An Enhanced Approach to Live Migration of Virtual Machines

Thesis submitted in partial fulfillment of the requirements for the degree of

M.S. by Research

in

Computer Science and Engineering

by

Sanidhya Kashyap
200802033
sanidhya.kashyap@research.iiit.ac.in

Center for VLSI and Embedded Systems
International Institute of Information Technology
Hyderabad - 500 032, INDIA
May 2014
It is certified that the work contained in this thesis, titled “An Enhanced Approach to Live Migration of Virtual Machines” by Sanidhya Kashyap, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Dr. Suresh Purini
To my mother: for her strength, wisdom and love.

To my father: for his unconditional belief in me.

To my brother: for being my critic.

To my background thread!
Acknowledgments

I owe my deepest gratitude to my adviser - a very humble and down to earth person - Dr. Suresh Purini. Without his advice, this work would have not have shaped up the way it has. Throughout my stay in IIIT-H, he has been the most supportive and understanding person, and has helped me achieve whatever I have craved for. It is only through his views, that I have learned mostly about a researcher, a professor and most importantly - an adviser.

I cannot forget the support given by my parents (the gods of my life) for always being there beside me and specially my brother - Siddharth, from whom I have a lot to learn in the arena of life, philosophy and about myself.

Life without friends is incomplete, like music to a song. I would like to thank my friends - Aditya, Ankit, Gaurav, Harshit, Himanshu, Jaspal, Kunal, Mihir, Mohit, Nehal, Rachit, Saumay, Varun and Vipul, who have made me realize their importance. I will never forget my endless discussions with Jaspal who has taught me how to question everything.
Abstract

Today, Infrastructure-as-a-service (IaaS) providers are trying to minimize the cost of data center operations, while maintaining the Service Level Agreements (SLAs). This can be achieved by one of the advanced state-of-the-art services of virtualization - the live migration capability. Live migration is defined as the process of transferring an active virtual machine (VM) from one physical machine to another without any disconnection. This is achieved by transferring all of the encapsulated states of the VM from one host to another. Thus, it has become an essential tool for efficient management of resources in a data center by enabling server consolidation and load balancing among other possible benefits with a minimal effect on client services.

There are two classical migration techniques, namely - pre-copy and post-copy, which employ different memory transfer mechanism during the offloading of a VM. In the case of pre-copy, the memory is iteratively transferred from the source to destination until the remaining dirty memory along with the other device states is left to be transferred in a shorter last iteration. While in the case of post-copy, all the device states except memory is immediately transferred to the destination. Source VM’s memory is only transferred if needed by the running VM. However, these existing techniques suffer from high network bandwidth utilization, large network data transfer, large migration time as well as destination’s VM failure during migration.

In this thesis, we propose a novel hybrid live migration technique by combining the existing pre-copy and post-copy approaches. Our hybrid technique is a fast, efficient and a reliable migration technique compared to its counterparts. This technique provides a reasonable solution to high efficiency and less disruptive migration scheme by utilizing all the three phase of the process migration. For effective network bandwidth utilization and reducing the total migration time, we introduced a learning phase to estimate the writable working set (WWS) prior to the migration process. Only those pages which are not present in the estimated WWS are transferred to the destination initially. Pages from the estimated WWS are retrieved from the source machine through page faults as the virtual machine resumes execution on the destination. Our learning technique helps us in achieving almost a single time transfer of the pages when compared against the naive hybrid technique. Several other optimizations like compression, block based paging, page pre-fetching and push phase parallel transfer are incorporated in the migration scheme.

Similar to the post-copy technique, hybrid live migration results in virtual machine failure if the destination machine fails during the pull phase of migration. We introduced reliability into the pull
phase by using a fail-stop model which periodically checkpoints the virtual machine state during the pull phase. If the destination machine fails during the pull phase, the virtual machine can be recovered on the source host from the latest checkpointed state on the disk.

We implemented the proposed hybrid approach on the top of a KVM hypervisor and conducted experiments on diverse workloads. Our compression based hybrid learning approach decreases the total data transfer by 1.83 - 7.47 times for pre-copy, and 1.16 - 12.21 times for post-copy. And it decreases the total migration time by 1.42 - 9.84 times for pre-copy, and 2.43 - 8.57 times for post-copy.
Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>5</td>
</tr>
<tr>
<td>1.5.1</td>
<td>5</td>
</tr>
<tr>
<td>1.5.2</td>
<td>5</td>
</tr>
<tr>
<td>1.5.3</td>
<td>5</td>
</tr>
<tr>
<td>1.5.3.1</td>
<td>5</td>
</tr>
<tr>
<td>1.5.3.2</td>
<td>6</td>
</tr>
<tr>
<td>1.6</td>
<td>7</td>
</tr>
<tr>
<td>1.6.1</td>
<td>7</td>
</tr>
<tr>
<td>1.6.1.1</td>
<td>8</td>
</tr>
<tr>
<td>1.6.1.2</td>
<td>8</td>
</tr>
<tr>
<td>1.6.1.3</td>
<td>9</td>
</tr>
<tr>
<td>1.6.2</td>
<td>9</td>
</tr>
<tr>
<td>1.6.3</td>
<td>9</td>
</tr>
<tr>
<td>1.7</td>
<td>10</td>
</tr>
<tr>
<td>1.8</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2.1</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2.2</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1.1</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1.2</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2.1</td>
<td>17</td>
</tr>
</tbody>
</table>
CONTENTS

2.3.2.2 FHS .................................................. 17

2.4 Experimental Work .................................................. 17
  2.4.1 Setup .................................................. 17
  2.4.2 Applications and Workloads Used for Evaluation .................. 18
  2.4.3 Performance metrics ........................................ 18
    2.4.3.1 Total data transfer .................................. 18
    2.4.3.2 Total migration time .................................. 21
    2.4.3.3 Application performance degradation ...................... 21
    2.4.3.4 Downtime ............................................... 21
  2.4.4 Block based paging and Proactive Background Prepaging ............ 22
  2.4.5 Perceivable downtime .......................................... 22

2.5 Summary .......................................................... 22

3 Hybrid Live Migration ............................................... 24
  3.1 Why should we consider hybrid algorithm? ......................... 24
    3.1.1 Post-copy Pitfalls ....................................... 25
  3.2 Hybrid Approach for Live Migration ................................ 25
    3.2.1 Basic Hybrid Migration Algorithm .......................... 26
    3.2.2 Learning phase ............................................ 26
      3.2.2.1 Discussion ............................................ 27
    3.2.3 Other optimizations ........................................ 27
      3.2.3.1 Push phase page compression ............................ 27
      3.2.3.2 Multi-threaded push phase .............................. 28
  3.3 Implementation details ............................................ 28
    3.3.1 Learning phase ............................................ 28
    3.3.2 Push phase page compression ................................ 28
    3.3.3 Modification to the post-copy’s pull phase .......... 29
      3.3.3.1 Server-Client model .................................. 29
      3.3.3.2 Page-Fault Handler .................................... 29
  3.4 Results .......................................................... 30
    3.4.1 Block selection for the pull phase .......................... 30
    3.4.2 Performance metrics ........................................ 31
      3.4.2.1 Total data transfer ................................... 31
      3.4.2.2 Total migration time ................................... 33
      3.4.2.3 Application performance degradation ................. 34
      3.4.2.4 Downtime ............................................... 34
    3.4.3 Push phase page compression ................................ 34
    3.4.4 Push phase parallel data transfer ............................ 37
    3.4.5 Block Based Paging and Proactive Background Prepaging ........ 38
    3.4.6 Perceivable downtime ........................................ 38
    3.4.7 The Learning Phase ......................................... 39

3.5 Summary .......................................................... 39
4 Reliable Hybrid Live Migration ............................................. 40
  4.1 Introduction .................................................................... 40
  4.2 Approach ................................................................. 41
    4.2.1 Checkpointing Memory and device states ................. 41
    4.2.2 Network buffering .................................................. 42
    4.2.3 Disk State Consistency .............................................. 42
  4.3 Implementation details .................................................. 43
    4.3.1 Checkpointing Memory and device states ................. 43
    4.3.2 Network buffering .................................................. 44
    4.3.3 Disk State Consistency .............................................. 44
  4.4 Results ........................................................................... 44
    4.4.1 Application performance degradation .................... 44
    4.4.2 Perceivable downtime ............................................... 46
    4.4.3 Total migration time ............................................... 46
    4.4.4 Correctness verification ............................................ 46
  4.5 Summary ........................................................................ 46

5 Conclusion and Future work ................................................. 47

Bibliography .......................................................................... 52
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Scenario for consolidation of the virtual machines avoiding server sprawl.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Load balancing scenario of the virtual machines for equal distribution of the resources.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Total data transferred during the pre-copy and post-copy migration mechanism.</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Total migration time during the pre-copy and post-copy migration mechanism.</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Application performance degradation during the pre-copy and post-copy migration mechanism.</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Downtime occurred in the stop-and-copy phase of the pre-copy and post-copy migration mechanism.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Perceivable downtime is the duration between the two vertical lines. Downtime (60 ms) is negligible when compared to the perceivable downtime. Apache Bench tool was used to generate the network requests.</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Hybrid VM migration workflow. Thick lines indicate the interaction between different components. Thin lines indicate several tasks executed by the components in the chronological order.</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Perceivable downtime measured for various block sizes using Apache Bench. Hybrid algorithm is used for VM migration.</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Total data transferred during different migration mechanisms.</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Total migration time during different migration mechanisms.</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Application performance degradation during different migration mechanisms.</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Downtime occurred in the stop-and-copy phase for different migration mechanisms.</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Total data transferred between the hybrid learning and compression based hybrid learning.</td>
<td>35</td>
</tr>
<tr>
<td>3.12</td>
<td>Total migration time between the compression based hybrid learning and parallelized compression based hybrid learning.</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>% CPU utilization during the push phase for hybrid-learning (HL), hybrid-learning-compression (HLC) and parallel hybrid-learning-compression (HLCP) approach.</td>
<td>36</td>
</tr>
<tr>
<td>3.13</td>
<td>Total migration time between the hybrid learning and compression based hybrid learning.</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>Application performance degradation between the hybrid learning and compression based hybrid learning.</td>
<td>37</td>
</tr>
<tr>
<td>3.11</td>
<td>Application performance degradation between the compression based hybrid learning and parallelized compression based hybrid learning.</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>VM is suspended and its state is checkpointed during the time intervals marked (i). The VM state at the source gets asynchronously updated after each checkpoint.</td>
<td>42</td>
</tr>
</tbody>
</table>
4.2 Network buffering during the pull phase of the hybrid migration. . . . . . . . . . . . . 43
4.3 Application degradation (%) when reliable pull phase is enabled. . . . . . . . . . . . . 45
4.4 Change in ratio (on-demand thread vs background thread) when the reliability is enabled for compression based hybrid-learning algorithm. . . . . . . . . . . . . . . . 45
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Data transfer overhead while transferring page blocks of different size.</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Average page block size fetched by both the threads along with the ratio of background vs on-demand thread.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Average page block size fetched by both the threads along with the ratio of background vs on-demand thread.</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Learning Phase performance metrics</td>
<td>39</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Virtualization [16, 29] has become one of the key technologies in the era of Cloud Computing. It is loosely defined as an abstraction of the computing resources, that can be achieved by either dividing the resources into multiple computing environments or merging various resource components into one. The division of the resources can be applied using various concepts and techniques such as time sharing, hardware and software partitioning, simulation, emulation etc. Thus, Virtualization technology has enabled efficient utilization of hardware resources by abstracting away the underlying resources like processors, main memory, secondary storage and networking.

Today, with the help of Virtualization, the data centers are continuously employing the virtualized architecture to execute multiple applications that are mapped on to the physical machines. This has been enabled with the help of virtual machines that forms the software abstraction of a physical machine. This abstraction is achieved by the various virtualization techniques [12, 16, 73] such as:

- **Full Virtualization**: It is defined as the isolated execution of the unmodified guest OS by simulating the hardware resources including full instruction set, input/output operations, interrupts, and memory access etc.

- **Para Virtualization**: It is defined as the isolated execution of the guest OS with modified resources in the form of hooks. The para virtualized interface is used to decrease the performance degradation caused by the time spent by the guest in performing certain operations which are substantially more difficult to execute in a virtual environment compared to a non-virtualized environment.

- **Hardware Assisted Virtualization**: It is defined as the execution of the guest OS with the capabilities provided by the hardware, primarily from the host processor. In this case, the resources can be either fully virtualized or para virtualized.

Therefore, the cloud providers that are providing the infrastructure resources\(^1\); completely rely on Full Virtualization with hardware assistance. This is because of the client requirements of an unmodified guests as a part of the service level agreements (SLAs). Overall, Virtualization helps in enabling

\(^1\)provisioning of the infrastructure in the form of service provided by the cloud providers is known as Infrastructure-as-a-Service.
agility, dynamism, adaptability [43, 58] within a data center. VMs therefore, form the basic building blocks of the Infrastructure-as-a-Service (IaaS) [58] with benefits such as flexible resource provisioning, monitoring and administration by the system administrators.

Resource provisioning allows the provisioning of the infrastructure based resources either at the starting or during the life-cycle of a service [13]. At the IaaS level, these resources consists of servers, network, and self-service for the clouds. Server provisioning, a part of resource provisioning, consists of well defined configuration of the servers based on the requirements of the client. It consists of the following two types:

1. **Static server provisioning**: The configuration of the VMs at the beginning of the life-cycle of a service constitutes the static part of server provisioning. During this process, a physical machine is selected from a pool of physical nodes. Today, a VM is instantiated by using the existing template based provisioning [46] before executing other services provisioning depending upon the requirements.

2. **Dynamic server provisioning**: Both from the client’s point of view and cloud provider, the dynamic resource provisioning plays an important role in allocation or removal of the resources during the life-cycle of a service. Currently, there are multiple ways to modify the resources either horizontally or vertically.

   (a) **Horizontal scaling**: It refers to adding multiple independent physical resources to provide more resources in terms of computing power, memory, disk space and even network bandwidth. This form of scaling employs multiple instances of the applications running on different physical servers.

   (b) **Vertical scaling**: It refers to the addition of the resources on the same physical machine by adding CPUs, memory or even disk for that particular application residing on a single physical instance.

   Currently, vertical scaling is not supported by cloud providers as it requires changes to the guest OS or VMs running on the bare physical machines, which may result in several security issues [56].

A VM, which is a software implementation of a physical machine, always uses the same physical resources (CPU, memory and I/O) utilized by a physical machine. At the beginning of the life-cycle of a service, the initial provisioning of a VM is determined by the resource-usage profile provided by the client. These profiles also try to include the estimations to meet the future load requirements. But, either due to the changes in the workload conditions on VMs or load on the physical machines can lead to ‘hotspots’ - not enough resources to meet the load spikes / demands or ‘coldspots’ - inefficient utilization of the provisioned resources [49]. Thus, to mitigate these issues, live migration of VMs plays an important role in the dynamic resources management of the physical machines inside a data center.
Live migration [20, 32, 44] is the process of transferring a running virtual machine from one physical machine to another over the network. This will alleviate the hotspots to meet the SLA guarantee and handling coldspots for efficient resource utilization. Therefore, from a cloud provider’s point of view, live migration plays a very important role in the following scenarios:

- **Server consolidation**: With the help of server consolidation, we can avoid server sprawl as shown in the Figure 1.1. The VMs residing on a lightly loaded physical machines can be packed onto a fewer physical machines while maintaining the SLA. This will not only lead to low power usage by the data center but also higher resources usage by a host [50].

- **Load Balancing**: The primary purpose of load balancing is to ensure the equal distribution of the resources resulting in almost equal residual resource capacity across the physical machines (Figure 1.2). This can be achieved via live migration of the VMs which will mitigate the resource utilization discrepancy [79].
• **Hotspot mitigation**: During the life-cycle service of a VM, if the resource utilization increases for some time window, it will result in hotspot in the future. To mitigate this issue, either additional resources are locally allocated (vertical scaling) or globally i.e. among the physical machines. If the local resources are not sufficient, VMs can be migrated to another physical node to offload the resources, thus resulting in hotspot mitigation [79].

• **System maintenance**: In order to keep the servers running smoothly inside a data center, the physical resources need to be physically repaired \(^2\), replaced or removed. Thus, during the maintenance, cloud provider will try to migrate the VMs on to different physical machines while adhering the SLAs [26].

### 1.1 Motivation

From the above section, it is quite clear that live migration is one of the most important services of virtualization. From a cloud provider perspective, a VM migration should be transparent enough to have an unnoticeable fast migration by neither exposing the latency to the user nor affecting the collocated VMs on the same physical machine. However, existing migration approaches, such as pre-copy [20] and post-copy [33], are inefficient for highly utilized physical machines inside a data center [45]. These approaches perform quite poorly on workloads with large working set size or even memory intensive workloads. Furthermore, they can incur significant performance overhead by consuming huge amount of network bandwidth. Besides that, the post-copy approach is efficient than pre-copy but does not provide any destination VM guarantee during the migration period. This becomes another concern from a cloud provider perspective.

### 1.2 Background

In this Section, we do a background survey about various migration techniques that have been employed in the past. First we discuss about the process migration and its importance and is live migration related to the process migration. We also discuss various performance metrics that are important from the algorithmic aspect.

### 1.3 Process Migration

Process migration [53] is defined as the migration of processes from one computing environment to another. This concept gained momentum with the introduction of clusters of workstations for providing high performance facility. As it was quite possible to transfer files between various machines in a

\(^2\)Closing the security loopholes and applying the patches for the critical updates.
distributed environment, this introduced a demand for dynamic allocation as well as re-allocation of the computational resources by migrating the process’ execution states.

With the introduction of distributed computing, process migration showed various benefits such as load balancing, fault-tolerance etc. But, it did not achieve widespread use because of the complexity involved in the transparent migration of the processes.

### 1.4 Virtual Machine Migration

A VM is a software implementation of a physical machine which emulates the behaviour of a physical machine. A running VM comprising of its states constitute as a process visible to the host OS. This process is a special process where every state information is in an encapsulated format and can be extracted and used, whenever required, by the host OS. This has been made possible by the virtual machine monitor (VMM) [61]. Thus VM migration is a special case of process migration where VMM plays an important role in providing the required VM’s execution state, thus bringing transparency to the migration process.

### 1.5 Types of Live migration

Following are the three kinds of migration techniques that have been developed for migrating a VM from one host to another.

#### 1.5.1 Cold Migration

In case of cold migration, a powered-off VM is relocated to a new host. The attached disk and configuration of a VM can be migrated to a new location. This can be also used for migrating from one data center to another. In this approach, there is no notion of VM downtime, and performance degradation. A powered-off VM can be easily migrated to mitigate hotspots if there are any.

#### 1.5.2 Suspend / Resume migration

In this case, a live VM is first suspended and then it is migrated to a new host [42]. Currently, this technique is not used as it will result in huge performance degradation and VM downtime if the VM is executing some task.

#### 1.5.3 Live VM migration

Live migration is the process of transferring an active VM’s resources from one host to another over the network. The run time information of VM consists of device states such as disk, memory, network
etc, that gets transferred to the destination. This process helps in dynamic consolidation of the resources by re-allocating the resources on the fly. The migration approach which is derived from the process migration, consists of the following three phases:

1. **Push phase**: This phase is active on the source. In this phase, all the device states with large memory size such as RAM and disk states are transferred to the destination.

2. **Stop-and-copy phase**: This phase begins after the push phase. In this phase, the VM at the source is halted and the dirtied pages of the disk and memory are transferred to the destination along with the other device states. The other device states are not transferred in the push phase not only because of the very frequent updates compared to memory and disk, but also being very small in memory size.

3. **Pull phase**: This phase marks the resumption of the VM on the destination which might fetch the dirty pages if required.

Thus, depending upon the usage of the aforementioned phases, following are the performance metrics that get affected while a live VM is being migrated under different scenarios and constraints:

### 1.5.3.1 Live VM Migration Types

- **Pre-copy live migration**: This approach consists of the first two phases of the process migration. During the push phase, the memory data is iteratively transferred followed by the transfer of the dirty pages along with the device states in the stop-and-copy phase.

- **Post-copy live migration**: The last two phases - stop-and-copy phase and the pull phase constitutes this approach. During the stop-and-copy phase, only the device states is transferred to the destination, while in the pull phase, all the pages are fetched on demand by destination from the source over the network.

- **Hybrid live migration**: This approach incorporates all the three phases of process migration. During the push phase, the memory content is streamed to the destination once. After the conclusion of the push phase, the VM is stopped and all the device states along with the dirty bitmap is transferred to the destination. The VM is then resumed on the destination and the dirtied pages are obtained from the source when requested by the destination.

### 1.5.3.2 Performance Metrics

The effectiveness of a live migration technique can be measured using the following metrics:

1. **Performance degradation**: The performance degradation of an application running on the VM due to the migration process.
2. **Data transferred:** The total data transferred during the migration process. Although it is lower bounded by the size of the VM and the device states, it can be larger than that depending on the migration technique used.

3. **Total migration time:** The time taken to completely finish all the three phases of migration: push, stop-and-copy and pull phases. This is an important metric as the resources allocated to a VM on the source machine will be completely released only after the total migration time.

4. **Downtime:** Downtime is the duration of the stop-and-copy phase. The VM is neither active on the source nor on the destination during the downtime.

5. **Perceivable Downtime:** This is the time period during which the VM is unresponsive after resuming execution on the destination. This is a qualitative metric and is only relevant to VMs whose state is externally visible such as network based workloads.

1.6 **Area Survey**

In this Section, we discuss the earlier work that has been done in the area of virtual machine live migration, and fault-tolerance. We present a literature survey of related work done up till now and we will explain the similarities and differences between the earlier work and ours.

1.6.1 **VM Migration techniques**

Live migration is similar to the process migration problem in earlier distributed systems such as Sprite [27]. Process migration has not come into practice as there is no widespread deployment of distributed systems [6, 9, 57, 68, 70] and also applications requiring true mobility [48]. However, it took a central role in the form of virtual machine migration with the advent of data centers hosting applications with high availability requirements. Further, increase in the available network bandwidth and hardware support for Virtualization made live migration a feasible option. In the context of process migration, post-copy technique was first implemented as “Freeze Free” using a file server [62] and then evaluated the technique via simulations [59] and later it was evaluated as an actual implementation in Linux [53] and in Mosix [36].

Before the beginning of the live VM migration, several approaches were based on non-live technique. In this technique, all the VM’s execution was suspended and then a migration step was performed. Schmidt [65] used the concept of capsules as migration units which consisted of groups of related processes along with their IPC and user space states. While, Zap [55] used both user space and kernel space states for migration of the pods. Also, the Denali project [76, 77] used the checkpointed VMs for the migration process. On the other hand, Sapuntzakis et al. [63] transferred the complete OS in the form of capsule from one host to another. While Internet suspend/resume [64] discussed the implication
of saving and restoring of computing states by cutting the tight bonding between the hardware and the software.

1.6.1.1 Pre-copy live migration

Virtualization based live migration gained its momentum with the pre-copy based self-migration technique proposed by Hansen and Jul [30]. Pre-copy approach for live migration of VMs has been a widely studied topic during the last decade [51, 20]. Clark et al. [20] proposed a pre-copy approach on top of the Xen VMM with dynamic rate limiting to increase the available network bandwidth utilization and reduce large downtime due to high page dirtying rates. Svard et al. [69] proposed a technique wherein, the incremental changes in dirtied pages are computed and the incremental changes are transmitted to the destination after applying a run-length encoding (RLE) compression algorithm. Liu et al. [44] proposed a checkpointing mechanism to trace the execution of a VM. The VM on the destination is synchronized with the VM on the source by iteratively transferring the log to the destination and replaying it there. The total size of the log files is much less than those of dirty pages, thus resulting in a drastic decrease of the downtime and the total migration time. Jin et al. [40] proposed a migration scheme which uses an adaptive compression technique. Ibrahim et al. [39] proposed an online adaptive pre-copy algorithm for VMs running HPC applications. Zhang et al. [80] tried to reduce the total data transferred during the migration process by using hash based fingerprints to find similar pages. Besides this, other works [14, 80, 78] used page level deduplication to reduce the complete transfer of a dirtied page by only transferring the difference between the dirtied and the original. VM-Flock [14] performed the non-live migration of VMs using memory deduplication while Cloudnet [78] and Shrinker [60] optimized the live migration of a single VM over wide area network (WAN). In case of Cloudnet, Wood et al. came up with an optimized pre-copy WAN live migration which optimized both disk and memory over high latency internet links with low bandwidth. Deshpande et al. [26] proposed pre-copy based live migration of multiple VMs from one physical machine to another.

Song et al. [67] designed and implemented a migration scheme exploiting the available data and pipeline parallelism. In our approach also, we use multiple threads during the push phase for parallel data transfer.

1.6.1.2 Post-copy live migration

Hines et al. [33] proposed the post-copy live migration approach for para-virtualized VMs and implemented it on Xen hypervisor. Using dynamic self-ballooning technique, free pages in the guest VM are released back to the hypervisor. This reduces the total data transferred from source to destination. Further, network faults are also reduced by using pre-paging technique. Cavilla et al. [43] used post-copy approach to rapidly provision VM clones on the fly. They introduced the concept of VM fork which is similar to process forking. Similarly, Hirofuchi et al. [35, 34] showed that post-copy migration is suitable for VM consolidation when the VM workloads are suddenly changed. They also used the
post-copy approach for instantaneous consolidation of the VMs. Hines et al. [33] mentioned the idea of combining both pre-copy and post-copy in their work. Our prepaging work approach is different from the one proposed by Hines et al. [33]. The background thread always tries to fetch the pages in block from the starting whereas the on-demand thread fetches the block of pages, which satisfy the spatial locality of the dirtied pages for the application.

1.6.1.3 Hybrid live migration

The closest work to our approach is that of Peng et al. [45] and Liang et al. [38]. Peng et al. [45] provides a hybrid self-migration technique for guests without the intervention of the hypervisor. Whereas our approach is guest agnostic and hence requires no modifications to the guest operating system. Also, their pre-paging technique speculates on the locality of the running processes inside the guest which is possible only in para-virtualized VMs. In our approach, we use block based page prefetching to gain on the spatial locality of programs. Another work by Liang et al.[38] has designed and developed a hybrid technique, that uses XOR based RLE delta compression mechanism while transferring the data in the pull phase. Their approach requires the availability of memory equal to the main memory size of VM which is a constraining factor. All of the aforementioned works do not provide any insight into the reliability issue and how to handle it. Other related work on hybrid migration include Luo et al. [47], Nocalae and Cappello [52] and Zheng et al. [81]. They support hybrid live migration approach only for the disk blocks, whereas our technique concentrates on live migration where the disk is already shared and memory becomes the critical state to be transferred to the destination.

1.6.2 Prepaging technique

The prepaging technique is used to hide the latency of page faults in the pull phase, which has different titles associated with it. In virtual-memory and application, it is called pre-paging. At the I/O level or the actual paging device level, it can be referred to as ‘adaptive prefetching’. For process migration, it can be also referred to as ‘adaptive distributed paging’. But, it has been extensively studied in the context of disk based storage systems [54, 72, 37, 71]. These systems rely on reactive and history based approaches to predict and prefetch the application’s working set, whereas we use prepaging to decrease the network faults as well major page faults by relying on the page offsets for which the network fault occurs.

1.6.3 Reliability

Pull phase reliability which requires state replication of which the virtualization replication is the most easiest one as it is implemented at the software level and provides the same replication guarantee as provided by the hardware based state replication. Earlier works [17, 23] have considered State replication as a way to provide fault-tolerance for a running instance. To achieve state replication, the
VM based solutions used VM logging and replay for high availability. But, ReVirt [28] used to better forensic evidence with the use of changed logging state for the intrusion detection.

Bressoud and Schneider [18] deterministically replayed to replicate the primary’s state forwarded to the backup system. Whereas, Cully et al. [23] provided fault tolerance to fail-stop failures of a physical host by asynchronously propagating its state to a backup host at high frequency. Our reliable pull phase is similar to that of Remus [23]. In Remus, the I/O operations are performed asynchronously to reduce the I/O overhead associated with synchronous writes. Whereas in our approach the disk operations are synchronous and the VM state checkpointing is done by suspending the VM. We adapt this approach in spite of its overhead due to the short pull phase period. This is not applicable in the context of high availability where checkpointing has to go on continuous basis.

1.7 Contributions

The main contribution of this thesis has been the development of the hybrid live migration technique. In addition, the research required for developing such algorithm has resulted into the following additional contributions (including the hybrid approach):

- As a part of the literature survey, we designed a server-client model to handle the major page-fault during the pull phase of the post-copy approach. To improve the fetching of the data over the network, we designed two new approaches - block based paging and low priority page transfer in the background.

- We have also done a thorough investigation of our post-copy approach against the existing pre-copy approach, by implementing it on the top of QEMU 0.13 and Linux 3.6. During our investigation, we formulated a new performance metric - *perceivable downtime*, which is evident in case of workloads with external visible states such as network and disc I/O.

- Realizing the weakness of the post-copy approach in case of memory intensive workloads, and large downtime for workloads with large writable working set size, we implemented a hybrid live migration algorithm incorporating all the three phases [20]. We further decrease the total data transfer by introducing compression in the push phase. We also realized that our approach will end up transferring the writable working set twice. To mitigate this issue, we introduced a *learning phase* prior to the push phase. Further, we parallelize the push phase for enhancing the network data transfer at the cost of CPU cycles.

- We evaluate hybrid along with the optimizations against the existing migration techniques for the same workloads used during the post-copy approach. Our evaluation shows that the hybrid approach (coupled with pull phase compression and learning phase) outperforms the existing pre-copy and the post-copy approaches in case of total data transfer, total migration time and the application performance degradation. We also observe that with the introduction of the learning
phase, the total page transfer gets almost restricted to the VM’s memory size, thus showing the effectiveness of our learning phase.

- Another major contribution in this thesis has been in the case of reliability of the pull phase. We have designed and implemented a fail-stop model which detects the failure of the VM on the destination during the pull phase and resumes the VM on the source. This comes at the cost of some performance degradation at the time of migration when the reliability is enabled.

1.8 Thesis Structure

- **Chapter 2**: In this chapter, we design and implement a post-copy live migration technique in order to understand the limitations of the approach against the existing pre-copy approach.

- **Chapter 3**: This chapter outlines the limitations of the both of the pre-copy and post-copy approaches. Then we design and implement hybrid technique comprising of all the three process or live migration phases. We further enhance the push phase of the hybrid approach to incorporate real-time compression and parallel transfer of the data depending upon the availability of both CPU cores and network bandwidth.

- **Chapter 4**: Here, we discuss about the reliability issue of the pull phase of the live migration technique which has not been handled till now. We come up with a fail-stop design and implementation of the reliable pull phase in order to provide the fault-tolerance guarantee to the VM on the destination.

- **Chapter 5**: In the last chapter, we conclude the thesis by discussing our contributions along with the existing issues in our approach and the possible future directions.
Chapter 2

Post-Copy Live Migration

In this Chapter, we present the design, implementation and evaluation of the post-copy approach for the live migration of VMs in a LAN setup connected via gigabit switch. Post-copy utilizes the last two stages of a generic migration algorithm - stop-and-copy phase, and pull phase. This is in contrast to the classical pre-copy approach which iteratively transfers the memory state in the push phase followed by the remaining dirty pages and the device states’ transfer in the stop-and-copy phase. With the introduction of the pull phase, the time taken to complete the migration process is undetermined. We resolve this issue by including a proactive background pre-paging technique which requests the memory pages from the source when there is no demand of the pages from the destination VM. This optimization coupled with block based paging results in low network overhead as well as reduction in the major page-faults. The post-copy approach proves to be effective by having the least downtime for all the workloads when compared against the pre-copy approach. We have implemented the post-copy approach on the top of QEMU/KVM and have demonstrated our results on the same.

2.1 Why post-copy?

Pre-copy is the classical approach used by the vendors such as Xen [16], KVM [15], VMWare [75] etc., that tries to keep the smaller downtime by transferring less data during the stop-and-copy phase. The pre-copy approach is effective in minimizing the two performance metrics - VM downtime and application performance degradation - only when the VM is executing a read intensive workload. However, the effectiveness of this approach gets reduced for moderately write intensive workloads. In this section, we discuss the pitfalls of the pre-copy approach and why post-copy is effective for memory intensive workloads.

2.1.1 Pre-copy Pitfalls

In the pre-copy approach for VM migration, the pages that got dirtied in each iteration have to be re-transmitted in the next iteration. A large WWS and a high page dirting rate cause a huge number
of page re-transmissions. This results in unnecessary CPU and network bandwidth utilization. If the network bandwidth required for page re-transmissions per iteration exceeds the available limit, then the push phase has to be terminated to give way to stop-and-copy phase. This contributes to the increased VM downtime. Let $M$ be the VM memory size and $D_{ds}$ be the size of the device states. Let the push phase goes for $k$ iterations with the $0^{th}$ iteration corresponding to the total main memory transfer. If $WWS_i$ is the writable working set size in $i^{th}$ iteration, then the total data transferred, $D$, during the entire migration process is given as follows.

$$D = M + \sum_{i=1}^{k} WWS_{i-1} + D_{ds}$$

The downtime of a VM depends on the writable working set size in the $k^{th}$ iteration. If $T_{dt}$ represents the VM downtime, then:

$$T_{dt} \propto WWS_{k-1}$$

From the above equations, it is difficult to predict the number of transmissions that will last in the push phase of the pre-copy approach.

In case of workloads with high dirtying rate, the application performance degrades because of the continuous logging of the VM’s memory at the hypervisor level, thus slowing down the complete VM. Another reason that can be accounted is the implementation of the thread performing migration task. Currently, in KVM [5], on which we have done our survey, uses the same thread pool that is being used for the VCPU thread. This, results in excessive performance degradation if the amount of dirtying rate is very high.

Thus, in order to tackle the aforementioned problem, we defer the transfer of the memory pages by replacing the push phase with the pull phase. Thus, all the memory pages are transferred once the other device states have been transmitted to the destination. When the page is accessed by the VM during the pull phase, that page is requested from the source over the network. Thus, post-copy approach ensures the transfer of every single page at most once, which results in avoidance of the duplicate transmission overhead that is observed in case of the pre-copy approach.

### 2.2 Post-copy approach for live migration

In this section, we propose the design of the post-copy approach which tries to minimize the downtime in the pre-copy and the application performance degradation for the workloads with high dirtying rate.

#### 2.2.1 Basic post-copy algorithm

The post-copy approach consists of the two approaches - stop-and-copy phase and pull phase. It works as follows:
<table>
<thead>
<tr>
<th>Pages transferred</th>
<th>Size (KB)</th>
<th>Data transferred (KB)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4.53</td>
<td>13.28</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>16.72</td>
<td>4.52</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>65.49</td>
<td>2.23</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
<td>259.84</td>
<td>1.5</td>
</tr>
<tr>
<td>128</td>
<td>512</td>
<td>516.47</td>
<td>0.87</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>1028.92</td>
<td>0.48</td>
</tr>
<tr>
<td>1024</td>
<td>4096</td>
<td>4115.66</td>
<td>0.48</td>
</tr>
</tbody>
</table>

Table 2.1 Data transfer overhead while transferring page blocks of different size.

- **Stop-and-copy phase**: The VM is suspended as soon as the migration command is issued. Then the VM's states excluding the main memory pages are transferred to the destination. After this, the VM is immediately resumed on for execution on the destination, while the older memory states of the VM at the source are present for the demand paging.

- **Pull phase**: As the VM resumes its execution on the target machine, every time it accesses a page that is not yet transferred from the source, a page fault occurs. Then the page fault handler retrieves the corresponding page from the source over the network. Page faults of this kind are called network page faults.

### 2.2.2 Other optimizations

On the lines of Hines et al.[33] and Hirofuchi et al.[34], we have incorporated two optimizations to decrease the data transfer as well as the major page faults, which are discussed in the following subsections:

#### 2.2.2.1 Block based paging

When a page fault is handled, it involves two kinds of overhead. The first one is due to the process involved in *invoking the page-fault handler* while the second kind is the *network protocol overhead* as a page is fetched over the network from the source machine. These overheads can be reduced by fetching a contiguous group of pages (*page block*), for every page fault. Table 2.1 shows the data transfer overhead while transferring the page blocks of different size. If the block size is too large, then the page fault handling time increases, which proportionately hampers the VM performance. Thus, block size can be neither too small or too large. For post-copy, we chose a block size of 128 pages in our implementation via experimentation.
2.2.2.2 Proactive Background pre-fetching

In our implementation, we have also enabled a low priority background thread which proactively fetches the pages from the source instead of completely relying on the on-demand thread. As the background thread is of low priority, it gets suspended whenever the page-fault handler is activated. For pull phase enabled algorithms, the concept of background fetching becomes important as it enables the complete VM’s memory transfer in deterministic time. Otherwise, some pages could remain on the source for arbitrarily long times until a page fault occurs. This unnecessarily hogs the resources on the source machine and also makes the migration process less reliable.

2.3 Implementation details

We have implemented the post-copy approach on QEMU 0.13 [11] and KVM [5] on Linux 3.6.4. KVM is one of the most widely used virtual machine monitor (VMM) [19, 16]. KVM is present as a character device driver that works by loading it with Linux kernel. KVM exploits the underlying Linux resources, thus making the Linux kernel as the hypervisor that works with the KVM. QEMU is a userspace hardware emulator, of which KVM is an extension, which manages and monitors the VMs. We have modified the QEMU source at specific points related to memory allocations and migration. No source modifications are necessary for KVM. Since our implementation is specific to hardware-assisted VMs (HVMs), we do not consider the option of ballooning to bypass the transfer of unused pages.

2.3.1 Page-Fault Handler

fault handler (FH) provides memory sharing between a normal process and a process developed by us - as fault-handling process. The code is developed outside the QEMU and KVM for fetching the pages on demand basis. In the following Subsection, we discuss the design choice for the page fault detection followed by description of the implementation.

2.3.1.1 Page fault detection and resolution

Hines et al. [33] have explained the 3 ways via which page faults can be trapped on the destination, viz. Shadow Paging, Page Tracking and Pseudo-Paging. Page Tracking requires significant changes to guest kernel and since we adopted a black-box policy, we dropped it. Pseudo-Paging not only requires swapping out all pageable memory in VM to an in-memory pseudo-paging device but also a modified block driver as well, which retrieves the “swapped out” pages across the network. Shadow Paging effectively means trapping each major fault on destination through a mechanism and then servicing it. In case Xen based implementation, Hines et al. exploited the swap-in/out code of the Linux kernel for on-demand memory transfer. During live migration, all the pages are swapped out to a special swap device backed by physical memory. The aforementioned step is performed during the stop-and-copy
phase along with transferring all the device states to the destination. In the pull phase, the guest OS performs swap-in operation to load the memory pages. The retrieval of the dirtied pages is performed by using a special swap device. Further, this mechanism has been implemented for the para virtualized guest which leverages the Pseudo Physical Address and Machine Frame Number at the mapping table of the hypervisor which results in zero copy from the host OS to the guest OS.

The above implementation needs to modify the guest OS. It is dependent on the memory abstraction of the Xen hypervisor code. Since, the IaaS providers allow the users to customize VMs flexibly, it is difficult to enforce the rule for post-copy migration configuration. On the other hand, our implementation is guest OS agnostic, which does not require any change to the KVM hypervisor except the QEMU code which is the userspace part of the VMM. To achieve this, we decided to use Shadow Paging, not only because it is faster than Pseudo-Paging but also due to its straightforward implementation via a character device driver which fits our setup very well.

2.3.1.2 Fault Handler (FH)

\(FH\) is a character device driver (386 LOC) which is a small pluggable kernel module that is responsible for sharing the memory between the two processes as well as handling the major page-faults at the destination. The program at the userspace - Fault Handler Client (\(FHC\)) obtains the required virtual address space corresponding to the number of pages that will be accessed by the other process. As the allocation of the virtual address is linear, the first address is passed to the \(FH\) driver through an \(ioctl()\) call. Since in Linux, it is difficult to allocate large amount of memory in the kernel space, because of the number of reserved structures for kernel mapping is limited at boot time. Therefore, we allocate the memory for the VM from the userspace through Direct I/O and later remap the memory pages with the obtained virtual address space from the userspace. This is achieved by the Linux kernel function \(get_user_pages\).

When a QEMU process performs \(mmap()\) to /dev/faulthandler, the allocated memory pages get shared between the two process; thus, memory manipulation by one process can be seen by the other process.

When a page is accessed by the mapping process, it is trapped by the \(FH\) driver. The \(mmap()\) implementation of the Linux checks for the page table entry of that page, if it is accessed for the first time. If the page table entry for that page of the accessing process is missing, then the page is marked as not-yet prepared page. For the not-yet prepared page, Linux kernel calls the page fault handler of the \(FH\) driver along with its page number in order to build the page table entry for that page. Then the fault handler code of the \(FH\) notifies the fault handler client, via an \(ioctl()\), to set up the content of the page at the userspace.
2.3.2 Server-Client model

We use client-server model to fetch the pages from the source during the pull phase of the post-copy migration technique. The server on the source machines is called fault handler server (FHS) and the client on the destination is called fault handler client (FHC). The whole code comprises of around 1000 LOC.

2.3.2.1 FHC

On the destination, FHC is a userspace process which allocates the initial address space on the basis of source’s VM total memory requirement. We modify the QEMU’s memory allocation code to 

\texttt{mmap} from the /dev/faulthandler instead of using the /dev/zero for the memory allocation. Here, the FHC has complete access to the /dev/faulthandler. Whenever there is a page fault, the FHC is notified by the fault handler to fetch the page with the given page offset information.

2.3.2.2 FHS

On the source side, FHS is a userspace process that is used for transferring the pages to the destination when the FHC demands those pages. FHS has read-only access to the VM’s memory. We create a normal file on a memory file system (e.g. /dev/shm/ram) for VM memory and allow both QEMU and FHS to access the file through \texttt{mmap}().

\textbf{Lines of Code.} The kernel level implementation of the FH is about 350 lines of code. While the FHS and FHC consist of 446 and 537 lines of code. A 1900 line patch is applied to the migration and allocation mechanism of QEMU for supporting both of the pre-copy and post-copy live migration techniques.

2.4 Experimental Work

2.4.1 Setup

In this section, we present a detailed evaluation of the hybrid migration technique against the existing pre-copy and post-copy approaches. Our test environment consists of an Intel Xeon W3670 (6 cores @ 3.2 GHZ) machine with 12 GB of RAM as the source and an Intel Core i7 (4 cores @ 2.8 GHZ) machine with 8 GB of RAM as the destination. The source and the destination machines are connected via a Gigabit Ethernet switch. Network bandwidth is available for complete utilization by the source and the destination machines. Each VM is allocated 2GB of memory in all the experiments.
2.4.2 Applications and Workloads Used for Evaluation

We chose applications and workloads, both real and synthetic, with diverse characteristics for the evaluation process. From SPEC CPU2006 [31], a memory intensive benchmark (429.mcf) and a cpu intensive benchmark (401.bzip2) are chosen. They have been selected based on their CPI (cycles per instruction) and MPKI (LLC misses per kilo-instruction) values. To supplement our study with real world applications, we used Linux Kernel Compile (LKC), and memcached [2] server which caches multiple key/value pairs in the main memory. For load generation, we use memaslap as client and it resides on a different machine. It is configured to perform random get and set operations in 9:1 ratio with 11 percent overwrite option. The memaslap client utilizes 600 Mbps of network bandwidth, thus leaving 300 Mbps for migration. Finally, we used a synthetic benchmark memtester, which is a highly memory write intensive program used to find faults in RAM. The writable working set size of memtester [8] is set to 1 GB through an input program parameter.

We script the migration phase to begin after a certain time interval for all of the above mentioned benchmarks. For SPEC CPU, memtester, LKC, and memcached server, the migration starts after 15, 10, 10, and 30 seconds respectively. All the experiments have been conducted three times and we report the average of the three measurements.

2.4.3 Performance metrics

In this section, we compare our post-copy approach against the basic pre-copy algorithm. We use the post-copy approach with all the optimizations namely - block based pre-fetching along with the background paging. For pre-copy, we use the KVM’s implementation for our evaluation. The KVM’s pre-copy implementation applies page compression to the pages with the same content. These are generally zero pages and only single byte is transferred along with the page offset information.

2.4.3.1 Total data transfer

The total data transferred for the post-copy migration of a VM is around 2GB (Figure 2.1). This is as expected since the VM memory size is 2GB. All the data transfer happens during the pull phase except for a small amount (~ 16KB) corresponding to the device states. The device states get transferred during the stop-and-copy phase.
Figure 2.1 Total data transferred during the pre-copy and post-copy migration mechanism.

Figure 2.2 Total migration time during the pre-copy and post-copy migration mechanism.
Figure 2.3 Application performance degradation during the pre-copy and post-copy migration mechanism.

Figure 2.4 Downtime occurred in the stop-and-copy phase of the pre-copy and post-copy migration mechanism.
In the case of pre-copy approach, the total data transferred depends on the working set size and the page dirtying rate. The data transferred for write intensive applications like mcf and memcached is substantially higher than their VM size due to multiple page re-transmissions. For bzip2, the data transferred is less than the VM size. This is due to the compression of zero pages.

2.4.3.2 Total migration time

For post-copy, we observe a variation in total migration time for different workloads for transferring the same amount of data (Figure 2.2). This happens due to the variation in the major page faults that get generated for different workloads. Another reason that can be accounted is that the pull phase mechanism is request-response based while the push phase mechanism is stream based. For pre-copy approach, the migration time is proportional to the total data transferred which is naturally high for write intensive workloads. Post-copy approach outperforms the pre-copy in terms of total migration time for memcached and mcf as both of them are memory intensive workloads. Even though memtester is a write intensive workload, the dirtying memory rate is almost equivalent to the network data transfer. Thus, we do not observe very huge difference in the time for both of the approaches.

2.4.3.3 Application performance degradation

From Figure 2.3, except bzip2, we observe a significant decrease in the application degradation. This happens because of the following two reasons: (1) the continuous memory logging of the VM during the push phase of the migration; (2) the same thread pool used for transferring the data which slows down the VM for the existing pre-copy approach. The smaller working set size of the bzip2 accounts to the lower performance degradation when compared against the post-copy approach. However, all the migration algorithms show a relative performance degradation on write intensive workloads. For pre-copy approach, we observe a huge application degradation for the memcached. This is because of the workload being both memory intensive as well as network intensive, which hampers the network throughput performance of the set and get operations of the memcached.

2.4.3.4 Downtime

The general trend is that the downtime is higher for write-intensive workloads (refer Figure 2.4). This is due to the excessive resource (especially pages) utilization of the VM prior to the stop-and-copy phase. The performance of the thread that performs the stop-and-copy phase is affected due to that. Post-copy migration approach performs consistently better than the rest of the techniques. This is due to the minimal amount of data transfer that happens during the stop-and-copy phase. Pre-copy approach performs the worst due to the large number of dirtied pages that need to be transferred. These dirtied pages are from the last iteration of the push phase.
### Table 2.2

<table>
<thead>
<tr>
<th>Workload</th>
<th>Background / On-demand</th>
<th>major page faults decremented (factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>memtester</td>
<td>1.2</td>
<td>0.53</td>
</tr>
<tr>
<td>bzip2</td>
<td>5.2</td>
<td>0.84</td>
</tr>
<tr>
<td>mcf</td>
<td>1.2</td>
<td>0.54</td>
</tr>
<tr>
<td>LKC</td>
<td>8.7</td>
<td>0.90</td>
</tr>
<tr>
<td>memcached</td>
<td>1.6</td>
<td>0.53</td>
</tr>
</tbody>
</table>

Table 2.2 Average page block size fetched by both the threads along with the ratio of background vs on-demand thread.

#### 2.4.4 Block based paging and Proactive Background Prepaging

Table 2.2 shows the effectiveness of the background pre-fetching when compared against the on-demand based page fetching. The background thread fetches more pages by an average factor of 3.6 with the least being 1.2 and a maximum of 8.7 times than that of the on-demand thread. Table 2.2 also shows the ratio of network faults against the major page faults. In case of post-copy, since all the pages have to be fetched over the network, the block based paging decreases the network faults by 128.

From Table 2.2, it can be observed that the major page faults get reduced by 54% - 90%, with the percentage being higher for workloads with low memory dirtying rate. For workloads having high memory dirtying rate and large WWS, the ratio is a bit more than a half. This is because of the large number of pages being fetched by the background thread during the pull phase of the post-copy approach.

#### 2.4.5 Perceivable downtime

For memcached workload, the calculated perceivable downtime comes out to be 34 seconds. During this period, VM becomes unresponsive i.e. it is not able to serve any requests. This happens because of the absence of the working set on the destination which is still in transition from the source over the network. As soon as the complete working set is fetched, the VM becomes responsive, thus marking the end of the perceivable downtime period.

### 2.5 Summary

We have presented the post-copy live migration technique on the top of KVM hypervisor which utilizes background pre-fetching as well as block based paging. Our implementation have shown significant improvements for workloads with intensive memory writes. Since our implementation is specific to the HVMs, our technique ends up transferring all pages which is an overkill in case of large VMs with small WWS. There is a great deal of work left that can be done. By merging both the pre-copy and post-copy, we might be able to make the migration algorithm more efficient. This work has been
discussed in the next chapter. Besides that, another critical problem with the pull phase is its reliability which is originally provided by the pre-copy approach. We discuss this issue in Chapter 4.
Hybrid Live Migration

In this chapter, we present the design, implementation and evaluation of the hybrid live migration approach. This approach consists of both pre-copy and post-copy approaches. The hybrid algorithm utilizes all the three described phases of the process migration steps, namely push phase, stop-and-copy phase and pull phase. Thus, our basic hybrid approach tries to provide the best of two worlds - pre-copy and post-copy, by outperforming the both of these approaches in terms of total data transfer, total migration time, and the application performance degradation. With the introduction of the push phase, our hybrid approach decreases both of the perceivable downtime as well as the number of network faults that are quite high in case of post-copy approach. We have also introduced a histogram based learning phase which not only improves the performance metrics but also reduces the resource consumption of the source. This phase is introduced prior to the push phase, that assists the migration daemon in restricting the transfer of the writable working set, during the push phase. Depending upon the availability of the CPU and network bandwidth, we incorporate a compression technique using a real-time compression/decompression technique - lzo as well as parallelize the push phase in order to utilize the unused bandwidth. We have implemented the hybrid approach on the top of QEMU/KVM and have demonstrated our results on the same.

3.1 Why should we consider hybrid algorithm?

On the basis of the performance metrics discussed for post-copy against the pre-copy, we find that post-copy performance is better only in the case of write intensive workloads with a large working set size. We have already discussed the pre-copy pitfalls in the third chapter in the Subsection 3.1.1. In the following subsection, we discuss the pitfalls of the post-copy approach using experimental observations done in the previous chapter.
Figure 3.1 Perceivable downtime is the duration between the two vertical lines. Downtime (60 ms) is negligible when compared to the perceivable downtime. Apache Bench tool was used to generate the network requests.

3.1.1 Post-copy Pitfalls

Post-copy avoids the possibly excessive page transfers that occur across multiple iterations in the push phase of the pre-copy approach. However, the perceivable downtime could be larger due to the large number of network page faults to retrieve the current working set from the source machine. The downtime will be small though, since during the stop-and-copy phase only the device states are transferred. Figure 3.1 illustrates the difference between the downtime and perceivable downtime. The data is obtained by generating network requests using the Apache Bench tool. The perceivable downtime is approximately 6 seconds when compared to the actual downtime which is 60 milliseconds. In general, if $T_{pdt}$ represents the perceivable downtime and the working set size is $WSS$ then:

$$T_{pdt} \propto WSS$$

Another issue with the post-copy approach is the reliability of the algorithm. This is because of the presence of the pull phase in which the VM’s state is distributed across the source and destination when the migration process is taking place. Thus, if the VM on the destination machines fails during the pull phase, we cannot restore back the VM. Hines et al. [32] threw some light on the issue, but currently no one has addressed this issue.

In order to circumvent the shortcomings of both of the approaches, we propose the hybrid approach comprising of both pre-copy and post-copy that tries to retain the beneficial features of both approaches. To mitigate the issue of extra data transfer, we introduce a new phase known as learning phase which has been discussed in detail (Section 3.2.2).

3.2 Hybrid Approach for Live Migration

In this Section, we discuss the design of the basic hybrid approach along with the introduction of a new phase - learning phase, prior to the push phase and other optimizations incorporated in the push phase. The hybrid approach tries to optimally utilize the network bandwidth by a single transfer of the
VM’s memory to the destination in the push phase followed by only transferring the device state’s along with the dirty bitmap in the stop-and-copy phase and then the dirty pages in the pull phase. Our approach helps us in achieving lesser downtime when compared against the pre-copy technique and drastic reduction in the perceivable downtime when compared against the post-copy approach. We explain the basic hybrid migration algorithm in Section 3.2.1. In Section 3.2.2, we introduce the learning phase which comes prior to the push phase to estimate the WWS. Other performance improving optimizations are presented in Section 3.2.3.

3.2.1 Basic Hybrid Migration Algorithm

The various phases of the basic hybrid migration algorithm are as follows:

1. **Push phase:** The entire memory is transferred to the destination without suspending the VM. There are no multiple iterations and hence no multiple page retransmissions across iterations. Only single byte is transferred, if all the bytes in a page contain the same value.

2. **Stop-and-copy phase:** The VM is suspended and the dirty bitmap along with the device states are transferred to the destination. The dirty bitmap indicates the pages that got dirtied during the push phase. The VM is then resumed at the destination.

3. **Pull phase:** Whenever the VM accesses a page that got dirtied during the push phase, a major page fault gets generated which results in a network fault. This network fault is handled by retrieving the corresponding page from the source. The dirty bitmap transferred during the stop-and-copy phase is useful in deciding whether a page fault can be resolved locally or requires a page transfer from the source.

A page gets transmitted in the push phase and gets retransmitted again in the pull phase, only if its gets dirtied again. During the stop-and-copy phase, no pages get transmitted. So a page gets transmitted at most twice. We introduce a learning phase before the push phase to minimize the number of pages that get transmitted twice.

3.2.2 Learning phase

When the VM migration process is initiated, the learning phase starts to estimate the pages present in the writable workable set. These pages will not be transmitted during the push phase as they are most likely to be dirtied again and have to be retransmitted during the pull phase. If the migration is initiated at time \( t \), then the learning phase happens during the time interval \([t, t + \delta]\). The parameter \( \delta \) has an impact on the accuracy of estimated WWS and the overall migration time.

We use an adaptive histogram to estimate the WWS. The learning phase interval \([t, t + \delta]\) is divided into equal sized epochs and at the end of each epoch, Algorithm 1 is invoked. The algorithm estimates the WWS by computing a histogram of page usage with a forgetting factor \( \alpha \). The array \( hist \) holds the
Algorithm 1: Estimating WWS using adaptive histograms.

1: \( \text{sum} \leftarrow 0.0 \)
2: \( \textbf{for } i \leftarrow 1, \text{nopages} \textbf{ do} \)
3: \( \text{hist}[i] \leftarrow \alpha \text{db}[i] + (1 - \alpha)\text{hist}[i] \)
4: \( \text{sum} \leftarrow \text{sum} + \text{hist}[i] \)
5: \( \textbf{end for} \)
6: \( \text{average} \leftarrow \frac{\text{sum}}{\text{nopages}} \)
7: \( \textbf{for } i \leftarrow 1, \text{nopages} \textbf{ do} \)
8: \( \text{if } \text{hist}[i] \geq \text{average} \textbf{ then} \)
9: \( \text{wwsapx}[i] \leftarrow 1 \)
10: \( \textbf{end if} \)
11: \( \textbf{end for} \)

histogram of page usage and the array \( \text{db} \) holds the page dirty bitmap during the last epoch. The bitmap array \( \text{wwsapx} \) contains the estimated WWS. We set \( \delta \) to three seconds and \( \alpha \) to 0.8 in our experiments.

### 3.2.2.1 Discussion

An alternate approach for estimating WWS is by using splay trees which we did not explore in our current work. Splay tree is a dictionary data structure [66] which satisfies the working set [66] and dynamic finger [66, 21, 22] properties. The working set property states that it takes \( O(\log k) \) time to search for an element in a splay tree, if \( k \) distinct elements are accessed between the current and the last access of that element. The dynamic finger property states that it takes \( O(\log d) \) time to search for an element, where \( d \) is the rank difference between that element and the previously accessed element. These two properties together indicate that if maintain a splay tree of page numbers and keep updating the splay tree based on the page access sequence, then the pages near the top of the splay tree are present in the WWS with high probability.

### 3.2.3 Other optimizations

The pull phase optimizations such as block based paging (refer Section 3.2.2.1) and proactive background pre-fetching (refer Section 3.2.2.2) that were incorporated in the post-copy algorithm, have been used here as well. Besides that, we have included two more techniques for the push phase of the hybrid approach to decrease the total data transfer and the total migration time.

#### 3.2.3.1 Push phase page compression

To decrease the total data transfer, the VM’s entire main memory is divided into logical blocks. During the push phase, each block is iteratively transferred to the destination machine after applying a real-time compression algorithm. After receiving the data on the destination, the data is decompressed...
and copied at the associated logical block position. However, page blocks transferred during the pull phase are not compressed.

3.2.3.2 Multi-threaded push phase

If enough compute power is available and the available network bandwidth is not sufficiently used, we can deploy more than one thread to carry out the push phase. If there are \( k \) threads, then the main memory is divided into \( k \) segments of equal size. Each thread compresses a segment of memory and transfers it to the destination on a separate socket connection. We transfer the memory blocks in a non-contiguous fashion because of no correlation between the memory data, which has been comprehensively utilized by Song et al. [67]

3.3 Implementation details

We have extended the post-copy approach to the hybrid approach by incorporating the push phase in the code base of the post-copy code (refer Section 3.3). The \( FH \) (Section 3.3.1) and server-client (Section 3.3.2) codes are modified in order to make the hybrid algorithm work. We also discuss the learning phase implementation to save the data transfer in the push phase.

3.3.1 Learning phase

The learning phase is based on the checkpointing process of the KVM, which is utilized by the migration protocol. We rely on the same dirty bitmap that is used for the logging process during migration. The implementation of the learning phase is divided into three phases - `start_log`, `log_dirty_pages` and `stop_log`. The `start_log` marks the beginning of the learning phase, in which all the data structures are initialized and logging thread is added to the thread pool. In the `log_dirty_pages` period, the VM is logged and the histogram is updated. Also the processing is executed from a separate thread, that is not a part of the QEMU’s thread pool. `stop_log` marks the end of the logging period and the beginning of the migration process.

3.3.2 Push phase page compression

We have used LZO [7] as the real-time compression tool for compressing and then transferring the data on the destination during the push phase of the hybrid approach. As LZO requires a buffer of size \( B \) bytes, therefore the size of the logical blocks cannot be arbitrarily large, which is 32MB in our case. We save 32MB of memory by directly declaring the memory model of the LZO for the VM’s memory shared by the \( FH \). This also saves the memory copy operation performed for each block, thereby decreasing the push phase migration time by \( \sim 2\text{-}3 \) seconds for a VM with 2GB of allocated memory.
3.3.3 Modification to the post-copy’s pull phase

We have done slight modifications to the both of the server-client architecture and the fault handler to complete the hybrid algorithm.

3.3.3.1 Server-Client model

We have modified both of the FHS and FHC to transfer the memory data in the push phase of the hybrid approach. In the case of learning based hybrid approach, the FHS transfers the pages on the basis of the bitmap obtained from the learning phase. If the bit corresponding to a page offset is marked dirty, then that page is not transferred. Similarly, in the case of compression based hybrid learning, only those pages are compressed and transferred which are non-dirty.

During the stop-and-copy phase, a dirty bitmap is also transferred along with the other device states. The purpose of the dirty bitmap is to notify the destination VM about the dirtied page that needs to be fetched from the source.

3.3.3.2 Page-Fault Handler

We also modify the post-copy’s FH’s page fault handling code. The FH refers to the dirty bitmap, which was passed to the FH in the stop-and-copy phase, to check if the FHC process contains the latest version of the page in its address space (VMA). If the latest version is present, then a new page table entry (pte) is appropriately created. Otherwise, the page is fetched over the network by communicating with the FHS process through the FHC process. After that, the pte corresponding to that page is created and the QEMU process is notified. Figure 3.2 depicts the complete workflow for the compression based hybrid VM migration technique.
3.4 Results

In this Section, we present a detailed evaluation of the hybrid migration technique against the existing pre-copy and post-copy approaches. Our test environment consists of the same configuration that was described in the Section 3.4.1.

3.4.1 Block selection for the pull phase

We use perceivable downtime as the criteria for selecting the block size during the pull phase. For block size selection, we use the naive pull phase which does not support reliability. Figure 3.3 shows the perceivable downtime for different block sizes. Network requests to the Apache server running on the VM are generated by running the Apache client program (Apache Bench) [1] on a machine different from both the source and destination. Figure 3.3 shows that the perceivable downtime is minimal when the block size is 128 pages. The perceivable downtime is the time period in which the server does not respond to the client at the start of pull phase. As the block size increases, the thread which handles the demand paging requests will end up waiting for the prepaging thread running in the background, as only one of them will be active at the same time. This results in increased freeze time of the VM process, thus exacerbating the perceivable downtime. We also observe that for 32 pages as block size, the perceivable downtime is higher than 128 pages. This is because of the two reasons - large number of page faults resulting in high freeze rate of the VM during the pull phase and an incomplete fetching of the writable working set. The 128 pages is only representative of 1Gbps network bandwidth connection, since it is a function of both network bandwidth and working set size.
3.4.2 Performance metrics

In this Section, we compare the basic hybrid algorithm and its optimized variants against the existing pre-copy and post-copy algorithms. Figures 3.4, 3.5, 3.6, and 3.7 shows how pre-copy, post-copy, hybrid and hybrid-learning migration algorithms compare against each other on various workloads. The performance metrics used are application degradation, data transferred, migration time and downtime.

3.4.2.1 Total data transfer

The data transferred during the post-copy migration of a VM is around 2 GB (Figure 3.4). This is as expected since the VM memory size is of 2 GB. All the data transfer happens during the pull phase except for a small amount corresponding to the device states. The device states get transferred during the stop-and-copy phase.

In the case of pre-copy approach, the total data transferred depends on the working set size and the page dirtying rate. The data transferred for write intensive applications like mcf and memcached is substantially higher than their VM size due to multiple page retransmissions. For bzip2, the data transferred is less than the VM size. This is due to the compression of zero pages.
Figure 3.5 Total migration time during different migration mechanisms.

Figure 3.6 Application performance degradation during different migration mechanisms.
Figure 3.7 Downtime occurred in the stop-and-copy phase for different migration mechanisms.

The total data transferred during the hybrid approach is theoretically upper bounded by twice the memory size. This bound is almost reached for workloads with large working set sizes like memcached and mcf. Overall, it can be observed from the Figure 3.4, that the hybrid approach outperforms pre-copy on all workloads. When the learning phase is invoked in the hybrid approach, it can be noticed that even for write intensive applications with working set size as large as the allocated VM size (mcf and memcached), the data transferred in the hybrid approach almost matches that of post-copy. This shows that the learning phase is effective in estimating the writable working set which will not be transmitted during the push phase. Thus, for hybrid-learning, the total data transfer gets decreased by a factor of 1.04x to 1.65x against hybrid and 1.08x to 5.92x against pre-copy.

3.4.2.2 Total migration time

Total migration time for hybrid learning decreases by a factor of 1.3x to 8.4x for pre-copy and 1.6x to 9.6x for post-copy. For the post-copy approach, it can be noted from Figure 3.5 that the migration times are different for different workloads. This is in spite of the same amount of data being transferred during the migration process. This happens due to the variation in the number of major page faults that get generated for different workloads. Hybrid approach outperforms the pre-copy and the post-copy approaches, and hybrid-learning outperforms the hybrid approach on all the workloads. Hybrid outperforms post-copy even on workloads mcf and memcached for which it transfers more data than post-copy. This is due to the presence of read-only pages in the working set resulting in fewer network faults which leads to less network bandwidth contention during the pull phase. The background prepaging thread fetches pages as the VM makes steady progress.
3.4.2.3 Application performance degradation

From Figure 3.6, we can infer that the hybrid-learning algorithm outperforms all the migration algorithms on every workload with a maximum performance degradation under 6 percent. The application performance degradation computed for hybrid-learning includes the learning phase overhead. Hybrid algorithm outperforms post-copy (1.01x) and pre-copy (1.01x to 1.57x) approaches on all workloads, except on memtester where post-copy does better than hybrid. This is due to the load on the source and destination machines while transferring pages during the push and pull phases respectively. All the migration algorithms show a relative performance degradation on write intensive workloads. However, the performance degradation is substantial for the pre-copy approach, like in the case of benchmarks memcached and memtester. Even for a mixed workload like LKC, the performance degradation for pre-copy is substantial when compared with post-copy and hybrid approaches.

3.4.2.4 Downtime

As we have already discussed in Section 3.4.3.4 about the general trend downtime which is higher for write intensive workloads (refer Figure 3.7). This is due to the excessive resource (especially pages) utilization of the VM prior to the stop-and-copy phase. The performance of the thread that performs the stop-and-copy phase is affected due to that. Post-copy migration approach performs consistently better than the rest of the techniques. This is due to the minimal amount of data transfer that happens during the stop-and-copy phase. Pre-copy approach performs the worst due to the large number of dirtied pages that need to be transferred. These dirtied pages are from the last iteration of the push phase. Hybrid and hybrid-learning does far better than pre-copy but do not outperform post-copy. This is because of the transfer of the dirty bitmap in the stop-and-copy phase whose size is proportional to the bits representing VM’s total memory pages.

3.4.3 Push phase page compression

In this section, we present the impact of compressing pages using the real time LZO algorithm in the push phase. Pages are not compressed in the pull phase as the goal there is to serve the pages as quickly as possible. Figures 3.8, 3.13, and 3.10 show the impact of page compression on different migration metrics. The total data transfer gets decreased by a factor of 1.83x to 7.47x for pre-copy, 1.16x to 12.21x for post-copy and 1.26x to 2.94x for naive hybrid. Whereas the total data transferred in the push phase of the naive hybrid, gets decreased by a factor of 1.1x to 5x. The migration time denoted in the table also includes the compression time. The total migration time gets decreased by a factor of 2.43x - 8.57x for post-copy and 1.42x - 9.84x for pre-copy. Figure 3.9 shows the CPU utilization by various migration schemes. The CPU utilization increases by 16.78% to 30.13% in push phase. On the flip side, the push phase time period decreases by an average factor of 1.5x.
Figure 3.8 Total data transferred between the hybrid learning and compression based hybrid learning.

Figure 3.9 % CPU utilization during the push phase for hybrid-learning (HL), hybrid-learning-compression (HLC) and parallel hybrid-learning-compression (HLCP) approach.
Figure 3.10 Application performance degradation between the hybrid learning and compression based hybrid learning.

Figure 3.11 Application performance degradation between the compression based hybrid learning and parallelized compression based hybrid learning.
Figure 3.12 Total migration time between the compression based hybrid learning and parallelized compression based hybrid learning.

Figure 3.13 Total migration time between the hybrid learning and compression based hybrid learning.
Table 3.1 Average page block size fetched by both the threads along with the ratio of background vs on-demand thread.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Ratio</th>
<th>Effective Block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>memtester</td>
<td>1.6</td>
<td>80</td>
</tr>
<tr>
<td>bzip2</td>
<td>99</td>
<td>18.6</td>
</tr>
<tr>
<td>mcf</td>
<td>2.1</td>
<td>52.9</td>
</tr>
<tr>
<td>LKC</td>
<td>0.9</td>
<td>4.7</td>
</tr>
<tr>
<td>memcached</td>
<td>99</td>
<td>23</td>
</tr>
</tbody>
</table>

### 3.4.4 Push phase parallel data transfer

If adequate network bandwidth is available, then we can use multiple threads running in parallel to compress the pages and transfer them. Figures 3.11 and Figure 3.12 compare the performance of the parallel version when two threads are employed during the push phase with the single threaded version. One of the two threads share the same core with the running VM. It can be observed that the application performance degradation and migration time consistently improves in the parallel version. The performance degradation of `memcached` increases as the VM and the migration mechanism share the same network link. Due to parallelization, the migration mechanism demands more network bandwidth from the shared link. Figure 3.9) shows the increase in the CPU utilization due to push phase parallelization. If the source machine has under utilized compute power in the form of idle cores, this is actually an ideal usage.

### 3.4.5 Block Based Paging and Proactive Background Prepaging

Table 3.1 shows the average block size during the pull phase for various workloads. For memory intensive applications, the average block is large, as there is more likelihood of having continuous pages to constitute blocks. As the block size increases, the number of major page faults and network faults decrease. The network faults reduced by an average of 35.8 times with minimum being 4.7 times for LKC and the maximum being 80 for `memtester`. Table 3.1 also shows the ratio of pages fetched by the background thread versus the on-demand thread. Fetching pages in the background before they are faulted helps in reducing the perceivable downtime. It can be observed that more pages are fetched by the background thread when compared with the on-demand thread for all workloads except for LKC. This is because of irregular write access patterns.

### 3.4.6 Perceivable downtime

We measured the perceivable downtime for the network based workload, `memcached` server, for hybrid and post-copy migration algorithms. The perceivable downtime is 34, 19 and 14 seconds for post-copy, hybrid and compression based hybrid-learning migration algorithms. Post-copy shows the
largest perceivable downtime, as all the pages including those which are not dirtied, have to be fetched in the pull phase. For hybrid migration algorithms, only the dirtied pages need to be fetched in the pull phase reducing the number of major page faults.

### 3.4.7 The Learning Phase

In this section, we discuss the implications of having the learning phase prior to the push phase in the hybrid algorithm. The effectiveness of the working set prediction algorithm in the learning phase can be measured by counting the number of false positive and negative predictions. If WWS is the actual working set and WWSAPX is the estimated working set, then the false positive prediction percentage $FP$ is defined as:

$$FP = \frac{|WWS_{APX}| - |WWS \cap WWS_{APX}|}{|WWS|} \times 100.$$  

A false positive page prediction results in a network fault. Similarly the false negative page prediction percentage $FN$ is defined as:

$$FN = \frac{|WWS| - |WWS \cap WWS_{APX}|}{|WWS|} \times 100.$$  

A false negative page prediction results in a page re-transmission. Table 3.2 summarizes the percentage of false positives and false negatives.

We observe that generally for memory intensive workloads, the histogram approach performs well. While in the case of mixed and CPU intensive workloads, the false negative is quite high. This holds true for false positive as well. But, there is no performance degradation in case of CPU and mixed workloads because of their small working set size.

### 3.5 Summary

We have designed and implemented a hybrid migration scheme on the top of a KVM hypervisor, which not only utilizes both pre-copy and post-copy techniques, but also incorporates a novel learning phase prior to the push phase to estimate WWS. Our learning phase leads to effective utilization of the source machine CPU cycles and the available network bandwidth. Besides this, we also expedite the

<table>
<thead>
<tr>
<th>Workload</th>
<th>False Negative</th>
<th>False Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>memtester</td>
<td>0.015</td>
<td>1.34</td>
</tr>
<tr>
<td>bzip2</td>
<td>64.98</td>
<td>9.96</td>
</tr>
<tr>
<td>mcf</td>
<td>0.45</td>
<td>9.8</td>
</tr>
<tr>
<td>LKC</td>
<td>13.57</td>
<td>13.24</td>
</tr>
<tr>
<td>memcached</td>
<td>20.05</td>
<td>10.99</td>
</tr>
</tbody>
</table>

*Table 3.2 Learning Phase performance metrics*
push phase with the introduction of the compression and parallel data transfer, which drastically reduces the data transfer and total migration time for the push phase. Still, in the case of hybrid approach, which comprises of 'unreliable' pull phase, the question of reliability still remains open. This has been discussed in the next Chapter 4.
Chapter 4

Reliable Hybrid Live Migration

4.1 Introduction

Reliability is considered one of the three related key attributes among system designers with modest resources. It is a key attribute providing consistency according to its specifications. The same is expected from the VMs which are the software implementation of a physical machine and are responsible for executing programs like physical machine. But VMs tend to fail during the life-cycle of their service. This can be attributed to the failure of the underlying physical machine on which the VM is running or the host operating system on the hardware fails.

During the migration process, failure of either source machine or destination machine will have different implications depending on the phase of migration as discussed below:

- **Push phase**: This phase is active on the source in which the data is streamed to the destination. Failure of the source will result in permanent loss of VM. There will not be any affect if the destination fails as the source still has the current active state of the VM.

- **Stop-and-copy phase**: Similar to the push phase, this phase is also active on the source, except the data is transferred when the VM is has been suspended on the source. Thus, VM will be permanently lost if it crashes on source while there will not be any effect if the failure of the destination occurs.

- **Pull phase**: During this phase, the VM is active on the destination machine. Since the state transfer is still under progress, the source machine may contain the current state of the VM partially. Failure of either the source or destination result in VM failure.

Thus, **pre-copy** approach is tolerant to destination failure whereas **post-copy** and **hybrid** approaches are not. This is due to the presence of the ‘unreliable’ pull phase. Therefore, pull phase becomes a critical state in ensuring destination reliability during migration.

In order to mitigate the issue of VM failure during the pull phase, we discuss the approach and design of the ‘reliable’ pull phase for the hybrid approach.
4.2 Approach

We assume a fail-stop\(^1\) behaviour of all the components in the subsequent discussion. We address this problem through a synchronized epoch based system where the pull phase is divided into epochs. At the end of each epoch, the incremental changes in the VM state at the destination are checkpointed to a secondary storage device (such as NAS). The secondary storage device is accessible to both the source and destination machines through a network. After checkpointing the latest state, the destination notifies the source so that the source can update its state asynchronously. This reduces the VM recovery time on the source in case of a destination failure. The externally visible state of a VM during each epoch is buffered and released only after successfully checkpointing the latest VM state to the disk. Since the VM is paused at the end of each epoch, there will be a performance degradation. And this degradation occurs in the pull phase and it is the cost we have to pay for reliability. If the destination machine fails, then the VM can be restored at the source by using the latest checkpointed state. The hard disk space consumed by the log files generated during checkpointing depends on the duration of the pull phase and is usually small.

We have also assumed that there is a possibility of VM crashing during the commit phase which will make the source inconsistent. Therefore, we do not concentrate on making the source as a direct receptacle for checkpoints because it is possible that while committing the changes back to the source in the commit stage, the VM at the destination might crash. This will make the source inconsistent as some of the memory state might have been committed while some might be still in transition. Due to this, all the changed states are propagated to the shared log files.

4.2.1 Checkpointing Memory and device states

The memory and the device states constitute the dynamic state of a VM. So checkpointing them is equivalent to checkpointing the VM state. The main memory pages of a VM are marked read-only and all the writes are intercepted by the hypervisor. The intercepted writes are conveyed back to the QEMU process and dirtied pages are marked using a bitmap. At the end of each epoch, the VM is suspended; the dirtied pages and the device states are checkpointed. Then the source machine is notified which in turn starts updating the state of the VM asynchronously. After the notification to the source, the VM resumes execution and the next epoch starts. This whole mechanism is similar to the repeated execution of the stop-and-copy phase. Figure 4.1 summarizes this procedure. The logging of the VM starts at the beginning of the pull phase. After a fixed amount of time, i.e. at \( t_1, t_3 \) or \( t_5 \) in Figure 4.1, the VM is paused, and all the dirtied pages along with other device states is buffered to the disk. As soon as the checkpoint has been acknowledged by the source, the old epoch atomically transitions into a new epoch.

The migrated VM on the source, waits for the next epoch and then updates the memory and device states after receiving the states information from the destination for that epoch. This is similar to the

\(^{1}\) a process crashes and remains halted.
Figure 4.1 VM is suspended and its state is checkpointed during the time intervals marked (i). The VM state at the source gets asynchronously updated after each checkpoint. The save / load mechanism of the complete VM snapshot, except only modified states are updated. This whole process occurs independently at the source after the new epoch transition.

4.2.2 Network buffering

Today, most of the applications rely on TCP connections which provide strong service guarantees. Thus, there is no requirement of the packet replication, since their loss will be accounted as a transient network failure. This fact simplifies the network buffering problem in which the packets get queued for transmission and are only transmitted after the VM state is successfully checkpointed at the end of each epoch. Figure 4.2 depicts the network buffer and release mechanism. Thus, any VM state exposed to the external world can always be recovered from the checkpointed state. If the destination fails during the pull phase, the buffered packets can be reflected as lost state.

4.2.3 Disk State Consistency

As disks provide stronger reliability guarantees, it presents a more challenging aspect than network. During the pull phase, all the disk operations are made synchronous. Before an update is made to a disk during an epoch, the old data is copied to a shared log file and then the new version is synchronously committed to the disk. Algorithm 2 depicts the steps required to save the old and new data at their respective locations. If the destination VM fails, the disk changes are reverted back with the help of the old disk state that is present on the shared log file.
Algorithm 2 Checkpointed disk write operations.
1: let buffer := data to be updated on disk.
2: let sector_num := sector number at which the data will be updated.
3: let offset := the offset in the sector from which the buffer will be updated.
4: function DISK_WRITE(buffer, sector_num, offset)
5:   disk_read(temp_buffer, sector_num, offset);
6:   store the previous disk data in temp_buffer
7:   write_log(temp_buffer, sector_num, offset);
8:   write the old data to the log file.
9:   synchronous_disk_commit(buffer, sector_num, offset);
10:  ▶ update the disk data using the buffer vector.
11: end function

4.3 Implementation details

The reliable pull phase has been designed as a synchronous suspend / resume process. In the following subsections, we present various implementation details of our approach.

4.3.1 Checkpointing Memory and device states

We need to obtain the dirty bitmap of every epoch for checkpointing. The dirtied page information is obtained by tracking the guest writes to memory using the shadow page tables. KVM itself maintains a dirty bitmap which tracks the dirtied pages for a certain period of time. The bitmap is passed back to the QEMU through an ioctl, which helps in identifying the memory regions dirtied during the epoch. The identified dirtied pages and the device states are directly dumped to the epoch based shared log file that is directly accessible to the source. Other device states such as CPU, time etc., constitute a very small size and change very frequently. Therefore, the entire states’ snapshot is propagated during
suspension. On receiving the notification from the destination, the source updates its state using these
log files asynchronously and deletes them.

4.3.2 Network buffering

The network buffering mechanism has been implemented as a Linux queuing discipline. The inbound
traffic is directly delivered to the guest but the outbound traffic is queued until the current VM state has
been checkpointed. This is achieved by using the libnl library[10] which provides a plug module to
buffer and release the packets as and when required. The buffer which is a Linux queuing discipline
is applied to the VM’s network interface on the host OS, which responds to two RT-netlink messages.
During the pull phase, the network buffer receives a CHECKPOINT message, which inserts a barrier
into the outbound traffic. This prevents the release of the subsequent packets until a corresponding
release message is received. As soon as the epoch is transitioned, the buffer receives a RELEASE
message, which dequeues the message. There was a minor issue with this implementation as libnl
works by queuing the incoming traffic, whereas we want to buffer the outgoing traffic. We achieve this
by redirecting the outgoing traffic of the interface on to a special device called intermediate functional
block driver[4] and then redirecting the output to the bridge. Figure 4.2 clearly shows the mechanism
which prevents the release of the network state.

4.3.3 Disk State Consistency

In QEMU, all the reads and writes to the disk are asynchronous. KVM uses asynchronous I/O
(AIO) operations to issue read/write requests to the disk data and allows the I/O thread to handle the
AIO completion notification. We modify the bdrv_aio_writev and bdrv_aio_writev_em which are generic
functions that allow asynchronous writes to the disk. The data is saved on the shared log file before
updating the new data. Then the new data is synchronously written to the disk. The synchronized reads
and writes start at the beginning of the pull phase and lasts till its end.

4.4 Results

In this Section, we compare the compression based hybrid learning algorithm for the pull phase with
and without the reliability.

4.4.1 Application performance degradation

When we use the reliable pull phase in the HLC migration scheme, the application performance
degradation increases. This is due to synchronized disk writes and periodic suspension of VM for
checkpointing. Figure 4.3 shows the impact of reliability on the application degradation. The increase
in the degradation varies from 1 percent to 9.9 percent. The degradation is substantial for memory
intensive workloads such as memtester and mcf, since the time taken for checkpointing a VM state is proportional to its WWS size. Since bzip2 is compute intensive with small WWS, the increase in the performance degradation is small. Further, checkpointing the disk state requires that we perform \( n \) read and \( 2n \) write operations for every \( n \) write operations requested by the VM. Due to this, the performance degradation of LKC is also substantial.

**Figure 4.3** Application degradation (%) when reliable pull phase is enabled.

**Figure 4.4** Change in ratio (on-demand thread vs background thread) when the reliability is enabled for compression based hybrid-learning algorithm.
4.4.2 **Perceivable downtime**

Due to the reliable pull phase, we observe that the perceivable downtime increases from 13 seconds to 14 seconds when reliability is enabled. We do not observe any significant increase in perceivable downtime because it is proportional to the time taken to fetch the working set and those dirty pages are proactively fetched by the background thread, even if the VM is suspended (refer Figure 4.4). Therefore, there is no significant impact of reliability on our hybrid approach.

4.4.3 **Total migration time**

As there is a continuous suspend / resume procedure taking place during the pull phase, this results in lowering of the on-demand thread activity. This in turns increases the activity of the background thread which continuously fetches the pages. Therefore, we observe a small decrease in the total migration time (around 0.5 seconds) for all the given workloads.

4.4.4 **Correctness verification**

In order to verify the working of the designed reliable approach, we deliberately induced network failure / process kill during the following phases:

1. Before the resumption of the VM at destination when all the data has been transferred to the destination and it is about to resume.
2. During the transient state of a VM between two checkpoints.
3. During the checkpointing phase, when the VM’s state is written to the disk.

The checkpoints were taken every 50 milliseconds throughout the pull phase of the HLC approach. For every workload and at every failure point, the source successfully took over the execution with a perceivable downtime of around 2 seconds. All the workloads continued to run till successful completion.

4.5 **Summary**

In this chapter, we discussed the issue of reliability of the pull phase by logging the virtual machine during the pull phase on the destination. Our work, similar to Remus [23], encapsulates a protected state, during the pull phase, and performs frequent whole system synchronous checkpoints and replication. Thus, our approach provides a simple fail-stop model for all the three kinds of device states, namely - disk, network and memory along with other devices and tries to circumvent the issues raised against the ‘unreliable’ pull phase.
Chapter 5

Conclusion and Future work

In this thesis, we studied and implemented two kinds of live migration techniques for the hardware assisted virtual machines (HVMs). The first contribution of this thesis is the design and implementation of the post-copy approach. This approach comprised of the last two stages of the process-migration phases - stop-and-copy phase and pull phase. Due to the introduction of the pull phase, this approach becomes non deterministic in terms of the completion of the migration. This is because of the only on-demand fetching of the data from the source. To resolve this issue, we incorporated a prepaging technique, that proactively fetches the data in background, thus leading to the completion of the migration process as well as reduction in the major page-faults. Another optimization that has been included is the block based fetching of the pages from the source, which not only decreased the major page-faults and network faults, but also the network data overhead associated with the transfer of the pages over the network. The post-copy approach proves to be fruitful for the least downtime for any kind of workload. In case of memory intensive workloads, post-copy outperforms the pre-copy approach for every performance metric, namely - total data transfer, total migration time, downtime and the application performance degradation. While for other kinds of workloads, even though the application degradation is lesser than that of pre-copy approach, the total migration time varies a lot. This happens because of the processing of the major page-faults as well as the context switch between the on-demand thread and the background thread. One major disadvantage of the HVM based post-copy implementation is that we end up transferring unused pages to the destination (including zero pages), resulting in complete VM’s RAM transfer for workloads with small working set size. Thus, pre-copy performs better for the workloads with low dirtying rate and smaller working set size against the post-copy approach. Therefore, for the CPU intensive and read only kind of workloads, pre-copy approach is still better than the post-copy approach. Another important performance metric that comes into the picture is the perceivable downtime. This performance metric is only defined for the migration approach with the pull phase, and occurs at the beginning of the pull phase in which the VM remains inactive due to the large number of network faults. The perceivable downtime for post-copy approach is quite high due to the huge transfer of the pages, consisting of the working set (both read-only and write-only).
The second contribution of this thesis is the design and implementation of the HVM based hybrid approach comprising of both pre-copy and post-copy approaches. This approach consists of all the three phases of process migration phases - push phase, stop-and-copy phase and pull phase. The RAM pages are transferred only once in the push phase as compared to the iterative page transfer for the pre-copy technique. While in the stop-and-copy phase, we transfer all the device states and the bitmap information of the pages that got dirtied during the push phase. Thus, in the pull phase, only the dirtied pages are fetched with the help of the obtained dirty bitmap. During the evaluation, we observed that with the introduction of the push phase, the complete data transfer issue is mitigated that used to occur in the post-copy approach. Besides this, the push phase also helps in decreasing the perceivable downtime and the total migration time by transferring the read-only working set to the destination, thus resulting in further decrease of the network faults and the major page-faults. Our basic hybrid approach outperforms the post-copy approach and the pre-copy approach in terms of total data transfer, total migration time and the application performance degradation, while the downtime is greater than the post-copy technique because of the dirty bitmap transfer during the stop-and-copy phase. We also observed that during the push phase, even the writable working set is copied to the destination, resulting in two times transfer of the WWS in a complete migration process. We try to resolve this issue by introducing a new phase - learning phase, before the starting of the push phase in the hybrid approach. This phase tries to learn about the writable working set and skips those pages in the push phase. Our learning phase uses a histogram based sliding window technique with a forgetting factor, that tries to predict the WWS for the next epoch which is not transferred in the push phase. Due to this phase, the data transfer decreases from an average of 1.54x to 1.04x for various workloads against the simple hybrid approach. The learning based hybrid approach generally outperformed other techniques by a huge margin in all the performance metrics (except downtime). As a part of optimization, we incorporated a real-time compression of the block of pages during the push phase, that not only decreased the total data transfer but also the total migration time. Apart from this, we also observed that migration daemon was unable to use the complete available network bandwidth during the push phase. We try to utilize this bandwidth by transferring the RAM pages in parallel over the network at the cost of some CPU-cycles. This resulted in the reduction of the time taken to transfer the pages in the push phase, leading to the lowering of the total migration time. Our hybrid migration technique coupled with learning phase, compression and parallel data transfer during the push phase, proves to be an efficient migration technique, thus making it a promising approach for the future data centers.

The third and the final contribution of this thesis is the design and implementation of a reliable pull phase. As it has been already shown in the literature survey that the post-copy and hybrid approaches are not reliable, i.e. during the pull phase, if the VM on the destination fails, this will result in the permanent loss of the VM. This happens because of the shared state between the source and destination (dirty pages still residing on the source). In order to address this issue, we developed our own fault-tolerant mechanism that utilizes a fail-stop model by continuously checkpointing the destination’s VM during the pull phase and replicating the VM’s state on the source. This has been achieved by utilizing the
existing logging infrastructure of the QEMU to provide a complete ‘reliable’ hybrid migration approach. Although, with its introduction, the application performance degradation increases, but it is the cost that we have to pay for defining such reliability guarantee to the destination VM. With the help of the reliable pull phase, our migration scheme is tolerant to fail-stop failures of the destination machine during the pull phase. Therefore, our complete migration technique provides fast, efficient and a reliable migration at a minimum cost, leading to minimal source’s resource utilization and an unnoticeable latency to the user.

In future, we would like to exploit GPU for compression as well as for data transfer during the push phase of the hybrid migration [3]. Our technique can also take the advantage of RDMA over Ethernet which will result in decreasing the downtime period since it will not transfer the dirty bitmap which will become a bottleneck for very VMs with huge memory. Currently, our hybrid implementation is solely for HVM, which can be extended to support the para-virtualized VMs that will allow the host to exploit the ballooning technique already used by Hines et al. [33]. This will reduce the data transfer by transferring the read only working set in the push phase and the writable working set in the pull phase. Currently, we believe that the unused pages are zero pages, but there is a possibility of the pages being unused and might be containing the data, which may result in the complete transfer of those pages. By coupling the idea of working set based ballooning technique proposed by Chiang et al. [41] and our prediction phase, we can remove the need of transferring the unused pages in the push phase.

Currently, our learning based technique relies on histogram that tries to gauge the working set values on the basis of a forgetting factor. But, this technique can be enhanced with the help of splay trees. As the splay trees satisfy the working set [66] and the dynamic finger [21, 22, 66] properties, which indicate that the pages present in the top of the splay tree will be present in the WWS with high probability. But the pages should be present in the some sequence, which is not provided by the existing QEMU logging infrastructure. Therefore, in order to make the splay based prediction working, we need to improve working mechanism of the logging of the VMs, which is one of the major bottlenecks in case of memory logging either for pre-copy live migration or even for the reliability of the VMs for the hybrid approach.

From the device driver’s implementation point of view, the fault-handler does not support the memory deduplication provided by the KSM [15]. The madviser() system call will not work because of our implementation utilizing the shared memory model between the destination’s QEMU process and the FHC, whereas KSM only scans anonymous pages for exploiting the page sharing capability. Another future extension can be in either extending the KSM’s functionality to even support shared pages or find a work around for fault-handler.

Currently, the reliable pull phase uses disk logging which is quite slower as all the device writes have to be communicated to the disk before committing all the changes on the source. In future, this can be improved by saving the logging states of the destination’s VM either on the source’s buffered memory or on a network shared memory between the source and destination.
Apart from the algorithmic point of view for the migration approach, we can develop an exhaustive model which defines the QoS for a workload and depending upon that, we can use different migration algorithms to provide a user with strong reliability guarantees.

Another line of thought for the hybrid live migration can be exploited in the domain of clusters. Deshpande et al. [24, 25, 26] have exploited the memory deduplication [74, 15] for migrating multiple VMs from multiple servers to different servers in a rack and then in a cluster. They enhanced the pre-copy approach to migrate multiple VMs, which demonstrated huge potential over the naive pre-copy approach. We can also enhance the hybrid approach and exploit the cluster information for migrating multiple VMs in a cluster environment. In our opinion, the distributed hybrid approach will be a better choice as the data transfer is considerably lower than the pre-copy and lower application performance degradation. We have built our own prototype for the hybrid live migration approach but it requires rigorous testing and evaluation on a cluster.

Currently, there has been no such work on migrating VMs with large memory size i.e. more than 16GB of memory. Only in that case, the actual scalability of the migration algorithms will come into picture. We would like to further evaluate all the migration techniques on VMs with huge working set size in order to gauge the effectiveness of the existing approaches and come up with another effective migration technique to mitigate huge downtime as well as application performance degradation.
Related Publication(s)


Bibliography

[4] Intermediate functional block device -
[7] Lempel-ziv-oberhumer (lzo) real time data compression library -
   http://www.oberhumer.com/opensource/lzo/.
    Proceedings of the 12th international conference on Architectural support for programming languages and
    operating systems, ASPLOS XII, pages 2–13, New York, NY, USA, 2006. ACM.
    cloud. In Proceedings of the 20th international symposium on High performance distributed computing,
    HPDC ’11, pages 159–170, New York, NY, USA, 2011. ACM.
    Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems
    principles, SOSP ’03, pages 164–177, New York, NY, USA, 2003. ACM.


[27] F. Douglis and F. Douglis. Transparent process migration in the sprite operating system, 1990.


VMware. Understanding full virtualization, paravirtualization, and hardware assist.


