 3.2 and 3.3, for downtime and memory usage. So we further optimized it by developing memory-layout-aware and parallel snapshot mechanisms.

3.3.4 Memory-layout-aware and Parallel Snapshot

For both Seamless and Chained templating, when snapshots are taken during the downtime, the VM's downtime can be significantly larger. Figure 3.4 shows that the snapshot copy duration increases almost linearly with the size of the snapshot. In addition, calling `cp` from a bash script adds a few milliseconds overhead. To reduce this downtime, we first integrated the snapshot mechanism within QEMU, instead of calling the `cp` command from the bash script. Next, instead of using `memcpy()`, which

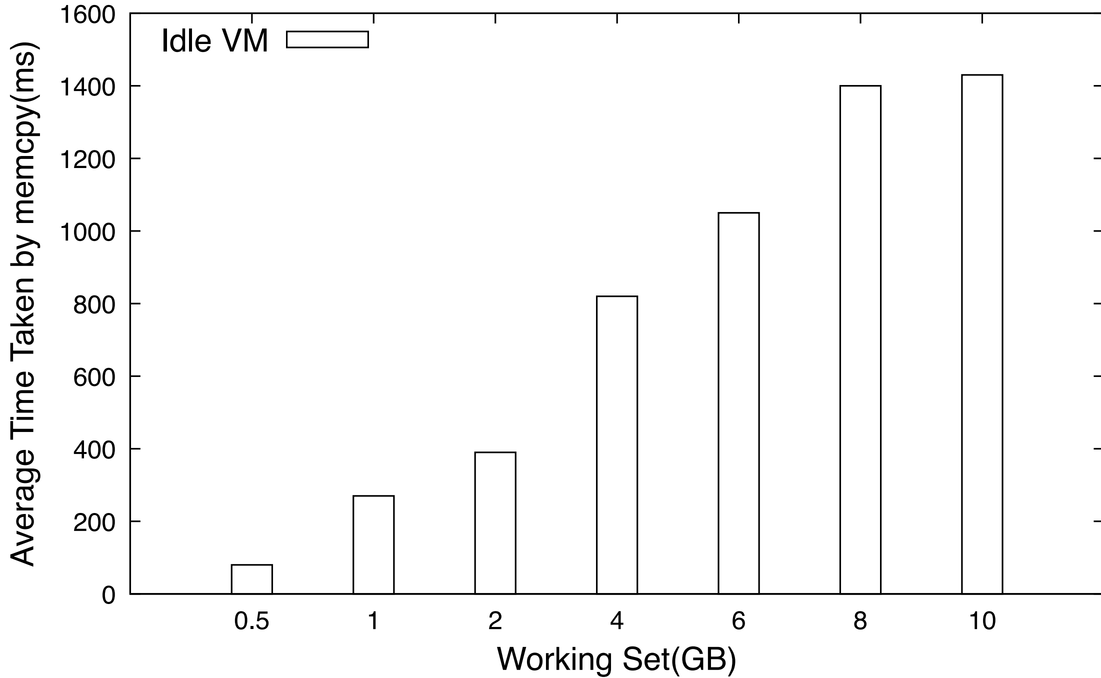


Figure 3.4: Snapshot copy time linearly increases with the working set size using memcpy.

copies the entire 8GB snapshot, we used the `lseek()` system call to seek and copy only the valid (allocated) data regions while skipping holes (unused/unallocated memory regions). This optimization significantly reduced the downtime.

Our second optimization was to parallelize the snapshot operation using multiple threads. We have divided the snapshot memory equally among multiple threads. Each thread performs lseek-based memory copy within its independent regions. In this manner, with the support of multi-threading, we reduced the VM downtime further, as shown in Figures 3.5 and 3.6 for downtime and memory usage.

3.4 Implementation

TLM uses dirty page tracking mechanism to keep track of the *delta* pages of the templated VMs before the migration begins. Templated VMs have read-only access to the template, so a write to a page triggers a trap to KVM, which updates a dirty

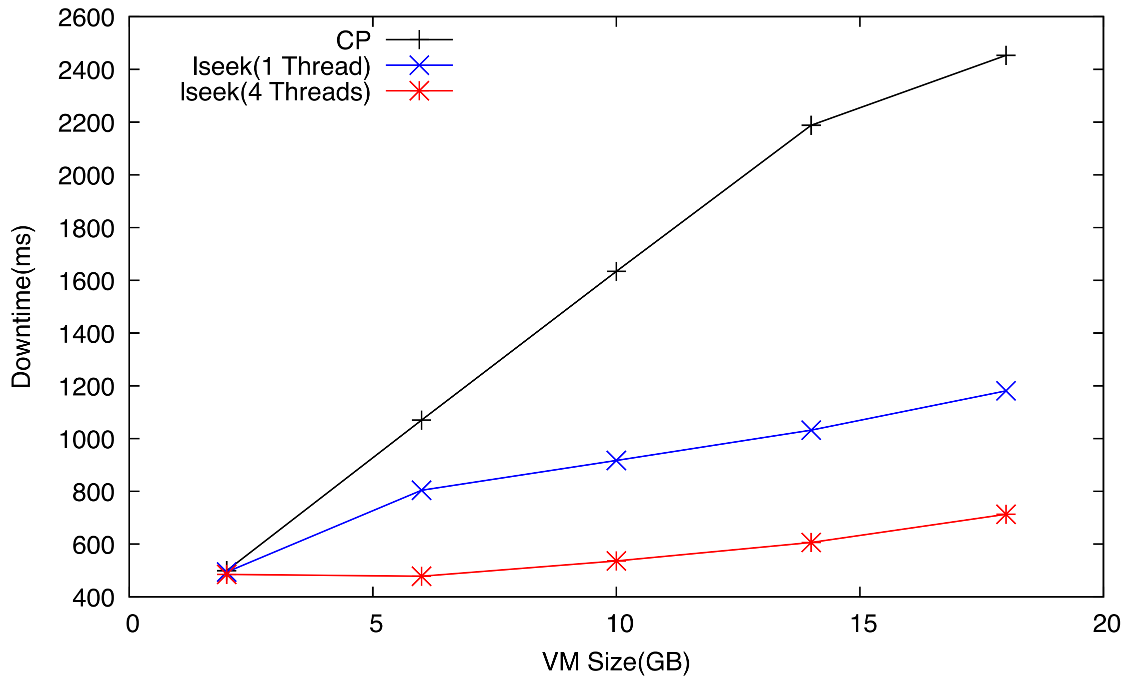


Figure 3.5: Lseek-copy performs better than both `cp` in terms of downtime.

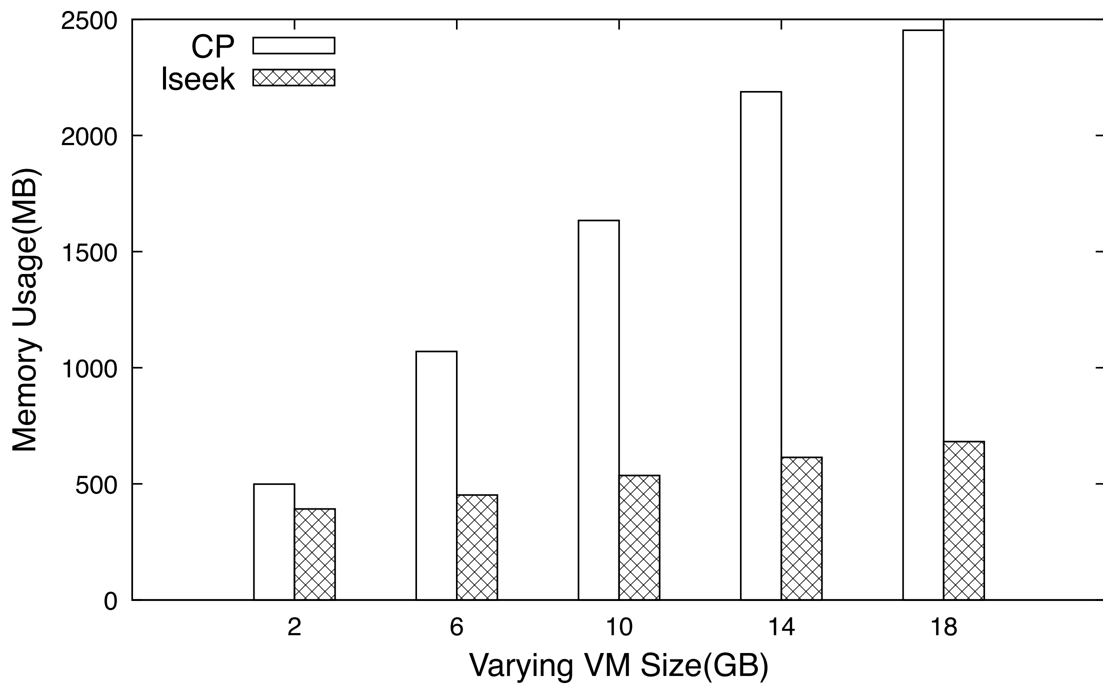


Figure 3.6: Lseek-copy performs better than both `cp` and `memcpy` in terms of memory usage

bitmap, and finally grants write permission on the trapped page. Any subsequent writes to the same pages are no longer trapped by KVM until the migration starts. Once live migration starts, TLM transfers only the delta pages that it had recorded earlier. This is unlike traditional pre-copy which transfers the entire memory of the VM during its first iteration. At the beginning of every round in TLM, QEMU retrieves the latest dirty bitmap from KVM, just like traditional pre-copy. The *delta* pages in each round are marked read-only again and transmitted to the destination. During downtime, TLM transfers any remaining dirty pages using stop-and-copy.

Limitation of TLM: While the above TLM approach works well in efficiently live migrating multiple templated VMs, we realized that the problem of sharing-awareness in live migration extends beyond just templated VMs. Specifically, TLM does not account for pages shared among VMs due to other memory sharing mechanisms besides templating, such as memory deduplication performed by Kernel Samepage Merging (KSM) [1] in Linux, or simple COW mappings due to process fork and file I/O operations. Figure 3.8 shows that memory expansion problem during live migration exists even for regular non-templated VMs, though to a lesser extent than templated VMs shown in Figure 3.7. The SLM technique, presented in the Chapter 5, improves upon TLM and retains all existing COW sharing during migration irrespective of the underlying memory optimization technique.

3.5 Evaluation

We evaluated the performance of TLM against generic pre-copy live migration. Our experimental setup consists of three machines with two Intel Xeon E5-2620 v2 processors and 128GB DRAM. We implemented TLM versions of pre-copy in the

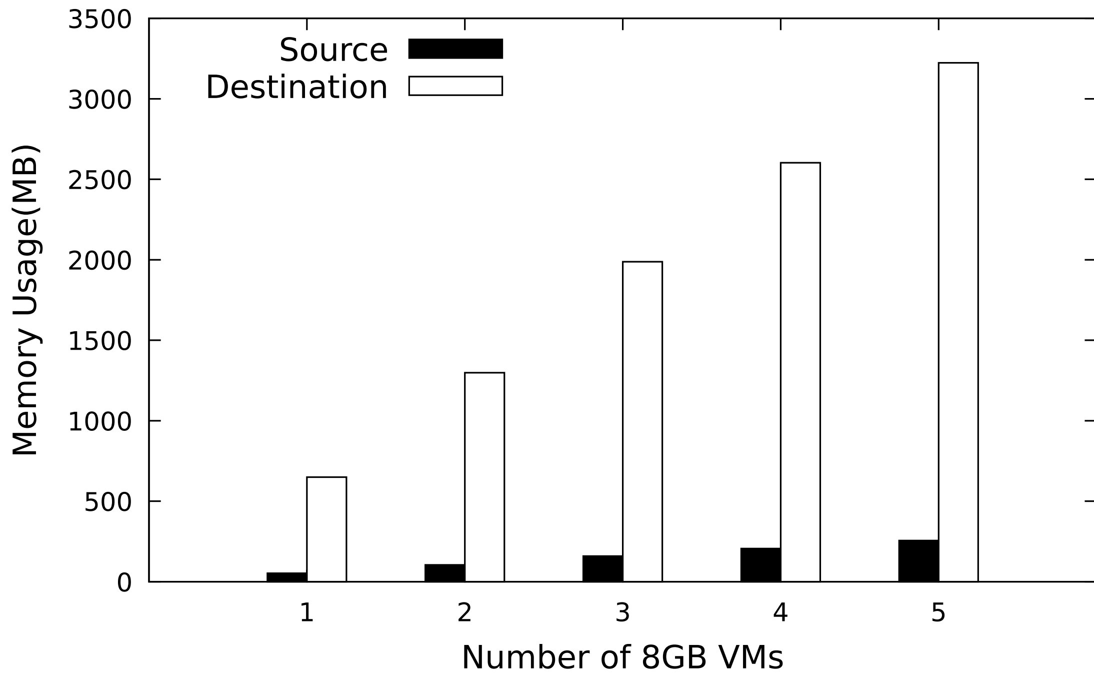


Figure 3.7: Memory footprint increase at destination when multiple templated VMs are migrated using generic pre-copy

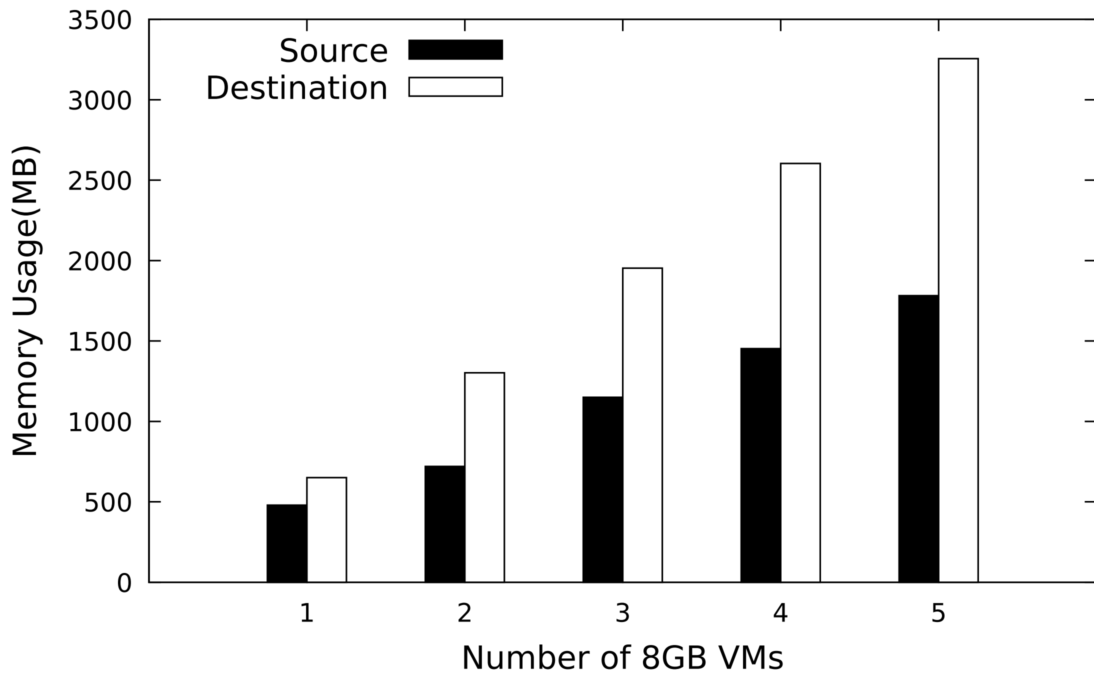


Figure 3.8: Memory footprint expansion problem in regular non-templated VMs at destination after live migration using generic pre-copy.

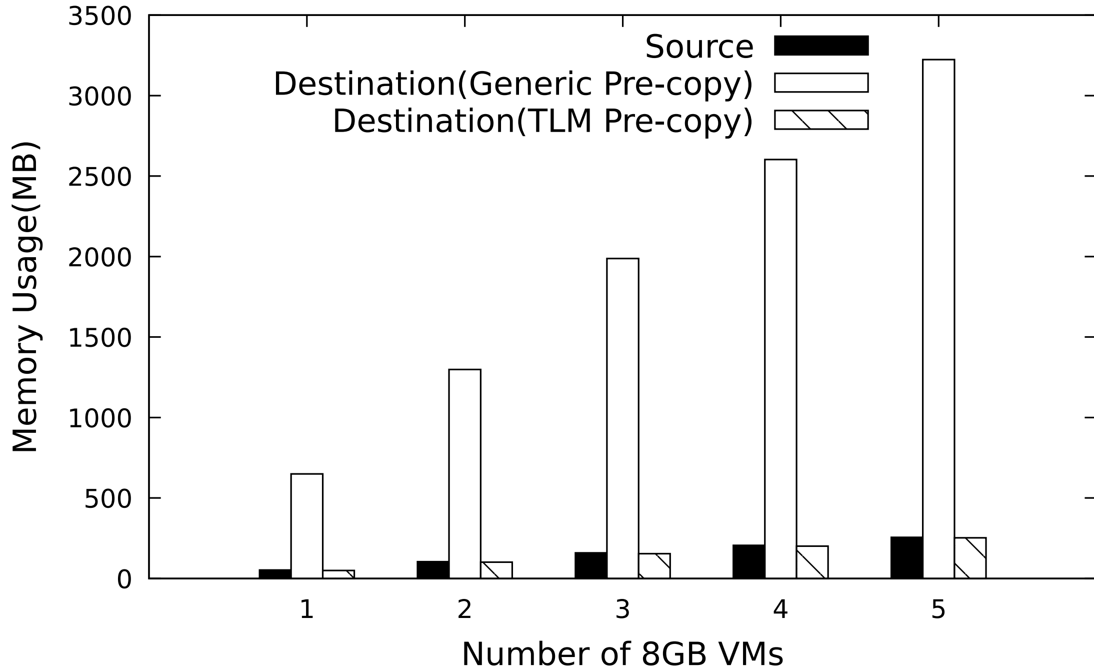


Figure 3.9: Memory footprint of templated VMs at the source before migration and destination after migration using Generic and TLM pre-copy.

KVM/QEMU [38] virtualization platform on Linux. We modified QEMU's default pre-copy algorithms, with no changes to the guest operating system. Each experiment was repeated at least five times on idle VMs to compute average values

Figure 3.9 shows the memory footprint of templated VMs migrated using generic and TLM pre-copy. The X-axis shows the number of VMs started from the same template, and the Y-axis shows their memory usage before migration at the source and after migration at the destination using `free` command. With increasing number of VMs, generic pre-copy migration results in significant expansion of memory footprint at the destination since it is unaware of memory sharing with the underlying template image. Hence pages that were originally shared with the base template at the source are transferred multiple times in addition to *delta* pages. In contrast TLM preserves the original memory footprint of templated VMs at the destination irrespective of the number of VMs started using the template, because shared pages are transferred only once and

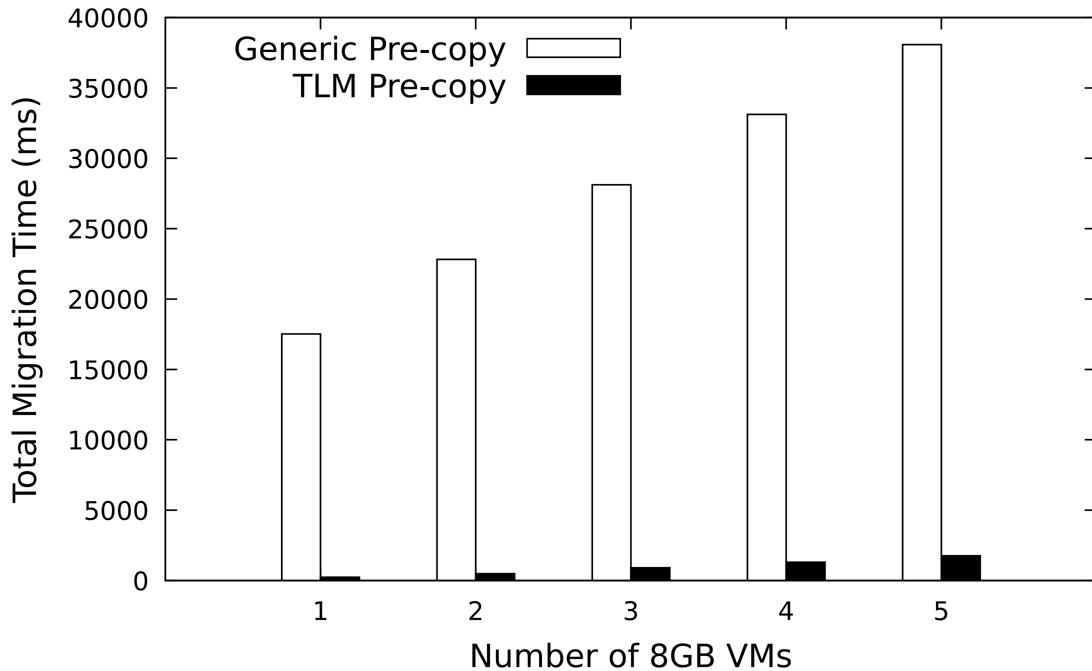


Figure 3.10: Total migration time of multiple VMs started from the same template and migrated concurrently using Generic and TLM pre-copy.

remapped to a common page at the destination.

Figure 3.10 shows the total migration time of multiple templated VMs using generic and TLM pre-copy. The X-axis indicates the number of VMs booted from the same template to be migrated concurrently. The Y-axis shows the total migration time. TLM reduces the total migration time up to 94% when considering only the transfer of *delta* pages.

Figure 3.11 shows that the downtime experienced during live migration of templated VMs is slightly longer (by a few tens of milliseconds) than that of generic VMs, even when the same number of dirty pages are transferred within the downtime window. The X-axis shows the number of pending pages transferred during the downtime and the Y-axis shows the downtime. Upon closer examination of the code, we identified that the increased downtime is influenced by the destination component of live migration. After the final packet arrives at the destination, the `vm_start()` function to resume the

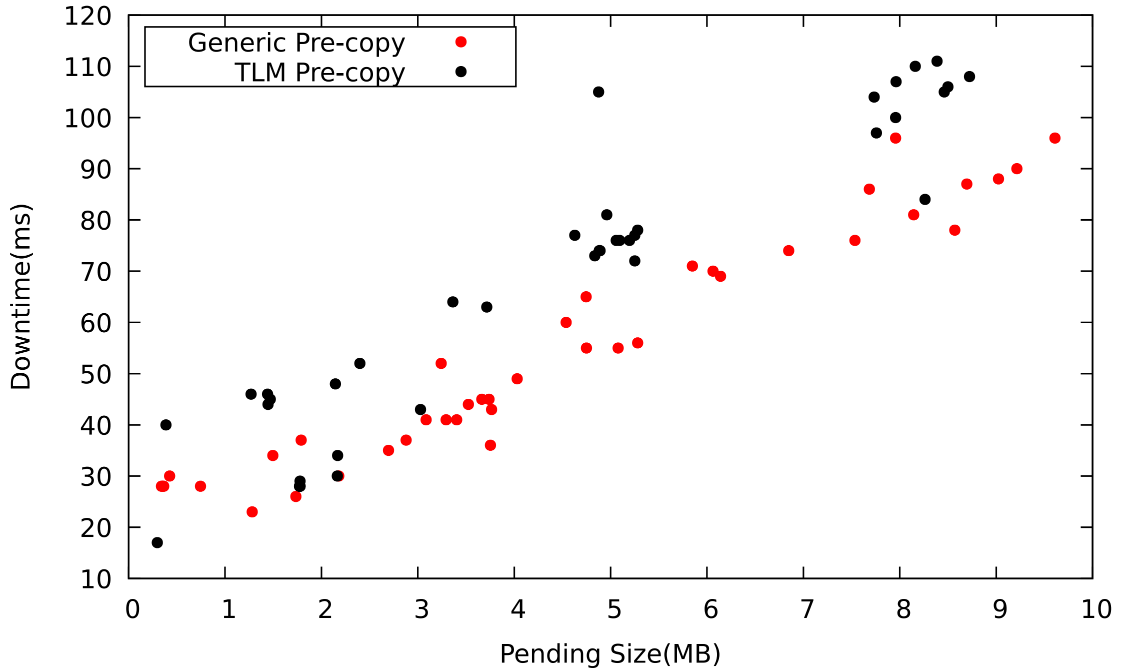


Figure 3.11: Downtime of multiple templated VMs migrated using Generic and TLM Pre-copy

VM can introduce variable overhead during the migration of both templated and generic VMs. However, in the case of templated VMs, the `vm_start()` function tends to result in higher resumption times more frequently. Addressing the downtime issue involves the VCPU thread invocation which will be addressed in our future work. Nonetheless, the downtime can still be minimized by configuring a smaller pending page count threshold for initiating the downtime phase.

Figure 3.12 shows the total 4KB pages transferred of multiple templated VMs using generic and TLM pre-copy. The X-axis indicates the number of templated VMs to be migrated concurrently. The Y-axis shows the total pages transferred during live migration. TLM significantly reduces the total pages transferred since it only transfers the delta pages.

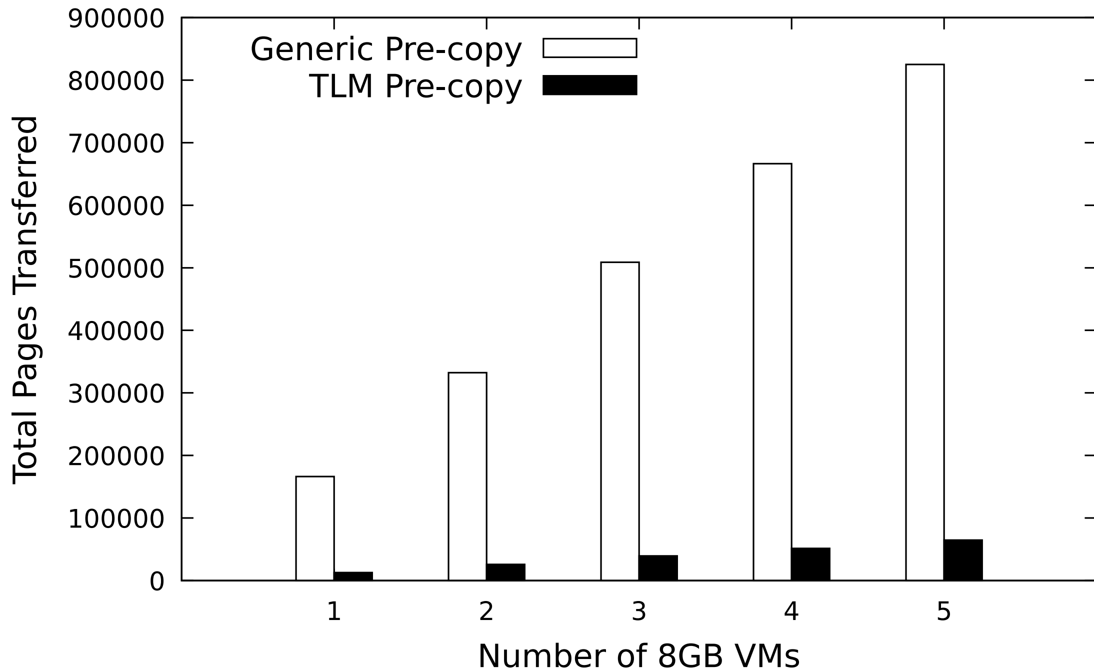


Figure 3.12: Total Pages Transferred of multiple VMs started from the same template and migrated concurrently using Generic and TLM pre-copy.

3.6 Related Work

Several VM templating techniques have been developed to efficiently launch multiple lightweight VMs from a common COW template image [40, 79, 81, 59]. VM templating can reduce the memory pressure and instantiation time of VMs on resource constrained edge computing nodes. To the best of our knowledge, these techniques lack support for live migration for templated VMs while maintaining COW sharing at the destination. Our TLM approach addresses this gap. Several studies have also attempted to minimize the migration time of containers or VMs in distributed edge platforms [52, 10, 87]. These efforts have employed techniques like delaying the transfer of writable working sets, using lightweight file systems, applying delta encoding, compressing data, and deduplicating identical pages. However, unlike TLM, these works do not focus on preserving COW page sharings at the destination to prevent an increase in memory footprint.

3.7 Chapter Summary

In this chapter, we addressed the problem that traditional live VM migration techniques do not maintain page sharing among templated VM instances that are migrated to the same destination nodes on edge platforms. This leads to increased memory footprint at a resource constrained destination node, longer migration time, and increased network traffic. We presented the design, implementation, and evaluation of a technique called TLM for pre-copy that retains the templating benefits at the destination after the live migration. Our evaluation of TLM on the QEMU/KVM platform shows that TLM not only avoids memory footprint expansion at the destination but also significantly reduces the migration time and the amount of data transferred. The SLM technique, presented in Chapter 5, improves upon TLM and incorporate other forms of inter-VM page sharing besides those due to templating.

4 Intra-host Template-aware Live Migration

As discussed in the previous chapter, edge computing nodes may execute tasks requiring low latency and high bandwidth [43, 53]. Edge nodes use VMs or containers encapsulated within VMs for security and isolation purposes. This approach allows applications to leverage the security and performance isolation provided by the VM while maximizing the provisioning and deployment capabilities inherent in VMs [52, 72, 23, 54, 32, 78, 59, 10, 87].

Live migration [8, 28] plays a pivotal role in cloud computing infrastructure by transferring running VMs from one physical node to another. This technique can be employed for migration either between two different hosts (inter-host) for purposes such as load balancing [2, 70, 20], meeting service level agreements [56], or within the same host (intra-host) for patching/updating VMMs [62, 63], or for instrumentation and taint analysis [82, 29].

At times, running VMs may require a replacement of the VMM (the QEMU process in KVM/QEMU) for various reasons, such as feature updates, security patches, bugs, etc. The conventional approach to tackle this problem involves terminating the VMs in the production environment and conducting the replacement offline [29, 14, 82]. However, this approach results in significant downtime, adversely affecting the liveness of the virtual machines. An better alternative is to update the VMM on-the-fly using live migration. By live migrating a VM from an old VMM (the source) to a new VMM

(destination), the downtime of the VM during replacement can be reduced.

4.1 Problem Statement

While live migration techniques have been extensively studied for migration between different hosts, there has been limited exploration [73] of live migration in intra-host environments. Current live migration techniques for both regular non-templated and templated VMs are inefficient when migrating VMs within the same host as they involve unnecessary copying of pages causing intermediate spikes in memory usage during migration. Besides memory contention, they also increase the total migration time, thereby prolonging the spikes in migration resource usage, which can adversely impact application performance in nodes with limited memory.

Existing pre-copy live migration involves iteratively transferring memory pages to the destination by copying them. This copying is necessary for migration between two different hosts. However, using the above technique is inefficient for migration within the same host, as it duplicates the memory pages, as shown in Figure 2.3 (a). This duplication is unnecessary for intra-host migration; instead, all we need to do is transfer the ownership of the memory pages from the source to the destination QEMU, which can be achieved with the help of a bypass memory flag thereby improving memory contention and total migration time, as shown in Figure 2.3 (b).

Our previous work Generic Template-aware Live Migration of Virtual Machines or Generic TLM addresses the shortcoming of pre-copy intra-host migration by ensuring that multiple VM instances maintain their COW page sharing with the base template and transfers only the *delta* pages that differ among various VM instances. Generic TLM requires that we first track delta (or dirty) pages for VM instances before migra-

tion. Then, during migration, only the delta pages are transferred for each instance. However, a limitation arises when applying the Generic TLM technique for intra-host live migration of templated VMs. Specifically, Generic TLM ends up copying delta pages from the source QEMU to the destination QEMU, which is unnecessary when both the QEMUs are present on the same host.

An earlier work by our group, Mwarp [73], addresses intra-host live migration of containers by transferring ownership of containers from one VM to another without copying pages. However, this solution is container-specific and cannot be applied to VMs. Several live patching works related to hypervisors, such as those presented in [86, 13, 3], focus on replacing old hypervisors with new hypervisors without disrupting VMs. But a drawback is that these techniques cannot be used for replacing VMMs (such as QEMU) when live patches/updates need to be performed on the VMM.

4.2 Contributions

Contributions: We propose Intra-host Template-aware Live Migration of Virtual Machines, or Intra-host TLM, as a new way to handle migration of templated VMs within the same host. The fundamental idea behind Intra-host TLM is to transfer ownership of delta pages, which reside on the same host, rather than duplicating them during migration. Intra-host TLM utilizes the *userfaultfd* mechanism [35, 45] to monitor delta pages and save them in a backend file. When the migration is issued, the source VMM only sends the corresponding offset in the backend file for delta pages so that the destination QEMU can claim ownership of those pages once the migration is complete. This technique completely eliminates intra-host transfer of delta pages over a TCP connection reducing the spike in memory usage, and improving the total migration time

significantly. Our contributions are as follows:

1. We identify and demonstrate the problems of memory usage spikes during intra-host live migration of templated VMs using Generic TLM.

2. We present Intra-host Template-aware Live Migration of VMs which migrates templated VM instances to a common destination within the same host without unnecessary memory footprint increase thereby reducing the total migration time.

3. We implement a prototype of Intra-host TLM in KVM/QEMU [38] virtualization platform and evaluate it using several benchmarks. Besides reducing the memory footprint, Intra-host TLM also significantly reduces the total migration time by 85%.

Outline: In the rest of this chapter, we first demonstrate the problem of migrating templated VMs within the same host. Next, we present the design and implementation of Intra-host TLM followed by its evaluation. The chapter concludes with a discussion of related work and a summary of the results.

4.3 Problem Demonstration: Memory Spikes During Intra-host Live Migration

In Figure 4.1, we show the problem that arises during the migration of templated VMs using pre-copy within the same host. The X-axis represents the number of VMs instantiated from a shared template that is being migrated, and the Y-axis represents their collective memory footprint as measured by the `free` command in Linux. The legend of the graph provides the following information: (a) represents the system memory before executing any QEMU (Q) or VM. (b) represents the system memory usage

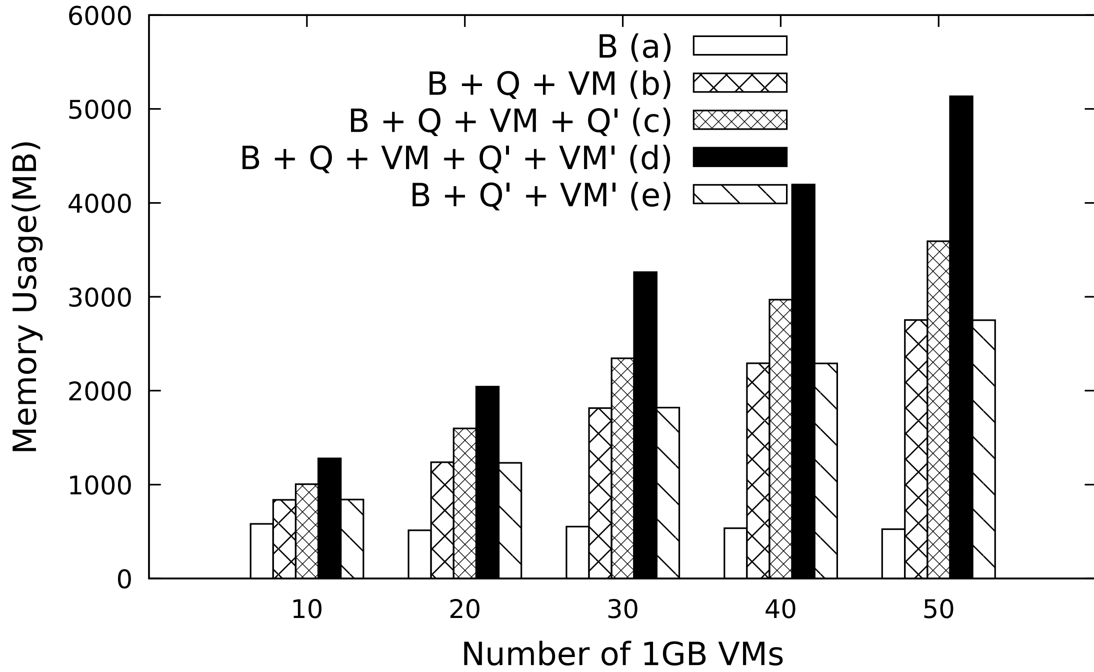


Figure 4.1: Memory footprint of templated VMs migrated within the same host using Generic TLM.

after booting a VM using Q. (c) represents the system memory usage after initiating a new Q' in listening mode. (d) represents the system memory usage after live migration, where Q, Q', VM, and VM' are all live. Finally, (e) represents the system memory state after terminating Q and VM, leaving only Q' and VM' live. When using Generic TLM for migration within the same node, there is an observable rise in the memory usage, particularly evident with an increase in templated VM instances (c) and (d). This is attributed to the duplication of delta pages during the migration process.

4.4 Design of Intra-host TLM

Intra-host TLM significantly mitigates this spike in memory usage by transferring the ownership of delta pages instead of unnecessarily copying them over a TCP connection. The outline of Intra-host TLM is described in Figure 4.2.

Preparation Stage – Step (1): The source QEMU registers the base template mem-

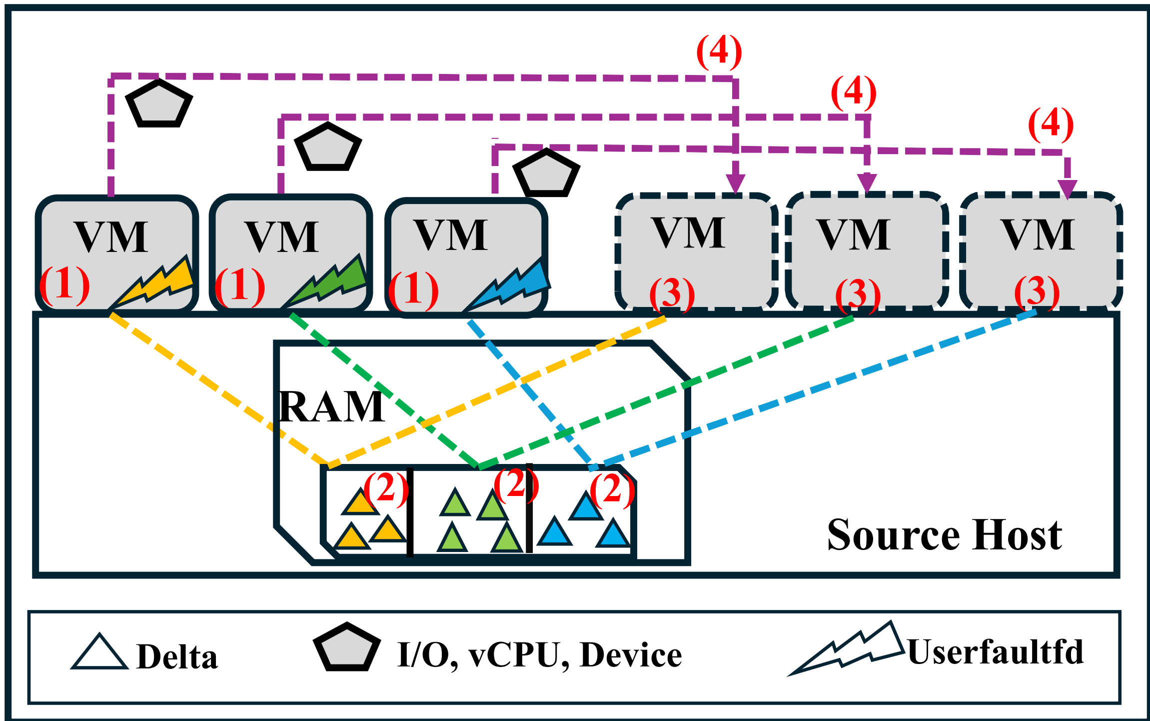


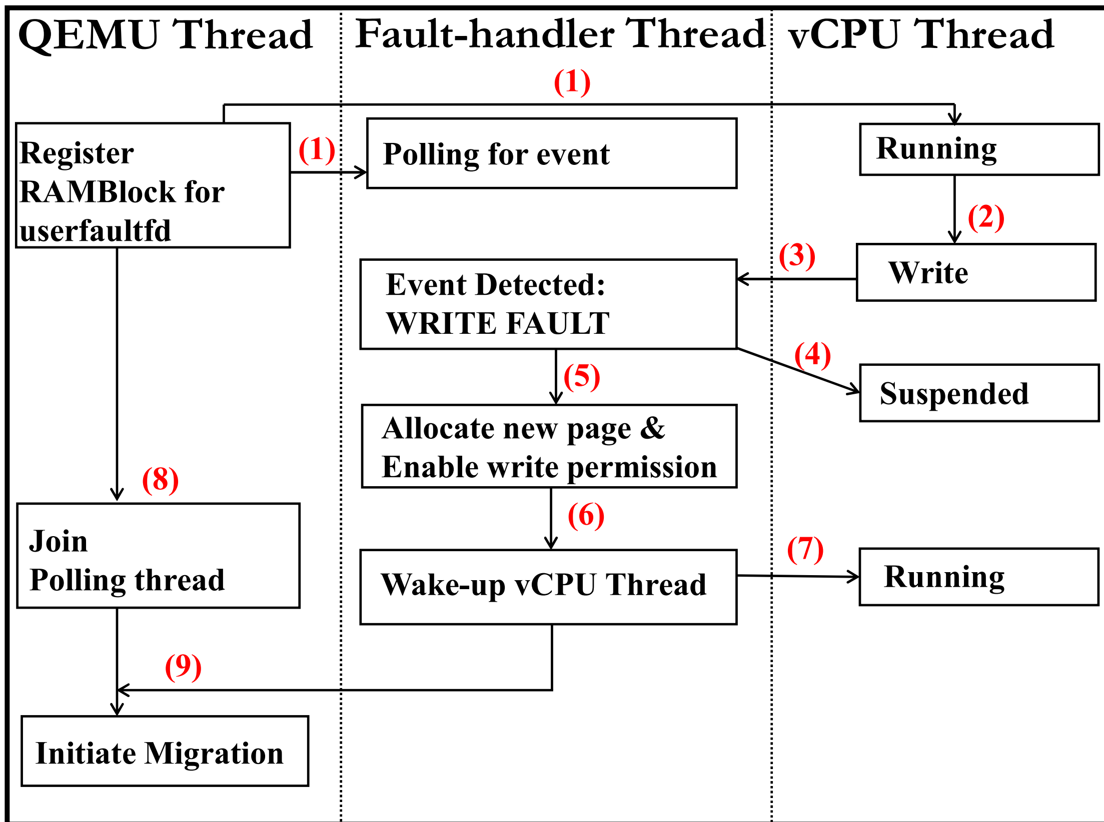
Figure 4.2: High-level overview of Intra-host TLM using userfaultfd.

ory from QEMU using `UFFIO_REGISTER_MODE_WP` with the help of userfaultfd. This is the initial setup required for the VM instantiation.

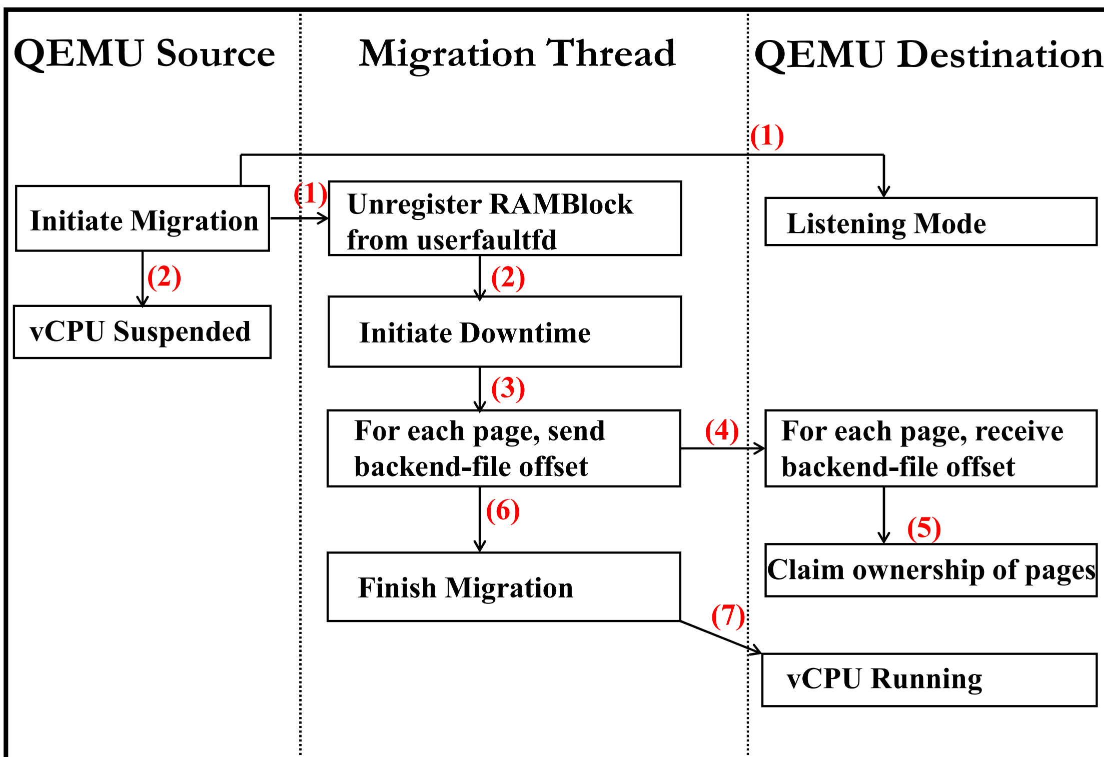
We create a new thread in the source QEMU equipped with a fault handler for userfaultfd, designed to continuously monitor write operations targeting the write-protected base template pages. This mechanism ensures that any write attempts are promptly detected and tracked by the fault handler. Lastly, we start the VCPUs of the VM. At this point, the templated VM is up and ready to use.

The essential aspect of this process is that the base template remains write-protected throughout. Consequently, any write operations to the base template will trigger the fault handler thread which then records the identity *delta* pages. This information is used by the subsequent steps.

Tracking Delta – Step (2): The fault handler thread operates in parallel with the VCPU threads and is responsible for intercepting write operations on the base memory



(a) Tracking the delta pages at the source QEMU using userfaultfd.



(b) Transferring the ownership of delta pages without copying.

Figure 4.3: Illustration of how Intra-host TLM transfers the ownership of delta pages using userfaultfd mechanism.

template. When a write operation is trapped, the following steps are performed.

The VCPU pauses execution and the fault handler thread and identifies the next available offset within the backend file to copy the content of the trapped write-protected page. Simultaneously, it unmaps and remaps the trapped virtual address with a new backend-file content using the `mmap` method with write permission, ensuring proper tracking, isolation of the changes, and preventing future write-faults to the virtual address. Any future writes to the page (including the trapped write) will be sent to the new page mapped into the backend file.

When migrating multiple templated VMs, each VM possesses its own dedicated backend-file to record delta changes.

Transferring Page Ownership – Steps (3) and (4): Our intra-host TLM exploits the fact that both the source and destination QEMUs reside on the same host, eliminating the necessity to transfer any pages during the migration process.

Typically, the traditional pre-copy migration involves three distinct phases: The first phase, referred to as *setup*, involves the preparation of RAM blocks and dirty bitmaps. The second phase is an iterative stage focused on transferring dirty pages. Finally, the last phase involves suspending the VM and transferring the remaining dirty pages, along with VCPU, I/O, and device states, and resuming the VM at the destination. In our intra-host TLM approach, we deliberately omit the second phase of iterative memory transfer to prevent copying delta pages.

Intra-host TLM executes the following steps. When the migration command is issued, the VMs immediately switch to the downtime phase. During downtime, we only transfer the corresponding offsets of dirty pages in the backend file which had been stored in the preparation stage. The destination VMs remap their virtual addresses to

the corresponding backend-file offset once they receive this information. Finally, we end the migration by transferring the I/O, VCPU and device states before resuming the VMs at the destination.

4.5 Implementation

4.5.1 Saving delta of Source VMs

Figure 4.3(a) shows the detailed steps involved in tracking delta pages using QEMU, polling, and VCPU threads.

QEMU Thread: (1) The RAMBlocks, which constitute the VMs' memory, must be registered for the userfaultfd mechanism to intercept any future write operations. Regardless of whether the VM is booted from a template or not, QEMU allocates the memory chunk in RAMBlocks [61]. For regular VMs, RAMBlocks are part of anonymous pages, and for templated VMs, they form part of the backend file. To enable userfaultfd to capture all writes to the backend file, the RAMBlocks associated with the backend file need to be registered with userfaultfd.

Memory template is a unique type of backend file; merely registering it with `UFFDIO_REGISTER_MOD` is insufficient due to the presence of unallocated pages. Registering unallocated pages with this mode results in a write fault. To address this issue, we preallocate the template during its creation

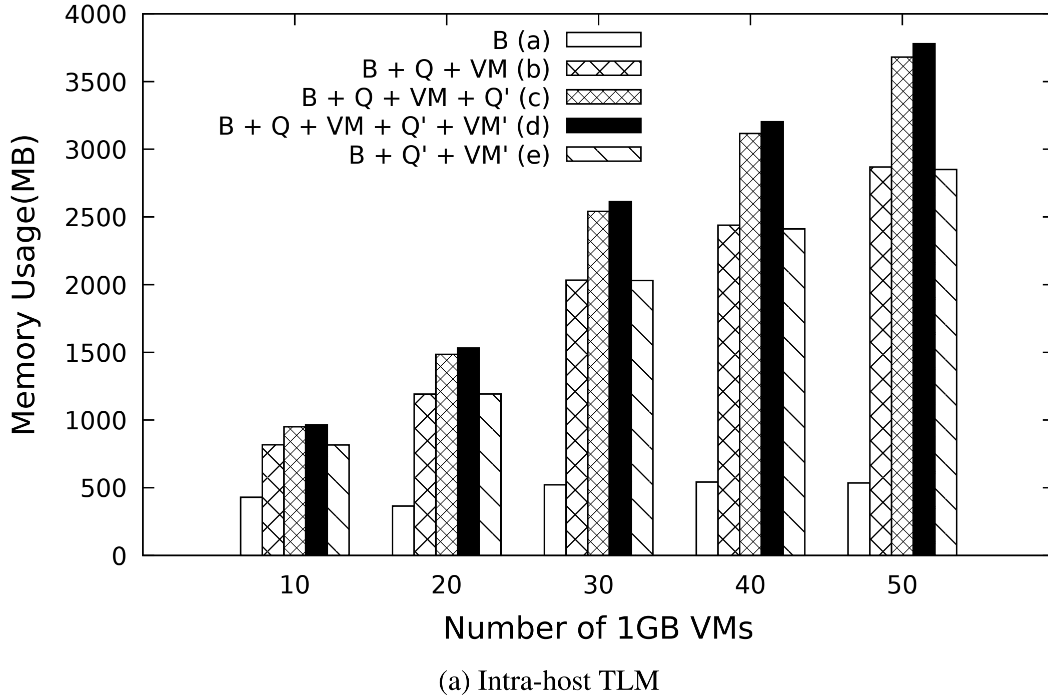
The source node prepares an in-memory backend file into which the content of the write-protected page is copied into. The backend file is stored in the main memory using the `tmpfs` file system to avoid the latency of accessing the virtual disk. Sometimes the OS uses lazy allocation when creating a backend file of large size. Doing so can result in bus errors when unallocated file regions are memory mapped to the virtual address

for unique pages. We eliminated the problem by writing zeros into the backend file before mapping the area for storing unique page contents.

Polling Thread (or Page Fault Handler Thread): In step (1), the page fault handler thread is spawned and continuously polls in parallel for `UFFD_EVENT_PAGEFAULT` to intercept any write faults in the protected region.

VCPU Thread: In step (1), since RAMBlock registration with userfaultfd and fault-handler thread setup occurs during the VM initialization phase, it has no impact on the VM's critical path.

In step (2), when a VCPU writes to a protected region, the polling thread receives the `UFFD_EVENT_PAGEFAULT` event (Step (3)) with the `UFFD_PAGEFAULT_WP` flag set. In Step (4), the VCPU thread enters a suspended state, awaiting a `UFFDIO_WAKE` from the fault handler to resume its operation. In step (5), the fault handler identifies the next available offset within the backend file to copy the content of the page on which write fault occurred. Simultaneously, it unmaps and remaps the trapped virtual address with new backend file content using the `mmap` method with write permission to avoid future traps. In step (6), the final action in this sequence involves waking up the suspended VCPU thread through the `UFFDIO_WAKE` ioctl parameter, allowing the VCPU thread to continue its execution with the newly allocated page (Step (7)). In cases involving multiple templated VMs, each VM has its dedicated backend file to record delta changes. This approach ensures that changes are tracked independently for each templated VM, maintaining the integrity and isolation of their respective environments. Finally, in step (8), before initiating the migration process, the polling thread joins with the main QEMU thread.



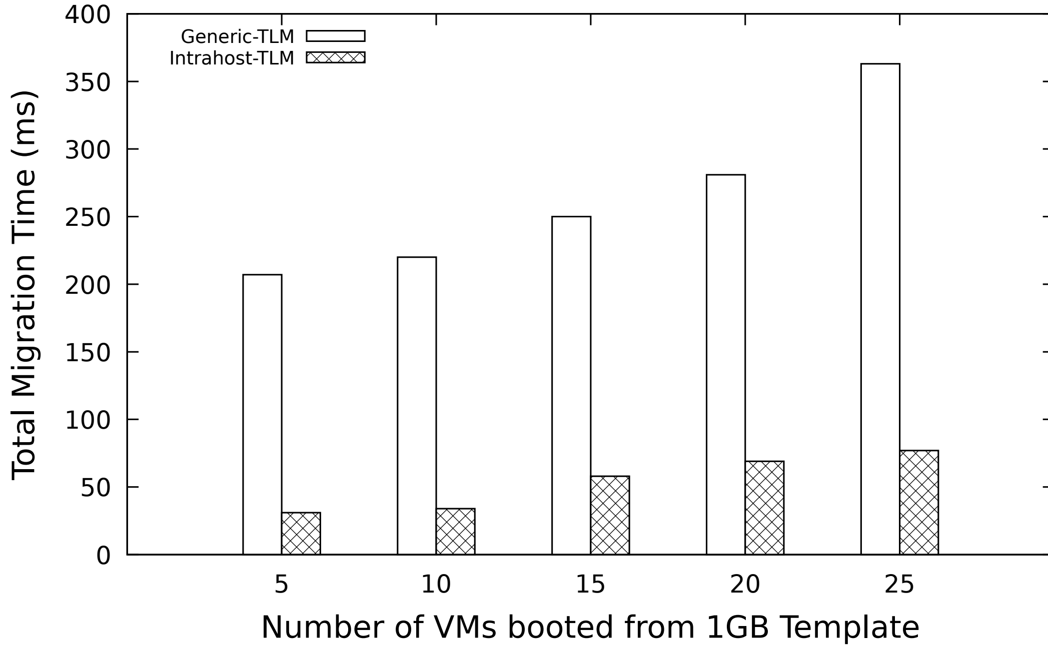
(a) Intra-host TLM

Figure 4.4: Memory footprint of templated VMs at the source before migration and destination after migration using Intra-host TLM

4.5.2 Transferring ownership of delta from Source VMs

Having tracked and mapped each delta page to the backend file, the next step is to transfer ownership of these pages to the destination QEMU. Two critical pieces of information are used for this process: identity the trapped pages and the corresponding backend file offset for the ownership transfer.

Figure 4.3(b) shows that before initiating the migration command, the source QEMU joins the polling thread and spawns the migration thread to transfer ownership of the delta pages. In step (1), the migration thread unregisters the RAMBlocks from the userfaultfd mechanism to prevent further write traps. In step (1), the destination QEMU is also started in listening mode with access to the backend file memory template. In step (2), once the migration is initiated, the VCPU at the source is suspended, and the downtime is initiated as well. In step (3), the migration thread examines the dirty bitmap



(a) Intra-host TLM

Figure 4.5: Total migration time of multiple VMs started from the same template and migrated concurrently using Intra-host TLM

of pages and transmits the backend file offset for those pages that are marked as dirty, using the information collected during the preparation stage. In step (4), for each page, the destination QEMU receives the offset, remaps the virtual address to the new location in the backend file with write permission and, in step (5), reclaims ownership of those delta pages. In step (6), once all the dirty pages' offsets have been sent, the QEMU source completes the migration. Finally, in step (7), the destination QEMU resumes the VMs.

4.6 Evaluation

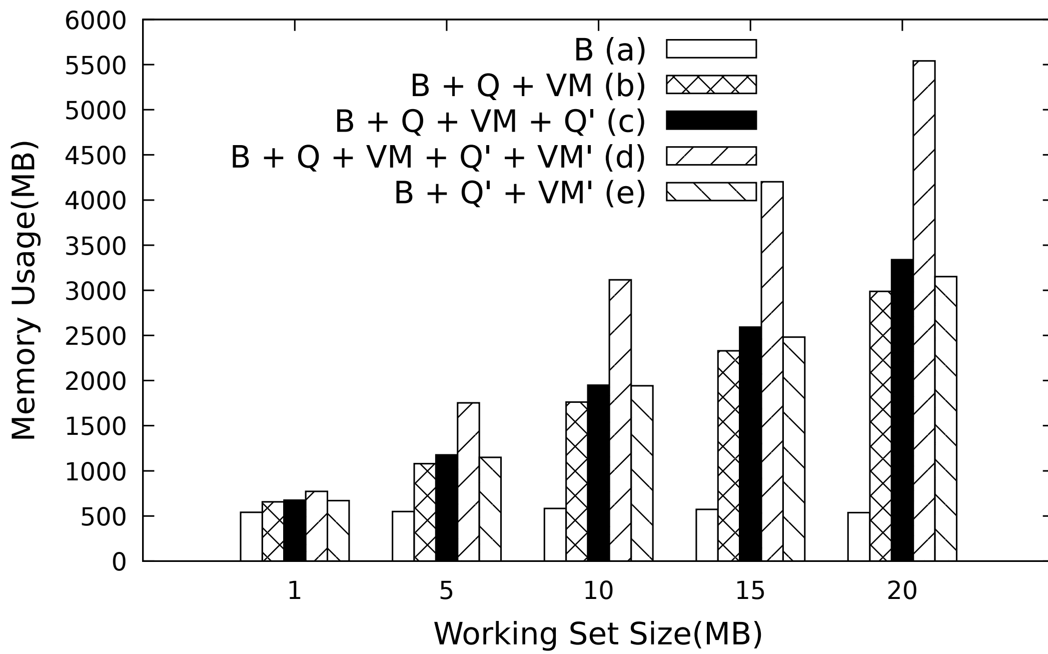
We evaluated the performance of intra-host TLM against Generic TLM. Our experimental setup consists of three machines with two Intel Xeon E5-2620 v2 processors and 128GB DRAM. We implemented Generic TLM and Intra-host TLM versions of pre-

copy in the KVM/QEMU [38] virtualization platform on Linux. We modified QEMU's traditional pre-copy algorithms, with no changes to the guest operating system. Each experiment was repeated at least five times to compute average values.

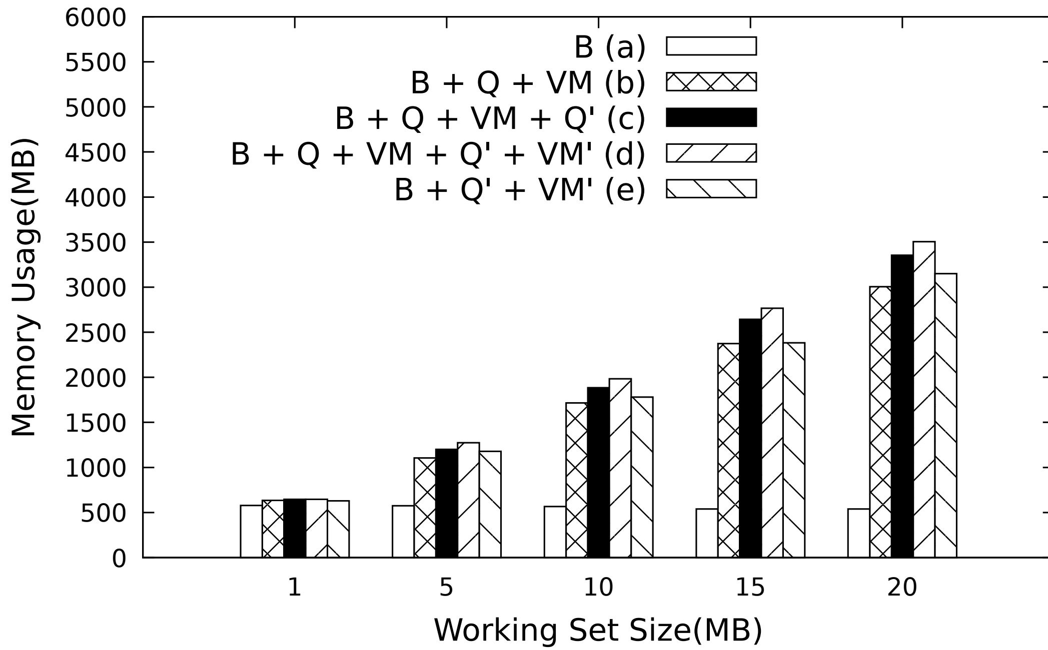
4.6.1 Reduced Memory Footprint

Figure 4.4 shows the memory footprint of templated VMs migrated using Intra-host TLM. The X-axis shows the number of VMs started from the same template, and the Y-axis shows their memory usage before migration at the source and after migration at the destination using `free` command. The legend (a) represents the system memory before executing any QEMU (Q) or VM. Legend (b) represents the system memory usage after booting a VM using Q. Legend (c) represents the system memory usage after initiating a new Q' in listening mode. Legend (d) represents the system memory usage after live migration, where Q, Q', VM, and VM' are all live. Finally, legend (e) represents the system memory state after terminating Q and VM, leaving only Q' and VM' live. For Generic TLM, we observed in Figure 4.1 legends (c)and (d) that the more the number of VMs, the more collective delta generated by them resulting in a significant increase in memory usage while they are migrated together. For Intra-host TLM, we observe in Figure 4.4 legends (c) and (d), that this memory usage spike is mostly eliminated by transferring the ownership of delta pages instead of copying them over a TCP connection.

To further evaluate Intra-host TLM, we migrated five templated VMs that each runs a write-intensive workload. The memory-write intensive application we used is a C program that writes random numbers to a large region of main memory. The program starts before the VM was migrated and continues writing to a large region of memory during migration. The size of the working set (i.e., the size of the memory written) for



(a) Generic TLM



(b) Intra-host TLM

Figure 4.6: Memory footprint of templated VMs running write-intensive benchmarks migration using (a) Generic TLM (b) Intra-host TLM.

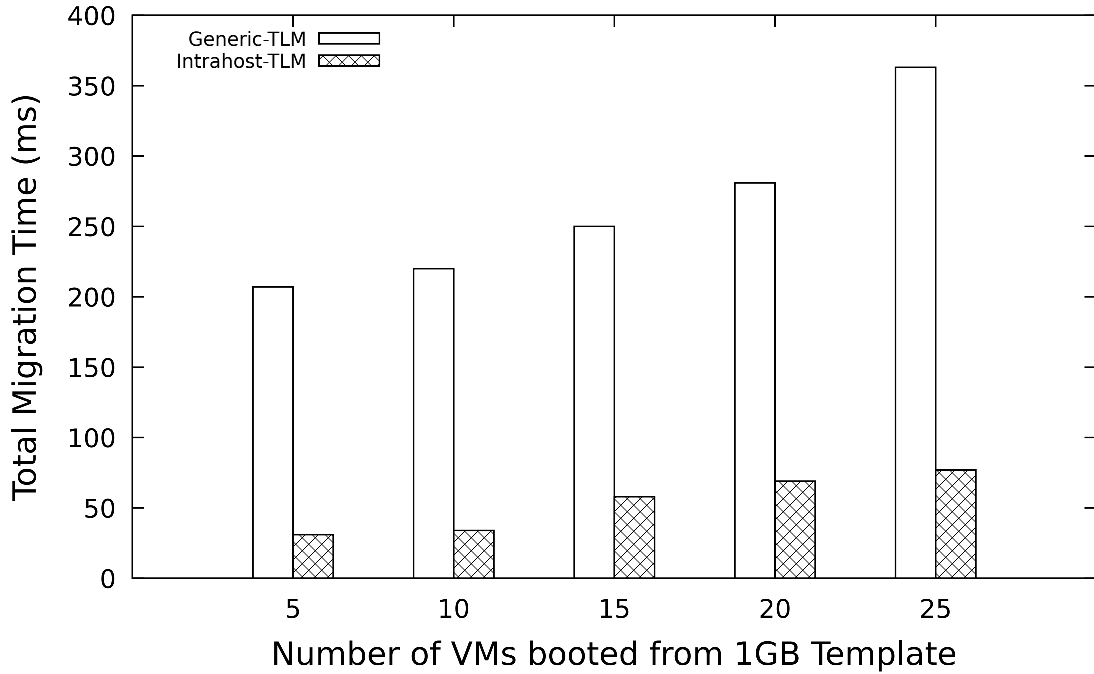


Figure 4.7: Total Migration Time (TMT) of idle multiple templated VMs migrated concurrently using Generic TLM and Intra-host TLM.

each of the 5 VMs ranges from 1MB to 20MB. As shown in Figure 4.6 (a), the memory usage increases in legends (c) and (d) with an increase in the working set size due to the copying of delta pages using Generic TLM. This effect is mitigated when using Intra-host TLM, as illustrated in legends (c) and (d) in Figure 4.6 (b).

4.6.2 Improvement in Total Migration Time

Figure 4.7, 4.8 shows the total migration time of multiple idle and busy templated VMs migrated using Generic TLM and Intra-host TLM. For Figure 4.7, the X-axis indicates the number of idle VMs booted from the same template and for Figure 4.8, the X-axis indicates the size of the dirtying memory block for each of the five templated VMs booted from the same template. The Y-axis shows the total migration time in milliseconds. As shown in Figure 4.7, for Generic TLM, there is an increase in total migration time with an increase in the number of templated VMs. The more the number of VMs the more delta generated by them causing the copying overhead contributing

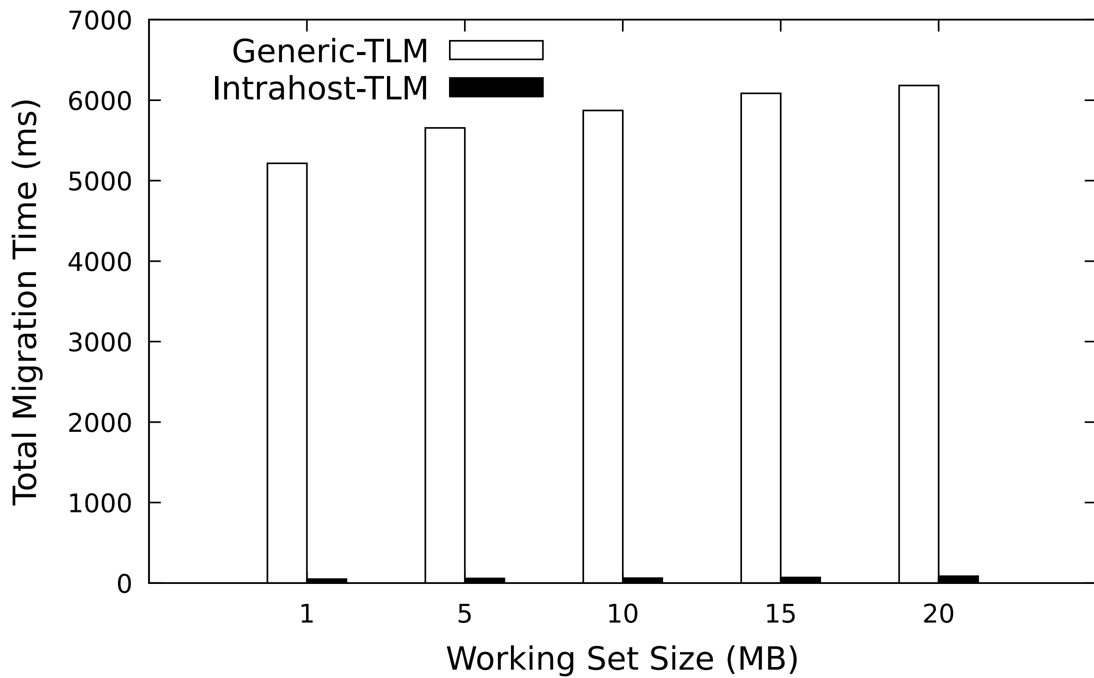


Figure 4.8: Total Migration Time (TMT) of multiple templated VMs running write-intensive workload of varying size in MB migrated concurrently using Generic TLM and Intra-host TLM.

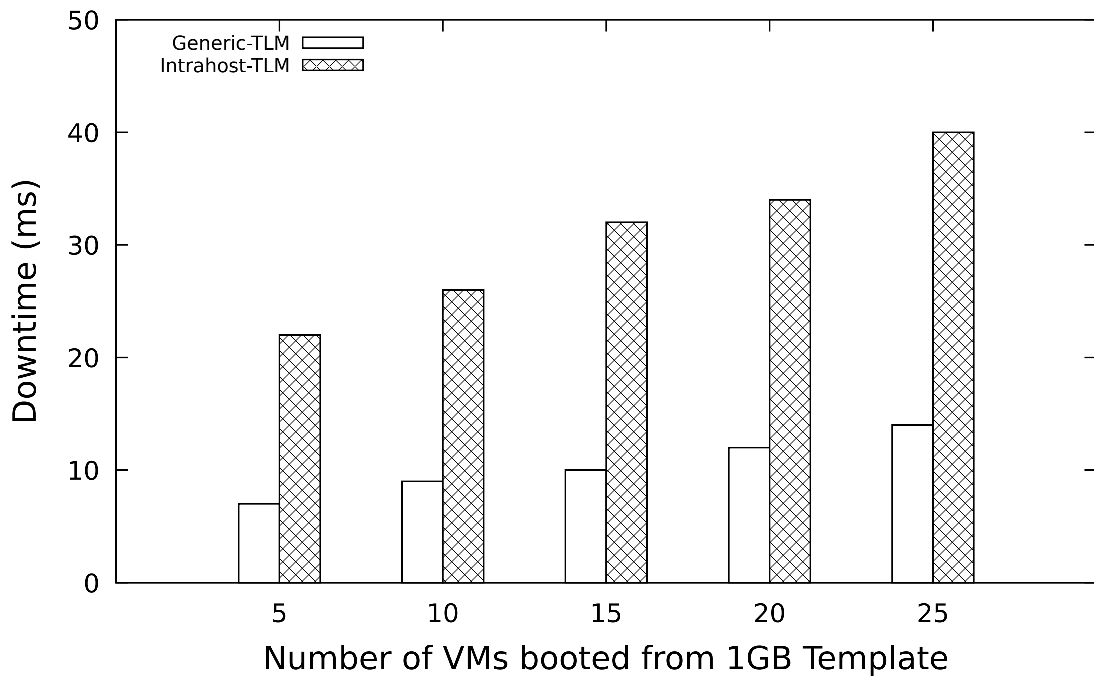


Figure 4.9: DownTime (DT) of multiple idle templated VMs migrated concurrently using Generic TLM and Intra-host TLM.

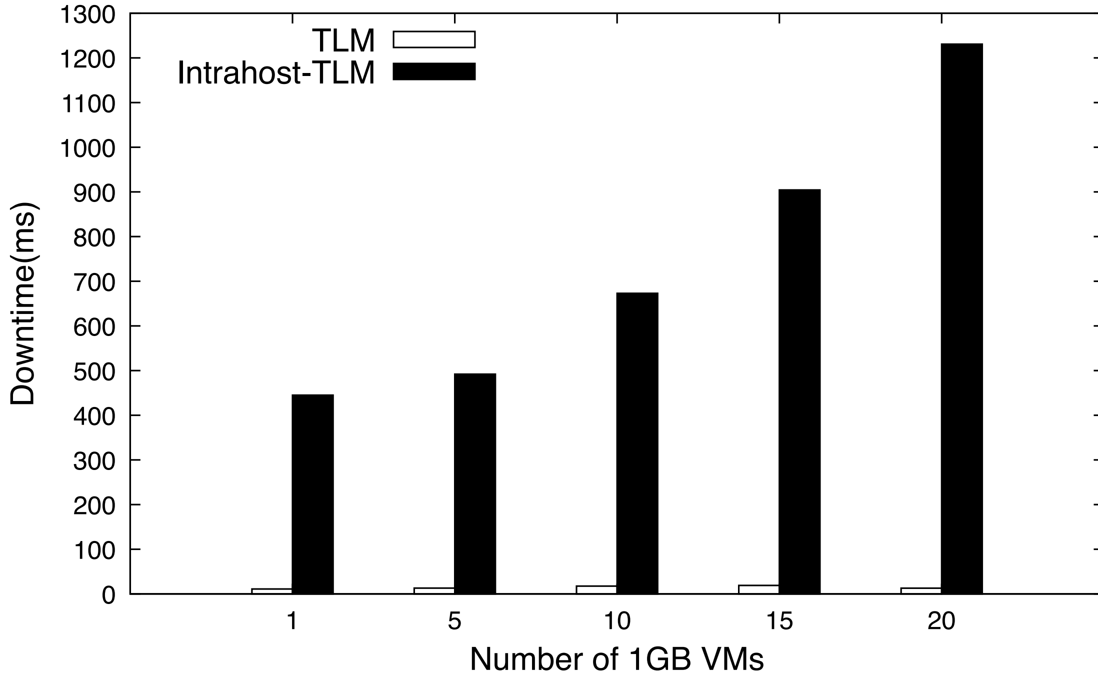


Figure 4.10: DownTime (DT) of multiple templated VMs running write-intensive workload migrated concurrently using Generic TLM and Intra-host TLM.

to the higher total migration time for Generic TLM. Our Intra-host TLM significantly reduced page copying by transferring ownership, resulting in an up to 85% reduction in total migration time. The total migration time remains consistent for all templated VMs regardless of their count until it reaches 11; after that, there is a slight increase in migration time due to CPU contention. For busy VMs, as shown in figure 4.8, both Generic and Intra-host TLM exhibit an increase in total migration time with an increase in the dirty rate size. However, the latter shows a significantly lower migration time due to the elimination of copy overhead.

4.6.3 Reduction in Pages Transferred

Figure 4.12 and 4.13 illustrates the total pages transferred of multiple idle and busy templated VMs migrated using Generic TLM and Intra-host TLM. For idle VMs as shown in Figure 4.12, both Generic and Intra-host TLM exhibit a slight increase in the total pages transferred with an increase in the number of templated VMs. In the case

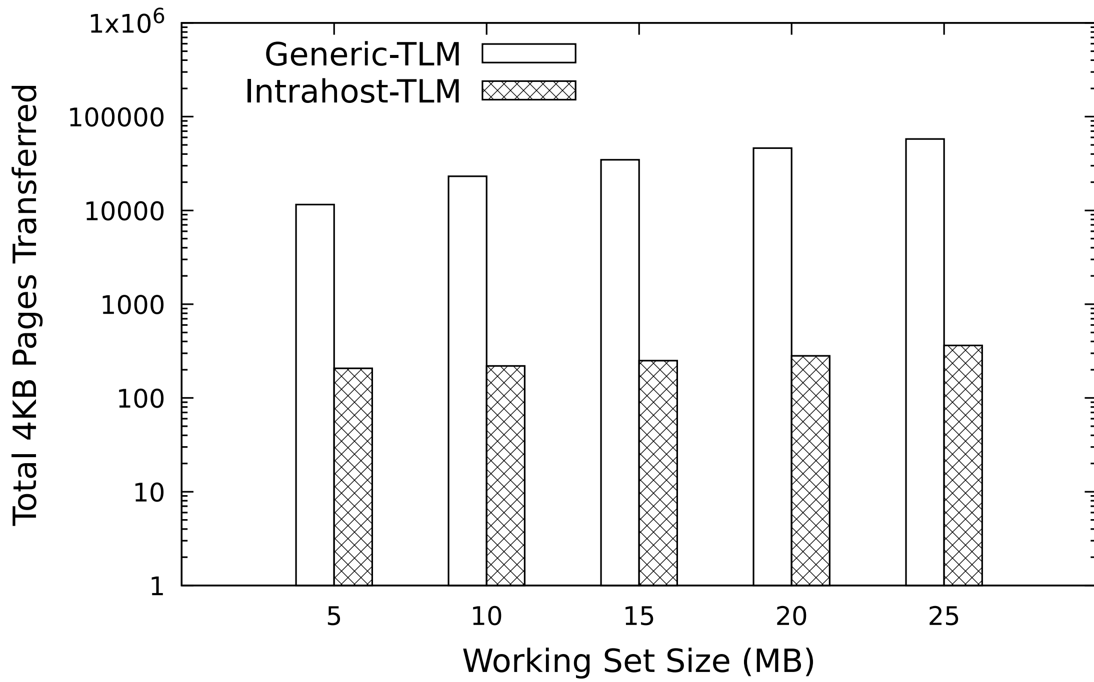


Figure 4.11: Total 4KB pages transferred of multiple templated VMs running write-intensive workload of varying size.

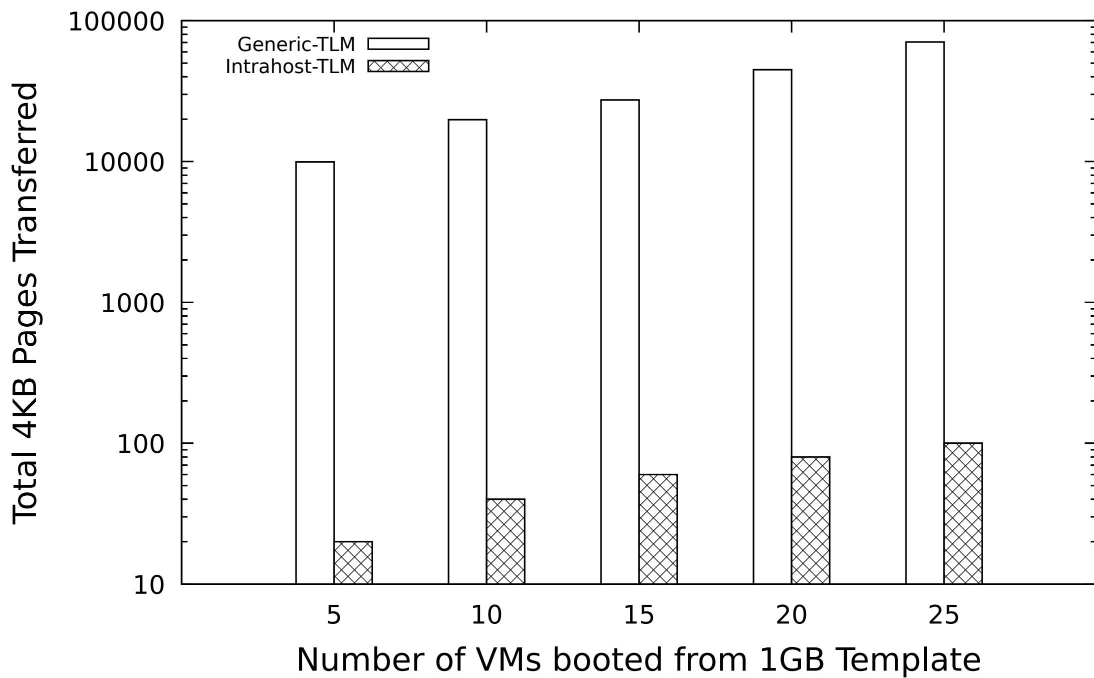


Figure 4.12: Idle Templated VMs

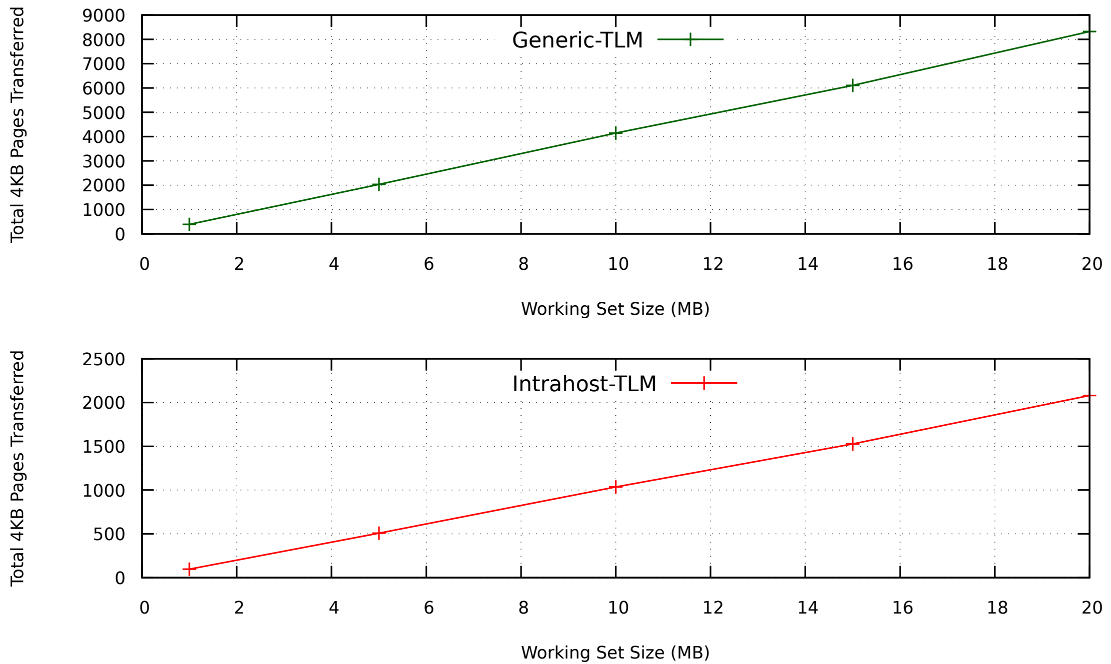


Figure 4.13: Busy Templated VMs

of Generic TLM, this increase is attributed to the duplication of delta pages within the same host. However, despite Intra-host TLM eliminating the unnecessary copying of pages, there is still some metadata information, such as block address, block name, page offsets, mmap offsets, and others, that needs to be transferred, thereby increasing the transferred pages. Four *special* pages from each of the different RAMBlocks cannot be write-protected; therefore, they also need to be copied along with the metadata. Figure 4.13 shows that with an increase in the working set size, the total pages transferred increases for both Generic and Intra-host TLM. For the Intra-host TLM, along with the four special pages and meta-data, the offset of the working set size needed to be transferred as well to the destination to reclaim the ownership of those pages.

4.6.4 Effect on Downtime

Figure 4.9, 4.10 shows the downtime of multiple idle and busy Templated VMs migrated using Generic and Intra-host TLM. Though there is unpredictability in the

downtime of Generic TLM [17], a more stable value can be obtained using a lower threshold value, we have set the threshold value as 2MB to trigger the downtime. For both idle and busy templated VMs, Generic TLM experiences minimal downtime due to the transfer of only a small number of delta pages and essential system state information such as I/O, VCPU, and device states. Conversely, in the case of idle VMs, Intra-host TLM exhibits a consistent downtime of approximately 22ms, regardless of the number of templated VMs. However, for busy VMs, the downtime increases proportionally with the rising memory dirty rate as this necessitates more pages requiring ownership changes, thus resulting in the transfer of additional offsets. This is likely due to an inefficient implementation of our offset transfer mechanism, which needs to be addressed in future work.

4.7 Related Work

Mwarp [73] addresses the issue of intra-host migration without copying pages by transferring ownership. However, this solution is container-specific and cannot be applied to VMs. Several live patching works related to hypervisors [86, 13, 3] focus on replacing old unstable hypervisors with new stable versions without disrupting virtual machines. If the patching only needs to be applied to QEMU and not KVM, then our Intra-host TLM offers a more flexible way to perform live migration within the same host for templated VMs without disrupting the entire host.

4.8 Chapter Summary

In this chapter, we addressed the problem that Generic TLM is inefficient for migrating templated VMs within the same host. We proposed Intra-host TLM that transfers the

ownership of delta pages without copying them when migration is performed within the same host. We designed, implemented, and evaluated Intra-host TLM on QEMU/KVM platform and showed that our technique not only avoid unnecessary copying of delta pages but also significantly reduce the migration time by upto 85% compared to Generic TLM.

5 Sharing-aware Live Migration

In this chapter, we continue our focus on the intersection of two essential techniques for managing co-located VMs: live migration and copy-on-write (COW) page sharing. As mentioned earlier, live migration [8, 28, 26] is a key technology in data centers that transfers running VMs from one physical machine to another. It is widely used for a variety of purposes, such as load balancing [4, 30, 70], meeting service level agreements [56], energy savings [77], and seamless maintenance of physical servers. As in Chapter 3, we reconsider the problem that co-located VMs may often need to be migrated to the same destination machine for various reasons.

5.1 Problem Statement

As mentioned in Section 1.1, COW page sharing reduces the collective memory footprint by sharing identical pages among co-located VMs, whenever doing so is feasible and safe. In Chapter 3, we presented the TLM technique, which performs maintains sharing of pages with the base template image when template instances are migrated together to another destination machine. However TLM does not handle shared pages besides those due to templating. For instance, deduplication mechanisms, such as KSM can share pages among unrelated co-located VMs that do not necessarily share a template base image. Hence there is a need for a more inclusive technique for sharing-aware live migration that can handle all types of page sharing, irrespective of the underlying

sharing mechanism.

5.2 Contributions

In this chapter, we address the general problem of preserving all pre-existing page sharings among multiple co-located VMs as they are live migrated together to a common destination machine. Our goal is to prevent the expansion of VMs' memory footprint at the destination for both pre-copy and post-copy live migration, for all types of VMs, irrespective of the underlying page sharing mechanisms. The contributions of this work are as follows:

1. We identify and demonstrate the problem of memory footprint expansion caused by traditional live migration techniques, specifically both pre-copy [8] and post-copy [28]. This expansion occurs because these techniques lack awareness of pre-existing COW page sharing among co-located VMs, both within and across VMs.

2. We then present a more general *Sharing-aware Live Mirgation* (SLM), which identifies and preserves all types of pre-existing page during live migration and works with any existing memory sharing techniques such as KSM, VM templating, or others.

3. We implement and evaluate SLM for both pre-copy and post-copy migration in the KVM/QEMU [38] virtualization platform using several workloads and microbenchmarks. Besides preserving all pre-existing page sharings at the destination machine, SLM reduces the total migration time by up to 59% and network traffic by up to 62%.

While not the focus of this chapter, we note that side-channels [27, 44, 9, 83] might exploit memory sharing among mutually untrusting VMs and solutions exist to mitigate these risks [58, 39]. This chapter assumes that appropriate mitigation strategies are deployed when page sharing is used, such as by sharing pages only among mutually trusting VMs [58]. Further, memory being a bottleneck resource, safe page sharing among mutually trusting VMs is important to retain consolidation and multiplexing benefits of virtualization. Finally, while we use the KVM/QEMU platform to demonstrate our techniques in this chapter, the core conceptual ideas of our solution are applicable to other hypervisors as well.

In the rest of this chapter, we demonstrate the problem of memory footprint expansion during live migration for pre-copy and post-copy. Next we present the design and implementation SLM followed by its evaluation. The chapter concludes with a discussion of related work and summary of contributions.

5.3 Problem Demonstration

To experimentally demonstrate this problem, we measured the memory footprint of multiple concurrent 1GB VMs at both the source and destination machines after migrating the VMs using traditional pre-copy and post-copy techniques. KSM is used at the source machine to deduplicate the memory of co-located VMs before migration begins, thus establishing preexisting COW-shared pages across VMs. The actual memory usage of each VM in this experiment is smaller than their maximum 1GB permitted since the VMs are not yet using their full allocation. Figure 5.1 shows that both pre-copy and post-copy, which are unaware of existing COW-shared pages among VMs, result in a larger memory footprint at the destination than at the source after live migration

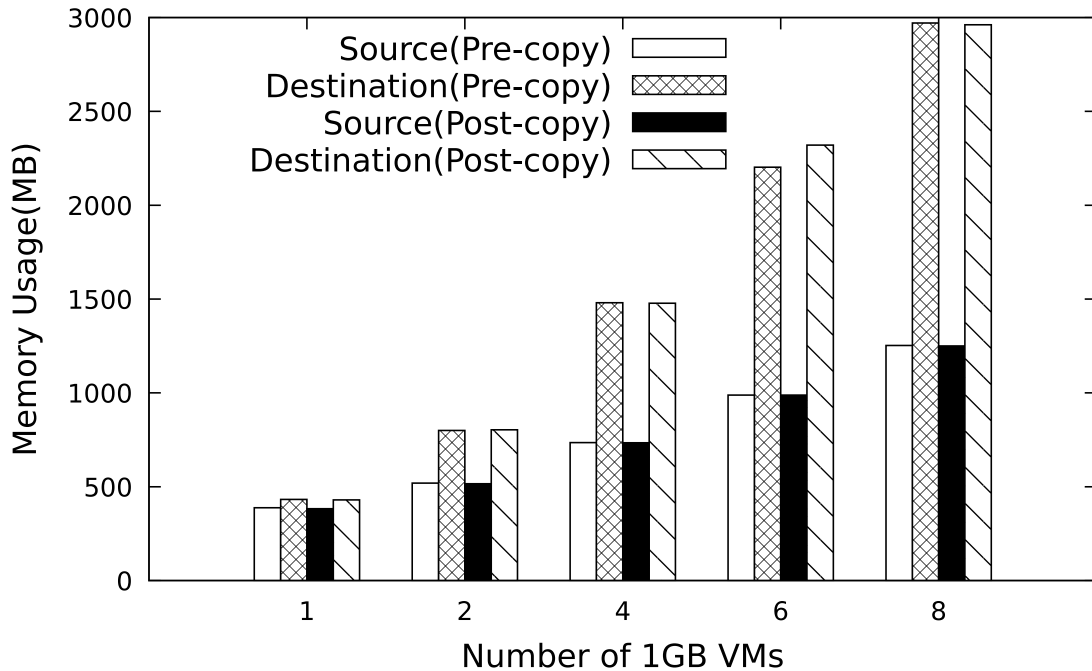


Figure 5.1: Memory footprint of VMs expands at destination after both pre-copy and post-copy live migration, because pages shared among VMs at the source are replicated for each VM at the destination.

completes.

While one can deduplicate again (say, using KSM) at the destination machine to reestablish page sharing and reduce the overall memory usage, this can take several minutes to converge depending on how aggressively KSM is configured to scan pages [66]. There is also a more severe possibility that, when migrating multiple VMs to the same destination, some VM migrations might fail due to a temporary lack of memory at the destination. However, sufficient memory exists if pre-existing page sharings from the source were faithfully reproduced at the destination during migration. Even if the destination has enough memory and the migration succeeds, it will cause higher memory pressure at the destination, more network traffic, and longer total migration time.

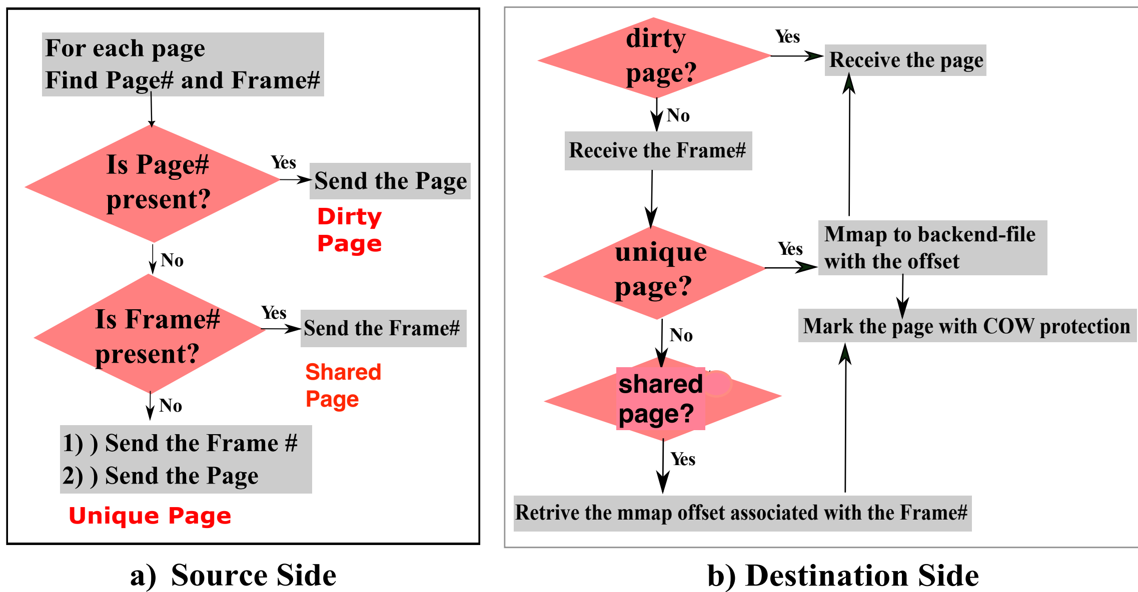


Figure 5.2: SLM Algorithm

5.4 SLM Design

We now present the design of a more general SLM technique, which preserves all pre-existing COW page sharings among co-located VMs being migrated concurrently. SLM is designed to operate effectively with both pre-copy and post-copy algorithms. The key insight behind SLM is that irrespective of the underlying page-sharing mechanism (such as KSM, VM templating, or others), multiple COW-mapped guest pages will map to the same physical page [74]. As shown in Figure 5.2 SLM examines the physical address of each guest page being transferred, identifies COW-mapped shared pages at the source node, and avoids transmitting them multiple times to the destination. Instead, such shared pages at the source node before migration are COW mapped to the same physical page at the destination node.

The traditional pre-copy migration transfers the memory pages of a VM over several rounds, where the initial round transfers the entire memory of the VM, while the subsequent rounds only transfer pages that the VM has dirtied (i.e., written to) in the

Algorithm 1 SLM: Source

Input:

- N is the total number of pages in a VM.
- $page[N]$ is the array of all pages.
- $vpn_list[N]$ is the array of Virtual Page Numbers (VPN).
- $pfn_list[N]$ is the array of Page Frame Numbers (PFN).

```
1:  function MIGRATE(VM)                                  ▷ Source
2:      for i ← 1 to N do
3:          Find VPN and PFN of page[i]
4:          if vpn not in vpn_list then
5:              Append vpn to vpn_list
6:              if pfn not present in pfn_list then
7:                  Append pfn to pfn_list                ▷ Unique page
8:                  Send pfn of page[i]
9:                  Send page[i]
10:             else                                      ▷ Shared page
11:                 Send pfn of page[i]
12:             end if
13:         else                                          ▷ Dirty page
14:             Send page[i]
15:         end if
16:     end for
17: end function
```

previous rounds. This dirtying operation during live migration may break pre-existing COW mappings at the source. SLM for pre-copy is designed to detect when such COW mappings break at the source across multiple pre-copy rounds and to disassociate the corresponding COW-mapped pages at the destination. On the other hand, post-copy migration transfers each page only once and, since the VM executes at the destination, there are no dirtied pages at the source to retransmit.

As shown in Algorithm 1 and 2 Figure 5.3, SLM operates on both the source and the destination nodes. At the source node, SLM classifies the type of each page (Unique, Shared, or Dirty) and transfers them to the destination according to their type. At the destination, SLM receives each page's information and maps it accordingly into the

Algorithm 2 SLM: Destination

Input:

- N is the total number of pages in a VM.
- $page[N]$ is the array of all pages.
- $identical[N]$ is the array of identifiers for pages.
- $offset_list[N]$ is the array of mmap_offsets from a memory-backend-file.

```
1:  function RECEIVE(VM)                                  ▷ Destination
2:      for i ← 1 to N do
3:          mmap_offset ← 0
4:          Receive identifier for page[i]
5:          if identical[i] = 0 then                      ▷ Unique page
6:              Append mmap_offset to offset_list
7:              Mmap page[i] with mmap_offset
8:              Receive page[i] from the network
9:              Increment mmap_offset
10:             COW-protect page[i]
11:         else if identical[i] = 1 then                 ▷ Shared page
12:             Retrieve mmap_offset from offset_list
13:             Mmap page[i] with mmap_offset
14:             COW-protect page[i]
15:         else                                          ▷ Dirty page
16:             Receive page[i] from the network
17:         end if
18:     end for
19: end function
```

VMs' memory. We describe these steps at the source and destination in more detail below.

5.4.1 Identifying Page Type at Source

As illustrated in Algorithm 1 and Figure 5.3, SLM follows a two-step process for each page transfer. In the first step, SLM determines the page's physical frame number (PFN) and the virtual page number (VPN). This information is stored in a hash table for efficient lookup during subsequent transfers. In the second step, SLM categorizes pages into one of the three types, as outlined in Table 5.1, based on the presence or absence of the PFN and VPN in the hash table.

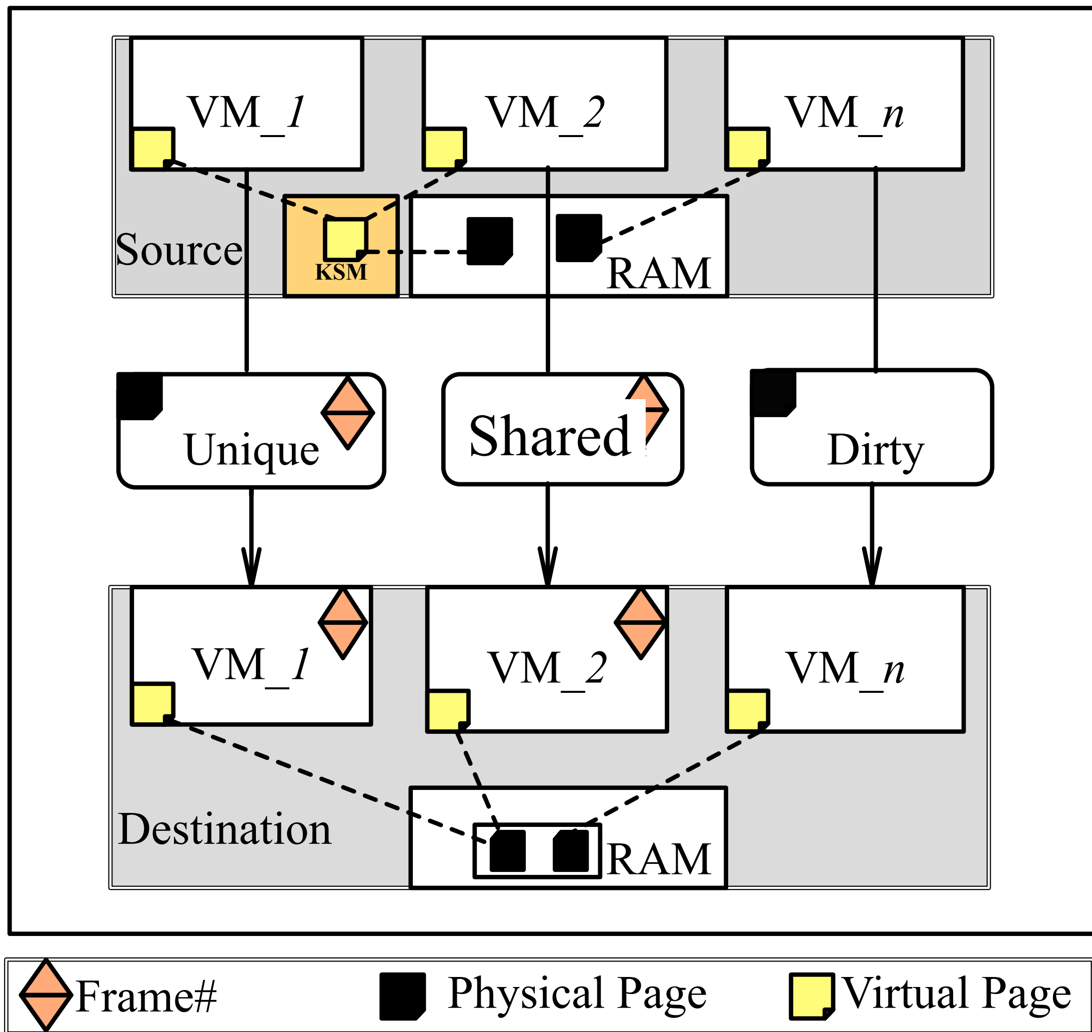


Figure 5.3: SLM classifies pages of VMs at the source as Unique, Shared, and Dirty. Shared pages are not re-transmitted; instead Destination COW-maps them into a common in-memory backend file.

PFN	VPN	Page Type
✗	✗	Unique
✓	✗	Shared
✓/✗	✓	Dirty

Table 5.1: Determining page type using PFN and VPN

1. *Unique Page:* Pages that have not been transferred yet are considered Unique. In this scenario, the corresponding PFN and VPN are not present in the hash table. Thus, both the PFN and VPN are inserted into the hash table before sending the page content.

2. *Shared Page:* Pages that have already been transferred are considered Shared. In this case, the PFN is present in the hash table, but the VPN is not. Therefore, only the VPN is inserted into the hash table before sending the PFN.

3. *Dirty Page:* Pages that require retransmission due to being dirty in the previous pre-copy round are referred to as Dirty pages. The PFN may or may not be present in the hash table, but the VPN is present. In this case, only the PFN is inserted into the hash table before sending the page content. Since post-copy only transfers pages once, this type of page doesn't exist for SLM post-copy.

For Unique and Dirty pages, SLM transfers the entire page, including the page type and its PFN at the source, as a unique identifier. However, for Shared pages, SLM does not send the page content but only the page type and the PFN.

5.4.2 Preserving COW Sharing at Destination

SLM at the destination node works as shown in Algorithm 2. At its core, the algorithm operates in two ways depending on whether a full page or only a source PFN is received from the source node. If a full page is received, it is copied into the corresponding VM's memory, and the source PFN is recorded for future reference. On the other hand, if only the source PFN is received, the corresponding virtual page in the VM is mapped to the previously received physical page with the same source PFN.

To facilitate COW sharing at the destination, we set up an in-memory backend-file into which each received page is memory mapped using the `mmap` system call [46]. There are two flags of importance, `MAP_SHARED` and `MAP_PRIVATE`. The `MAP_SHARED` flag causes any writes to a mapped virtual address to be written back to the backend-file. On the other hand, `MAP_PRIVATE` results in COW mapping, meaning that any writes to the mapped virtual address result in the allocation of a new private page to the process before the write is committed, ensuring that the write is not transmitted to the backend-file.

For SLM pre-copy, when a Unique page is received, the entire backend-file is initially configured with the `MAP_SHARED` flag. The received page is then written to the backend-file, and the `mmap` configuration for that page is changed to `MAP_PRIVATE` to enable COW mapping for any future transmissions of the same Shared page. The received PFN and its corresponding mmap offset are also recorded in a hashtable. When a Shared page is received, SLM retrieves the corresponding mmap offset from the hashtable using the received PFN and maps the virtual address to the backend-file using the `MAP_PRIVATE` flag. If a Dirty page is received, SLM skips any `mmap` operations and copies the entire page content directly from the network into the VM's address space.

In SLM post-copy, the migration thread is tasked with copying page content from the network socket, whether received through active-pushing or demand-paging. This algorithm operates in three stages: (1) For a Unique page type, the migration thread directly copies the temporary page to the backend-file and records the received PFN and its corresponding mmap offset in the hash table. For Shared page types, the migration thread retrieves the mmap offset from the hash table using the received PFN. (2) The migration thread maps the virtual address to the backend-file using the `mmap` system call

and configures the page as `MAP_PRIVATE`. (3) Finally, if a VCPU accessing this page was suspended due to a page fault, the migration thread wakes it up. At the destination, if the VMs introduce any new duplicated pages in the future, KSM continues to deduplicate them.

5.5 Implementation

5.5.1 Retrieval and Tracking of PFN

All virtual pages mapped to a shared physical page must have the same PFN, irrespective of which sharing mechanism generates such mapping. QEMU is a user-level management process whose address space has specific regions reserved for guest memory [38]. To get the guest's physical address of a page (VPN), we could directly access the addresses that are part of the reserved region.

The Linux kernel exposes page table information to userspace using `/proc/pid/pagemap` [76]. With this file, a userspace process can find the PFN for a specific VPN. Each entry in the pagemap contains 64-bit information indexed by the VPN, with the first 56 bits indicating the PFN. SLM takes advantage of the pagemap in the pseudo file system to retrieve accurate PFNs for each VPN, enabling the determination of the page type. SLM uses hashtable at the source for page type and at destination for $PFN \rightarrow mmap_offset$ mappings. A new mmap offset entry is inserted into the table using PFN as a key every time a Unique page arrives since they are guaranteed to have new page content. Whenever a Shared page arrives, SLM looks up the hashtable to retrieve the corresponding mmap offset using its PFN as the key and maps the VPN to the respective mmap offset with COW protection. Finally, SLM skips the lookup during the arrival of Dirty pages.

5.5.2 In-Memory Backend File

The destination node prepares an in-memory backend file into which the content of each Unique page is memory mapped. Subsequent Shared pages are mapped COW from this backend file. The backend file is stored in the main memory using the `tmpfs` file system to avoid the latency of accessing the virtual disk. Sometimes the OS uses lazy allocation when creating a backend file of large size. Doing so can result in bus errors when unallocated file regions are memory mapped to the virtual address for unique pages. We eliminated the problem by writing zeros into the backend file before mapping the area for storing unique page contents. By default, each VM has a limit on the number of `mmap` regions of the file that it can access; we increase the limit by adjusting the `vm.max_map_count` configuration parameter in `/etc/sysctl.conf`. In our prototype, the size of each `mmap` region in the backend file is the same as the page size of 4KB.

5.5.3 Synchronization Across Multiple VMs

One synchronization challenge we encountered relates to the order of arrival of Unique and Shared page information. In an ideal scenario, for a given PFN, a Unique page (comprising both its page content and PFN) should arrive at the destination before any Shared page (containing only the PFN). But, during the migration of multiple VMs, there are instances where the PFN for a Shared page arrives for a VM (e.g., VMx) before the Unique page content for the corresponding PFN arrives for another VM (e.g., VMy). However, the Shared page cannot be COW-mapped until the Unique page is mapped and its content is written into the backend-file.

For SLM post-copy, when a Shared page information arrives before its Unique page,

QEMU for VMx busy waits, anticipating the arrival of the Unique page with the expectation that the waiting time will be short. When the Unique page with page content for VMy arrives, QEMU for VMx proceeds to COW map the Shared page.

For SLM pre-copy, VMy's page content may change in subsequent pre-copy rounds. We update all pending Shared pages in VMx that depend on VMy at the end of each pre-copy round through busy waiting to prevent stale mapping entries. Busy waiting at the end of the last pre-copy round, just before downtime, can extend VM downtime. To mitigate this latency, SLM includes an additional live pre-copy round without busy waiting before the downtime phase. This issue doesn't arise with SLM post-copy since pages are sent only once in post-copy.

5.6 Evaluation

We now evaluate the performance of SLM against traditional pre-copy and post-copy live migration methods. Our experimental setup consists of three machines, each equipped with two Intel Xeon E5-2620 v2 processors and 128GB of DRAM, running Ubuntu. We implemented SLM versions of pre-copy and post-copy in the KVM/QEMU [38] virtualization platform on Linux. We modified QEMU's default pre-copy and post-copy algorithms with no changes to the guest operating system in the VMs. We used VMs of varying sizes, ranging from 1GB to 32GB, as needed. Each experiment was repeated at least five times to calculate average values.

We demonstrate that migration of VMs using SLM reduces the total migration time and network traffic, besides maintaining memory footprint at the destination and application performance during migration.

Throughout the evaluation, we use the term *generic* to refer to the traditional ver-

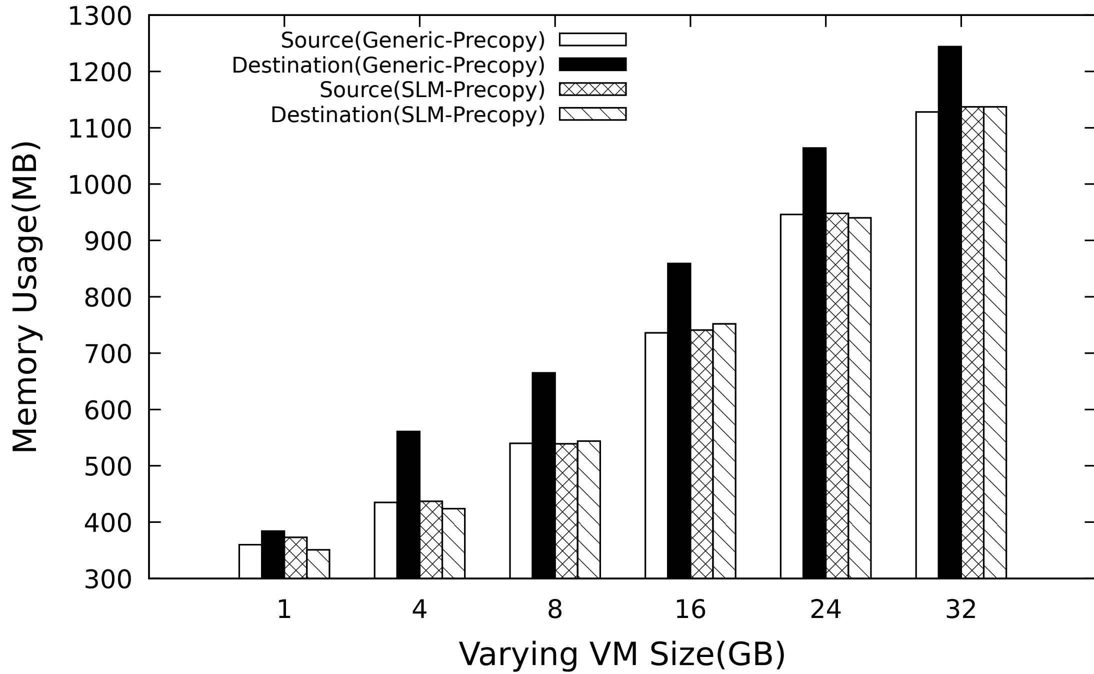


Figure 5.4: Memory footprint of virtual machine of varying size migrated using Generic and SLM pre-copy and post-copy.

sions of pre-copy and post-copy. In our experiments with KSM, we initiate live migration only after allowing the KSM daemon [16] to run for a sufficient period of time, ensuring that total memory usage has stabilized. This stabilization is confirmed by monitoring the output of the `free` command in the host system. Doing this ensures that our results capture all COW-shared pages among VMs.

5.6.1 Live Migration of Single VM

Figures 5.4and 5.5 depict the memory usage of a single VM of varying sizes during migration using the Generic, SLM pre-copy and post-copy methods. Our analysis reveals that as the VM size increases, SLM approach effectively preserves the memory footprint at the destination, mirroring that of the source. Additionally, our experiment demonstrates that there is minimal sharing of pages in the case of a single VM. SLM provides more improvements when handling multiple VMs.

Figure 5.6 illustrates the total migration time for single VMs of varying sizes, uti-

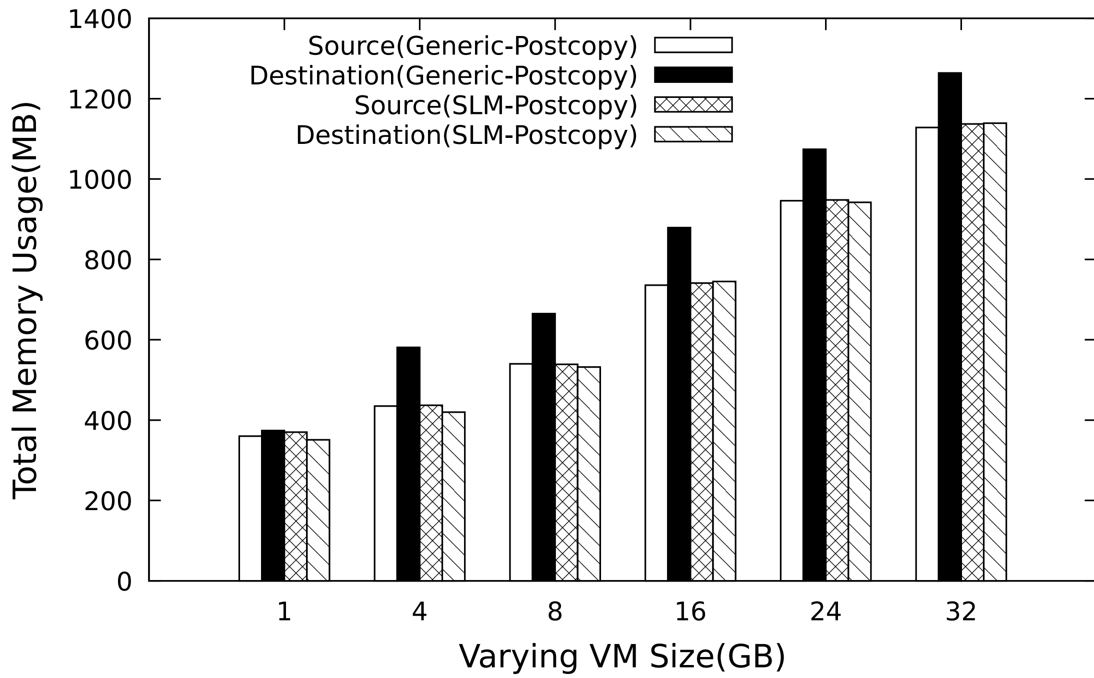


Figure 5.5: Memory footprint of virtual machine of varying size migrated using Generic and SLM post-copy.

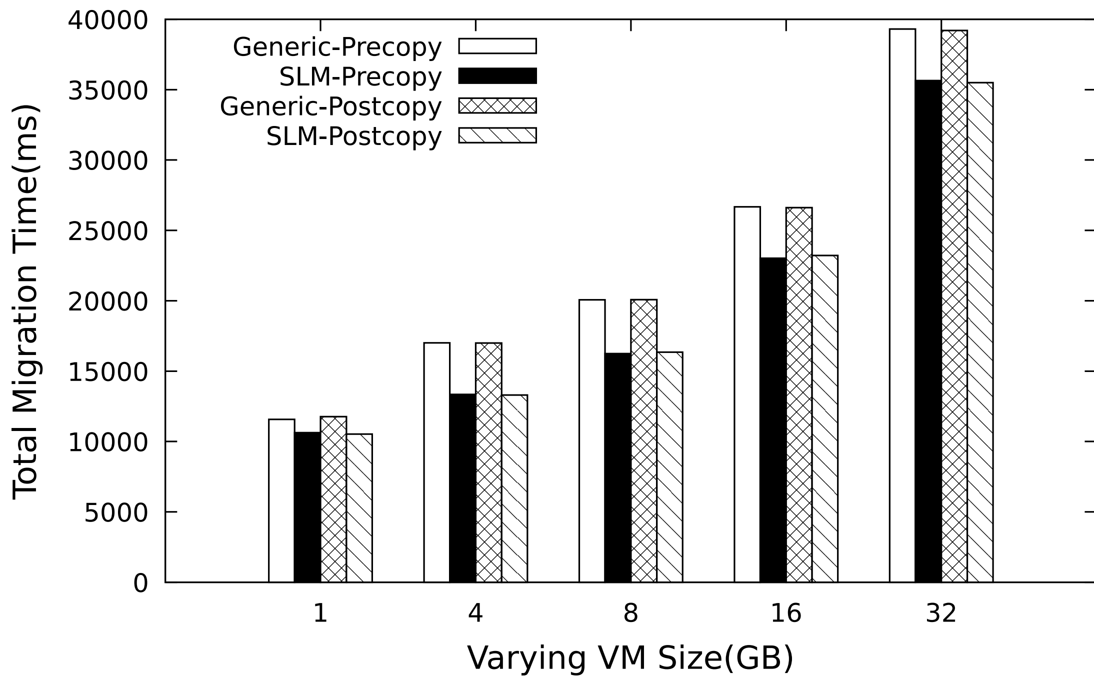


Figure 5.6: Total migration time of single VM of varying size migrated using generic, SLM pre-copy and post-copy.

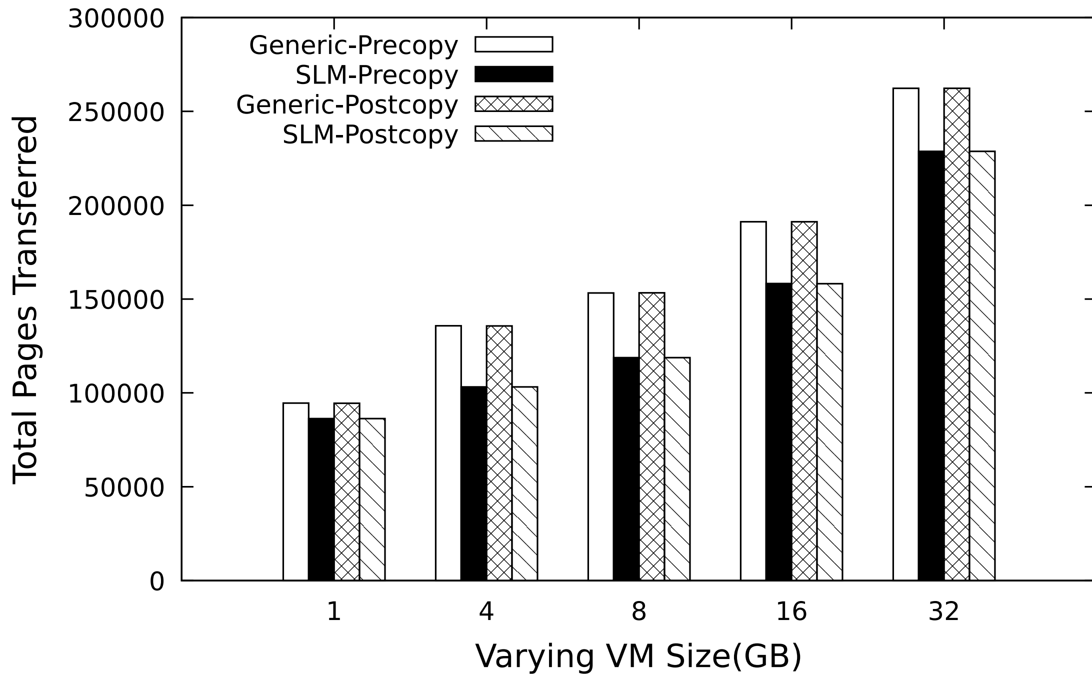
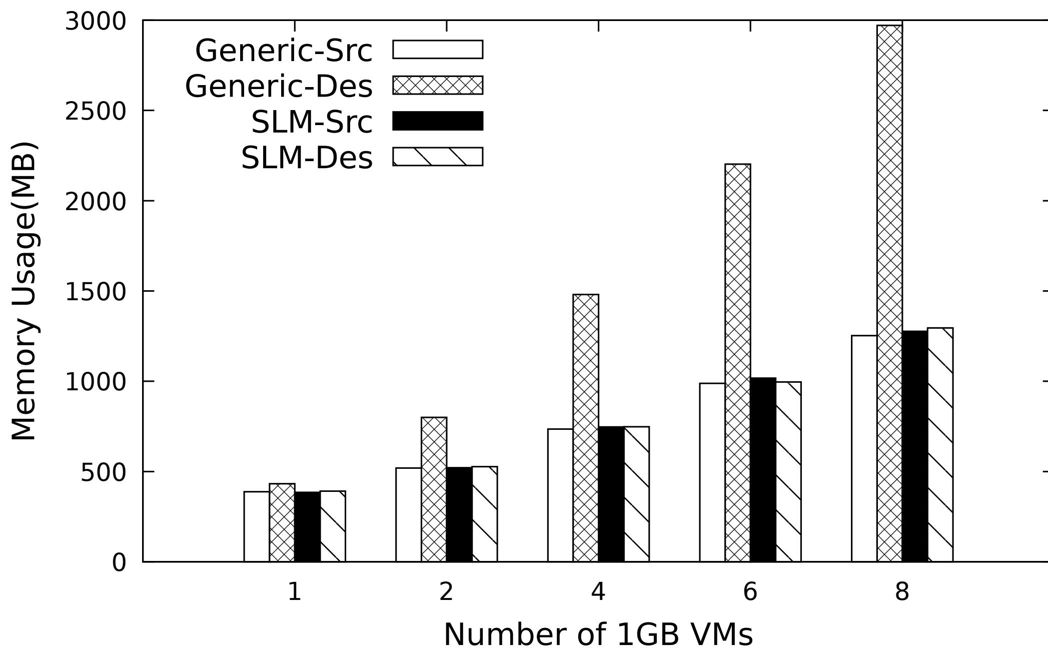


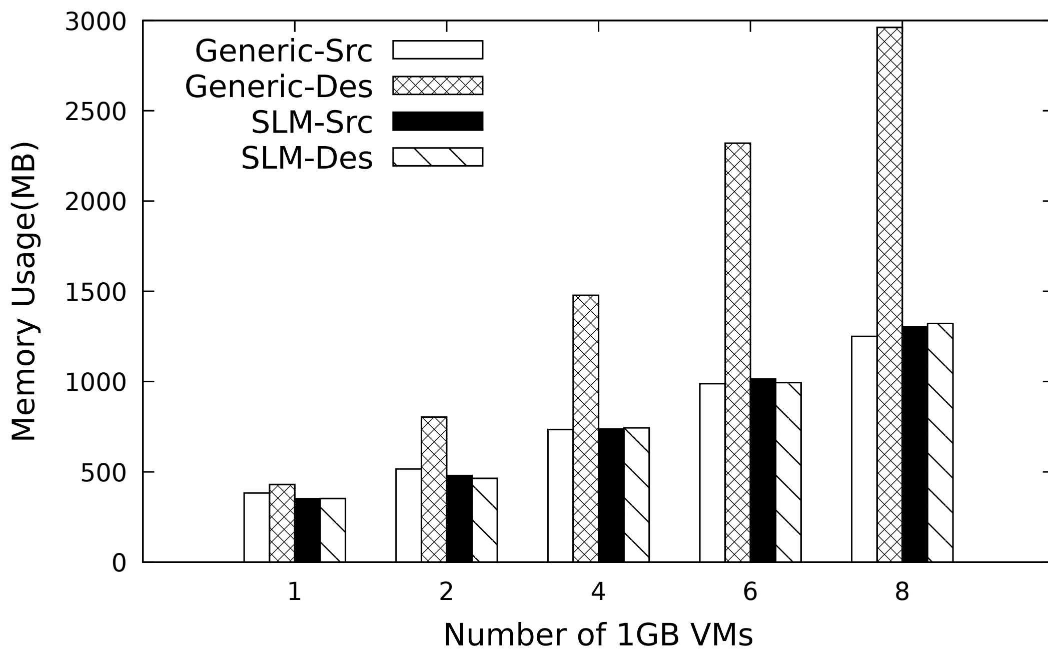
Figure 5.7: Total pages transferred of single VM migrated using generic, SLM pre-copy and post-copy

lizing both the Generic, SLM pre-copy and post-copy methods. Our experiments reveal that as VM size increases, the migration time for single VMs also increases. However, SLM pre-copy and post-copy effectively preserved shared pages, resulting in a reduction in migration time.

Figure 5.7 presents data on the total pages transferred for single VMs of various sizes when employing both the Generic, SLM pre-copy and post-copy methods. The page size was consistently set to 4KB, and the hugepages configuration was disabled throughout all experiments. Our analysis reveals that while the total pages transferred increase with VM size, our SLM technique skips the transmission of shared pages, resulting in a relatively lower volume of pages transferred compared to the Generic method.



(a) Pre-copy



(b) Post-copy

Figure 5.8: Comparison of memory footprint at source vs. destination when multiple VMs are migrated concurrently using generic vs. SLM (a) pre-copy and (b) post-copy. We observe a significant increase in memory footprint for generic pre-copy and generic post-copy vs. no significant increase for SLM pre-copy and SLM post-copy.

5.6.2 Memory Footprint of VMs After Migration

Figures 5.8(a) and (b) compare the memory footprint at source vs. destination when multiple VMs are migrated concurrently using generic vs. SLM techniques for pre-copy and post-copy. The X-axis indicates the number of concurrent 1GB VMs, and the Y-axis displays their collective memory footprint. Generic live migration, despite reducing memory usage at the source through KSM, leads to a significantly expanded memory footprint at the destination because the live migration mechanism is unaware of COW-shared pages among VMs at the source. In contrast, SLM preserves any pre-existing COW page mappings at the destination for both pre-copy and post-copy, resulting in no significant memory expansion at the destination. For SLM, slight differences in memory footprint between source and destination are due to differences in memory usage of the QEMU process associated with VM.

5.6.3 TMT, Downtime, and Network Traffic Reduction

In this section, we evaluate the performance of concurrently migrating multiple idle VMs between two hosts in terms of TMT, downtime, and network traffic reduction. We increase the number of VMs keeping the memory size of each VM constant at 1GB.

TMT is compared between generic and SLM versions of pre-copy and post-copy in Figure 5.9. The X-axis is the number of concurrent 1GB VMs being migrated, and the Y-axis shows the TMT in milliseconds. The results show up to 59% and 57% reduction in TMT for SLM pre-copy and post-copy, respectively, compared to their generic coureduction in is due to SLM eliminating the retransmission ofnterparts. This reduction in is due to SLM eliminating the retransmission of COW-shared pages from source to destination. Figure 5.10 compares the total number of pages transferred during

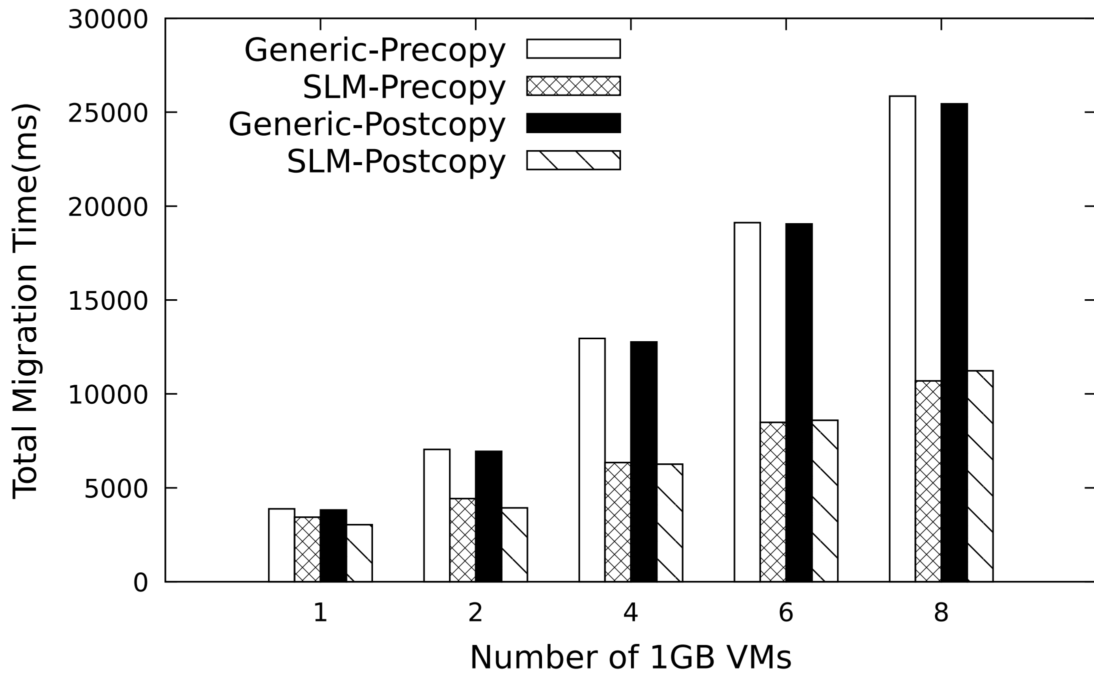


Figure 5.9: Total migration time of multiple VMs concurrently migrated using generic and SLM pre-copy and post-copy.

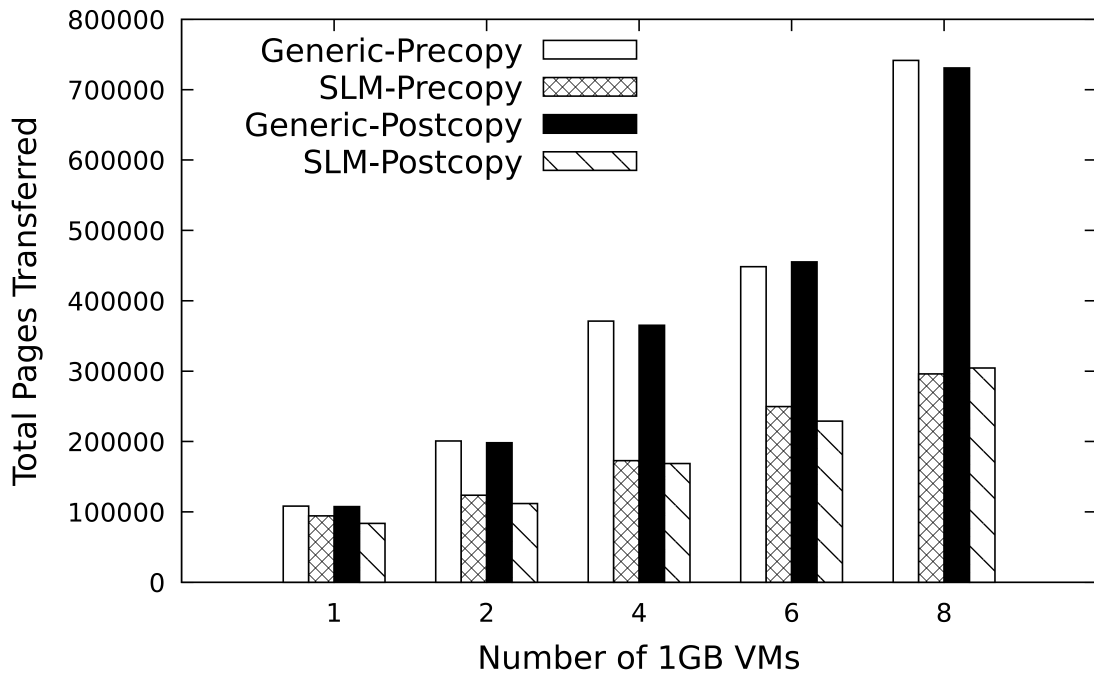


Figure 5.10: Pages transferred during concurrent migration of multiple VMs using generic and SLM pre-copy and post-copy.

generic and SLM pre-copy and post-copy. The experiments show a reduction of up to 60% and 62% in total pages transferred for SLM pre-copy and post-copy, respectively.

We compare downtime of 8 idle 1GB VMs during their concurrent migration using generic and SLM versions of pre-copy and post-copy. The maximum number of pages transferred during downtime is capped at 512 (2MB). The results indicate that VMs experience a comparable average downtime of around 93ms for generic pre-copy and 96ms for SLM pre-copy. Generic and SLM post-copy transfer minimal processor states and non-pageable memory, causing downtime of around 290ms and 300ms respectively. The higher downtime of post-copy for both generic and SLM versions may be attributed to various factors including VCPU thread invocation and demand-paging, leading to more remote page faults at resumption time. Application-observed downtimes for non-idle VMs (discussed later) tend to be higher than these numbers for idle VMs because of network state recovery.

5.6.4 Network Bandwidth Using iPerf

VM migration is a network-intensive procedure that can lead to network contention between the migration process and the applications running inside the VM. To measure the available bandwidth for the VM's applications during migration, we use iPerf [33], a network-intensive application benchmark. An iPerf server is set up on a third machine (i.e., neither the source nor the destination) within the same network, while the iPerf client is run inside the VM being migrated. The client then sends data to the server during migration through a TCP connection. All these machines are connected using a Gigabit Ethernet switch and the link is shared between the host and VM.

Figures 5.11 and 5.12 show network bandwidth measurements during live migration using pre-copy and post-copy techniques. At the beginning of the migration, both

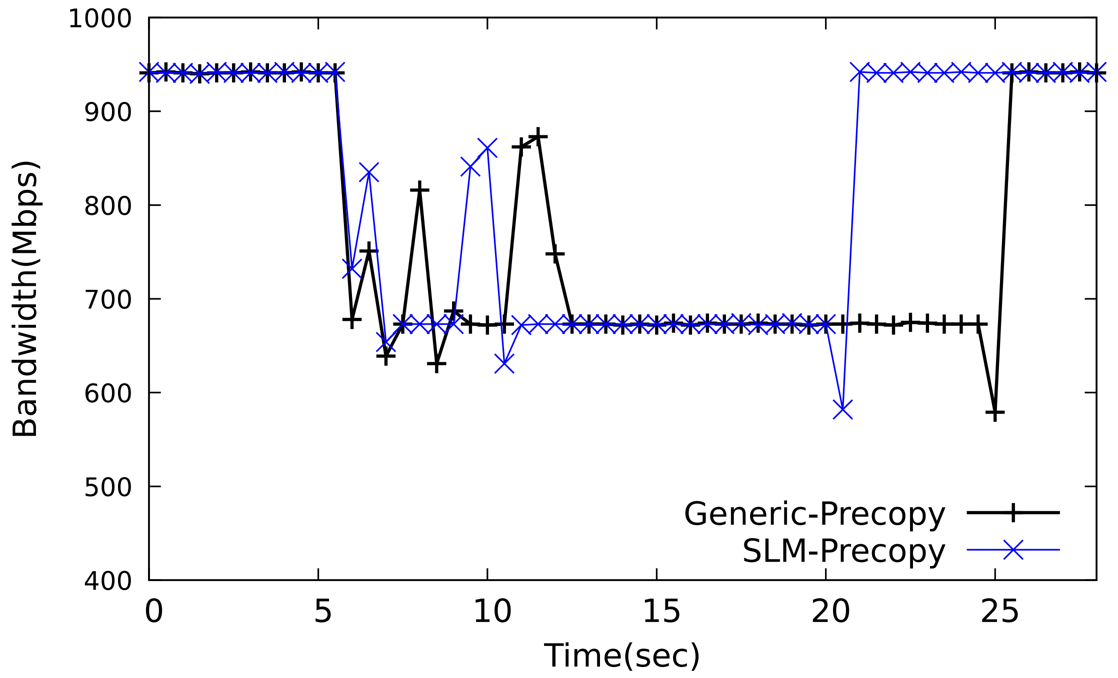


Figure 5.11: iPerf bandwidth for generic and SLM pre-copy.

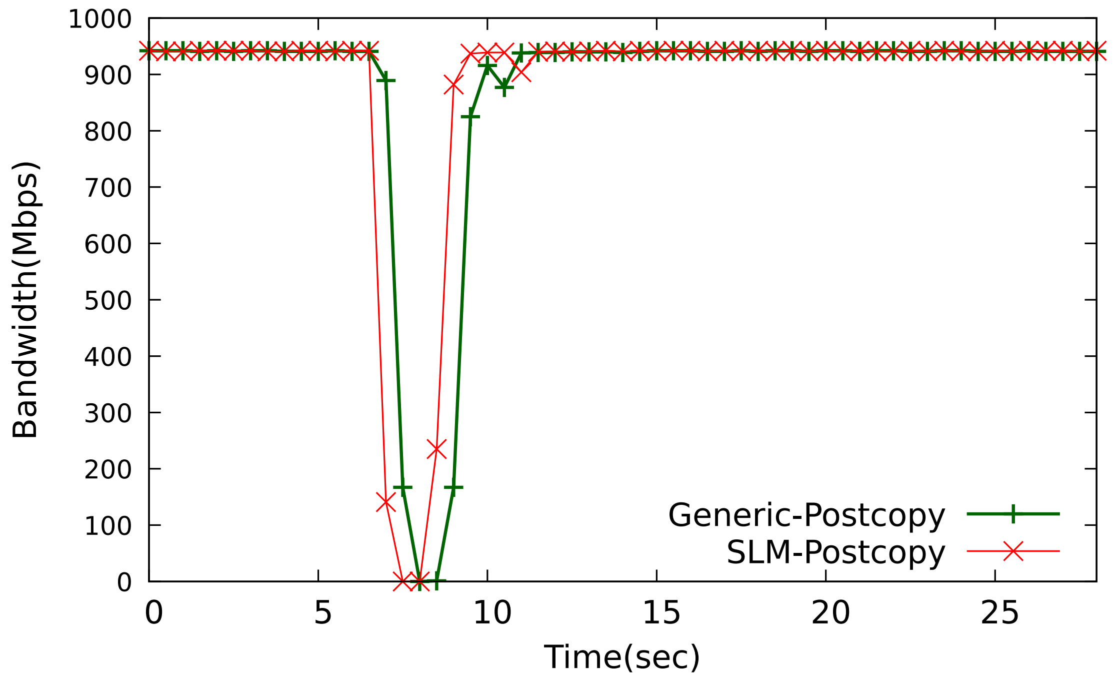


Figure 5.12: iPerf bandwidth for generic and SLM post-copy.

pre-copy techniques experienced a drop in network bandwidth from 940 Mbps to approximately 650 Mbps. However, the decline was more significant for both post-copy versions when the bandwidth dropped close to zero due to downtime. The sudden drop to 670 Mbps in both pre-copy versions is attributed to network contention between the migration thread and the iPerf client running inside the VM. Meanwhile, the fluctuations are a result of QEMU's optimization for zero pages, where only 8 bytes are sent to indicate a zero page instead of the entire page, freeing up bandwidth for iPerf traffic. In contrast, in both versions of post-copy, the fluctuations are due to page faults caused by the post-copy thread resulting in the retrieval of pages from the source via demand-paging and active-pushing of pages from the source to avoid network faults.

SLM pre-copy migration was completed in 14.8s, whereas generic took a longer time, approximately 19.1s. This reduction in TMT for SLM pre-copy is due to a reduced number of pages that needed to be transferred, a consequence of eliminating redundant transfers of COW-shared pages. However, the active-pushing nature of both generic and SLM post-copy techniques, in tandem with demand-paging, aided in the faster recovery of network bandwidth for both post-copy methods. SLM post-copy required approximately 2s, whereas generic took around 3.5s to complete their migration process. They reached full bandwidth faster without significant fluctuations when compared to their pre-copy counterparts. While SLM pre-copy experienced a downtime of approximately 182ms, generic pre-copy exhibited a comparatively lower downtime of around 106ms. This overhead is due to the busy waiting synchronization, as detailed in section III.D. Although both SLM and generic versions of post-copy exhibit a similar downtime of around 500ms due to network state recovery, this duration becomes significant when compared to their pre-copy counterparts. Our SLM technique for both

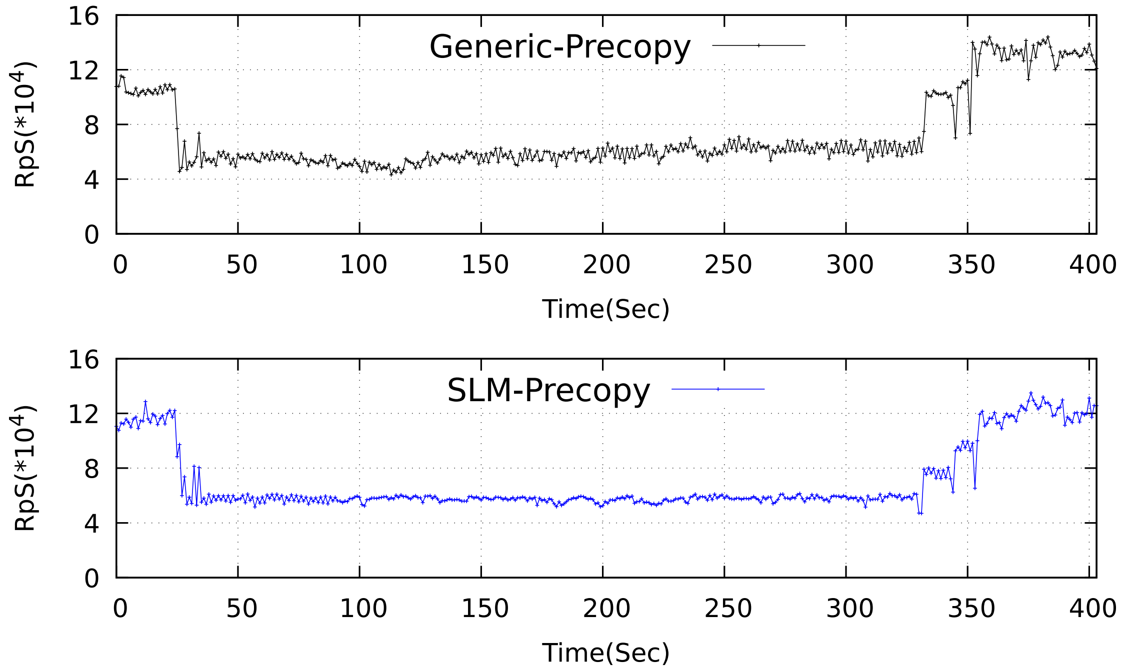


Figure 5.13: Redis-cluster read throughput when migrating 3 VMs using generic and SLM versions of pre-copy.

pre-copy and post-copy doesn't introduce any significant overhead in terms of application performance; in fact, it reduces TMT by eliminating the redundant transfer of shared pages.

5.6.5 Redis Cluster Benchmark

Redis is a real-world in-memory key–value database. Redis cluster is a way to run a Redis server by evenly distributing data across multiple nodes. We set up all three Redis cluster nodes as one Redis cluster server. Each VM is configured with 4GB RAM sharing a gigabit link and running a Redis cluster node instance. The Redis cluster server contains 5 million random key-value data entries, which are evenly distributed across all three nodes. We use Redis-Benchmark [67] to emulate 50 clients sending a `GET` command that randomly reads key-value data from the target Redis cluster server. Our experiments used a Redis pipeline of 16, allowing clients to send concurrent requests without waiting for server responses [49].

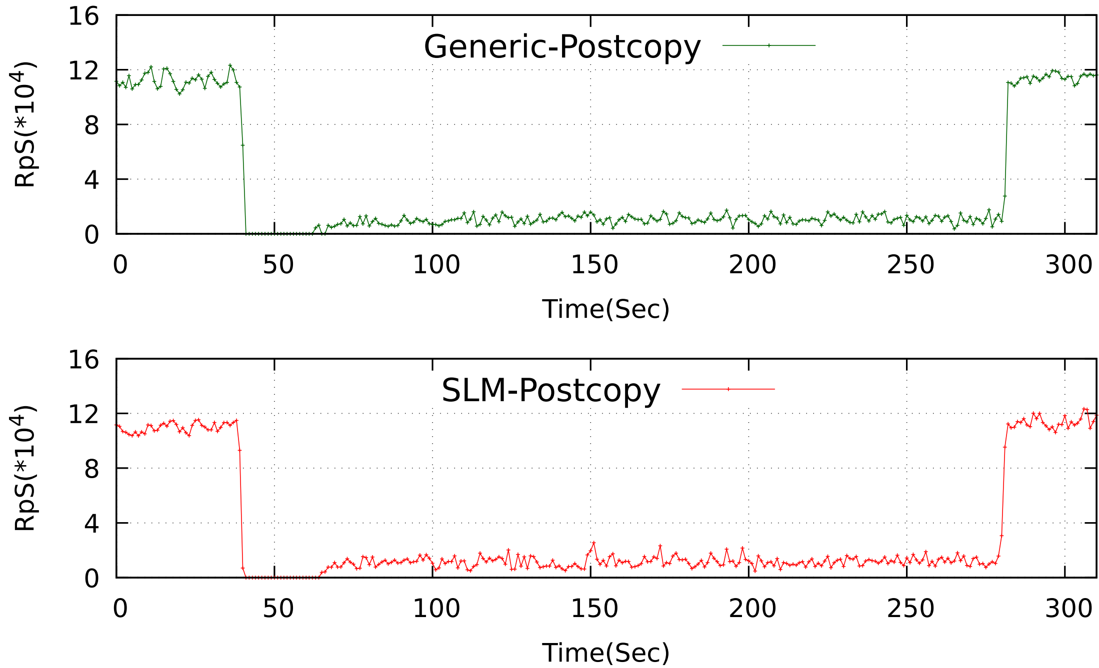


Figure 5.14: Redis-cluster read throughput when migrating 3 VMs using generic and SLM versions of post-copy.

To ensure a fair comparison, we synchronized the start time of migration for Figures 5.13 and 5.14. Initially, Redis demonstrated comparable throughput with both pre-copy and post-copy migration. However, during pre-copy migration, Redis experienced a 40% reduction in throughput due to network contention with migration traffic. Conversely, Redis' throughput during post-copy migration dropped to zero and consistently remained lower than pre-copy, primarily due to downtime and remote page faults which resulted in fetching pages across the network. During the downtime, with pre-copy, there were three brief drops in throughput towards the end of migration whereas. With post-copy, Redis experienced a significant downtime causing a complete disconnection.

In SLM versions of both pre-copy and post-copy, the advantages of COW page sharing are preserved during migration, resulting in a reduced TMT compared to their generic counterparts. SLM pre-copy took approximately 77s to complete migration,

whereas generic pre-copy required 87s with almost the same application-level downtime of around 5s. Due to multiple remote page faults, SLM post-copy took 58s to complete migration, while generic post-copy took about 65s. While post-copy had shorter TMT, the additional pages required by demand paging and active-push mechanisms took significantly longer to fetch, leading to a 30-second application-level downtime before full throughput was restored.

5.6.6 LAMP/ApacheBench Response Time

We created three VMs with a web server environment to represent a typical cloud host migration. Each VM was equipped with the Linux-Apache-MySQL-PHP (LAMP) software stack [41], which is a commonly used open-source software stack for web servers. These components work together to offer a platform for creating and hosting dynamic websites and web applications. On top of this software stack in each VM, we installed a WordPress website. To evaluate the web server's performance, we used ApacheBench [75], a benchmark tool for Apache servers. ApacheBench emulates multiple clients that continuously request the website and reports performance metrics such as the number of requests per second handled and the response time of each request.

Three ApacheBench instances run on a separate machine in the same network, each emulating five clients to request one of the three web servers. ApacheBench is a resource-intensive benchmark that stresses CPU, memory, disk I/O, and network. In our testing environment, CPU utilization is consistently above 95% while memory and I/O accesses are heavy enough to trigger the downtime phase. On average, ApacheBench indicates a rate of 20-23 responses per second across VMs, except for the downtime phase. During this phase, the response time increases proportionally to the duration required for the destination-side VM to resume

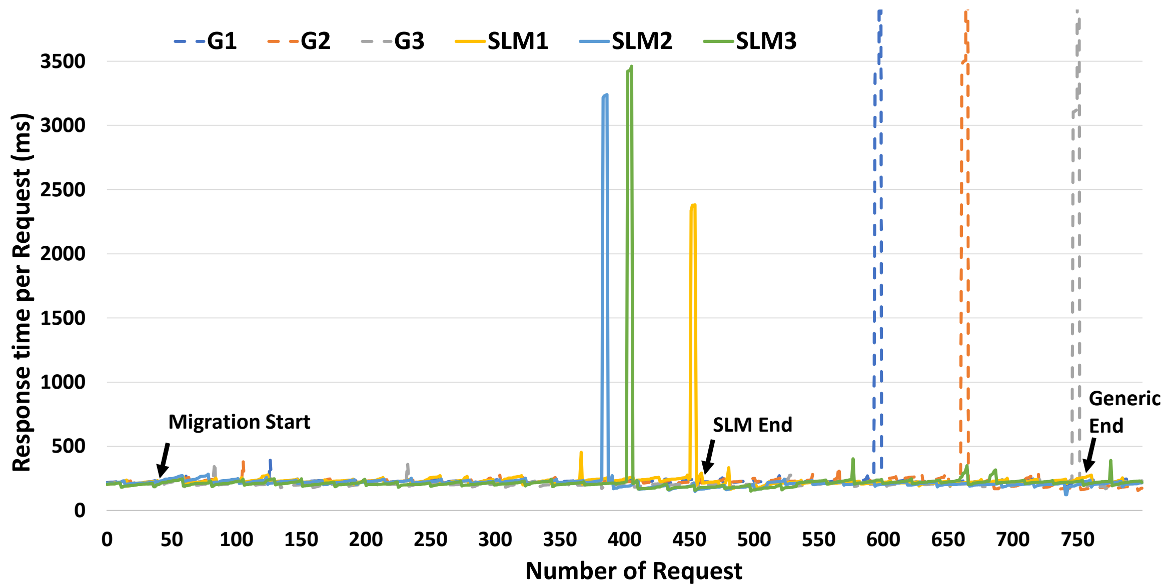


Figure 5.15: ApacheBench response times for generic and SLM pre-copy

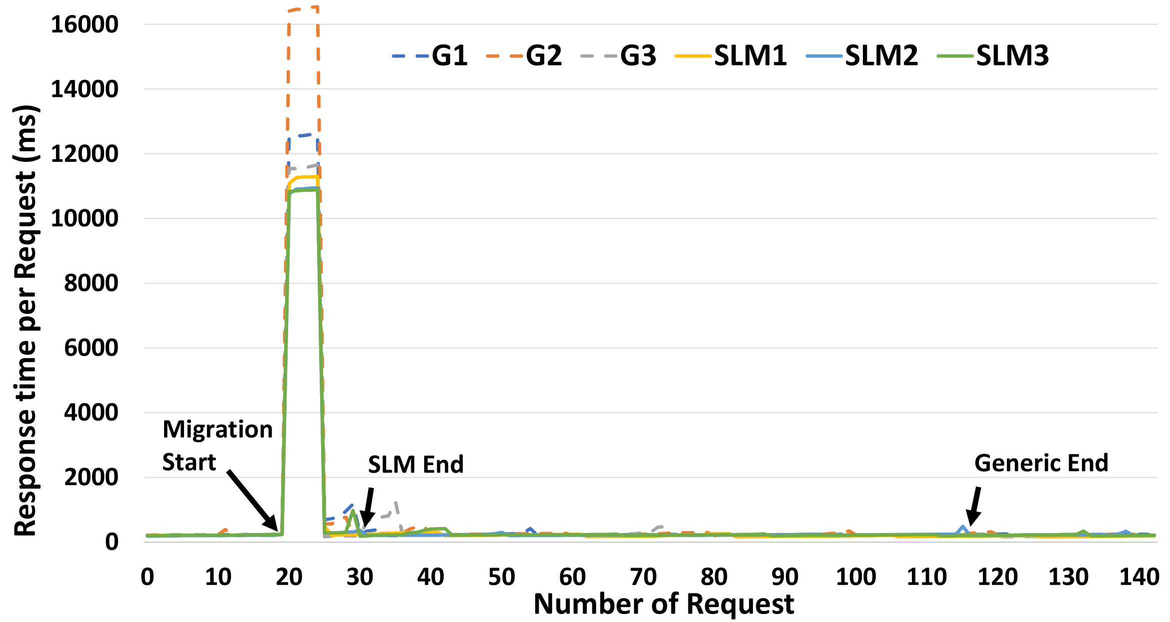


Figure 5.16: ApacheBench response times for generic and SLM post-copy

Figure 5.15 and Figure 5.16 show the response time of each web server VM during migration using different migration strategies. We aligned the time when the migration started. The solid and dashed lines represent VMs migrated using SLM and generic. Regarding the pre-copy strategy, both generic and SLM approaches exhibit three distinct spikes in the plot, each signifying one of the VMs entering the downtime phase, causing temporary degradation in web server performance. However, SLM pre-copy can reach the downtime threshold earlier than generic pre-copy due to the reduced number of pages transferred. SLM pre-copy takes about 20 seconds to complete the entire migration, whereas generic pre-copy takes around 30 seconds. As for the post-copy strategy, each VM experiences downtime immediately upon issuing the migration command, causing a significant increase in response time. Due to the post-copy nature of page faults across the network, it takes a while to resume the internal applications, as it only transfers the CPU state to the target machine during downtime. In the SLM approach, application resumption at the destination machine takes approximately 11 seconds, while in the generic method, it takes 12-16 seconds. Even though the application within the virtual machine has resumed, this does not imply that the entire migration process has been completed. The generic method takes approximately 20 seconds for the entire migration process, while the SLM method takes only 13 seconds due to the COW sharing of pages at the destination.

5.6.7 Performance of SLM on templated VMs

As previously noted, SLM offers broader compatibility compared to TLM when it comes to retaining various types of existing page sharing during migration. This experiment demonstrates that SLM is also applicable to templated VMs, although it may face challenges in surpassing TLM's migration speed. Nonetheless, SLM can prove valu-

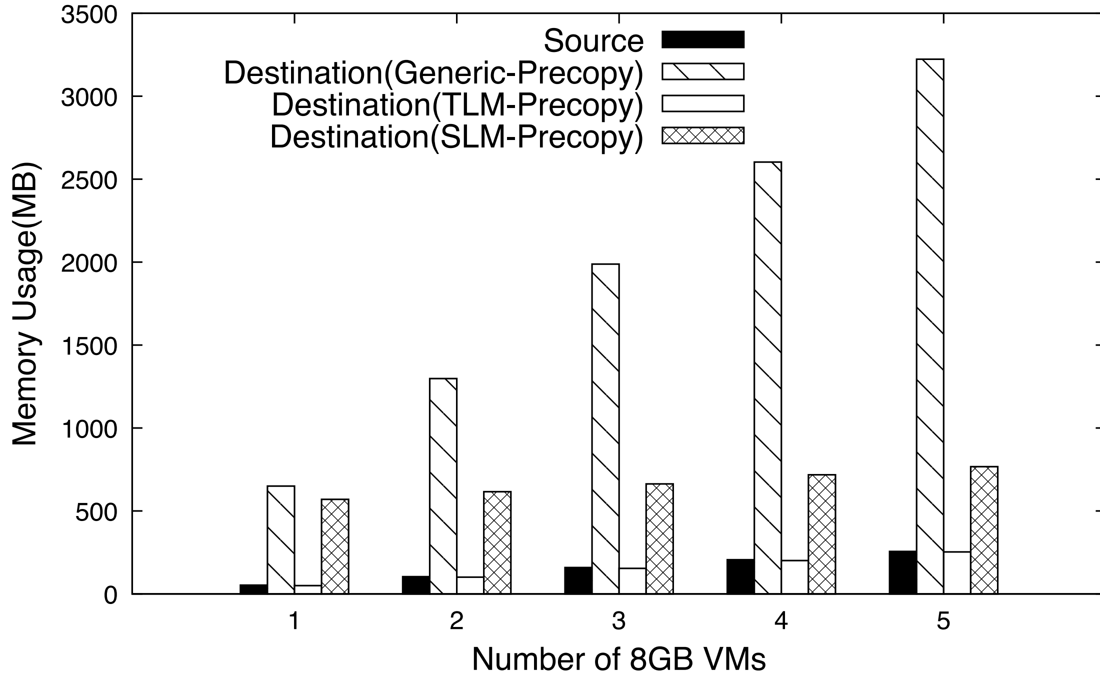


Figure 5.17: Memory footprint of templated VMs at the source before migration and destination after migration using generic, TLM and SLM pre-copy.

able for templated VMs in situations where the template file cannot be transmitted in advance. Nevertheless, since we initially implemented a TLM version of pre-copy, we briefly evaluate its performance in this section for completeness. We did not implement a TLM version of post-copy once we decided to move on to SLM implementation.

Figure 5.17 shows the memory footprint of templated VMs using generic, TLM and SLM pre-copy. The X-axis shows the number of VMs started from the same template, and the Y-axis shows their memory usage before migration at the source and after migration at the destination. With increasing number of VMs, the generic pre-copy results in significant expansion of memory footprint at the destination since it is unaware of memory-sharing with the underlying template image. Hence it transfers pages that were shared with the template multiple times in addition to *delta* pages. In contrast TLM preserves the memory footprint of templated VMs at the destination irrespective of the number of VMs started using the template but doesn't account for other exisitng

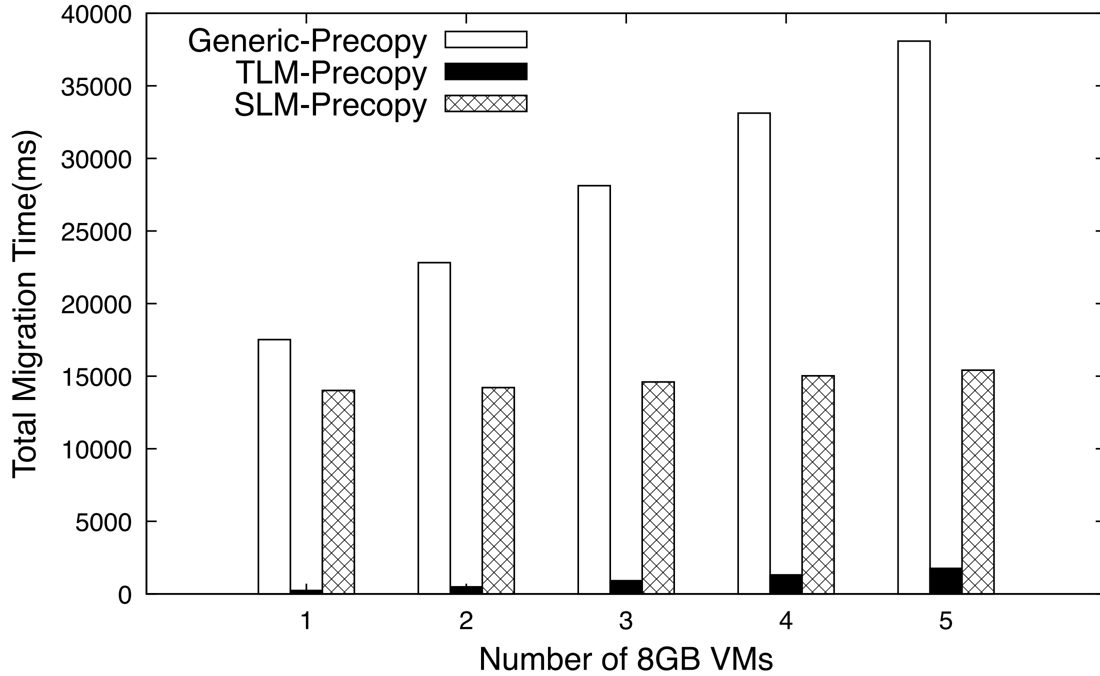


Figure 5.18: Total migration time of multiple VMs started from the same template and migrated concurrently using generic, TLM and SLM pre-copy.

COW page sharing among VMs such as KSM or process fork.Our SLM approach not only preserves COW page sharing at the destination through templating but also prevents the transfer of the base template ahead of time before the migration begins, thus ensuring comparable memory usage with the generic pre-copy approach.

Figure 5.18 shows the total migration of multiple templated VMs using generic, TLM and SLM pre-copy. The X-axis indicates the number of VMs booted from the same template to be migrated concurrently. The Y-axis shows the total migration time. Our TLM not only adds live migration capability to templated VMs but also reduces the total migration time up to 94% when considering only the transfer of *delta* pages. In addition to transferring the delta generated by the VMs, our SLM also transfers the COW shared base template, resulting in nearly identical total migration times, as the delta produced by idle VMs is negligible.

Figure 5.19 shows the total pages transferred of multiple templated VMs using

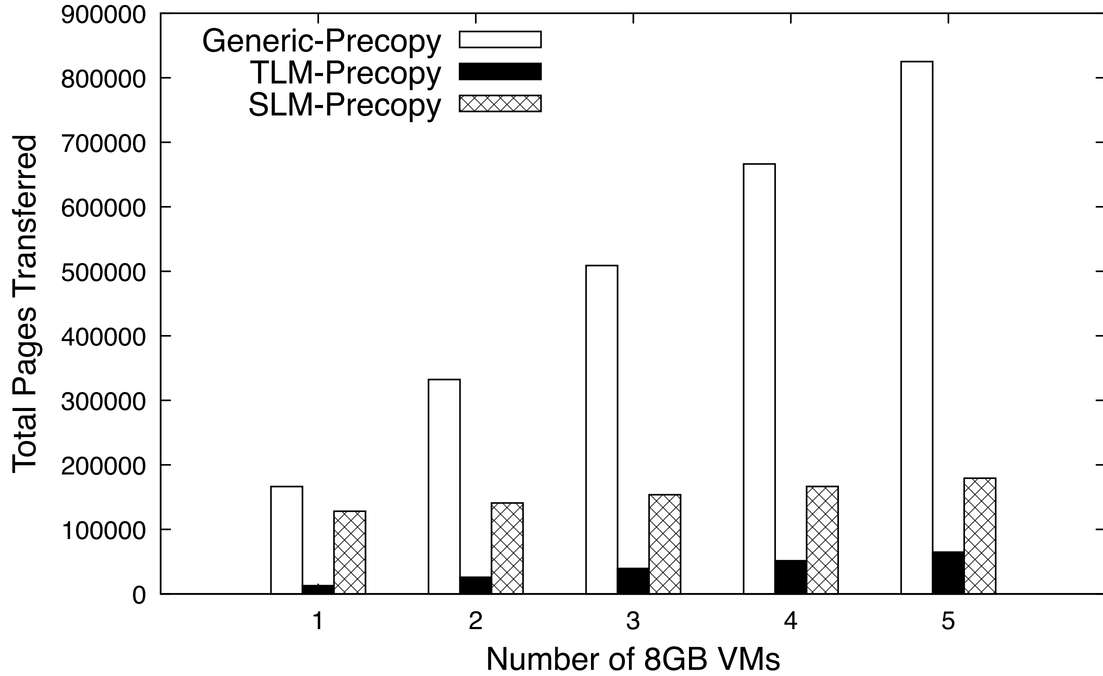


Figure 5.19: Total pages transferred of multiple VMs started from the same template and migrated concurrently using generic, TLM and SLM pre-copy.

generic, TLM and SLM pre-copy. The X-axis indicates the number of templated VMs to be migrated concurrently. The Y-axis shows the total pages transferred during live migration. Both TLM and SLM significantly reduces the total pages transferred when compared to the generic pre-copy

5.7 Related Work

We first discuss existing techniques for memory footprint reduction among co-located VMs within a single node followed by works related to page sharing in live migration.

Page sharing within a single node: Disco [5] was one of the first systems to propose and implement transparent page sharing to map multiple identical virtual pages to a single physical page. Satori [55], modifies guest OSes to identify sharing opportunities and communicate them to the hypervisor. KSM [1] uses a red-black tree indexed

with page content to find identical pages. Unlike Disco, it doesn't require any modification to the guest OS and doesn't need hash computation of page content. Performance of KSM depends on the location of the identical pages in the virtual address space. Since KSM sequentially looks for the potential candidate for merging, the further down the pages in the process address space, the less likely they will be merged. Difference Engine [24], in addition to standard COW full-page sharing, also supports sub-page level sharing and compression to improve memory savings through deduplication. Catalyst [21] offloads the identification of identical pages for deduplication to a GPU and eliminates sequential scanning of pages. Several techniques have been developed to efficiently launch multiple lightweight VMs from a common template image, which is COW-mapped into each VM's memory [40, 15, 50, 81, 59].

Live Migration with Page Sharing: Traditional pre-copy and post-copy [8, 28] are unaware of the source's memory optimizations. Because of this limitation, they send identical pages multiple times as if they are different, increasing total migration time and network traffic. Several prior techniques [12, 11, 85] have used content hashing to find identical and similar memory pages across multiple VMs to reduce/eliminate their transfer during live migration. However, these techniques do not detect and preserve pre-existing COW page sharings among co-located VMs at the destination node. Work in [34] aims to reduce the total migration time by identifying identical pages stored in the disk and sending only unique pages. This technique experiences high downtime because the destination fetches identical pages from the disk during downtime. Work in [7] deduplicates identical pages within the guest address space of a single VM via free memory pool and recreates page mappings at the destination. However, it does not address page sharings across multiple co-located VMs. Several studies have at-

tempted to minimize the migration time of containers or VMs in distributed edge platforms [52, 25, 10, 87, 6]. These efforts have employed techniques like delaying the transfer of writable working sets, using lightweight file systems, applying delta encoding, compressing data, and deduplicating identical pages. However, unlike SLM, these works do not focus on preserving COW page sharings at the destination to prevent an increase in memory footprint.

5.8 Chapter Summary

In this chapter, we addressed the problem that traditional live VM migration techniques do not preserve COW page sharing among co-located VMs. The resulting expanded memory footprint at the destination can lead to failed migrations, longer migration times, and increased network traffic. We presented the design, implementation, and evaluation of Sharing-aware Live Migration (SLM) to address this problem for both pre-copy and post-copy. SLM preserves all pre-existing page sharings among VMs at the destination machine irrespective of the underlying sharing mechanism. Our evaluation of SLM on the KVM/QEMU platform shows that SLM not only prevents memory footprint expansion but also significantly reduces the migration time by up to 59% and the amount of data transferred by up to 62% with no significant impact on application performance.

6 Conclusions and Future Directions

Traditional live VM migration is unaware of the underlying COW page sharing among VMs migrated to the same destination. Hence it ends up transferring the shared pages multiple times. In this dissertation, we designed, implemented, and evaluated three techniques to make traditional live VM migration aware of pages shared among VMs migrated to the same destination. The goal was to prevent memory footprint expansion and memory usage spikes during migration, while also reducing total migration time and network traffic.

6.1 Inter-host Template-aware Live Migration of Virtual Machines

We proposed Generic Template-aware Live Migration (Generic TLM) which addresses this shortcoming of pre-copy migration by ensuring that multiple templated VM instances maintain their COW page sharing with the base template even at the destination node and transfers only the *delta* pages that differ among various VM instances. Generic TLM requires that we first track delta (or dirty) pages for VM instances before migration. Then, during Generic TLM, only the delta pages are transferred for each instance. Generic TLM, besides preventing memory footprint expansion, also reduces the total migration time by up to 95.37% and network traffic by up to 92.15%.

6.2 Intra-host Template-aware Live Migration of Virtual Machines

Using the traditional pre-copy live migration is inefficient for migration within the same host as the existing memory pages are duplicated instead of transferring the ownership. Using Generic TLM still requires copying the delta pages generated by multiple templated VM instances. We designed an Intra-host TLM that tracks delta pages diverging from the base template and transfers their page ownership to the destination templated VMs, preventing unnecessary copying of additional dirtied pages. *Userfaultfd* - a Linux mechanism to handle page faults in user space - is used by QEMU to continuously monitor for any write events and redirect them to a dedicated memory backend file. During migration, the source only transfers the backend file offset for each page without copying the page content during the downtime. The destination then remaps the virtual address of the pages to the corresponding location at the backend file using the received offsets.

6.3 Sharing-aware Live Migration of Virtual Machines

While the Generic TLM approach works well in efficiently migrating multiple templated VMs, we realized that the problem of sharing-awareness in live migration extends beyond just templated VMs. Specifically, Generic TLM does not account for pages shared among VMs due to other memory sharing mechanisms besides templating, such as memory deduplication performed by KSM in Linux, or simple COW mappings due to process fork and file I/O operations. We proposed Sharing-Aware Live Migration (SLM) [18] to address the above problem. The key insight behind SLM is that irrespective of the underlying page-sharing mechanism, multiple COW-mapped guest pages will map to the same page in the physical memory. SLM examines the Physical Frame

Number (PFN) and Virtual Page Number (VPN) of each guest page being transferred and classifies the pages into three types. Unique Pages are those that have not been transferred yet. Shared Pages are those that have already been transferred. Dirty Pages are those that require retransmission due to being dirty in the previous pre-copy round. For Unique and Dirty pages, SLM transfers the entire page, including the page type and its PFN at the source, as a unique identifier. However, for Shared Pages, SLM does not send the page content, instead it only sends the page type and the PFN. At the destination, SLM COW maps the Shared Pages to the corresponding common page thus retaining the COW-shared mappings the same as the source. Besides preserving all pre-existing page sharings at the destination machine, SLM reduces the total migration time by up to 59% and network traffic by up to 62%.

Bibliography

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proc. of the Linux Symposium*, 2009.

[2] Wissal Attaoui, Essaid Sabir, Halima Elbiaze, and Mohsen Guizani. Vnf and cnf placement in 5g: Recent advances and future trends. *IEEE Transactions on Network and Service Management*, 2023.

[3] Hardik Bagdi, Rohith Kugve, and Kartik Gopalan. Hyperfresh: Live refresh of hypervisors using nested virtualization. In *Proc. of the Asia-Pacific Workshop on Systems*, 2017.

[4] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, 2007.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 1997.

[6] Lucas Chaufournier, Prateek Sharma, Franck Le, Erich Nahum, Prashant Shenoy, and Don Towsley. Fast transparent virtual machine migration in distributed edge clouds. In *Proc. of ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[7] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. *ACM SIGPLAN Notices*, 48(7):51–62, 2013.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines.

In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[9] Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. On the effectiveness of same-domain memory deduplication. In *Proc. of European Workshop on System Security*, pages 29–35, 2022.

[10] Rohit Das and Subhajit Sidhanta. LIMOCE: Live migration of containers in the edge. In *Proc. of International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2021.

[11] Umesh Deshpande, Brandon Schlinker, Eitan Adler, and Kartik Gopalan. Gang migration of virtual machines using cluster-wide deduplication. In *Proc. of International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[12] Umesh Deshpande, Wang Xiaoshuang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proc. of High Performance Parallel and Distributed Computing (HPDC)*, 2011.

[13] Spoorti Doddamani, Piush Sinha, Hui Lu, Tsu-Hsiang K. Cheng, Hardik H. Bagdi, and Kartik Gopalan. Fast and live hypervisor replacement. In *Proc. of ACM International Conference on Virtual Execution Environments (VEE)*, 2019.

[14] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. Qemu-based framework for non-intrusive virtual machine instrumentation and introspection. In *Proc.of the Joint Meeting on Foundations of Software Engineering*, 2017.

[15] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[16] Izik Eidus and Hugh Dickins. Kernel Samepage Merging. `https://www.kernel.org/doc/Documentation/vm/ksm.txt`, 2009.

[17] Roja Eswaran, Mingjie Yan, and Kartik Gopalan. Template-aware live migration of virtual machines. Accepted for publication in Proc. of ACM/IEEE Symposium on Edge Computing Workshop, 2023.

[18] Roja Eswaran, Mingjie Yan, and Kartik Gopalan. Tackling memory footprint expansion during live migration of virtual machines. In *Proc. of International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2024.

[19] Fatalerrors. Dirtybitmap. `https://www.fatalerrors.org/a/qemu-synchronization-dirty-pages-principle.html`.

[20] 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 *Proc. of IEEE Conference on Computer Communications Workshops Annual Computer Security Applications Conference*, 2019.

[21] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. Catalyst: GPU-assisted rapid memory deduplication in virtualization environments. In *Proc. of ACM International Conference on Virtual Execution Environments (VEE)*, 2017.

[22] GNU. bypass-Patch. `https://lists.gnu.org/archive/html/qemu-devel/2018-04/msg02250.html`.

[23] Google. Kubernetes engine. `https://cloud.google.com/kubernetes-engine`.

[24] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Communications of the ACM*. ACM New York, NY, USA, 2010.

[25] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: Agile VM handoff for edge computing. In *Proc. of ACM/IEEE Symposium on Edge Computing*, 2017.

[26] Jacob Gorm Hansen and Eric Jul. Self-migration of operating systems. In *Proc. of ACM SIGOPS European Workshop*, 2004.

[27] David Hildenbrand, Martin Schulz, and Nadav Amit. Copy-on-pin: The missing piece for correct copy-on-write. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[28] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS Operating Systems Review*, 2009.

[29] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2006.

[30] Jinhua Hu, Jianhua Gu, Guofei Sun, and Tianhai Zhao. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *International Symposium on Parallel Architectures, Algorithms and Programming*, 2010.

[31] Dong Huang, Bingsheng He, and Chunyan Miao. A survey of resource management in multi-tier web applications. *IEEE Transactions on Communications Surveys & Tutorials*, 2014.

[32] IBM. Kubernetes services. `https://www.ibm.com/cloud/containerservice`.

[33] iPerf. iPerf: The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects/Iperf/`.

[34] Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger. Efficient live migration of virtual machines using shared storage. *ACM SIGPLAN Notices*, 2013.

[35] Kernel.org. Admin-guide Userfaulfd. `https://docs.kernel.org/admin-guide/mm/userfaultfd.html`.

[36] Kernel.org. Userfaultfd. https://docs.kernel.org/admin-guide/mm/userfaultfd.

[37] Shinji Kikuchi and Yasuhide Matsumoto. Impact of live migration on multi-tier application performance in clouds. In *Proc. of IEEE International Conference on Cloud Computing*, 2012.

[38] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proc. of the Linux Symposium*, 2007.

[39] Santhosh Kumar T, Debadatta Mishra, Biswabandan Panda, and Nayan Deshmukh. CoWLight: Hardware assisted copy-on-write fault handling for secure deduplication. In *Proc. of Intl. Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.

[40] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2009.

[41] James Lee and Brent Ware. *Open Source Web Development with LAMP: Using Linux, Apache, MySQL, Perl, and PHP*. Addison-Wesley Professional, 2003.

[42] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, and Chita R Das. Migration, assignment, and scheduling of jobs in virtualized environment. In *Proc. of USENIX Hotcloud Workshop*, 2011.

[43] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. Computation offloading toward edge computing. *Proc. of IEEE International Conference on Cloud Computing*, 2019.

[44] Jens Lindemann and Mathias Fischer. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In *Proc. of the Annual ACM Symposium on Applied Computing (SAC)*, pages 183–192, 2018.

[45] Linux-manpages. Man-userfaultfd. `https://man7.org/linux/man-pages/man2/userfaultfd.2.html`.

[46] Linux-manpages. MMap Description. https://man7.org/linux/man-pages/man2/mmap.2.html.

[47] Haikun Liu and Bingsheng He. Vmbuddies: Coordinating live migration of multi-tier applications in cloud environments. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[48] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He, and Cheng-zhong Xu. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[49] Redis Ltd. Redis pipelining. `https://redis.io/docs/manual/pipelining/`.

[50] Costin Lupu, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Răzvan Deaconescu, and Costin Raiciu. Nephele: Extending virtualization environments for cloning unikernel-based VMs. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2023.

[51] Quang-Trung Luu, Sylvaine Kerboeuf, and Michel Kieffer. Admission control and resource reservation for prioritized requests with guaranteed sla under uncertainties. *IEEE Transactions on Network and Service Management*, 2022.

[52] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing*, 2019.

[53] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *Proc. of IEEE International Conference on Communications(ICC)*, 2017.

[54] Microsoft. Kubernetes service. `https://azure.microsoft.com/en-us/products/kubernetes-service`.

[55] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 1–1, 2009.

[56] Saad Mubeen, Sara Abbaspour Asadollah, Alessandro Vittorio Papadopoulos, Mohammad Ashjaei, Hongyu Pei-Breivold, and Moris Behnam. Management of service level agreements for cloud services in IoT: A systematic mapping study. *IEEE access*, 2017.

[57] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 391–394, 2005.

[58] Fangxiao Ning, Min Zhu, Ruibang You, Gang Shi, and Dan Meng. Group-based memory deduplication against covert channel attacks in virtualized environments. In *Proc. of IEEE Trustcom/BigDataSE/ISPA*, pages 194–200, 2016.

[59] Open Infrastructure. Kata Containers. `https://katacontainers.io`.

[60] Patchwork. Template-Patch. `http://patchwork.ozlabs.org/project/qemu-devel/list`.

[61] QEMU. Memory-QEMU. `https://www.qemu.org/memory.html`.

[62] QEMU. System emulation. `https://www.qemu.org/system/index.html`.

[63] QEMU. Userspace bugs. `https://gitlab.com/qemu-project/qemu/-/issues`.

[64] QEMU.org. Migration-Streams. `https://www.qemu.org/docs/master/devel/migration/main.html`.

[65] QNX. MMap mechanism. https://www.qnx.com/developers/docs/7.1/.

[66] Shashank Rachamalla, Debadatta Mishra, and Purushottam Kulkarni. Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems. In *International Conference on High Performance Computing*, 2013.

[67] Redis Ltd. Redis benchmark. https://redis.io/docs/management/optimization/benchmarks/.

[68] Yi Ren, Renshi Liu, Qi Zhang, Jianbo Guan, Ziqi You, Yusong Tan, and Qingbo Wu. An efficient and transparent approach for adaptive intra-and inter-node virtual machine communication in virtualized clouds. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2019.

[69] Christoph Rohland. Tmpfs. https://docs.kernel.org/filesystems/tmpfs.html.

[70] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. VM live migration at scale. *ACM SIGPLAN Notices*, 2018.

[71] Shashank Sahni and Vasudeva Varma. A hybrid approach to live migration of virtual machines. In *IEEE international conference on cloud computing in emerging markets (CCEM)*. IEEE, 2012.

[72] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proc. of Middleware*, 2016.

[73] Piush K Sinha, Spoorti S Doddamani, Hui Lu, and Kartik Gopalan. mwarp: Accelerating intra-host live container migration via memory warping. In *Proc. of IEEE Conference on Computer Communications Workshops Annual Computer Security Applications Conference*, 2019.

[74] Jonathan M. Smith and Gerald Q Maguire Jr. Effects of copy-on-write memory management on the response time of unix fork operations. *Computing Systems*, 1988.

[75] The Apache Software Foundation. Apachebench. https://httpd.apache.org/docs/2.4/programs/ab.html, 2022.

[76] The Linux kernel user's and administrator's guide. Examining process page tables. `https://www.kernel.org/doc/html/latest/admin-guide/mm/pagemap.html`.

[77] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: Power and migration cost aware application placement in virtualized systems. In *Proc. of Middleware*, 2008.

[78] VMware. Container Services. `https://cloud.vmware.com/vmwarepks`.

[79] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 148–162, 2005.

[80] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. XenLoop: A transparent high performance inter-VM network loopback. In *Proc. of High Performance Parallel and Distributed Computing (HPDC)*, 2008.

[81] Kun Wang, Jia Rao, and Cheng-Zhong Xu. Rethink the virtual machine template. *ACM SIGPLAN Notices*, 2011.

[82] Jinpeng Wei, Lok K. Yan, and Muhammad Azizul Hakim. Mose: Live migration based on-the-fly software emulation. In *Proc. of the 31st Annual Computer Security Applications Conference*, 2015.

[83] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[84] Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. iAware: Making live migration of virtual machines interference-aware in the cloud. *IEEE Transactions on Cloud Computing*, 2013.

[85] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proc. of IEEE International Conference on Cluster Computing*, 2010.

[86] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[87] Zhe Zhou, Xintong Li, Xiaoyang Wang, Zheng Liang, Guangyu Sun, and Guojie Luo. Hardware-assisted service live migration in resource-limited edge computing systems. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 2020.

ProQuest Number: 31490188

INFORMATION TO ALL USERS
The quality and completeness of this reproduction is dependent on the quality
and completeness of the copy made available to ProQuest.

Part of **Clarivate**

Distributed by
ProQuest LLC a part of Clarivate (2024).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work is protected against unauthorized copying under Title 17,
United States Code and other applicable copyright laws.

This work may be used in accordance with the terms of the Creative Commons license
or other rights statement, as indicated in the copyright statement or in the metadata
associated with this work. Unless otherwise specified in the copyright statement
or the metadata, all rights are reserved by the copyright holder.

ProQuest LLC
789 East Eisenhower Parkway
Ann Arbor, MI 48108 USA