D-Track: Efficient Fake Dirty Page Tracker for Live VM Migration

Samindu Cooray
Index Number : 20000251

Supervisor: Dr. Dinuni K Fernando

April 2025

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

Declaration

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

Student Name : P.L.S.R. Cooray
Registration Number : 2020/CS/025
Index Number : 20000251

[illegible] 28/06/2025

Signature & Date

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

Supervisor Name : Dr. Dinuni K Fernando

[illegible] 29/06/2025

Signature & Date

Abstract

Cloud computing offers efficient and reliable services to millions of users worldwide, ensuring uninterrupted service. They use virtualization to offer services by hosting multiple virtual machines (VMs) on a single physical server. To maintain service continuity during hardware failures or server maintenance, live virtual machine migration techniques transfer virtual machines to different physical servers without interrupting service. It is challenging to accurately predict the total migration time in Pre-Copy migration due to the unpredictable nature of convergence. The iteration count required to transfer memory pages cannot be precisely estimated due to varying dirty page rates and the highly dynamic nature of network bandwidth. In extreme cases, rapid dirty page generation and low network bandwidth can lead to a prolonged total migration time or even failure in migration. Therefore, rapid dirty page generation is a significant problem in Pre-Copy migration. A key issue in this process is generating fake dirty pages, mainly caused by silent store instructions and defects in dirty page tracking. Previous solutions, such as SHA1-based fake dirty prevention algorithms, did not account for the overhead associated with hash computation on migration. This paper presents D-Track, an efficient fake dirty page tracking mechanism that mitigates redundant page transfers by continuously tracking dirty pages during migration. D-Track reduces the total migration time by 20% - 40% when combined with compression techniques while maintaining acceptable application performance degradation. Evaluations of memory-intensive, CPU-intensive, and multiple-intensive workloads demonstrate its effectiveness in improving migration efficiency without significant degradation in VM performance.

Acknowledgements

First and foremost, I would like to express my heartfelt appreciation to Dr. Dinuni Fernando, whose insightful mentorship and domain knowledge steered me through this research project. Her unwavering support and constructive feedback were invaluable in ensuring its depth and direction. Also, I would like to thank Pratheek Senevirathna and B.F. Ilma, my seniors who assisted me in understanding and exploring the QEMU code base and adjusting to the server environment.

I should also appreciate WSO2 Lanka (Pvt) Ltd. for generously providing the servers and necessary infrastructure to establish the test environment in this research endeavor and Dr. Dinuni Fernando for facilitating their collaboration. Furthermore, I would like to extend my appreciation to the staff at NOC for their hard work in setting up the servers and granting us access to the server room.

Finally, I express my gratitude to my family and colleagues, whose constant love and understanding provided the essential foundation for this endeavor. Their unwavering support and encouragement fueled my resolve throughout this journey.

Contents

List of Figures

List of Tables

Acronyms

AWS Amazon Web Services. 6

CC Cloud Computing. 6

CDCs Cloud Data Centers. 6

DT Downtime. 1, 7, 10

FDP Fake Dirty Page. ii, 1, 10, 11

FDPS Fake Dirty Pages. 1, 6–8, 10, 11

GCP Google Cloud Platform. 6

HBFDP Hash-Based Fake Dirty Prevention. ii, iv, 7, 8, 18, 20–22, 34–36

HCM Hybrid Migration. iv, 1, 6

JIT Just-In-Time. 16

PosC Post-Copy. 1, 6

PosCM Post-Copy Migration. iv, 1, 4, 5

PreC Pre-Copy. iv, 1, 3, 4, 6, 8, 11

PreCM Pre-Copy Migration. iv, 1, 3–8, 10

TMT Total Migration Time. 1, 7, 10

VM Virtual Machine. ii, 1–6, 10, 16, 42

VMs Virtual Machines. 1, 6–8

XBZRLE XOR Binary Zero Run Length Encoding. 7, 8, 18, 20–23, 34, 36

1 Introduction

Live Virtual Machine (VM) Migration is the process of migrating a VM from a source host machine to a destination host machine without stopping the VM at the start of the migration procedure, providing uninterrupted services to end-users throughout the migration. The efficiency and the performance of a Live VM Migration technique are evaluated using attributes such as Total Migration Time, Downtime, Service Degradation, Network Traffic, and Network bandwidth utilization.

When considering Pre-Copy Migration, total migration time is regarded as the primary bottleneck due to its unpredictability (Elsaid et al. 2022). The iteration count required to transfer memory pages cannot be precisely estimated due to varying dirty page rates and network transmission speed. In extreme cases, rapid dirty page generation and low network bandwidth can lead to prolonged total migration time or even migration failure. Therefore, rapid dirty page generation is a significant problem in Pre-Copy Migration (Shribman & Hudzia 2013).

However, most of these generated dirty pages are not dirty. They are identical to their existing copies stored in the destination host, where transferring them wastes network bandwidth and migration time. Exploring the root of this inefficiency, Li et al. (2019) identified two reasons for the generation of Fake Dirty Pages. The primary reason is the "write-not-dirty" request issued by silent store instructions. The secondary reason is the defects in the mechanism that tracks dirty pages. Furthermore, Li et al. (2019) conducted experiments on Pre-Copy Migration to observe Fake Dirty Page generation when migrating VMs with memory-intensive workloads. The experiments showed that some workloads generated high amounts of Fake Dirty Pages, while others generated significant amounts. Therefore, by eliminating these Fake Dirty Page performance matrices, such as total migration time, downtime and application performance degradation can be improved.

On the other hand, if the Post-Copy Migration approach is used to migrate a VM running memory-intensive workloads, the problem of prolonged total migration time for memory-intensive workload in Pre-Copy Migration is resolved. But Post-Copy Migration inherently contains two weaknesses: Less robust and performance degradation of an application running in the VM which is migrated. When considering Hybrid Migration, it allows memory-intensive workloads to be handled without migration failures by combining Post-Copy and Pre-Copy migration techniques. However, it still inherits the problem of missing the optimal converging point from Pre-Copy Migration (Li et al. 2019).

1.1 Background

Migrating a VM to a destination host machine from a source host machine is called VM Migration . VM migration can be categorized mainly into two categories **Non-Live Migration** and **Live Migration** as shown in the Figure 1. When considering Non-Live Migration (Milo̊jičić et al. 2000), the VM is stopped before starting the migration procedure and again resumes it in the destination host machine after transferring the VM completely. Unlike Non-Live Migration, in Live Migration, before starting the migration procedure, the VM is not stopped, providing uninterrupted services to end-users throughout the migration.

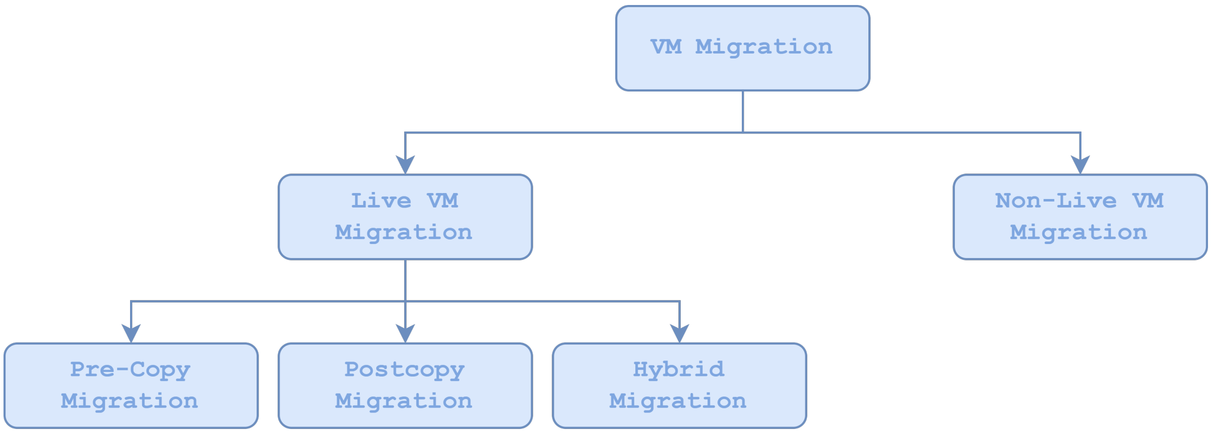


Figure 1: Classification of VM Migration

Following are some of the benefits as stated in Jul et al. (2005),

- After migration completion, the original host machine does not have to be available for access.
- The clients are not required to reconnect again after migration.
- The operators of the data centers can perform live migration without being concerned about the operations running in the VM.

Further, Live VM Migration has adapted over time to minimize the service downtime (Zhu et al. 2013), efficiently utilize the network bandwidth (Svärd et al. 2011), and decreasing the performance degradation (Ibrahim et al. 2011).

1.1.1 Live VM Migration

In Live VM Migration, the main concern is to minimize both *total migration time* (the duration from initiating the migration to completing the migration process) and *downtime* (the period where the VM is suspended).

Generally, there are three phases when transferring the memory of a VM:

1. **Push Phase**

   Memory pages assigned to the VM are migrated over the network to the destination from the source. Since the VM is continuously working in the source host, some memory content may get modified; therefore, these modified copies should be again sent to the destination.

2. **Stop-and-Copy Phase**

   The source stops the execution of the VM and transfers every memory content to the destination. Upon the migration completion, the destination starts the VM

3. **Pull Phase**

   After starting the VM in the destination, memory content is fetched from the source upon page faults.

Out of the above-mentioned phases, one or two are selected in practical solutions. Sapuntzakis et al. (2002) and Kozuch & Satyanarayanan (2002) state a *pure stop-and-copy* method that only uses the stop-and-copy phase and Zayas (1987) states a *pure demand-migration* method which uses both stop-and-copy and pull phases. However, applying them in Live Migration may lead to unacceptable results.

Based on the method of memory content transferring, there are three Live Migration techniques. They are *Pre-Copy Migration* (Jul et al. 2005), *Post-Copy Migration* (Hines et al. 2009*a*) and *Hybrid Migration* (Sahni & Varma 2012)

1.1.1.1 Pre-Copy Migration

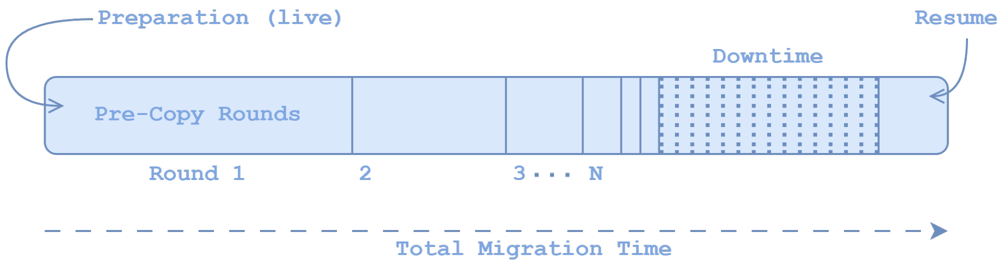


Figure 2: Pre-Copy Migration Timeline (Hines et al. 2009*a*).

At the initial round of *Pre-Copy Migration*, every page assigned to the VM are migrated into the destination host. Then, in successive rounds, the memory content modified (dirtied) during the earlier round by the VM is sent to the destination. These rounds are stopped when the page dirtying rate converges to a specific point. Then the remaining dirty pages (Writable Working Set) and the CPU state are transferred to the destination after the source stops the VM. Figure 2 graphically demonstrates the Pre-Copy Migration procedure. Then the destination starts the VM, making it the primary host, and the source host discards the VM from it (Jul et al. 2005).

Therefore the proposed *Pre-Copy Migration* by Jul et al. (2005) combines iterative push phase with short stop-and-copy phase.

Pre-Copy Migration Stages

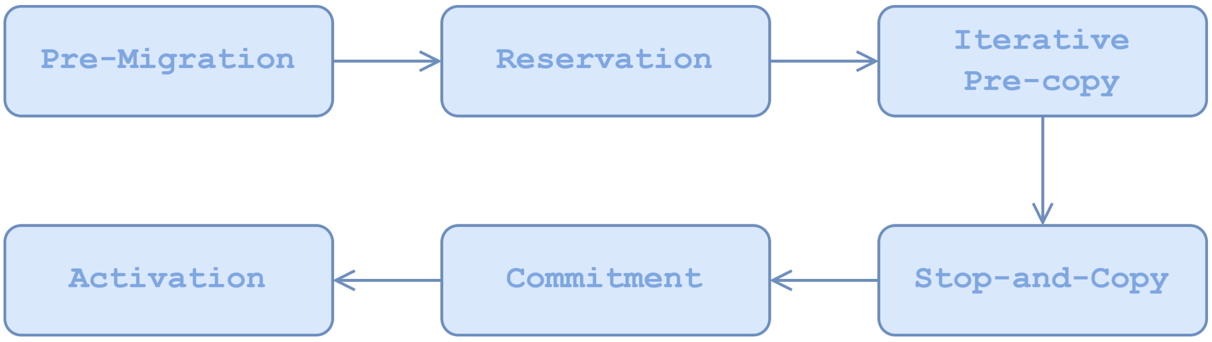


Figure 3: Stages in Pre-Copy VM Migration (Jul et al. 2005).

The Figure 3 shows the migration stages of Pre-Copy Migration. The functionality of each stage are described below:

1. **Pre-Migration:** The VM is active in the source host.
2. **Reservation:** Reserves a container on the destination host after confirming the resource availability.
3. **Iterative Pre-Copy:** All the memory pages are transferred to the destination host in the initial iteration. Then, only the modified (dirtied) pages are transferred in the next iterations.
4. **Stop-and-Copy:** The CPU state and the remaining memory pages are transferred to the destination after the source stops the Virtual Machine.
5. **Commitment:** The destination informs the source that the migration of the VM is completed the destination received it, and the source sends an acknowledgment to this message. The source discards the VM.
6. **Activation:** The destination host becomes the primary host and runs a post-migration code.

Pre-Copy Migration Techniques

There are various number of Pre-Copy Migration techniques, given below are some of those techniques:

- **Dynamic Rate-Limiting :** Handle the Bandwidth Utilization in the iterative push phase and stop-and-copy phase (Jul et al. 2005).
- **Data Compression :** Exploits word-level duplication in data to reduce data transfer (Jin, Li, Wu, Shi & Pan 2009).
- **Migration Control Method :** Detects memory modification patterns and terminated migrations with high downtime (Ibrahim et al. 2011).
- **Delta Compression :** The modifications done to the data are stored without storing the full data sets (Svärd et al. 2011).
- **Memory Compaction Technique :** A technique that is a combination of memory snapshot and disk-memory deduplication cache (Piao et al. 2014).
- **Adaptive VM downtime control technique** (Piao et al. 2014)
- **Deduplication :** Transfer only one copy of duplicate pages (Wood et al. 2015).
- **Three Phase Optimization :** Minimizes the memory page transfer in transferring rounds (Sharma & Chawla 2016).
- **Page Content Reduction Technique :** Transferring the encoded form of the XOR differential page without transferring the whole page. (Bhardwaj & Krishna 2019)

1.1.1.2 Post-Copy Migration

Pre-Copy Migration effectively minimizes the application performance degradation and downtime when executing CPU-intensive workloads in the VM. However, its performance for memory-intensive workloads is less effective because of the high page dirty rates. To solve this problem, Hines et al. (2009*a*) introduces Post-Copy Migration as another Live VM Migration technique.

Post-Copy Migration initially transfers the CPU states to the receiving host after stopping the VM in the source. Then, the destination host starts the VM without memory content. After that, the migration thread starts the active push of memory pages to the destination from the source. Concurrently, fetching of the page-faulted memory pages in the destination is done on-demand from the source (Hines et al. 2009*a*). Figure 4 graphically demonstrates the Post-Copy Migration procedure.

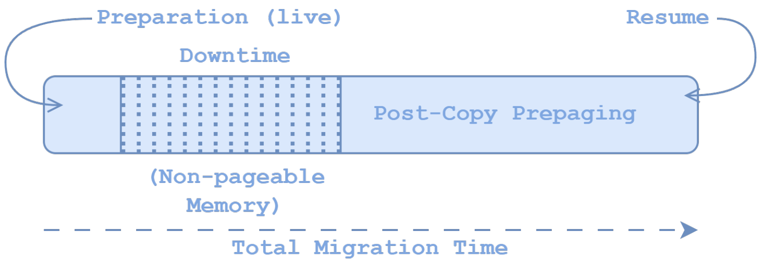


Figure 4: Post-Copy Migration Timeline (Hines et al. 2009*a*).

Post-Copy Migration Variants

There are different variants of Post-Copy Migration, pivoting on different techniques of page retrieval from source to destination. These techniques are:

- **Active Push :** Proactively pushes pages from source to destination (Hines et al. 2009*a*).
- **On-Demand Paging :** Only page-faulted pages are retrieved from the source (Hines et al. 2009*a*).
- **Pre-Paging :** Guess future page faults and adjust retrieval patterns to avoid page faults (Hines et al. 2009*a*).
- **Self Ballooning :** Reduces the transfer of unused pages (Hines et al. 2009*a*).

Weaknesses in Post-Copy Migration

Although the Post-Copy Migration resolves Pre-Copy Migration's problem with dirty pages not converging to a minimum value to complete the migration (Li et al. 2019), it has two inherent weaknesses:

1. **Less Robust:** When considering Pre-Copy Migration, the VM in the source host will still be working, although the migration fails due to a problem in the migration network or destination host. But in Post-Copy Migration, since both the destination and source hosts contain a piece of the latest state of memory, it will cause the failure of the VM.
2. **Performance Degradation:** In Post-Copy Migration, access latency for memory is increased due to continuous page faults resulting in performance degradation of applications in the VM.

1.1.1.3 Hybrid Migration

By combining migration techniques Post-Copy and Pre-Copy, Hybrid Migration is performed. The migration procedure starts with a few Pre-Copy rounds, as shown in Figure 5. Next, the CPU state is migrated to the destination host after the VM is stopped in the source, and upon the completion of the transfer, the destination host starts the VM. Then, the destination host retrieves the pages using one or more techniques mentioned in the above section. Therefore, as Hybrid Migration contains combined advantages of both Post-Copy and Pre-Copy Migrations, memory-intensive workloads can be handled without any migration failures (Sahni & Varma 2012).

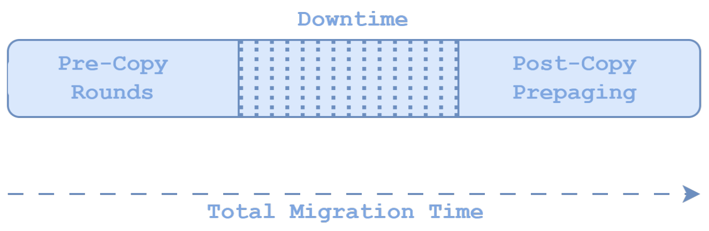


Figure 5: Hybrid Migration Timeline

When looking into the optimizations in Hybrid Migration, Li et al. (2019) introduces an Intelligent Hybrid Migration approach. In traditional Hybrid Migration, the datacenter admin might have to switch manually to Post-Copy from Pre-Copy or set to shift after executing fixed two or three Pre-Copy rounds. But the proposed intelligent Hybrid Migration will minimize the number of page faults in the Post-Copy phase by shifting to Post-Copy from Pre-Copy at a nearly ideal point.

1.2 Motivation

Cloud Computing (CC) is an essential technology in the modern world due to its efficient and reliable services. At present, platforms like Microsoft Azure, Amazon Web Services (AWS), Alibaba Cloud, Google Cloud Platform (GCP) are some of the CC service providers (Rayaprolu 2024). As these services are used by millions of users worldwide, Cloud Data Centers (CDCs) manage these resources, maintaining high-speed network connections for uninterrupted services. However, CDCs consume a significant amount of electricity, with over 30% of the servers using energy while being idle. As a solution, CDCs are embracing virtualization (Jul et al. 2005), allowing OS instances to run concurrently on a physical machine, providing individual OS isolation, efficient resource usage, and high performance (Barham et al. 2003).

With the use of virtualization, CDCs maintain servers, which host multiple Virtual Machines (VMs) in each server. These servers can be subjected to various hardware or software failures. This affects the interruptions to the services provided by the VMs in the failing server. When these failures are identified, the VMs in the server are migrated to another physical server by using Live VM Migration techniques. However, migrating VMs running memory-intensive workloads is an intractable problem that may increase the total migration time and downtime or lead to migration failure.

The recent work of Nathan et al. (2016) discovered a previously unidentified phenomenon: the presence of **Fake Dirty Pages** during Pre-Copy Migration. While marked as modified and sent to the destination host, these pages are identical to their existing copies, which are already stored in the destination. According to the observations of Li et al. (2019), they have identified that during Pre-Copy Migration of VMs running memory-intensive workloads, some workloads generated high amounts of Fake Dirty Pages while others generated significant amounts. Transferring these Fake Dirty Pages wastes the bandwidth and increases the total migration time and downtime.

1.3 Research Problem

The state-of-the-art technique for reducing redundant data transfers in Pre-Copy Migration for memory-intensive workloads proposed by Li et al. (2019) uses SHA1 secure hashes to identify and eliminate Fake Dirty Pages, marking significant advancements in reducing redundant data transfers. However, this approach raises questions about the efficacy and potential benefits of alternative, more sophisticated hashing methods. While using SHA1 hashes effectively minimizes the transfer of Fake Dirty pages, exploring different hashing techniques could further enhance this process, reducing the data transfer load even more efficiently.

Furthermore, the existing study has not considered applying different optimization techniques along with the Hash-Based Fake Dirty Prevention (HBFDP) algorithm. This application can significantly enhance the efficiency of VM migration by minimizing total migration time and downtime.

By addressing these research gaps, this study aims to improve hash computation speed and improve the performance of Pre-Copy Migration when migrating Virtual Machines running memory-intensive workloads.

1.4 Research Questions

1. How to mitigate redundant data transfers by identifying fake dirty pages?
    - This research question investigates how to identify fake dirty pages with minimal overhead on migration and Virtual Machines by solving the issues of the dirty page tracking mechanism in Qemu
2. How to reduce the data transferring load in Pre-Copy Migration to improve the performance of memory-intensive workloads?
    - This research question investigates how to incorporate different optimization techniques *(XBZRLE, Delta compression, etc)* to improve migration efficiency.

1.5 Aims and Objectives

1.5.1 Aim

Improving the performance matrices: Total Migration Time, Downtime, and Application Performance Degradation in Live Migration by Mitigating Redundant Memory Page Transfers in Pre-Copy Migration

1.5.2 Objectives

- To evaluate the performance improvement of fake dirty tracking mechanism to reduce redundant memory page transfers by considering Fake Dirty Pages

- To evaluate the performance improvement of combining different optimization techniques to reduce the data transferring load in Pre-Copy Migration to improve the performance for memory-intensive workloads?

1.6 Scope

1.6.1 In Scope

The following areas will be covered under the scope of this research.

- Develop a prototype of the proposed fake dirty tracking mechanism.
- Develop a prototype of the algorithm proposed by Li et al. (2019) using SHA1 hashes in QEMU-KVM to use a baseline technique to evaluate the proposed fake dirty tracking mechanism.
- Evaluate the D-Track with and without incorporating different optimization techniques to show that the total migration time can be reduced by comparing against Vanilla Pre-Copy, XBZRLE, and the algorithm proposed by Li et al. (2019) *(Prototype developed with SHA1 hashes).*
- The research focuses on running single VM migrations on Ubuntu Server as the host OS within a LAN environment, with potential for future extension to WAN and multiple VM migrations.

1.7 Significance of the Research

As stated in Section 2.3, the proposed solution of Li et al. (2019) avoids the transfer of Fake Dirty Pages using **SHA1 hashes**. However, their study only used the SHA1 hashing technique in the algorithm and did not explore the improvement that could be achieved using other hashing techniques. Furthermore, the study of Li et al. (2019) hasn't focused on incorporating their algorithm with other optimizations for further improvements.

This study evaluates the performance improvement of various hashing techniques over SHA1 to reduce redundant memory page transfers by considering Fake Dirty Pages. This approach provides a method to enhance the hash computation speed, thus improving the efficiency of Pre-Copy Migration when migrating VMs running memory-intensive workloads.

Another significance of this research is the potential to combine different optimization techniques with the Hash-Based Fake Dirty Prevention algorithm. This combined approach is expected to minimize total migration time and downtime more effectively than existing methods.

This is particularly significant for ensuring uninterrupted services to end-users by improving key performance matrices such as total migration time, downtime, and Application Performance Degradation.

1.8 Outline of the Dissertation

The organization of the rest of this dissertation is as follows. Section 2 provides an overview of the fake dirty problem. Section 4 provides the architecture of D-Track and implementation details. Section 6 provides the evaluation results. Sections 8 provide the study's conclusions.

2 Literature Review

The efficiency and the performance of a Live VM Migration technique is evaluated using attributes such as Network Traffic, Downtime, Total Migration Time, Service Degradation, and Network bandwidth utilization. When considering Pre-Copy Migration, total migration time is regarded as the primary bottleneck due to its unpredictability. Two significant factors directly impacting the total migration time and downtime are the dirty page generation rate and VM size. The total migration time of Pre-Copy Migration relies on two distinct phases: Initial iteration that transfers whole memory and subsequent iterations that transfer dirty pages. The time allocation between these phases depends heavily on the VM's memory behavior.

Problems that directly affect the time consumption in the push phase, resulting in higher total migration time in Pre-Copy Migration, are Rapid Dirty Page Generation, Page Level Content Redundancy, Lower Network Transferring Rates, and Fake Dirty Pages.

Additionally, Performance and energy consumption concerns also arise, including service degradation due to resource competition and data transfer overhead and increased CPU activity, VM size and network bandwidth, resource consumption on hosts, and state transfer and I/O bandwidth, respectively.

2.1 Fake Dirty Problem

As mentioned in Section 2, *Rapid Dirty Page Generation* is one of the significant problems in Pre-Copy Migration. However, most of these generated dirty pages are actually **not dirty**. These pages, marked as modified and transferred, are identical to their existing copies stored in the destination host. This unnecessary duplication wastes network bandwidth and migration time (Li et al. 2019, Nathan et al. 2016). Li et al. (2019) observed the Fake Dirty Page generation in Pre-Copy Migration when migrating VMs running memory-intensive workloads.

2.2 Reasons for Fake Dirty Page Generation

There are two reasons identified by Li et al. (2019) for Fake Dirty Page generation. The primary reason is **"write-not-dirty"** request issued by *silent store instructions*. The secondary reason is the defects in the mechanism that tracks dirty pages.

2.2.1 "Write-not-dirty" Pages

The first reason for generating Fake Dirty Pages are "Write-not-dirty" Pages. These **Write-not-dirty** pages are memory pages marked as dirty but do not have actual content change(Li et al. 2019).

Silent store instructions are the main cause for the write-not-dirty requests (Molina et al. 1999, Lepak & Lipasti 2000*a*,*b*). These instructions write a value to a memory address exactly like the existing value, resulting in no system state change. According to the evaluations done by Lepak & Lipasti (2000*a*), an average amount of 20% to 68% are silent store instructions among all the store instructions executed. Also, the analysis done by Lepak & Lipasti (2000*b*) identified that silent store instructions execute in compiler optimizations and all levels of program executions and have an algorithmic nature. Therefore, in workloads wide amounts Fake Dirty Pages are generated due to write-not-dirty requests resulting in prolonged total migration time (Li et al. 2019).

2.2.2 Defective Dirty Page Tracking

The second reason for generating Fake Dirty Pages is the method used by the migration thread for dirty page tracking during migration. Both hypervisors, KVM(Bellard 2005) and XEN(Barham et al. 2003) use bitmaps to keep track of these dirty pages.

2.2.2.1 KVM

The virtualization platform KVM/QEMU (QEMU 2024, KVM 2024) can be divided into KVM (kernel module) and QEMU (userspace tool). Both these parts contain their dirty bitmap. The dirty bitmap of the kernel traces the write request to a page and marks the page as dirty. This process is continued throughout an iteration, marking all the dirty pages. When the iteration ends, the kernel's dirty bitmap is synced with userspace's dirty bitmap using API: *ioctl*, and the dirty bitmap of the kernel is reset to track the next iteration. In the next iteration, the synced dirty bitmap of the userspace is used to transfer modified pages. Therefore, the tracking of dirty pages is limited to one Pre-Copy round, resulting in Fake Dirty Page generation. This will be discussed in detail in Section 3.1.

2.2.2.2 XEN

XEN (XEN 2024) maintains three bitmaps in the migration thread to mark the pages that must be transferred.

Below are the functionalities of the three bitmaps maintained by XEN:

1. **to_send:** Pages dirtied during an iteration are marked in this bitmap. Therefore, this bitmap is referred to by the migration thread to transfer pages in the next iteration. The to_send bitmap is updated at the end of every iteration by referring to the kernel bitmap and then resetting it.

2. **to_skip:** Pages that should not transfer in the current iteration but are transferred in the next iteration are marked in this bitmap. This bitmap is updated by the migration thread from the kernel bitmap in a random iteration without resetting the kernel bitmap.

3. **to_fix:** Pages that are transferred in the stop-and-copy phase are marked in this bitmap.

Compared with KVM, XEN's tracking mechanism works well to minimize Fake Dirty Page generation with the use of *to_skip* bitmap as it avoids written dirty pages to be transferred in the current iteration. But if a page is dirtied before being transferred and after the *to_skip bitmap* is updated, it is the same problem as the KVM's tracking mechanism, making the page fake dirty (Li et al. 2019).

In this next section 2.3, let's explore the proposed solutions to avoid the transmission of Fake Dirty Pages (FDPS) during migration.

2.3 Related Work

Various optimizations have been proposed over the years to accelerate live virtual machine (VM) migration. Many of these techniques focus on reducing the amount of data transferred, which directly impacts the overall migration time.

Content-based optimizations, such as deduplication (Deshpande et al. 2011*a*, 2013, Kiswany et al. 2011, Riteau et al. 2011, Wood et al. 2011, Zhang et al. 2010, Svärd et al. 2011) and compression(Jin, Deng, Wu, Shi & Pan 2009, Deshpande et al. 2011*a*, Svard et al. 2011, Nathan et al. 2016), are commonly used to reduce the data volume. However, while these techniques are effective in minimizing data transfer, they still inherit the drawback of transmitting fake dirty pages, which contributes to redundancy.

Post-copy migration(Hines et al. 2009*b*, T. Hirofuchi and I. Yamahata n.d.) offers an alternative to the traditional pre-copy approach. In post-copy, each memory page is transferred only once, reducing unnecessary retransmissions. This method is particularly effective for memory write-intensive workloads, offering lower total migration time and typically smaller downtime compared to pre-copy. For example, Reactive Cloud(Hirofuchi et al. 2012) leverages post-copy to handle sudden overloads with rapid VM relocation. Similarly, Scatter-Gather VM migration (Deshpande et al. 2015) enables fast VM eviction when destination resources are constrained.

Other approaches attempt to avoid the transfer of unnecessary memory pages. For instance, VMware introduces a per-VM swap device(Banerjee et al. 2014), shared between source and destination, to skip the migration of swapped-out pages. Jo et al.(Jo et al. 2013) propose bypassing cached pages at the source, fetching them directly from network-attached storage instead. Jettison(Bila et al. 2012) presents the idea of partial VM migration, where only the active working set of an idle VM is initially transferred, with the remainder paged on demand. Agile migration(Deshpande et al. 2016) takes a hybrid pre-/post-copy approach, where non-working set pages are swapped to a per-VM swap device, allowing only the working set to be migrated upfront.

Vecycle (Knauth & Fetzer 2015) explores checkpoint reuse, where a VM's previous state is stored on machines it has visited before. When the VM returns, that checkpoint is reused, reducing both traffic and migration time. However, due to the lack of regular snapshot updates, the memory state can diverge significantly, potentially increasing migration overhead.

While all the above techniques aim to reduce migration time primarily by minimizing transferred data, none leverage eliminating redundant fake dirty pages to reduce the total migration time, which could further optimize live migration efficiency.

In parallel, a set of techniques has been proposed to detect resource pressures and alleviate them via VM migration. Solutions such as VMware DRS(Scheduler n.d.) and SandPiper(Wood et al. 2009) monitor host-level resource usage to detect hotspots that trigger migration. Zhang et al.(Zhang et al. 2011) use access-bit scanning to estimate a VM's working set, while Chiang et al.(Chiang et al. 2013) propose resizing VMs based on working set size to enhance consolidation. Overdriver (Williams et al. 2011) tackles transient hotspots with network memory swapping and sustained hotspots via migration.

An enhancement of pre-copy, known as SDPS(VMWare Inc. n.d.), reduces the migration time for write-intensive VMs by throttling vCPUs during migration. However, this comes at the cost of degrading application performance(VMWare Knowledge Base n.d.).

In addition to the aforementioned optimizations, Nathan et al. (Nathan et al. 2016) state that delta compression, as proposed by Zhang et al. and Svärd et al. (Zhang et al. 2010, Svärd et al. 2011), and deduplication techniques, as presented by Deshpande et al. and Wood et al. (Deshpande et al. 2011*b*, Wood et al. 2015), inherently help prevent the transfer of fake dirty pages, thereby reducing total migration time.

Furthermore, Li et al. (Li et al. 2019) proposed a solution to eliminate the transfer of fake dirty pages using secure hashing. Their approach stores the secure hashes of all pages transferred during the initial iteration. In subsequent iterations, each page marked as dirty is compared against its previously stored hash. If the new hash differs from the stored one, the page is treated as genuinely dirty, transferred to the destination, and its hash is updated. If the hashes match, the page is identified as a fake dirty page and is not transferred. In their implementation, Li et al. use SHA1 (SHA1 2024) to compute the hashes and consider identical hashes to indicate unchanged pages, as the probability of an SHA1 collision is less than 10^{-31}—effectively negligible.

3 Research Methodology

This research adopts the Design Science Research (DSR) methodology, a well-established approach for developing and evaluating artifacts aimed at solving real-world problems. DSR is particularly suitable for technical projects where both innovation and evaluation are integral to the research process. In this study, DSR guided the design, implementation, and evaluation of D-Track, a fake dirty page tracking mechanism designed to improve live virtual machine (VM) migration performance.

3.1 Justification for Using Design Science

Design Science Research focuses on creating and evaluating artifacts that offer solutions to identified problems. The primary goal of this study was to solve inefficiencies in QEMU/KVM's dirty page tracking mechanism, specifically the issue of fake dirty page generation. Since the solution required both a novel mechanism (the artifact) and empirical validation, DSR was chosen as the most appropriate methodology.

The research followed the three core cycles of DSR:

- **Relevance Cycle:** The problem of fake dirty pages during Pre-Copy live migration was identified from existing literature and practical challenges faced in data center virtualization. This established the need for a more efficient mechanism for tracking dirty pages.
- **Design Cycle:** The D-Track mechanism was iteratively designed and implemented, with refinements based on technical feasibility and experimental results.
- **Rigor Cycle:** Existing knowledge from prior research on hash-based tracking, silent store instructions, and compression techniques informed the artifact design and evaluation.

4 Design

This section provides the design and implementation details of D-Track, a fake dirty page tracking mechanism to mitigate redundant memory page transfer when live migrating VMs executing memory-intensive workloads.

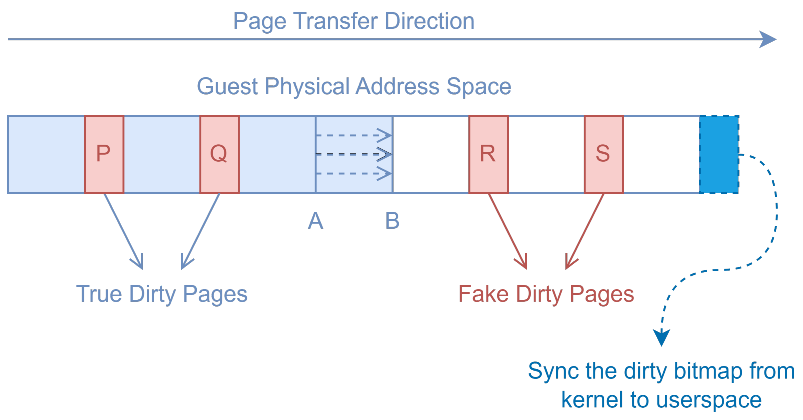

Page Transfer Direction
Guest Physical Address Space
P
Q
R
S
A
B
True Dirty Pages
Fake Dirty Pages
Sync the dirty bitmap from kernel to userspace

Figure 6: Tracking Mechanism of Dirty Pages in KVM Li et al. (2019).

The D-Track mechanism is introduced as the solution to the defective dirty page tracking mechanism of the hypervisor, as mentioned in Section 3.A.2.

The virtualization platform KVM/QEMU QEMU (2024), KVM (2024) can be divided into KVM (kernel module) and QEMU (userspace tool). Both parts contain their dirty bitmap. The dirty bitmap of the kernel traces the write request to a page and marks the page as dirty. This process is continued throughout an iteration, marking all the dirty pages. When the iteration ends, the kernel's dirty bitmap is synced with userspace's dirty bitmap using API: *ioctl*, and the dirty bitmap of the kernel is reset to track the next iteration. In the next iteration, the synced dirty bitmap of the userspace is used to transfer modified pages. Therefore, the tracking of dirty pages is limited to one Pre-Copy round, resulting in fake dirty page generation.

Li et al. Li et al. (2019) used Figure 6 to explain how fake dirty pages are generated due to the tracking mechanism. During the first iteration of Pre-Copy Migration, the pages in the memory region allocated to the VM are transferred by the migration thread to the destination, starting from a low address in a sequential order. Let's assume that during the transfer of pages to the destinations from *address* A to *address* B, four pages, P, Q, R, and S, are modified and marked as dirty in the kernel dirty bitmap. At that moment, since P and Q are already transferred in the ongoing iteration, the modifications done to them are not sent to the destination, making them dirty pages in the second iteration. However, pages R and S are dirtied before they are transferred, as a result, the modified versions of R and S are transferred in this current iteration, although pages R and S are marked as dirty in the kernel's dirty bitmap. When the kernel's dirty bitmap syncs with the userspace's dirty bitmap, pages R and S are selected to resend it in the second iteration. Therefore, in the second iteration, when pages R and S are resent, they already exist in the destination, making pages R and S fake dirty pages.

D-Track is designed to identify pages like R and S and avoid transferring them in the current iteration. Figure 7 provides the D-Track architecture. D-Track's main component

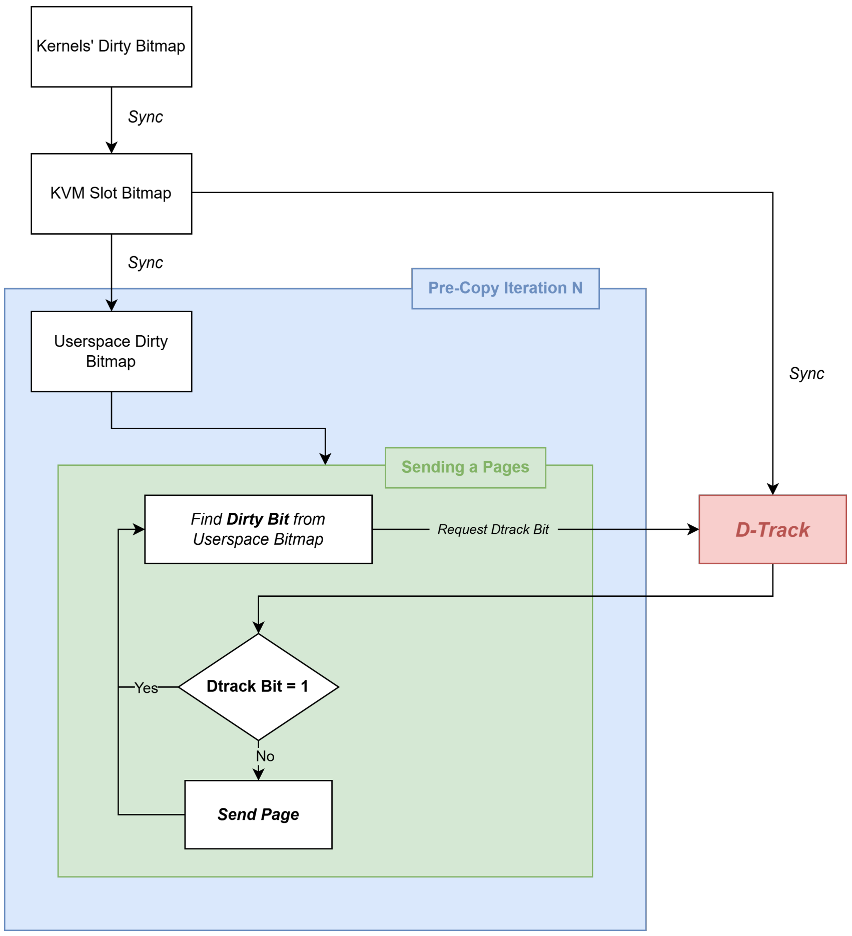


Figure 7: The architecture of D-Track

is keeping track of pages that become dirty during an iteration.

4.1 Why D-Track?

When considering the proposed algorithm of Li et al. Li et al. (2019), it uses secure hashes to avoid transferring fake dirty pages. As mentioned in Section 3.B., the hash value of the 4KB pages selected to send must be calculated to identify the fake dirty pages. Since this is done within the migration thread, it puts an overhead on the migration thread. Li et al. have implemented their solution in QEMU 2.5.1 Li et al. (2019), but with modifications made to the QEMU code, this overhead significantly impacts the Total Migration Time. As shown in Figure 8, the total migration time taken when migrating using Li's algorithm (HBFDP SHA1) is higher than the total migration time taken by Vanilla Pre-Copy.

D-Track provides a mechanism to avoid transferring fake dirty pages without placing an overhead on the migration thread.

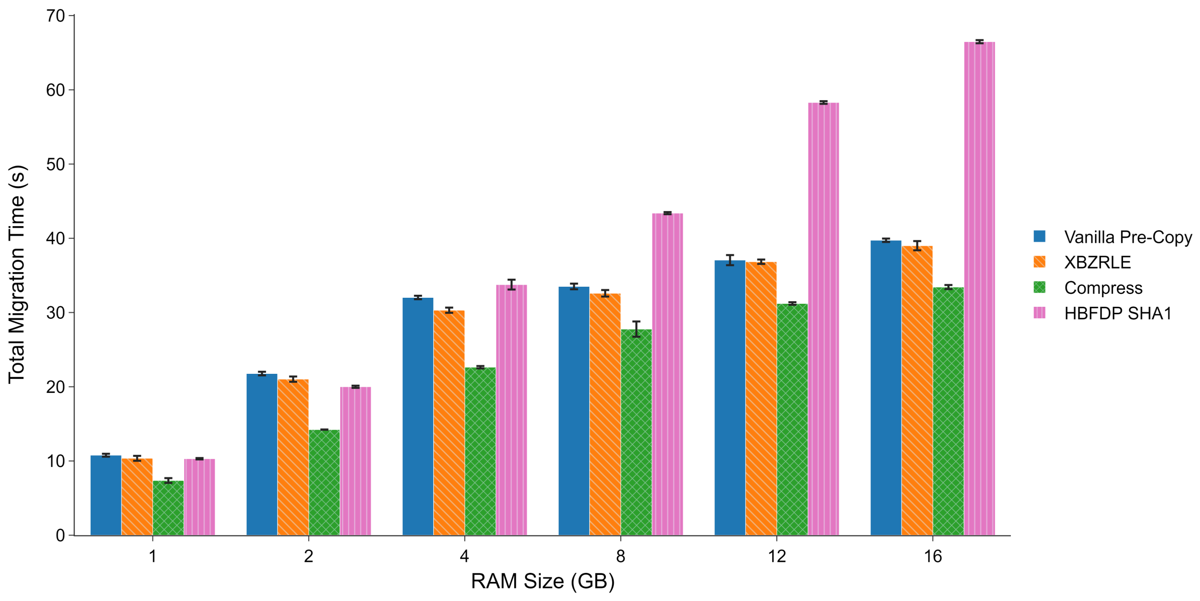


Figure 8: Variation of Total Migration Time with Ram sizes in Vanilla Pre-Copy, XBZRLE, Compression and Li's Algorithm

4.2 Exploration of Optimizations to Dirty Page Tracking Mechanism

Due to the computational overhead placed by the hashing mechanism to identify the fake dirty pages, optimizing the dirty page tracking mechanism in Qemu to prevent Fake Dirty Page transfers as it is one of the reasons for generating fake dirty pages which will be the ideal way to prevent the transferring Fake Dirty Pages without computational overhead on the migration thread.

Initial approach was to find function(s) in QEMU that handles *write requests* issued by the Virtual Machine. After some extensive searching, it was found that there is no single function that all guest writes go through. As a primary developer confirmed in cases where a guest write is directed to host RAM, an optimization is employed within the Just-In-Time (JIT) compilation process. Therefore the writes happen directly from the host code that the JIT generates, without going out to any C code in QEMU itself.

Due to the limitations identified in the initial approach, the next approach was to implement a memory listener method to capture write requests using the SIGSEGV signal issued by the operating system's kernel. The idea was to lock the memory allocated to the Virtual Machine and capture segfault signals issued when Virtual Machine tries to write to the memory. This approach was also ineffective since the signals the Virtual Machine issued can not be captured as a SIGSEGV signal in the QEMU source code.

Following the limitations observed in the first (no single function handling all guest writes due to JIT optimization) and second (inability to capture SIGSEGV signals from VM writes), the D-Track mechanism was identified.

4.3 D-Track's Fake Dirty Tracking

The logic behind D-Track is as follows: at the start of the migration, the D-Track's page tracking mechanism is enabled, but the tracking of dirty pages is not started. This mechanism keeps track of the dirty pages using a bitmap. The initial value of this bitmap is zero, indicating there are no modifications to pages. At the start of an iteration, D-Track starts tracking dirty pages by continuously syncing with the kernel's dirty bitmap. During the iteration, when a page is selected to transfer, the corresponding bit in the D-Track's bitmap is checked. If that bit is one, it means the page is transferring in the next iteration, making it a fake dirty page. And that page is not transferred. Else if that bit is zero, it means the page is not transferred in the next iteration, and the page is approved to transfer in the current iteration.

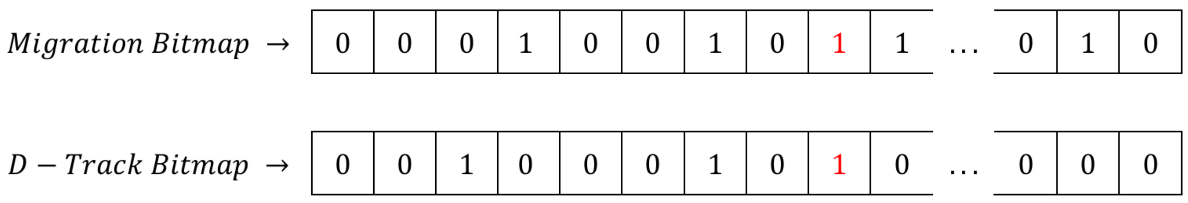


Figure 9: Migration Bitmap and Dtrack Bitmap at moment of transferring Page 9

Lets consider the example shown in Figure 9: when page 9 is selected to transfer in the iteration, the bit value of page 9 has to be selected from the D-Track's bitmap. At that instance, the 9th bit (marked in red) of the D-Track bitmap is one indicating it is a fake dirty page. Hence, page 9 is not transferred.

5 Implementation

KVM/QEMU QEMU (2024), KVM (2024) virtualization platform is used to implement D-Track. KVM is a kernel module that coordinates with QEMU to execute the VM in guest mode (ring 3) using hardware virtualization features. And QEMU manages the userspace associated with each VM, performing device emulation and various management functions like live migration. The QEMU code is modified to implement the D-Track algorithm. The guest OS and applications have not been modified in the implementation.

5.1 Implementation of Li's Algorithm and Its Variations

The algorithm proposed by Li et al. (2019) is taken as a baseline technique to evaluate D-Track along with *Vanilla Pre-Copy*, *XBZRLE*, and *Compression.* This theoretical algorithm will be referred to as the Hash-Based Fake Dirty Prevention (HBFDP) algorithm for the rest of this dissertation. The initial implementation steps were to implement HBFDP in the QEMU code base.

5.1.1 Primary HBFDP Prototype

According to the algorithm given by Li et al. (2019), at the start of the migration, an array to store the hash values of memory pages for each RAMBlock is created with the size of the total number of pages in the RAMBlock. Then, in the initial iteration, the hash of each page is computed before transferring and store it in the specific index (page address) of the corresponding array (created for the RAM block the page belongs to). In subsequent iterations, the hash of each selected dirty page is computed, and the computed hash is compared with the stored previous hash. If the hashes are similar, we mark the page as a fake dirty page and skip the page from transferring. Otherwise, replace the previous hash stored in the array with the current hash and transfer the page to the destination. SHA1 hashing mechanism is used in this primary prototype for the hash computation of the pages.

5.1.2 HBFDP Variations

Next, the two variations of the HBFDP was created by replacing SHA1 hashing with MD5 and Murmur3 hashing. A separate hash type selection API was implemented using QAPI provided by KVM/QEMU; the default hashing was set to SHA1 and enabled to set MD5 and Murmur3 using the API. Therefore, the user has to set the hashing type by sending the request to QMP (QEMU Monitor Protocol) for controlling qemu.

5.2 Implementation of D-Track

D-Track contains two main components: a Dirty page tracking component and a Fake Dirty Identification component. The dirty page tracking component handles the concurrent tracking of dirty pages without placing overhead on the migration thread. The fake dirty identification component identifies the fake dirty pages and avoids transferring them to the destination.

5.2.1 Dirty Page Tracking

D-Track is implemented as a separate thread from the migration thread, which concurrently tracks dirty pages without placing an overhead on the migration thread.

Initially, the D-Track thread is created and put on hold to track dirty pages until an iteration is started. After the initial setup in the migration thread, it starts migrating pages in iterations. When an iteration is started, a signal is sent from the migration thread to the D-Track thread to concurrently track dirty pages while the pages are transferred from the source host to the destination host. D-Track uses a separate dirty bitmap, called *dtrack bmap*. The tracking is done by continuously syncing the *dtrack bmap* with the kernel's dirty bitmap using an *ioctl()* call. After the final page is sent in the iteration, the synchronization process of *dtrack bmap* is stopped, and the *dtrack bmap* is cleaned to use in the next iteration. This process happens in all the iterations, excluding the last iteration where the synchronization of *dtrack bmap* is disabled.

5.2.2 Selection of Pages to Send

Algorithm 1 Identifying Fake Dirty Pages in Migration Thread

```
Input: last_stage      // Flag to indicate last iteration
       page            // Page Selected to Transfer
Output: The page sending status.
1: procedure SELECTION(last_stage, page)
2:     if last_stage is false then
3:         bit ← dtrack_dirty_page_state(page)
4:         if bit == 1 then
5:             skip page transfer
6:         end if
7:     end if
8: end procedure
```

As mentioned in Section 3.B.1, the D-Track thread continuously keeps track of the pages that are dirtied during the migration. While the page tracking mechanism runs separately, when a page is selected to transfer, the corresponding bit in the *dtrack bmap* is checked to identify whether the page is a fake dirty page in the next iteration, as shown in the algorithm 1. If the corresponding *bit* has a value of 1, that means the page is modified and is a fake dirty page in the next iteration, hence skipping the page from transferring. Else if the corresponding *bit* is 0, the page is transferred in the current iteration.

6 Evaluation

6.1 Evaluation Metrics

This section provides the D-Track's experimental results evaluated on the following metrics:

- **Total Migration Time:** Total time to migrate a VM from the source host to the destination host.
- **Downtime:** Time Duration where the VM is Suspended to transfer CPU/IO states.
- **Application performance:** Performance of the applications running in the VM during migration.

D-Track is evaluated with and without incorporating different optimization techniques (Compression and XBZRLE) against *Vanilla Pre-Copy*, *XBZRLE enabled Pre-Copy*, *Compression enabled Pre-Copy*, and *algorithm proposed by Li et al. Li et al. (2019)*. Each data point is an average of three rounds of experiment.

6.2 Testbed Setup

The setup testbed contains two physical servers and an NFS server. The two physical servers act as source and destination servers, each equipped with a 48-core Intel Xeon E5-2697 v2 @ 3.500GHz processor with 314.8 GiB of RAM. The two servers and the NFS server are interconnected with Gigabit Ethernet. The host OS is Ubuntu 22.04.3 LTS x86_64 Server, and the chosen hypervisor is QEMU-KVM v8.1.2, running various VMs utilizing Linux-based OSs (guest OS). Figure 10 presents the high-level architecture of the testbed used for data collection.

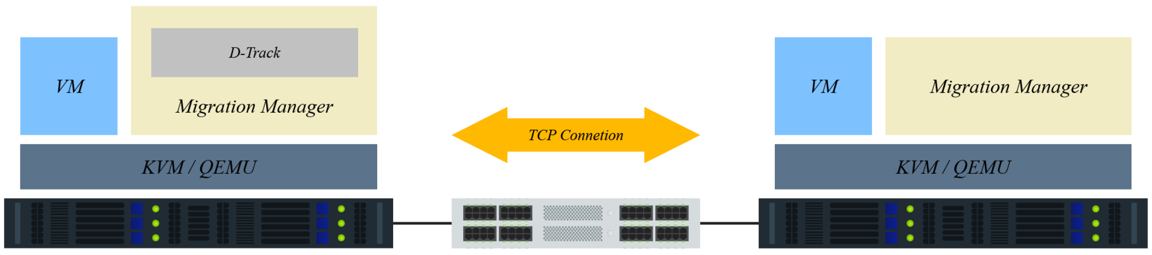


Figure 10: High-level architecture of the Testbed

6.3 Evaluation on Variations of HBFDP Algorithm

The initial evaluation was carried out on the prototypes implemented for variations of the HBFDP algorithm. The Figure 11 shows the total time migration of *Vanilla Pre-Copy*, *XBZRLE*, *Compression*, HBFDP with SHA1 (*Li's algorithm Li et al. (2019)*), HBFDP with MD5 and HBFDP with Murmur3. When considering the graph, it can be seen that, as mentioned in Section 3.1, the total migration time taken when migrating using HBFDP with SHA1 (*Li's algorithm Li et al. (2019)*) is higher than the total migration time taken by Vanilla Pre-Copy. The variations of HBFDP algorithm with MD5 and Murmur3 hashing instead of SHA1 also has higher total migration time than *Vanilla Pre-Copy*. Hence, it can be deduced that the computational overhead placed by the HBFDP algorithm on the migration thread is irrespective of the hashing technique.

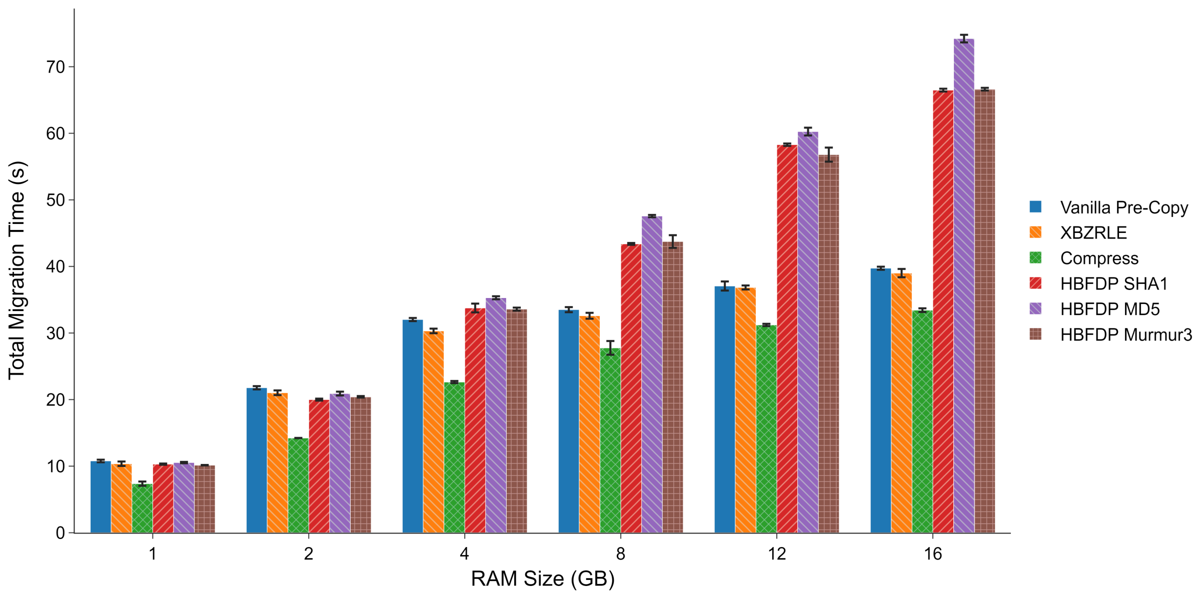


Figure 11: Total Migration Time for HBFDP Variations

Next the HBFDP algorithm was combined with XBZRLE and compression optimizations and then evaluated. The Figure 12 shows the total time migration of *Vanilla Pre-Copy*, *XBZRLE*, *Compression*, HBFDP with SHA1 combined with Compression, HBFDP with MD5 combined with Compression and HBFDP with Murmur3 combined with Compression. When considering the HBFDP variations combined with compression optimization, they all have a higher total migration than the *Vanilla Pre-Copy*.

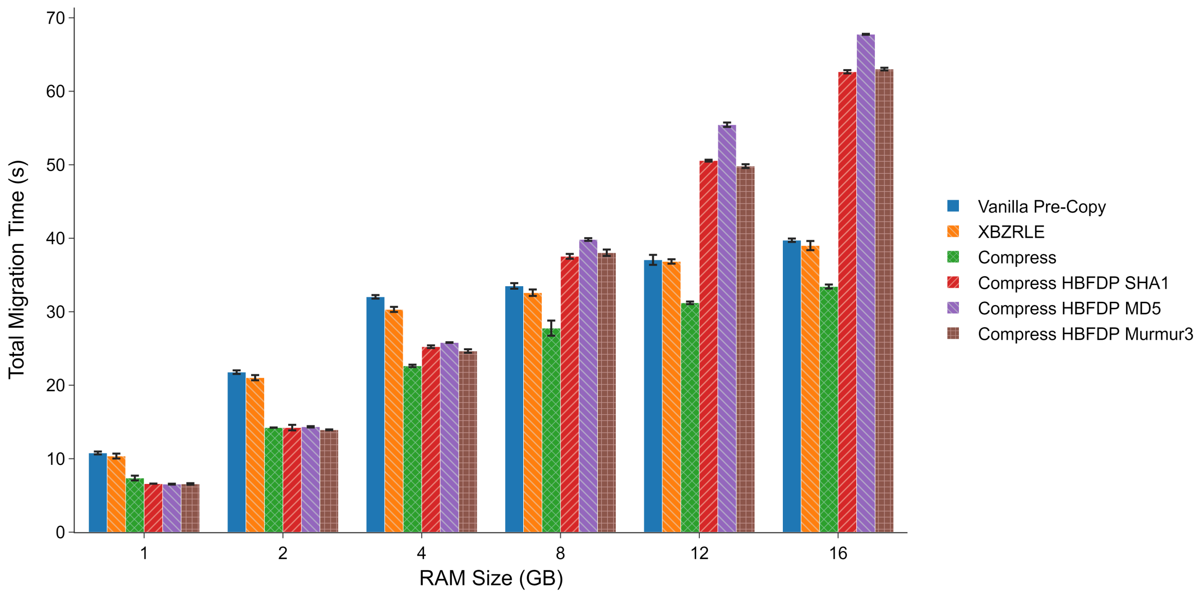


Figure 12: Migration Performance for various techniques

The Figure 13 shows the total time migration of *Vanilla Pre-Copy*, *XBZRLE*, *Compression*, HBFDP with SHA1 combined with XBZRLE, HBFDP with MD5 combined with XBZRLE and HBFDP with Murmur3 XBZRLE with Compression. Similarly, when considering the HBFDP variations combined with XBZRLE optimization, they all have a higher total migration than the *Vanilla Pre-Copy*. Hence, the computational overhead placed by the HBFDP algorithm on the migration thread is still not reduced by incorporating different optimizations to HBFDP.

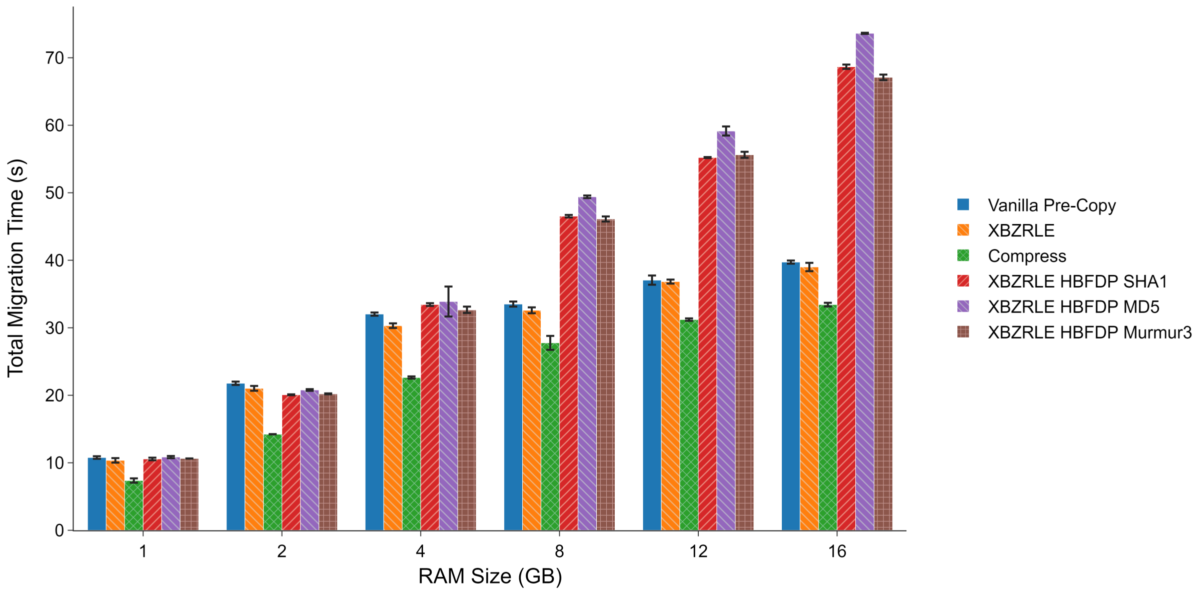


Figure 13: Migration Performance for various techniques

6.4 Impact on Memory-Intensive Workload

The performance of D-Track for memory-intensive workloads was captured using Memcached Memcached (2024). The Memcached workload was configured by allocating 4 threads to handle requests, and 90% of the VM memory size was allocated as Memcached cache. The Memaslap benchmark was configured in a client server to send requests to the Memcached in the VM with set and get the ratio of 1:0.

Figure 14 shows the total time migration of *Vanilla Pre-Copy*, *XBZRLE*, *Compression*, HBFDP with SHA1 (*Li's algorithm Li et al. (2019)*), D-Track, D-Track combined with Compression and D-Track combined with XBZRLE. As the graph depicts, D-Track and D-Track combined with Compression perform better than *Vanilla Pre-Copy*, *XBZRLE*, HBFDP with SHA1 and both techniques has more than 10% improvement in total migration time when compared to *Vanilla Pre-Copy* when migrating VM executing memory-intensive workload. When considering D-Track combined with XBZRLE, it performs better than *Vanilla Pre-Copy*, *XBZRLE*, and HBFDP with SHA1 for VM RAM sizes 1GB, 2GB, 4GB, and 8GB, but there is no improvement in total migration time for VM RAM sizes more than 8GB when compared with *Vanilla Pre-Copy* and *XBZRLE*.

Although D-Track performs better compared to *Vanilla Pre-Copy*, Compression optimization performs better than D-Track, although it does not prevent the transfer of Fake Dirty Pages. But it can be seen that, the D-Track combined with Compression performs better than Compression Optimization when migrating VM executing memory-intensive

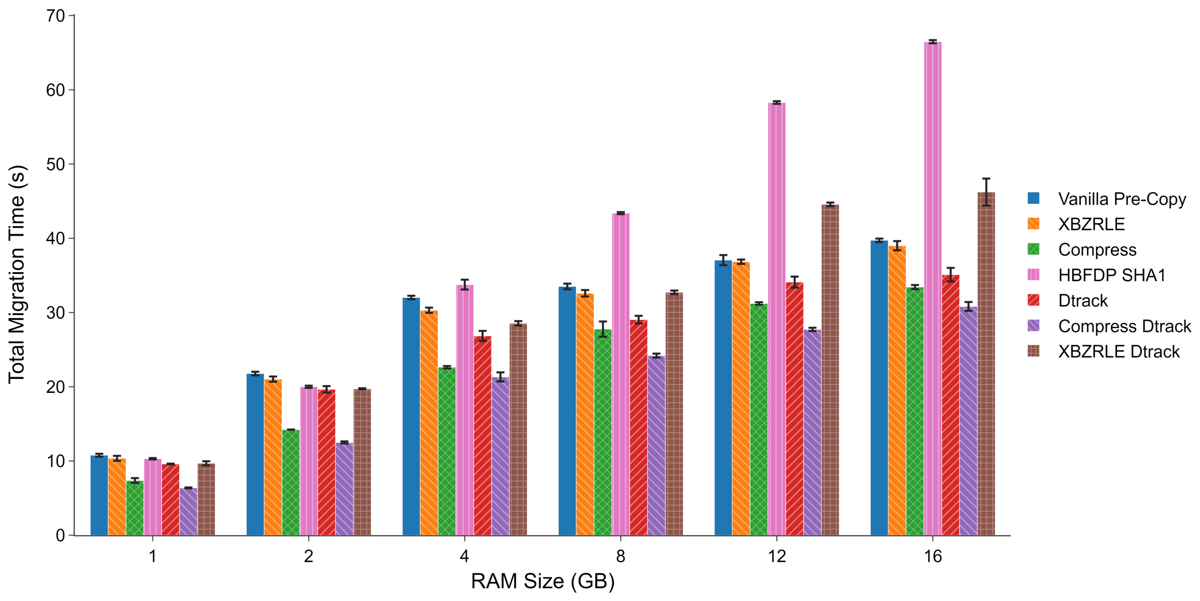


Figure 14: Migration Performance for various techniques

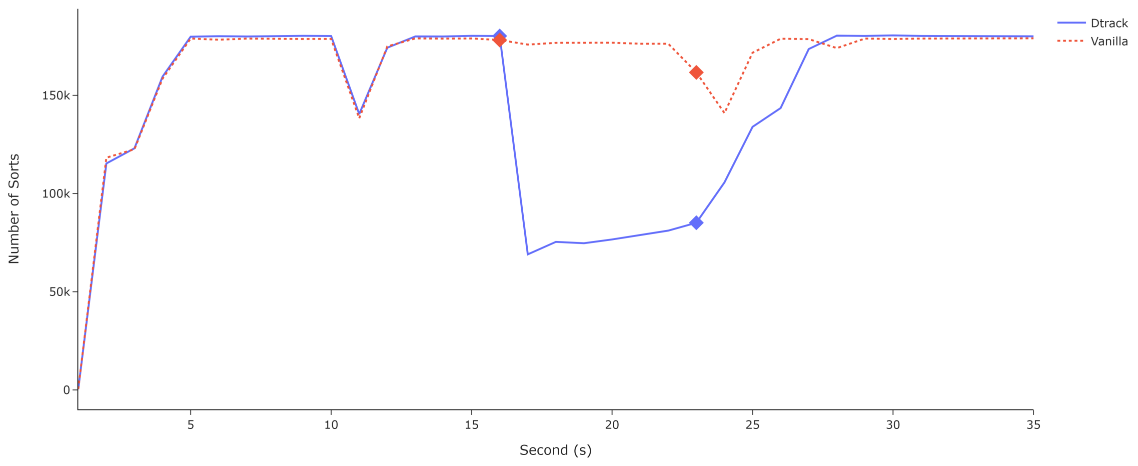
Ram	XBZRLE	Compression	Li's Algorithm	D-Track	D-Track + XBZRLE	D-Track + Compression
1	3.88%	31.62%	4.41%	10.92%	5.66%	**40.87%**
2	3.45%	34.66%	8.16%	9.74%	9.19%	**42.62%**
4	5.36%	29.33%	-5.41%	16.16%	6.46%	**33.41%**
8	2.75%	17.14%	-29.47%	13.33%	1.51%	**27.80%**
12	0.58%	15.77%	-57.29%	7.98%	-20.25%	**25.19%**
16	1.84%	15.86%	-67.33%	11.65%	-13.71%	**22.44%**

Table 1: Total Migration Time Improvement Percentage Compared to Vanilla Pre-Copy

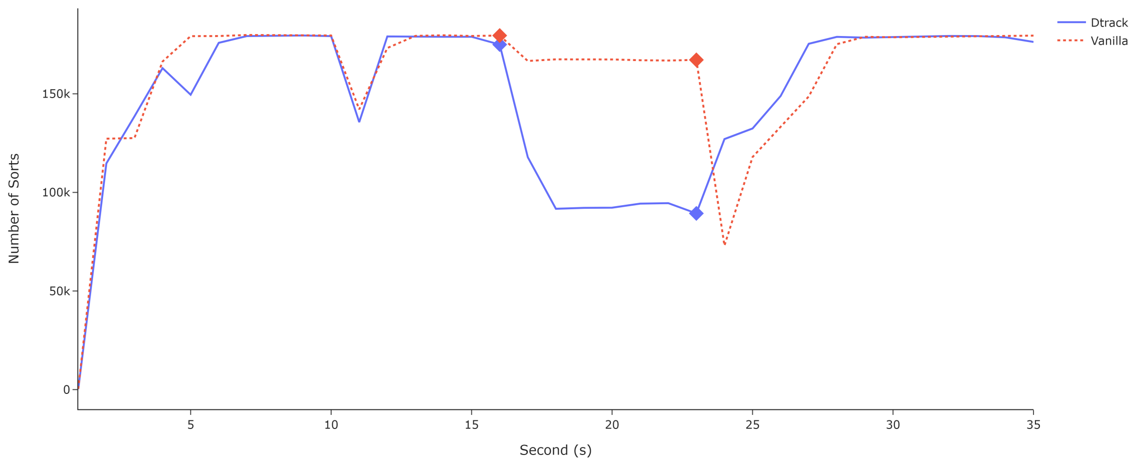
workload. The results in the Table 1 presents the total migration time improvement percentage of *XBZRLE*, *Compression*, and *algorithm proposed by Li et al. Li et al. (2019)*, D-Track, D-Track combined with Compression and D-Track combined with XBZRLE techniques compared to *Vanilla Pre-Copy*. D-Track combined with the Compression technique shows the highest improvement percentage (over 20% upto 40%). These results highlight that Dtrack reduces the total migration time and performs better compared to other techniques when migrating VMs running memory-intensive workloads.

6.5 Impact on CPU-intensive Workload

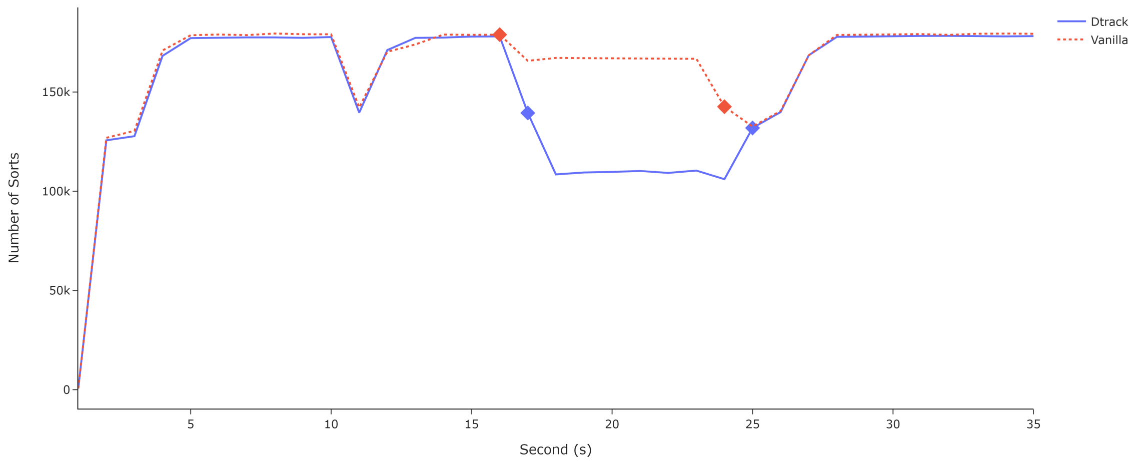
This section shows how D-Track affects the performance of CPU-intensive applications executing in a VM using Quicksort to determine the impact of D-Track on the applications executing on the VM. The Quicksort benchmark repeatedly allocates 512MB of memory, writes random integers to the allocated memory, and performs quicksort on the values. Figure 15 shows the number of sorts performed per second for Vanilla Pre-Copy and D-Track version 2 during migration. As seen in the Figure 15 (a) to (f), the D-Track technique reduces the Number of Sorts performed during the migration. According to the Table 2, from 1GB to 16GB ram sizes, the percentage of reduction of the number of sorts decreases from 30% to 15%. This degradation of application performance is due to the continuous invoking of *ioctl()* call in the dtrack thread.



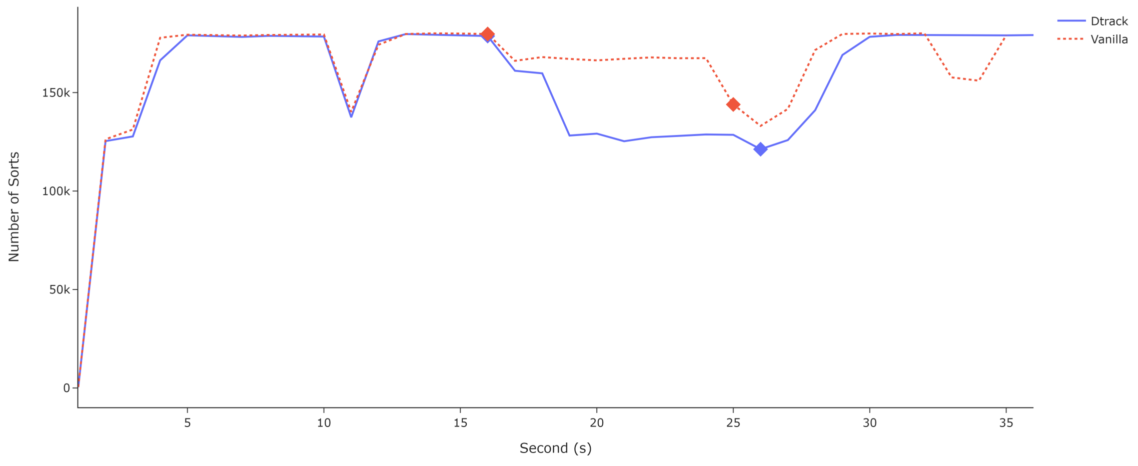
(a) RAM 1GB



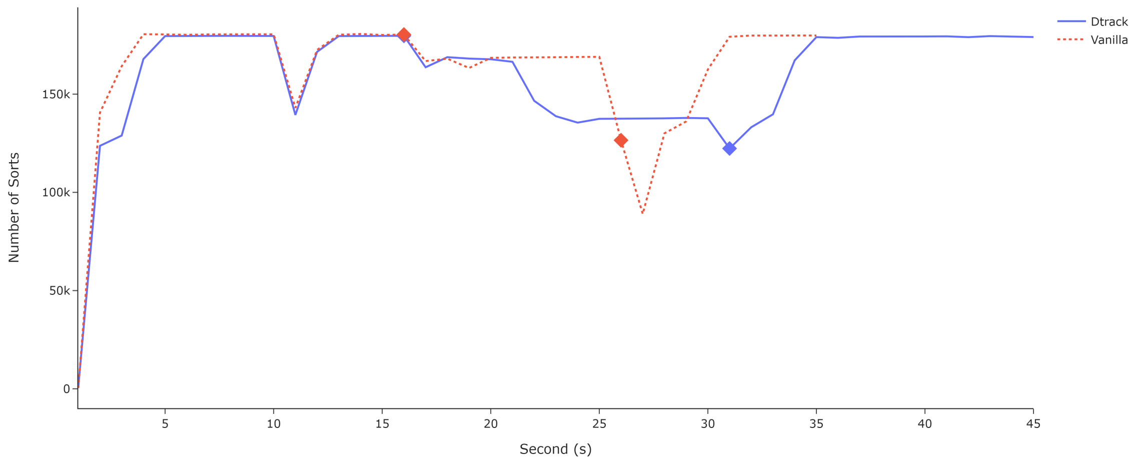
(b) RAM 2GB



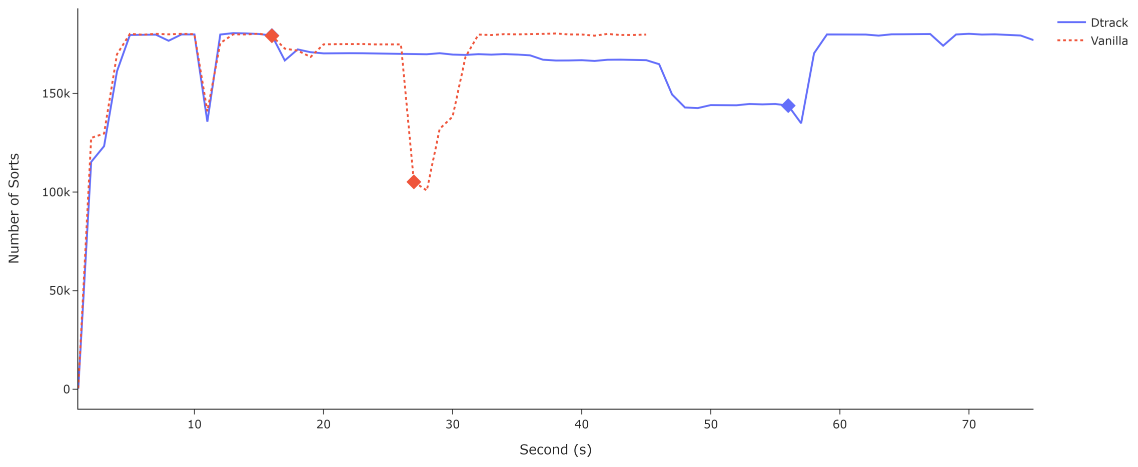
(c) RAM 4GB



(d) RAM 8GB



(e) RAM 12GB



(f) RAM 16GB

Figure 15: Impact of D-Track on the CPU-intensive workload *(The start and the end of the migration are denoted by the Markers)*

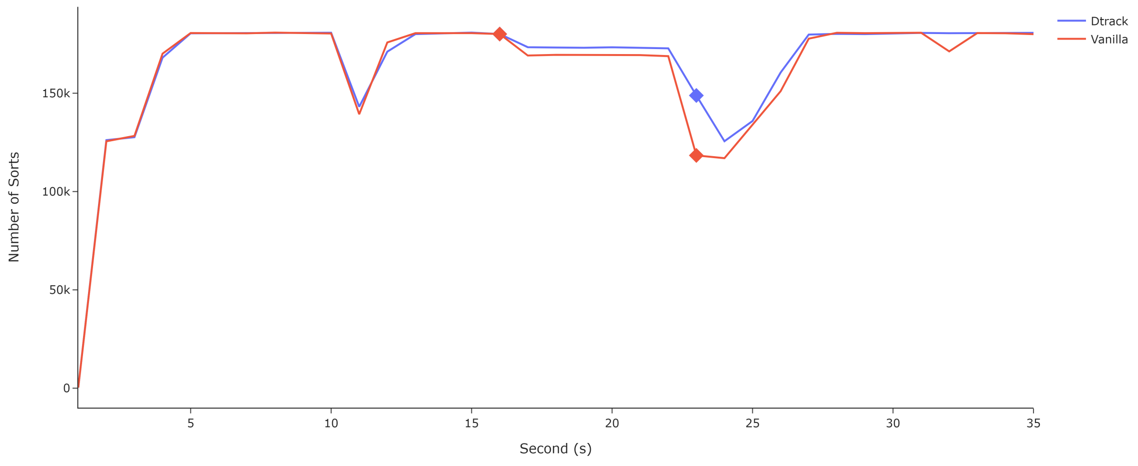
Ram Size	Reduction in Average # of Sorts per Second
1	32.29%
2	32.00%
4	25.38%
8	22.98%
12	15.34%
16	16.38%

Table 2: Reduction in Number of Sorts per Second in Quicksort Algorithm in D-Track Compared to Vanilla Pre-Copy

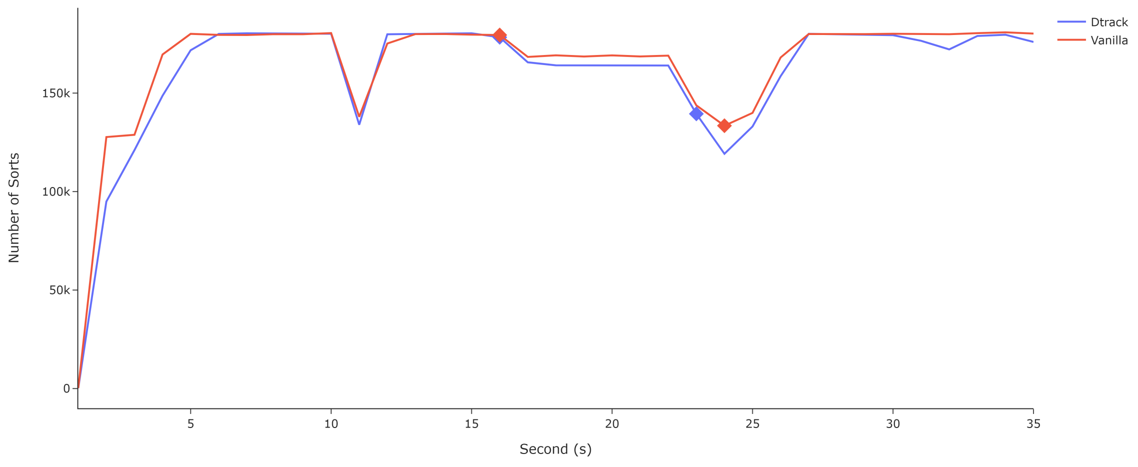
To overcome this impact on application performance, a delay is introduced between *ioctl()* calls in the dtrack thread. This delay reduces the number of *ioctl()* calls but ensures the tracking of fake dirty pages as before. Figure 16 shows the number of sorts performed per second for Vanilla Pre-Copy and D-Track version 2 during migration. Accordingly, as the results in Table 3 show, it can be seen that Dtrack version 2 does not have an observable impact on CPU-intensive workloads.

Ram Size	Reduction in Average # of Sorts per Second
1	-7.54%
2	1.93%
4	5.98%
8	-3.86%
12	1.61%
16	1.41%

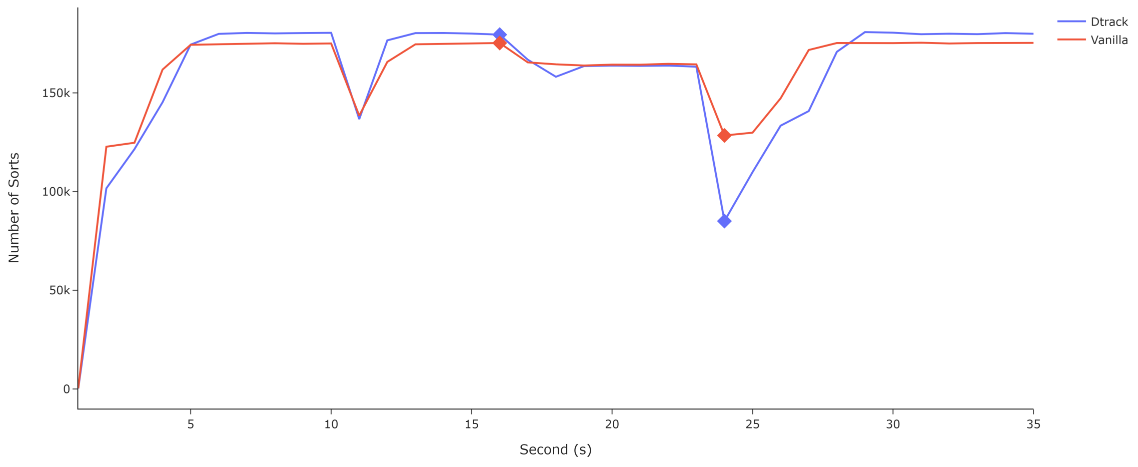
Table 3: Reduction in Number of Sorts per Second in Quicksort Algorithm in D-Track Version 2 Compared to Vanilla Pre-Copy



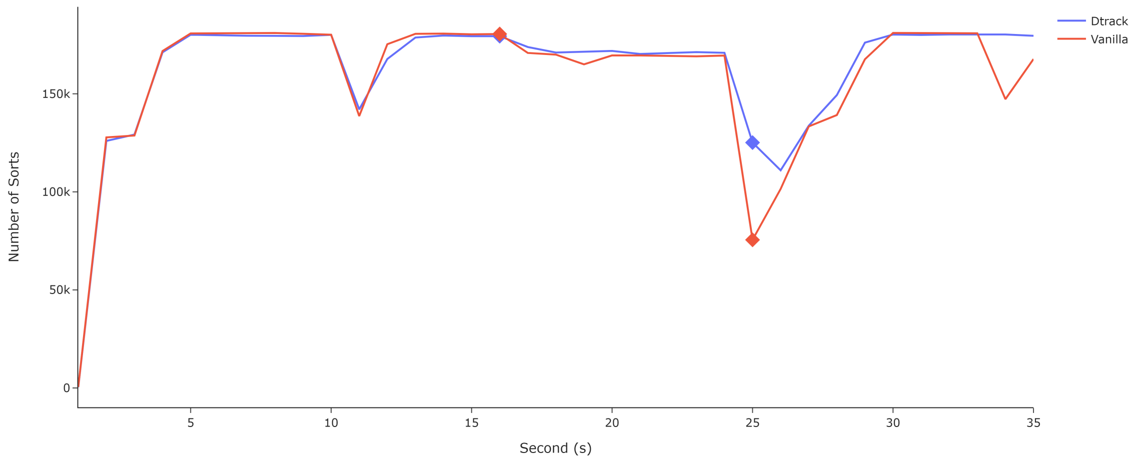
(a) RAM 1GB



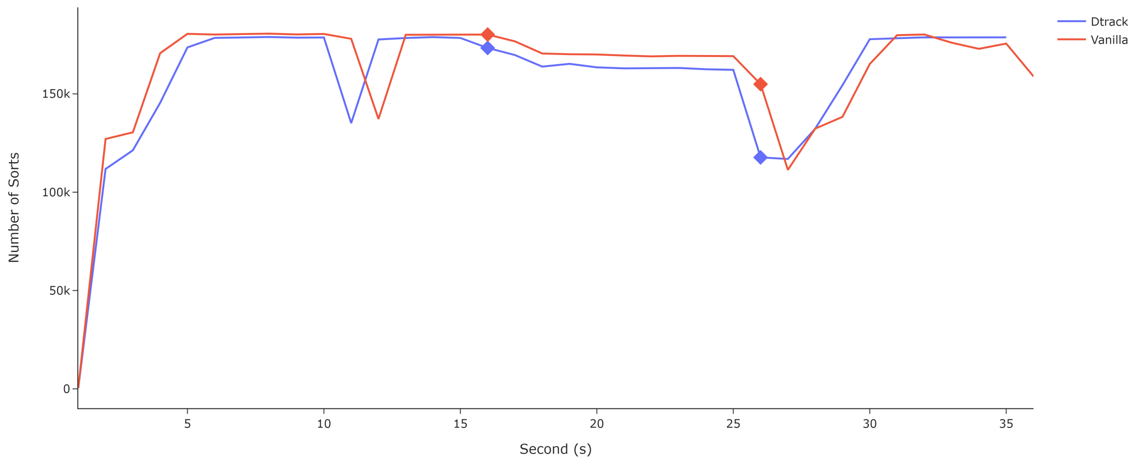
(b) RAM 2GB



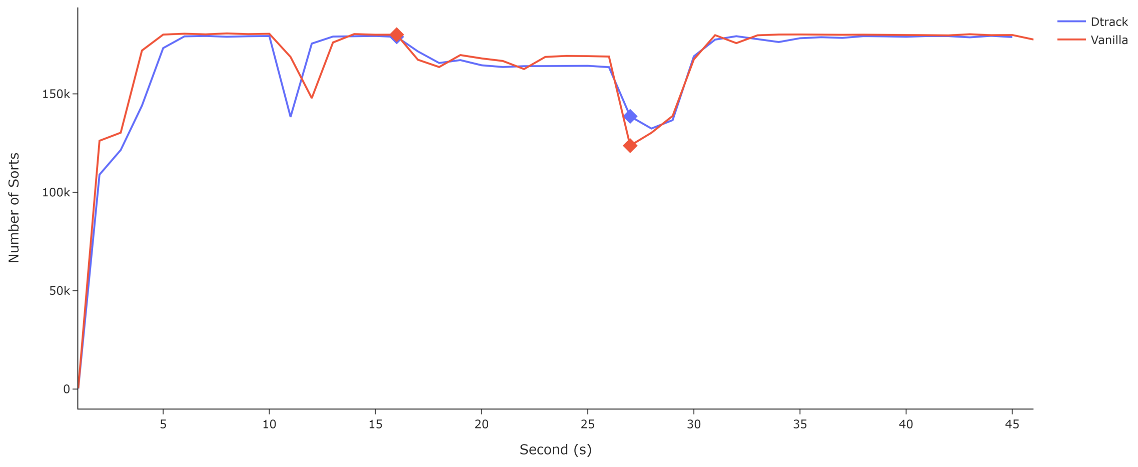
(c) RAM 4GB



(d) RAM 8GB



(e) RAM 12GB



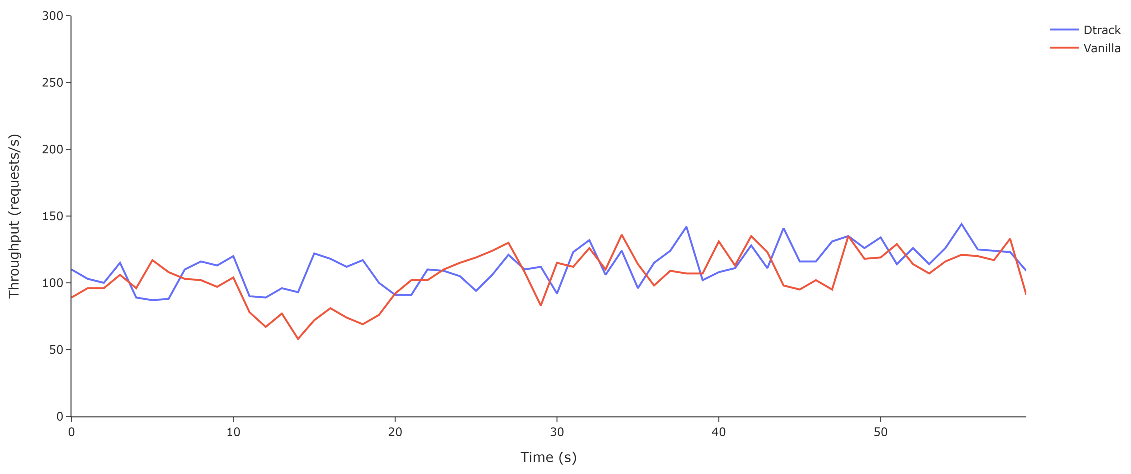
(f) RAM 16GB

Figure 16: Impact of D-Track Version 2 on the CPU-intensive workload *(The start and the end of the migration are denoted by the Markers)*

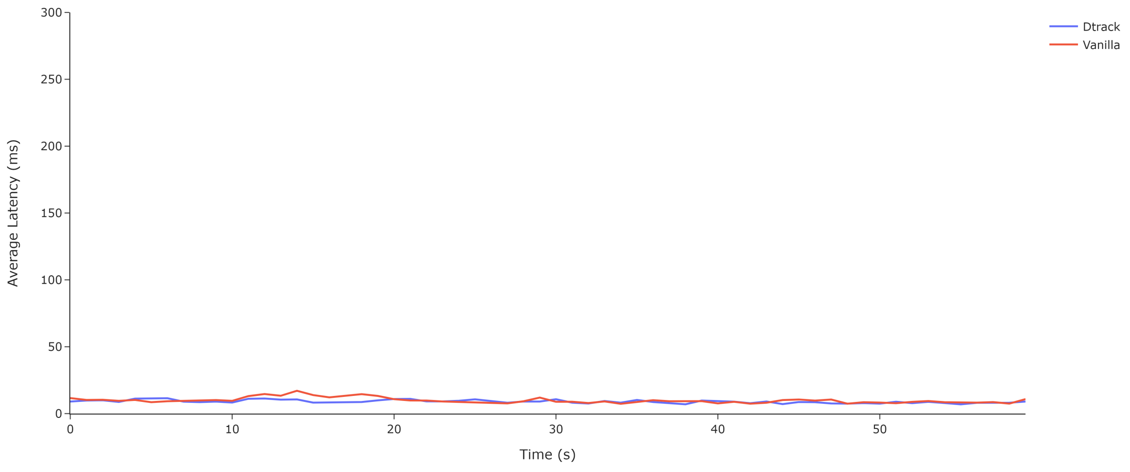
6.6 Impact on Multiple-Intensive Workload

The impact on application throughput and latency when using D-Track for multiple-intensive workloads that are (network, CPU, and memory intensive) was captured using Yahoo Cloud Serving Benchmark (YCSB) Benchbase (2024) and PostgreSQL PostgreSQL (2025). The YCSB benchmark was executed in the VM, and PostgreSQL was hosted in a client server. The PostgreSQL first loads its database. Then, the YCSB client queries data using update operations (50% read & 50% update) when the VM is migrated from the source host to the destination host.

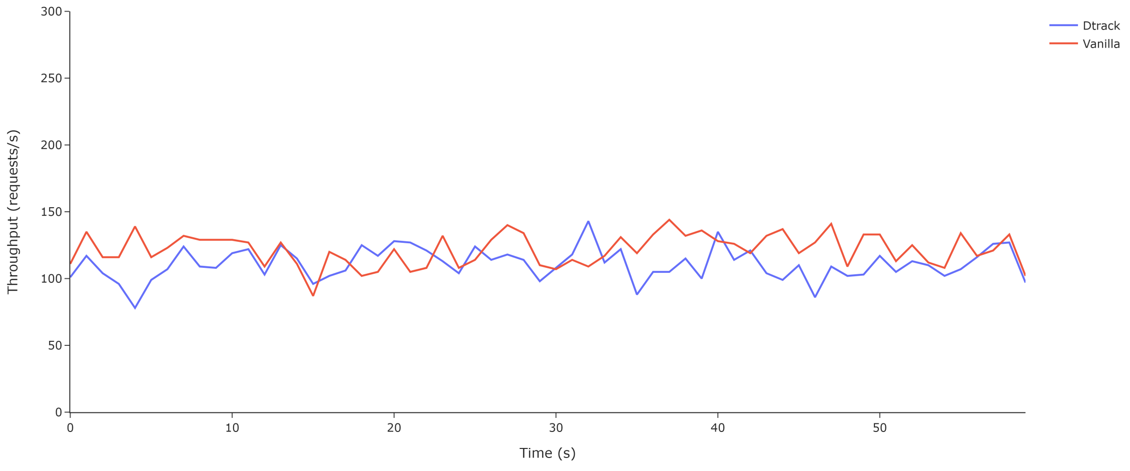
Figure 17 shows the throughput and latency variation. When considering both throughput and latency, the D-Track version 2 performs similarly to Vanilla Pre-Copy without any considerable application performance degradation.



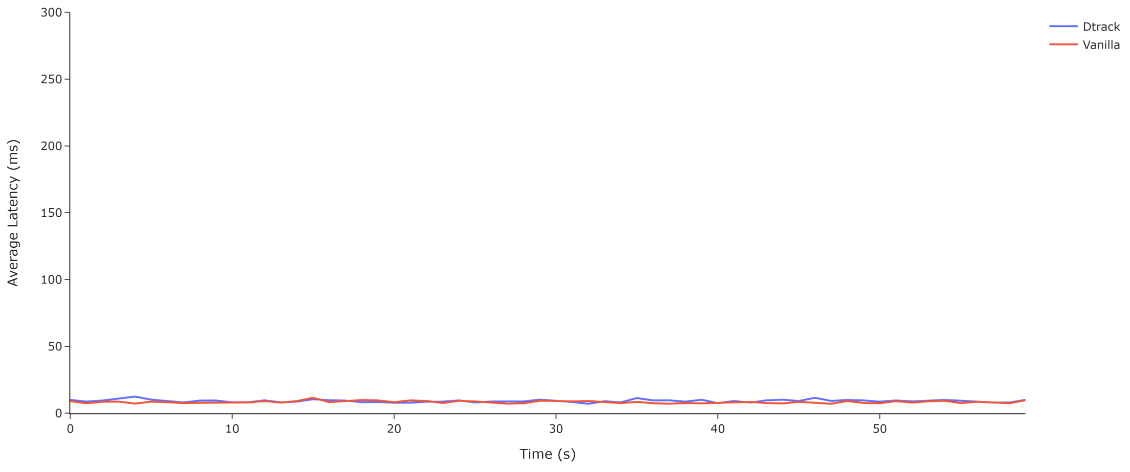
(a) Throughput - RAM 1GB



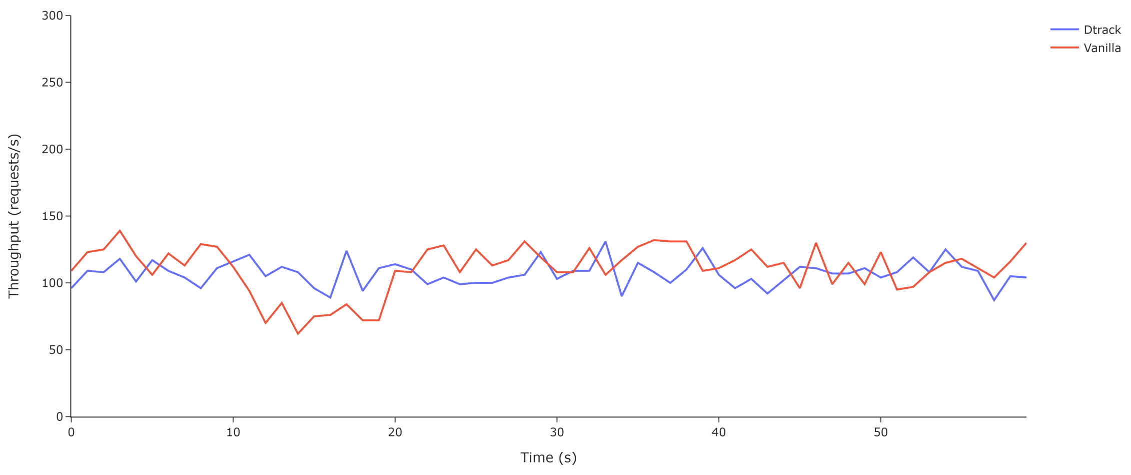
(b) Latency - RAM 1GB



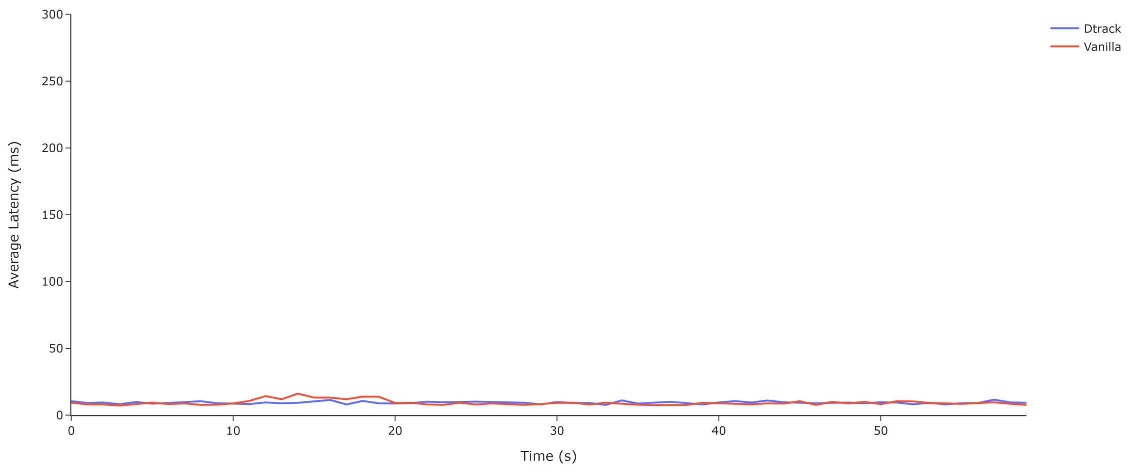
(c) Throughput - RAM 2GB



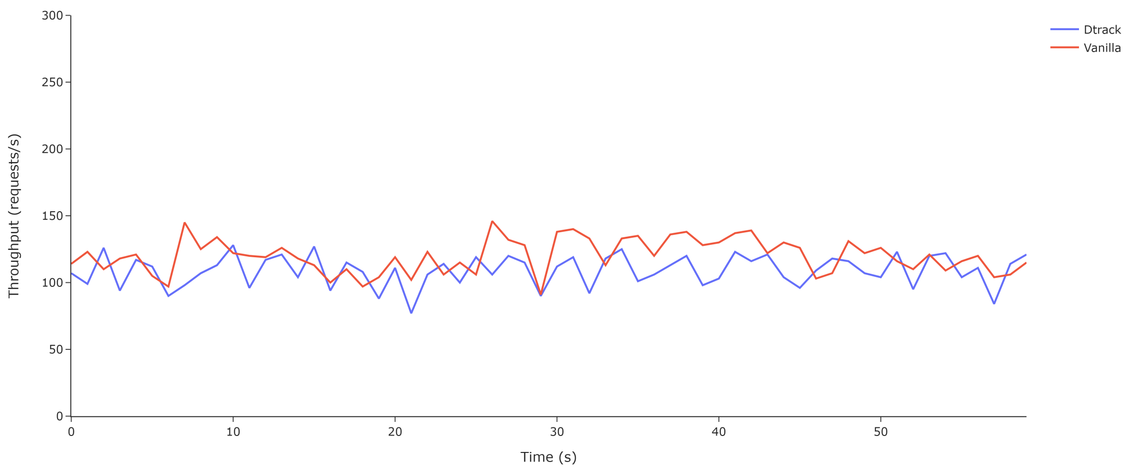
(d) Latency - RAM 2GB



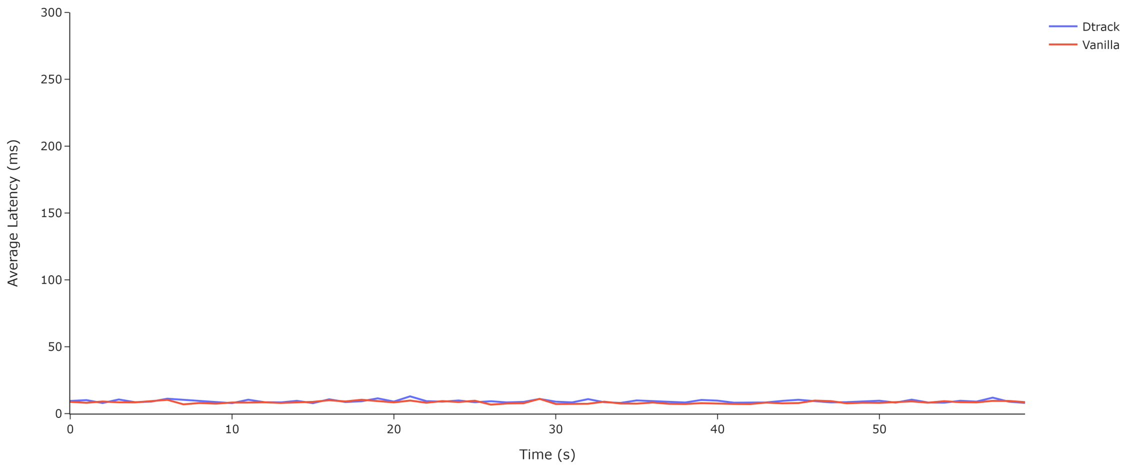
(e) Throughput - RAM 4GB



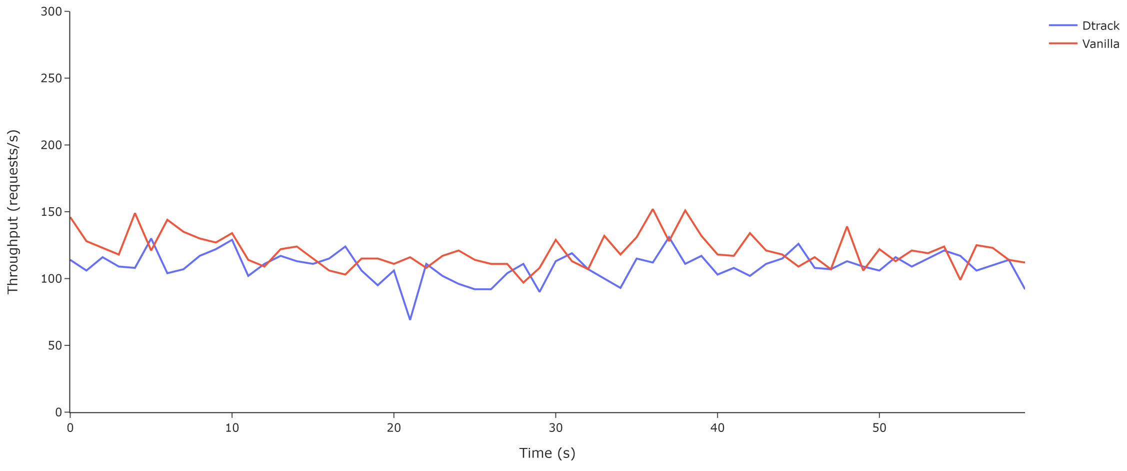
(f) Latency - RAM 4GB



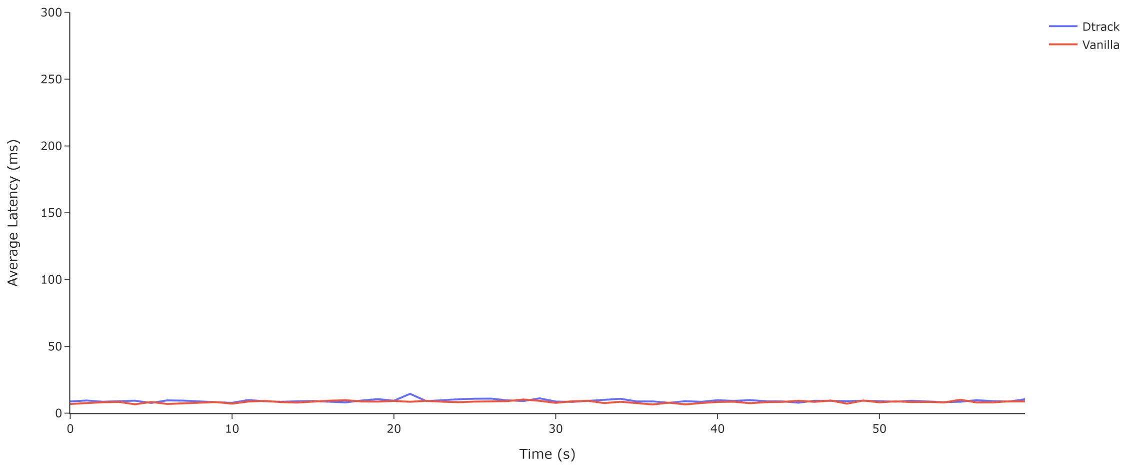
(g) Throughput - RAM 8GB



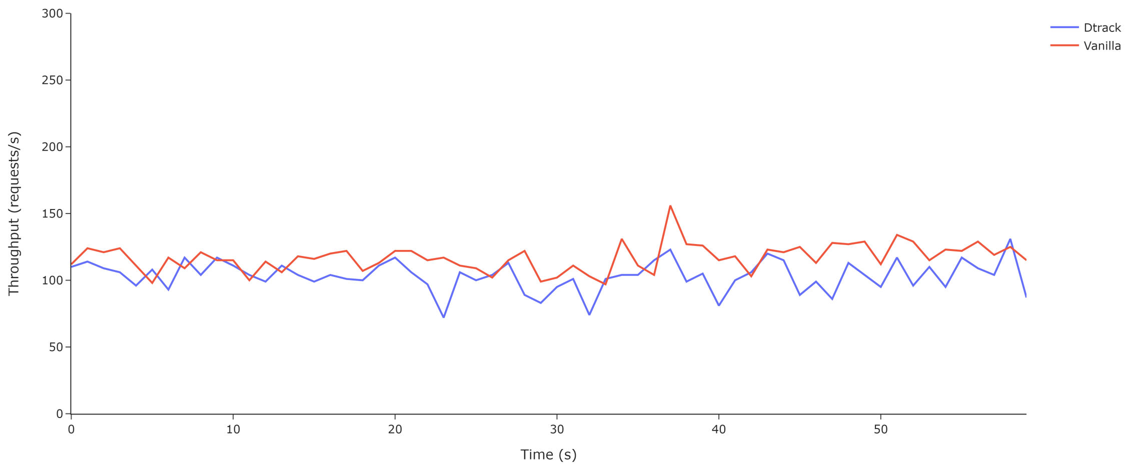
(h) Latency - RAM 8GB



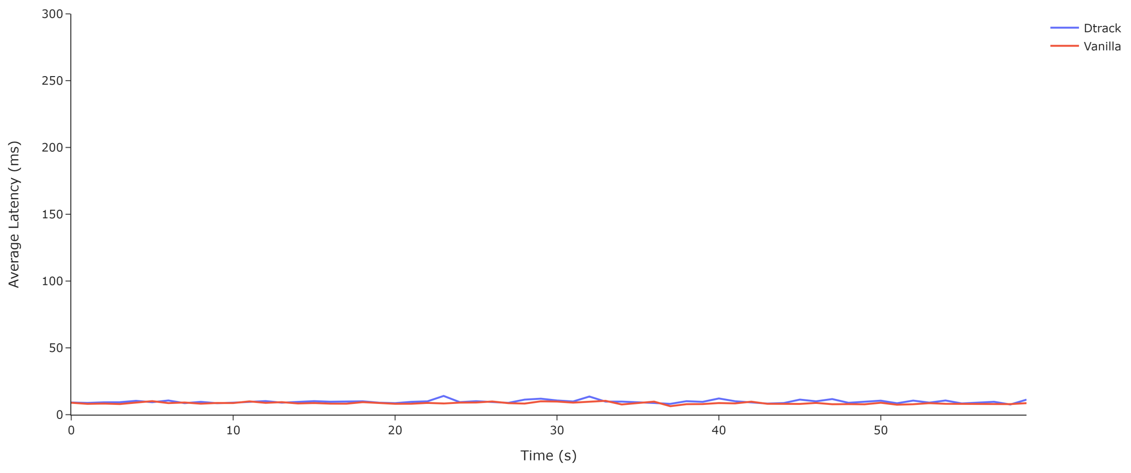
(i) Throughput - RAM 12GB



(j) Latency - RAM 12GB



(k) Throughput - RAM 16GB



(l) Latency - RAM 16GB

Figure 17: Impact of D-Track Version 2 on the Multiple-intensive workload

7 Discussion

The evaluation of the HBFDP algorithm variations and the proposed D-Track mechanism provides critical insights into their effectiveness in optimizing live VM migration by mitigating fake dirty pages, particularly under different workload conditions. The results highlight both the strengths and limitations of these techniques, offering a foundation for further improvements in migration efficiency and application performance when migrating VMs with memory-intensive workloads.

7.1 Computational Overhead of HBFDP and Its Variations

The initial evaluation of HBFDP with different hashing techniques (SHA1, MD5, and Murmur3) revealed a consistent increase in total migration time compared to Vanilla Pre-Copy. This suggests that the computational overhead introduced by the hashing mechanism in HBFDP is significant enough to offset any potential benefits from reducing redundant page transfers. Notably, even when combined with existing optimizations like XBZRLE and compression, HBFDP still underperformed compared to Vanilla Pre-Copy. This indicates that the primary bottleneck lies in the hashing and page-tracking mechanism itself, rather than the data transfer optimizations.

A critical observation here is that the choice of hashing algorithm (SHA1 vs. MD5 vs. Murmur3) did not significantly alter the performance, implying that the overhead is intrinsic to the HBFDP approach rather than being hash-specific. This raises questions about the efficiency of HBFDP algorithm, where the cost of continuous hashing outweighs the benefits of reduced network transmission.

7.2 D-Track's Superiority in Memory-Intensive Workloads

In contrast to HBFDP, D-Track demonstrated a substantial improvement in total migration time for memory-intensive workloads, particularly when combined with compression optimization. The results showed up to a 40% reduction in total migration time compared to Vanilla Pre-Copy, outperforming both XBZRLE and standalone compression optimizations. This improvement can be attributed to D-Track's ability to efficiently track fake dirty pages, thereby reducing unnecessary data transfers without incurring the same computational overhead as HBFDP.

Interestingly, while compression alone also reduces migration time, D-Track combined with compression delivered even better results. This suggests that fake dirty page elimination and compression are complementary optimizations, where D-Track reduces the volume of data that needs to be compressed, further enhancing efficiency. However, when D-Track was combined with XBZRLE, the benefits diminished for VM sizes beyond 8GB. This could be due to XBZRLE's inefficiency in handling larger memory footprints or increased computational overhead when paired with D-Track's tracking mechanism.

7.3 Mitigating Performance Degradation in CPU-Intensive Workloads

The initial implementation of D-Track introduced a noticeable performance degradation (up to 30% reduction in Quicksort operations per second) for CPU-intensive workloads.

This was primarily due to frequent *ioctl()* system calls, which disrupted the VM. However, by introducing a controlled delay between *ioctl()* calls in **D-Track version 2**, this overhead was significantly reduced, bringing performance degradation down to negligible levels.

This finding is crucial because it demonstrates that real-world VM migrations must balance tracking accuracy with computational intrusiveness. While aggressive page tracking improves migration efficiency, it must not come at the cost of application performance—especially for latency-sensitive or CPU-bound workloads. The success of D-Track version 2 in mitigating this issue highlights the importance of adaptive synchronization mechanisms in live VM migration.

7.4 Robustness in Multiple-Intensive Workloads

For workloads that are simultaneously network, CPU, and memory-intensive (e.g., YCSB with PostgreSQL), D-Track version 2 performed comparably to Vanilla Pre-Copy in terms of throughput and latency. This indicates that the optimizations introduced in D-Track do not introduce significant overhead in complex, multi-threaded environments. Given that real-world cloud workloads often exhibit such mixed characteristics, this result strengthens the case for D-Track's practical applicability.

8 Conclusion

This dissertation presents D-Track, a fake dirty page tracking mechanism to mitigate redundant memory page transfer when migrating VMs executing memory-intensive workloads. D-Track solves the inherent problem in QEMU: defects in the dirty page tracking mechanism by introducing a dirty page tracking mechanism during Pre-Copy iterations without placing an overhead on the migration thread. The dissertation presented the design, implementation, and evaluation of D-Track on the KVM/QEMU platform.

8.1 Addressing Research Questions

8.1.1 Research Question 1

The first research question focused on **how to mitigate redundant data transfers by identifying fake dirty pages with minimal overhead while solving the limitations of QEMU's existing dirty page tracking mechanism**.

The proposed D-Track mechanism introduces an efficient bitmap-based tracking mechanism that identifies fake dirty pages. D-Track avoids fake dirty page transfers without introducing excessive computational overhead. Key findings are as follows: HBFDP, despite its theoretical benefits, introduced significant hashing overhead, increasing migration time compared to Vanilla Pre-Copy. D-Track, in contrast, achieved up to **40%** reduction in total migration time for memory-intensive workloads by eliminating fake dirty pages without relying on costly hashing. Also, the performance degradation of D-Track on CPU-intensive workloads was less than **32%**, which is considerable when compared with the Vanilla Pre-Copy. This performance degradation was reduced to **1%** by modifying the D-Track's bitmaps synchronization by adding a delay between *ioctl()* calls.

This research demonstrates that bitmap-based tracking is more efficient than hash-based approaches for fake dirty page detection, as it avoids computational overhead while still mitigating redundant transmissions.

8.1.2 Research Question 2

The second research question focused on **how to reduce the data transferring load in Pre-Copy Migration to improve the performance of memory-intensive workloads**.

Since D-Track reduces the number of pages transferred, applying compression on the remaining data further improves efficiency. This hybrid approach achieved the best overall performance, outperforming standalone compression or XBZRLE. Key findings are as follows: D-Track combined with Compression reduced migration time by **20%-40%** compared to Vanilla Pre-Copy for memory-heavy workloads like Memcached. D-Track combined with XBZRLE was less effective than D-Track combined with compression, particularly in larger VMs, due to its higher CPU overhead. D-Track alone still outperformed HBFDP and *Vanilla Pre-Copy*, proving that fake dirty page elimination is a fundamental optimization before applying further data reduction techniques.

These results highlight that combining optimization with D-Track improves the live VM migration by reducing fake dirty pages and reducing the transferring load during the migration process for memory-intensive workloads.

9 Limitations & Future Work

9.1 Limitations

While D-Track shows promising improvements in live VM migration efficiency, several limitations must be acknowledged:

- **Applying to Multiple VM Migration:** The current implementation and evaluation of D-Track have been limited to single VM migration scenarios. Its performance and effectiveness in scenarios involving concurrent migration of multiple VMs remain untested. Shared resource contention and cumulative overheads may impact the results.
- **Static Tracking Interval:** D-Track currently uses a fixed delay interval for synchronizing the bitmap using *ioctl()* system calls. While effective in reducing performance degradation, this static approach may not adapt optimally to varying workload intensities (e.g., dynamic CPU or memory usage fluctuations).
- **Applications in Post-Copy Migration:** D-Track is designed specifically for Pre-Copy migration and has not yet been extended to Post-Copy mechanisms, specifically in Checkpointing. Thus, its applicability is limited in systems where Post-Copy is preferred.
- **No Integration with Modern Orchestration Systems:** The current version of D-Track has not been integrated with large-scale VM orchestration platforms (e.g., OpenStack or Kubernetes-based VM managers), which limits its immediate deployability in production-grade cloud systems.

9.2 Future Work

To address the limitations outlined above and further improve the utility of D-Track, several directions for future work are proposed:

- **Multi-VM Migration Optimization:** Future research should evaluate and enhance D-Track's effectiveness when handling simultaneous migrations of multiple VMs. This involves managing shared I/O bottlenecks, inter-VM dependencies, and ensuring fairness in resource allocation during migration. Potential strategies include workload-aware scheduling and parallel D-Track instance coordination.
- **Dynamic Synchronization Intervals:** To better balance overhead and accuracy, D-Track can be extended with adaptive bitmap synchronization mechanisms. These would dynamically tune the delay between *ioctl()* calls based on runtime metrics such as CPU usage, memory write rate, or application responsiveness, thereby minimizing intrusion during peak activity.
- **Integration with Post-Copy Migration:** An important avenue for enhancement is adapting D-Track to work with Post-Copy migration models. Specifically, D-Track can assist during the checkpointing phase by identifying and excluding fake dirty pages from initial transfer sets, reducing both downtime and total data transferred.
- **Workload-Aware Optimization:** Incorporating machine learning or heuristic-based policies to classify workload types in real time (e.g., CPU-bound, memory-bound, or I/O-bound) can help D-Track adjust its behavior accordingly, choosing optimal tracking and compression strategies on the fly.
- **Cloud Platform Integration and Automation:** Integrating D-Track into modern virtualization orchestration systems such as OpenStack, Proxmox, or VMware vSphere will make it practically viable for enterprise use. Automation scripts, APIs, and dashboard tools can be developed to manage D-Track settings and monitor its performance across data center operations.
- **Security and Fault Tolerance Considerations:** Since D-Track interacts with low-level memory operations, its design can be extended to detect abnormal page dirtiness patterns, contributing to security (e.g., intrusion detection) and fault tolerance during migration.

References

Banerjee, I., Moltmann, P., Tati, K. & Venkatasubramanian, R. (2014), VMware ESX Memory Resource Management: Swap, *in* 'VMWare Technical Journal'.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. & Warfield, A. (2003), Xen and the art of virtualization, Vol. 37.

Bellard, F. (2005), Qemu, a fast and portable dynamic translator.

Benchbase (2024), 'Benchbase: Benchbase (formerly oltpbench) is a multi-dbms sql benchmarking framework via jdbc.'. `https://github.com/cmu-db/benchbase`.

Bhardwaj, A. & Krishna, C. R. (2019), Impact of factors affecting pre-copy virtual machine migration technique for cloud computing, Vol. 18.

Bila, N., de Lara, E., Joshi, K., Lagar-Cavilla, H. A., Hiltunen, M. & Satyanarayanan, M. (2012), Jettison: Efficient idle desktop consolidation with partial vm migration, *in* 'Proc. of European Conference on Computer Systems'.

Chiang, J., Li, H. & Chiueh, T. (2013), Working Set-based Physical Memory Ballooning, *in* 'The International Conference on Autonomic Computing (ICAC)'.

Deshpande, U., Chan, D., Chan, S., Gopalan, K. & Bila, N. (2015), 'Scatter-gather live migration of virtual machines', *IEEE Transactions on Cloud Computing* **PP**(99), 1–1.

Deshpande, U., Chan, D., Guh, T.-Y., Edouard, J., Gopalan, K. & Bila, N. (2016), Agile live migration of virtual machines, *in* 'IEEE International Parallel and Distributed Processing Symposium, Chicago, IL, USA'.

Deshpande, U., Schlinker, B., Adler, E. & Gopalan, K. (2013), Gang migration of virtual machines using cluster-wide deduplication, *in* 'IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing', pp. 394–401.

Deshpande, U., Wang, X. & Gopalan, K. (2011*a*), Live gang migration of virtual machines, *in* 'ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)'.

Deshpande, U., Wang, X. & Gopalan, K. (2011*b*), Live gang migration of virtual machines.

Elsaid, M. E., Abbas, H. M. & Meinel, C. (2022), 'Virtual machines pre-copy live migration cost modeling and prediction: a survey', *Distributed and Parallel Databases* **40**.

Hines, M. R., Deshpande, U. & Gopalan, K. (2009*a*), Post-copy live migration of virtual machines, Vol. 43, pp. 14–26.

Hines, M. R., Deshpande, U. & Gopalan, K. (2009*b*), 'Post-copy live migration of virtual machines', *SIGOPS Operating System Review* **43**(3), 14–26.

Hirofuchi, T., Nakada, H., Itoh, S. & Sekiguchi, S. (2012), 'Reactive cloud: Consolidating virtual machines with postcopy live migration', *IPSJ Transactions on Advanced Computing Systems* pp. 86–98.

Ibrahim, K. Z., Hofmeyr, S., Iancu, C. & Roman, E. (2011), Optimized pre-copy live migration for memory intensive applications.

Jin, H., Deng, L., Wu, S., Shi, X. & Pan, X. (2009), Live virtual machine migration with adaptive, memory compression, *in* 'Proc. of Cluster Computing and Workshops'.

Jin, H., Li, D., Wu, S., Shi, X. & Pan, X. (2009), Live virtual machine migration with adaptive memory compression.

Jo, C., Gustafsson, E., Son, J. & Egger, B. (2013), Efficient live migration of virtual machines using shared storage, *in* 'ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)', pp. 41–50.

Jul, E., Warfield, A., Clark, C., Fraser, K., Hand, S., Hansen, J. G., Limpach, C. & Pratt, I. (2005), Live migration of virtual machines.
URL: https://www.researchgate.net/publication/220831959

Kiswany, S. A., Subhraveti, D., Sarkar, P. & Ripeanu, M. (2011), Vmflock: Virtual machine co-migration for the cloud, *in* 'Proc. of High Performance Distributed Computing'.

Knauth, T. & Fetzer, C. (2015), Vecycle: Recycling vm checkpoints for faster migrations, *in* 'Proceedings of the 16th Annual Middleware Conference', pp. 210–221.

Kozuch, M. & Satyanarayanan, M. (2002), Internet suspend/resume.

KVM (2024), 'Kvm'. https://linux-kvm.org/page/Main_Page.

Lepak, K. M. & Lipasti, M. H. (2000*a*), 'On the value locality of store instructions', *ACM SIGARCH Computer Architecture News* **28**, 182–191.

Lepak, K. M. & Lipasti, M. H. (2000*b*), Silent stores for free, IEEE, pp. 22–31.

Li, C., Feng, D., Hua, Y. & Qin, L. (2019), 'Efficient live virtual machine migration for memory write-intensive workloads', *Future Generation Computer Systems* **95**.

Memcached (2024), 'Memcached: Free & open source, high-performance, distributed memory object caching system.'. https://memcached.org/.

Milojičić, D. S., Douglis, F., Paindaveine, Y., Wheeler, R. & Zhou, S. (2000), 'Process migration', *ACM Computing Surveys* **32**.

Molina, C., Gonzàlez, A. & Tubella, J. (1999), Reducing memory traffic via redundant store instructions, Vol. 1593.

Nathan, S., Bellur, U. & Kulkarni, P. (2016), On selecting the right optimizations for virtual machine migration.

Piao, G., Oh, Y., Sung, B. & Park, C. (2014), Efficient pre-copy live migration with memory compaction and adaptive vm downtime control.

PostgreSQL (2025), 'Postgresql'. https://www.postgresql.org/.

QEMU (2024), 'Qemu'. https://www.qemu.org/.

Rayaprolu, A. (2024), 'How Many Companies Use Cloud Computing in 2024? All You

Need To Know'.
URL: https://techjury.net/blog/how-many-companies-use-cloud-computing/

Riteau, P., Morin, C. & Priol, T. (2011), Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing, *in* 'Proc. of EURO-PAR'.

Sahni, S. & Varma, V. (2012), A hybrid approach to live migration of virtual machines.

Sapuntzakis, C. R., Chandra, R., Pfaff, B., Lain, M. S., Rosenblum, M. & Chow, J. (2002), Optimizing the migration of virtual computers, Vol. 36.

Scheduler, V. D. R. (n.d.), 'http://www.vmware.com/products/vi/vc/drs.html'.

SHA1 (2024), 'Sha1'. https://www.geeksforgeeks.org/sha-1-hash-in-java/.

Sharma, S. & Chawla, M. (2016), 'A three phase optimization method for precopy based vm live migration', *SpringerPlus* **5**.

Shribman, A. & Hudzia, B. (2013), Pre-copy and post-copy vm live migration for memory intensive applications, Vol. 7640 LNCS.

Svard, P., Hudzia, B., Tordsson, J. & Elmroth, E. (2011), Evaluation of delta compression techniques for efficient live migration of large virtual machines, *in* 'VEE'.

Svärd, P., Hudzia, B., Tordsson, J. & Elmroth, E. (2011), 'Evaluation of delta compression techniques for efficient live migration of large virtual machines', *ACM SIGPLAN Notices* **46**, 111–120.

T. Hirofuchi and I. Yamahata (n.d.), 'Postcopy live migration for qemu/kvm', http://grivon.apgrid.org/quick-kvm-migration.

VMWare Inc. (n.d.), *VMware vSphere vMotion Architecture, Performance and Best Practices in VMware vSphere® 5, https://www.vmware.com/files/pdf/vmotion-perf-vsphere5.pdf.*

VMWare Knowledge Base (n.d.), *Virtual machine performance degrades while a vMotion is being performed, http://kb.vmware.com/kb/2007595.*

Williams, D., Jamjoom, H., Liu, Y.-H. & Weatherspoon, H. (2011), Overdriver: Handling memory overload in an oversubscribed cloud, *in* 'ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)', pp. 205–216.

Wood, T., Ramakrishnan, K. K., Shenoy, P. & Merwe, J. V. D. (2011), Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines, *in* 'Proc. of Virtual Execution Environments'.

Wood, T., Ramakrishnan, K. K., Shenoy, P., Merwe, J. V. D., Hwang, J., Liu, G. & Chaufournier, L. (2015), 'Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines', *IEEE/ACM Transactions on Networking* **23**.

Wood, T., Shenoy, P., Venkataramani, A. & Yousif, M. (2009), 'Sandpiper: Black-box and gray-box resource management for virtual machines', *The International Journal of Computer and Telecommunications Networking* **53**(17).

XEN (2024), 'Xen'. https://xenproject.org/.

Zayas, E. R. (1987), Attacking the process migration bottleneck.

Zhang, I., Garthwaite, A., Baskakov, Y. & Barr, K. C. (2011), Fast restore of checkpointed memory using working set estimation, *in* 'ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)', pp. 87–98.

Zhang, X., Huo, Z., Ma, J. & Meng, D. (2010), Exploiting data deduplication to accelerate live virtual machine migration.

Zhu, L., Chen, J., He, Q., Huang, D. & Wu, S. (2013), 'Lncs 8147 - itc-lm: A smart iteration-termination criterion based live virtual machine migration'.

Index