Distributed Shared Memory based Live VM Migration

By

Tariq Ghaleb Daradkeh

A Thesis

In

The Department of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

November 2015

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Tariq Daradkeh

Entitled: "Distributed Shared Memory based Live VM Migration"

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

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair

Dr. Yousef Shayan

_____ Examiner, External

Dr. Amr Youssef (CIISE) To the Program

_____ Examiner

Dr. Yan Liu

_____ Supervisor

Dr. A. Agarwal

Approved by: _____

Dr., Chair

Department of Electrical and Computer Engineering

_____20_____ _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Distributed Shared Memory based Live VM Migration

Tariq Daradkeh

Cloud computing is the new trend in computing services and IT industry, this computing paradigm has numerous benefits to utilize IT infrastructure resources and reduce services cost. The key feature of cloud computing depends on mobility and scalability of the computing resources, by managing virtual machines. The virtualization decouples the software from the hardware and manages the software and hardware resources in an easy way without interruption of services. Live virtual machine migration is an essential tool for dynamic resource management in current data centers. Live virtual machine is defined as the process of moving a running virtual machine or application between different physical machines without disconnecting the client or application. Many techniques have been developed to achieve this goal based on several metrics (total migration time, downtime, size of data sent and application performance) that are used to measure the performance of live migration. These metrics measure the quality of the VM services that clients care about, because the main goal of clients is keeping the applications performance with minimum service interruption.

The pre-copy live VM migration is done in four phases: preparation, iterative migration, stop and copy, and resume and commitment. During the preparation phase, the source and destination physical servers are selected, the resources in destination physical server are reserved, and the critical VM is selected to be migrated. The cloud manager responsibility is to make all of these decisions. VM state migration takes place and memory state is transferred to the target node during iterative migration phase. Meanwhile, the migrated VM continues to execute and dirties its memory. In the stop and copy phase, VM virtual CPU is stopped and then the processor and network states are transferred to the destination host. Service downtime results from stopping VM execution and moving the VM CPU and network states. Finally in the resume and commitment phase, the migrated VM is resumed running in the destination physical host, the remaining memory pages are pulled by destination machine from the source machine. The source machine resources are released and eliminated.

In this thesis, pre-copy live VM migration using Distributed Shared Memory (DSM) computing model is proposed. The setup is built using two identical computation nodes to construct all the proposed environment services architecture namely the virtualization infrastructure (Xenserver6.2 hypervisor), the shared storage server (the network file system), and the DSM and HPC cluster. The custom DSM framework is based on a low latency memory update named Grappa. Moreover, High Performance Computing (HPC) cluster is used to parallelize the work load by using CPUs computation nodes. HPC cluster employs OPENMPI and MPI libraries to support parallelization and auto-parallelization. The DSM allows the cluster CPUs to access the same memory space pages resulting in less memory data updates, which reduces the amount of data transferred through the network.

The thesis proposed model achieves a good enhancement of the live VM migration metrics. Downtime is reduced by 50 % in the idle workload of Windows VM and 66.6% in case of Ubuntu Linux idle workload. In general, the proposed model not only reduces the downtime and the total amount of data sent, but also does not degrade other metrics like the total migration time and the applications performance.

Acknowledgement

All praises to Almighty “ALLAH” who enabled me to complete this task successfully. This thesis would have never been completed without the will and blessing of ALLAH, the most gracious, the most merciful.

It has been my privilege and the honor of my academic life to work closely with my thesis supervisor, Dr. Anjali Agarwal, I have learned a lot from her knowledge and experience. Her frequent insights and patience with me are always appreciated. I am very proud of what we have achieved together. Also, I would like to thank my family for their encouragement in all of my pursuits to follow my dreams. I am especially grateful to my mother Husnieh, my father Ghaleb, my sisters Kefah, Roba, Asma, Maryam, Ahlam and Maha, and my brothers Mohammed, Ibrahim, and Ahmed. Also, I must thank all my friend who helped me to achieve my goals.

TABLE OF CONTENTS

List of Figures

List of Tables

List of Acronyms

Acronyms	Meanings
VM	Virtual Machine
CPU	Central Processing Unit
RAM	Random Access Memory
NIC	Network Interface Card
vCPU	Virtual Central Processing Unit
SOA	Service Oriented Architecture
IT	Information Technology
SaaS	Software as a Service
PaaS	Platform as a Service
IaaS	Infrastructure as a Service
API	Application Programming Interface
OS	Operating System
WWS	Writable Working Set
DSM	Distributed Shared Memory
PM	Physical Machine
VMM	Virtual Machine Manager
I/O	Input/Output
KVM	Kernel Virtual Machine
HPC	High Performance Computing
DSB	Dynamic Self Ballooning
ASLR	Address Space Layout Randomization
CBP	Context Based Prediction
PPM	Prediction by Partial Match
ME2	Memory Exploration and Encoding
LRU	Least Recently Used
MMU	Memory Management Unit
SC	Second Chance
PFN	Page Frame Number

DMA	Direct Memory Access
TLM	Thread Based Live Migration
WAN	Wide Area Network
RDMA	Remote Direct Memory Access
PTS	Persistent Temporal Stream
LRC	Lazy Release Consistency
ScC	Scope Consistency
ERC	Eager Release Consistency
EC	Entry Consistency
SMP	Symmetric Multiprocessors
MTA	Multi-Threaded Architecture
HPF	High Performance FORTRAN
MPI	Message Passing Interface
NFS	Network File System
SAN	Storage Area Network
RAID	Redundant Array of Independent Disks
LUN	Logical Unit Number
iSCSI	Internet Small Computer Interface
NAS	Network Attached Storage
CIFS	Common Internet File System
RPC	Remote Procedure Call
LVM	Logical Volume
GUI	Graphical User Interface
VNC	Virtual Network Computing

LAN	Local Area Network
MBR	Master Boot Record
GPT	Global Partition Table
SR	Storage Repository
SDK	Standard Development Kit
API	Application Programming Interface
CLI	Command Line Interface
SSH	Secure Shell
MP	Message passing
PCB	Process Control Block
RRD	Round Robbin Demon
CSV	Comma Separated value
DS	Data Storage
URL	Unified Resource Locator
GPU	Graphical Processing Unit
SLA	Service Level Agreement

Chapter 1

Introduction

Live Virtual Machine (VM) migration process is a major service provided by modern cloud service providers. It can be defined as transferring the Virtual Machine (VM) state while it continues to run and serve clients from one physical machine to another physical machine without disrupting the clients accessing that VM where a condition of shared storage between the two physical machines exists. The VM can be defined as a fully software computer that can run its own operating system and applications as if it were a physical computer. A VM behaves same like a physical computer and contains its own virtual (software-based) CPU, RAM, hard disk and network interface card (NIC). Live VM migration in cloud datacenters is crucial for IT administrators to manage and maintain servers and provide different cloud agility functions like load balancing, power management (server consolidation), fault tolerance in addition to low-level system maintenance that come from the separation between hardware and software.

The VM state is dynamically changed during the live migration process. As a result of serving live clients, these changes affect the memory state, the virtual VM CPU (vCPU) registers and state, and network state. The way to transfer these three work spaces (vCPU state, Memory and Network) safely while continuing to run the VM is to maintain sending the changes in a coherent way until a stop condition occurs. However, clients should be unaware of any changes in the cloud infrastructure due to the fact that live VM migration is a seamless migration process.

As cloud resources change dynamically the live VM migration has become very important for cloud service providers. Since the existing method of live VM migration have certain types of limitation, this work introduces enhancement to the live VM migration process to overcome the common migration model problems, which are discussed in Section 1.5 on live VM migration.

Cloud computing services depends on the virtualization technologies and its capabilities, where hypervisors provide these virtualization features. In order to understand the concept of live VM migration and why it is important in cloud environment we need to understand the concepts of cloud computing, virtualization and hypervisors first. Therefore, the next sections discuss the

following topics: cloud computing, hypervisors and virtualization, and live VM migration process (pre-copy model).

1.1 Cloud Computing

Cloud computing provides cost efficiency, enables collaboration and sharing of resources, improves access methods to resources, saves power, and achieves better resources utilization. Nowadays, cloud computing has become a big player in business and education markets. Cloud computing has become a state-of-art IT technology, defined as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources such as networks, servers, storage, applications, and services that can be rapidly provisioned and released with minimal management effort or service provider interaction [1]. Cloud achieves service or resource on demand irrespective of the type of the service or the resource (hardware like processing, storage, network connectivity, memory; and software like platform, operating system, application). Cloud computing is implemented through a service architecture model, which contains four main components: Virtualization, Automation, Provisioning, and Management. These components may merge or separate upon the model design of cloud architecture as cyber infrastructure and Service Oriented Architecture (SOA) [2, 3, and 4].

Cloud SOA is classified as following:

- Software as a Service (SaaS): Applications offered by service providers and residing in the cloud (e.g. Google Docs, Drop Box, and OneDrive). Such applications provide two kinds of interfaces: (Programmatic that support web-services, and non- programmatic),
- Platform as a Service (PaaS): Platforms used for the development and management of the applications offered as SaaS to end-users and other applications (e.g. Google Apps Engine, Microsoft Azur and Cloud foundry),
- Infrastructure as a Service (IaaS): (Infrastructure provider perspective) Virtualized resources (CPU, memory, storage and service substrates) used (on a pay per use basis) by applications (e.g. IBM Blue Cloud, Amazon EC2).

The dynamicity of resource provisioning based on resource request demands needs a flexibility in resource management that allows cloud provider to dynamically adjust the number of running cloud datacenter servers (increasing or decreasing) based on the demands. Live VM migration is the key feature that aids to achieve this goal.

1.2 Hypervisor and Virtualization

In order to understand the live VM migration we need to understand virtualization first and what kinds of hypervisors we have. Hypervisor allows many OS instances to run concurrently on a single physical machine with high performance, providing better use of physical resources and isolating individual OS instances. In general we can define the hypervisor as a hardware abstraction translator or emulator, based on how VM can get access to the hardware. The hypervisor is responsible for low-level tasks such as CPU scheduling and is responsible for memory isolation for resident VMs. The hypervisor abstracts from the hardware for the VMs. Many people call hypervisor with its management application as Virtual Machine Manager VMM.

Typically, there are two types of hypervisor approaches, the hosted approach and the bare-metal approach. In the hosted approach, the virtual machine monitor runs as an application. The services are presented inside a hosting operating system and the access to the hardware resources is done in cooperation with the hosting operating system. The bare-metal VMM is runs directly on server hardware without requiring an underlying operating system, which results in an efficient and scalable system. This kind of VMM works by abstracting elements from the physical machine (such as hard drives, resources and ports) and allocating them to the virtual machines running on it. The details of the two schemes are shown in Figure 1.1.

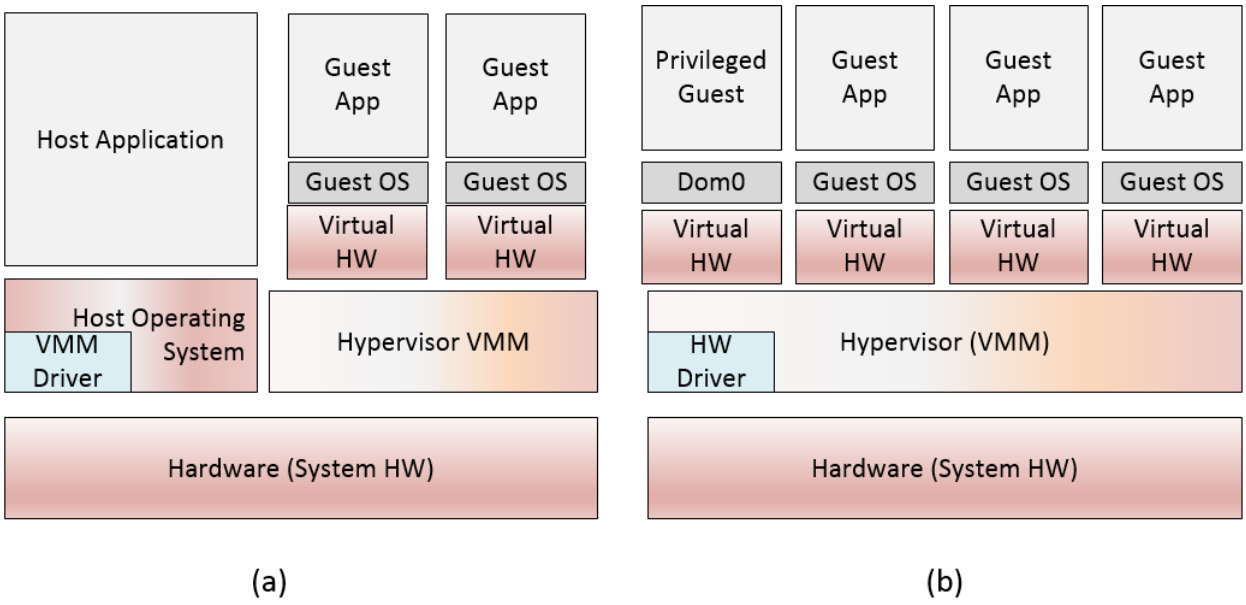


Figure 0.1: Types of Hypervisors (a) Hosted (b) Bare-Metal

Each type of the VMM can support two to three types of virtualization techniques based on the CPU instruction execution that may support virtualization or not. CPUs instructions are divided into privilege and non-privilege instructions. The privilege instructions are the critical instructions that control the hardware and the CPU execution usually through system calls. The non-privilege instructions are the user mode instructions that represent user's applications. The two set of instructions are provided in VMM, however hypervisor privilege instructions are called sensitive instructions whereas, the user application instructions are called non-sensitive instructions. In order for the CPU to be fully virtualized, a computer system architecture must be capable of trapping all sensitive instructions and calling the VMM. A CPU architecture is fully virtualized if the set of sensitive instructions for that computer is a subset of the set of privileged instructions [5]. If this is not the case, and some sensitive instructions are not capable of being trapped, then the architecture is not fully virtualized (called para- virtualized). The differences are depicted in Figure 1.2.

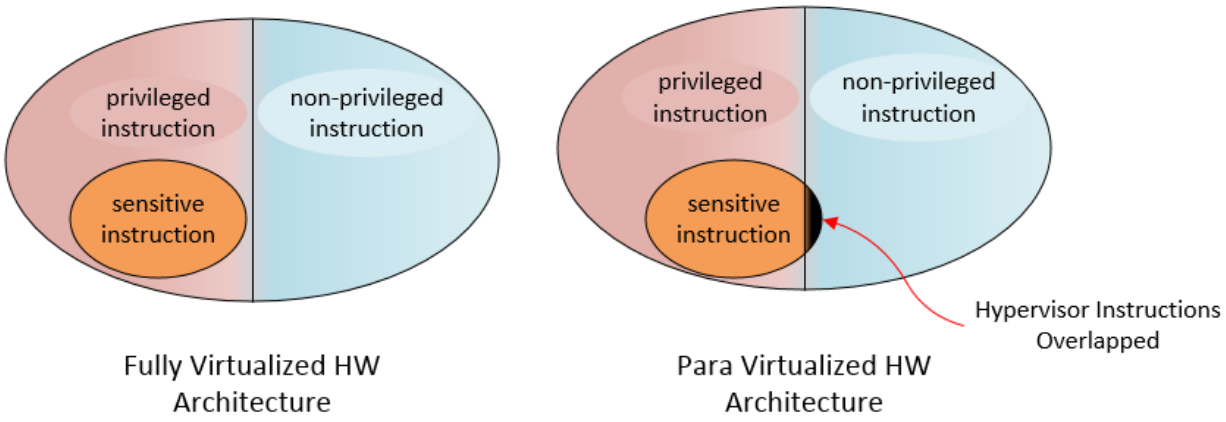


Figure 1.2: CPU Virtualization support [5]

The full virtualization support allows running the guest OS without any modification on the kernel, however the guest OS must have a driver that is installed to make the guest OS compatible with the CPU instruction execution in para-virtualization scheme.

For non-virtualized architectures, we need to emulate the hardware using a third technique called binary translation which is very similar to emulation, and involves running guest code (both OS and application code) on an interpreter that handles any sensitive instructions correctly. The different kinds of VMMs and virtualization technologies are summarized in Tables 1.1 and 1.2.

VMM Types	Properties
Bare Metal VMM	VMM is installed directly into hardware as primary boot system on the hardware. The VMM has full control over any virtual machines that use it and executes at the highest level of privilege.
Hosted VMM	The hosted VMM sits above a host operating system or alongside the hardware, and may share drivers from the host operating system to handle Input/output (I/O).

Table 1.1: Summary of VMMs Types

Virtualization Types	Properties
Fully virtualization	If the set of sensitive instructions for that computer is a subset of the set of privileged instructions. (No need to modify guest OS)
Para virtualization	If the set of sensitive instructions for that computer is not subset of the set of privileged instructions. (Must modify guest OS)
Binary Translation	Similar to emulation, and involves running guest code (both OS and application code) on an interpreter that handles any sensitive instructions correctly.

Table 1.02: Summary of Virtualization Techniques

There are also many virtualization technologies available, including Xen hypervisor [6], VMWare ESXi [7], Kernel based Virtual Machine (KVM) [8], Oracle VirtualBox [9], QEMU [10], XenMicrosoft's Hyper-V [11], OpenVZ [12], Parallels Virtuozzo [13], Oracle VM [14], and many more. The most widely used hypervisor, which has been chosen for the most of open platforms (especially within academic clouds) over the past 8 years, is the Xen hypervisor and it is chosen to implement work in this thesis. Recently, VMWare ESXi, KVM and Oracle VirtualBox are becoming more commonplace [14].

1.3 Live VM Migration Process

Live VM migration is a technique that migrates the entire operating system (OS) and its associated applications from one physical machine to another where the user does not notice any interruption in his service. The main approach of live migration process is pre-copy method: that works iteratively to copy memory state to a set threshold (or until a condition is met) while

still executing on host machine, then execution is suspended, processor state and remaining memory state are copied, and VM is restarted on target machine. The pre-copy approach is divided into three phases [15]:

- Push phase: where the source VM continues to run while certain pages are pushed across the network to the new destination. In order to ensure consistency during this process pages modified must be re-sent.
- Stop-and-copy phase where the source VM is stopped, CPU state, network state and remaining memory pages are copied across to the destination VM, then the new VM is started.
- Pull phase where the new VM executes and, if it accesses a page that has not yet been copied, this page is faulted in ("pulled") across the network from the source VM.

The iterative process requires that the pre-copying occurs in rounds, in which the pages to be transferred during round N, are those that are modified during round N − 1 (all pages are transferred in the first round). Every VM will have some (hopefully small) set of pages that is updated very frequently, which are therefore poor candidates for pre-copy migration. Hence a good way to limit the number of iterations is by binding the number of rounds of pre-copying, based on an analysis of most dynamic memory changed during the VM execution, these pages called writable working set (WWS) behavior of typical server workloads. Stop and copy is the optimal choice where those pages could be transferred. The other (possibly large) set of pages will seldom or never be modified and hence are good candidates for pre-copy, to be transferred at the iterative stage.

Figure 1.3 depicts the steps to pre-copy live VM migrating in sequence stages as a transactional interaction between the two hosts (host A source and host B destination), with respect to a safe data consistency movement approach to the management of migration with regard to safety and failure handling. The stages of interaction could be listed as:

- Step 0: Pre-Migration, it begins with an active VM on the physical host A. In order to ensure speeding up any future migration, a target host may be preselected where the resources required to receive migration will be guaranteed, this step is triggered by cloud manager.
- Step 1: Cloud manager resource reservation, where a request is issued to start migrating an OS from host A to host B. Initially, a resource confirmation check is conducted to ensure

that the necessary resources are available on host B and to reserve a VM container based on that size. Failure to reserve the resources in this stage means that the VM simply continues to run on host an unaffected.

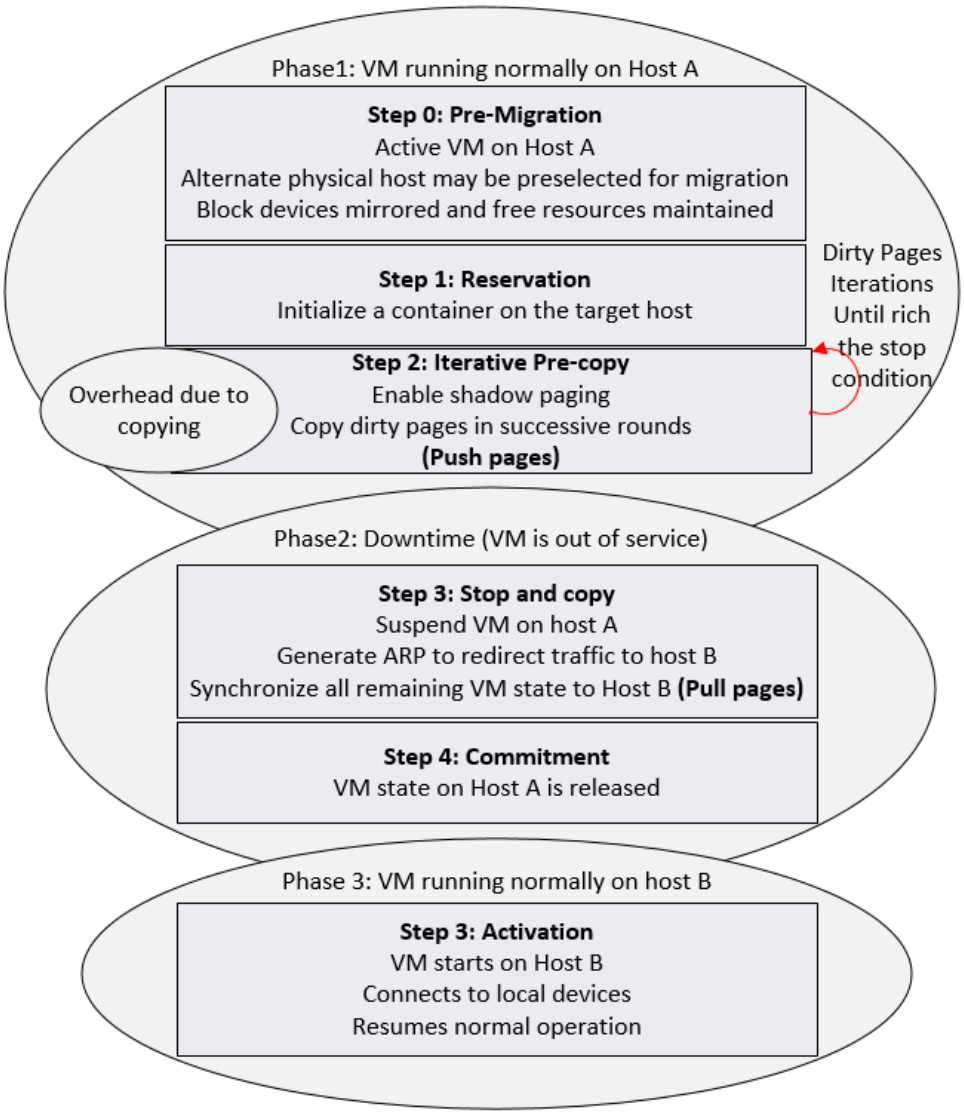


Figure 1.3: Pre-copy Live VM migration [15]

- Step 2: Iterative Pre-Copy, this stage involves pages coping between the two machines. During the first iteration, all pages are transferred from host A to host B. Next iterations copy only those pages dirtied during the previous transfer phase.
- Step 3: Stop-and-Copy, the VM that is running on host A is suspended and redirected its network traffic to host B. CPU state and any remaining inconsistent memory pages are then

transferred. At the end of this stage there is a consistent suspended copy of the VM at both hosts. The copy at host A is still considered to be essential and is resumed in case of failure.

- Step 4: Commitment, where host B indicates to A that it has successfully received a consistent OS image. Host A acknowledges this message as commitment of the migration transaction. Host A may now discard the original VM, and host B becomes the primary host.
- Step 5: Activation, the migrated VM on B is now activated. Post-migration producer runs to reattach the device drivers to the new machine and advertise the VM moved IP addresses.

In general, based on the pre-copy procedure following metrics are used to measure the performance of live migration process:

1. Preparation Time: The time when the live VM migration has started and transferring the VM's state to the target node. The VM continues to execute and dirty its memory.
2. Downtime: The time during which the migrating VM's is not executing. It includes the transfer of processor state.
3. Resume Time: This is the time between resuming the VM's execution at the target and the end of migration, all dependencies on the source are eliminated.
4. Pages Transferred: This is the total amount of memory pages transferred, including duplicates, across all of the above time periods (preparation, downtime and resume time).
5. Total Migration Time: This is the total time of all the above times from start to finish. Total time is important because it affects the release of resources on both participating nodes as well as within the VMs.
6. Application Degradation: This is the extent to which migration slows down the application executing within the VM.

In pre-copy live VM migration, an approach of failure avoidance ensures that at least one host will have a consistent VM image at all times during migration. It depends on that the original host remains has stable consistence image until the migration commits, and that the VM may be suspended and resumed on that host with no risk of failure. Based on previous failure avoidance, a migration request essentially attempts to move the VM to a new host and in case of any failure, the execution is resumed locally and the migration is aborted.

A drawback of pre-copy migration technique is that its total migration time is very high and this technique consumes high bandwidth due to the iterative copying of dirty memory pages, then moving memory and CPU states until a threshold condition is met.

On the other hand, a post-copy technique is proposed in [17] to overcome pre-copy limitation. Post-copy technique starts by copying processor state, then resuming execution on target machine, followed by actively pushing memory state to target machine, there by escaping the iteration phase. It has a smaller total migration time than pre-copy migration technique, but the challenge is to render the needed memory pages to the destination physical machine in the right time to avoid interruption. Also a big drawback of post-copy approach is the migration failure that will cause to lose the VM state since both the source and receiver machines do not maintain consistent VM images during the migration. Table 1.3 summarizes the pros and cons of each technique.

Migration Method	Pros	Cons
Pre-copy	1. Less down time 2. Failure resiliency (sender always keep a consistence VM image until migration finished).	1. Higher total migration time 2. Large amount of data sent during the migration due to iterative phase.
Post-copy	1. Less total migration time 2. The size of data transferred is very close to the VM image size.	1. Higher down time. 2. Migration failure means losing the VM image.

Table 1.3: Post-copy and Pre-Copy Properties.

1.4 Motivation

A big challenge in business is to keep its services available to end users in spite of dynamic changes and maintenance in its infrastructure. For many reasons, IT system administrators need to upgrade software and add or remove hardware. Applying old solution techniques (such as switching off the service, do the upgrade) leads to a long time of interruption (long downtime), where these

operations can be done without any interruption for any service with cloud and virtualization techniques. End users satisfaction can therefore be achieved by company services.

A live VM migration is used to achieve the goals of system upgrade or maintenance. Virtual machine migration also facilitates online maintenance, load balancing and energy management, with minimum administrators' effort. Migrating an entire OS and all of its applications as one package allows us to avoid many of the difficulties measured in process-level migration approaches. Specifically, the narrow interface between a virtualized OS and the virtual machine monitor (VMM) makes it easy to avoid the problem of residual dependencies in which the original host machine must remain available and network-accessible in order to service certain system calls or even memory accesses on behalf of migrated processes. With virtual machine migration, on the other hand, the original host may be out of service (not using the hardware anymore) once migration has completed. This is particularly valuable when migration is occurring in order to allow maintenance of the original host.

In addition, migrating at the level of an entire virtual machine means that in-memory state can be transferred in a consistent and (as will be shown) efficient fashion. This applies to kernel-internal state (e.g. the TCP control block for a currently active connection) as well as application-level state, even when this is shared between multiple cooperating processes. A separation of concerns between the users and operator of a datacenter or cluster, is needed and live migration of virtual machines allows that.

Users have full authority regarding the software and services they run within their virtual machine, and need not provide the worker with any OS-level access at all (e.g. a root login to quiescence processes or I/O prior to migration). Similarly the operator need not to be concerned with the details of what is occurring within the virtual machine; instead they can simply migrate the entire operating system and its attendant processes as a single unit.

Overall, live VM migration is an extremely powerful tool for cluster administrators, allowing separation of hardware and software considerations, and consolidating clustered hardware into a single comprehensible management domain. If a physical machine needs to be removed from service, an administrator may migrate the VM instances including the applications that they are running to alternative machine(s), releasing the original machine for maintenance. Likewise, VM

instances may be rearranged across machines in a cluster to relieve load on congested hosts. In these situations the combination of virtualization and migration considerably improves manageability.

1.5 Problem Statement

The migration of active OSs hosting live services is critically important to minimize the downtime during which services are entirely unavailable. Moreover, main migration matrices must be considered such as the total migration time and down time, during which state on both machines is synchronized and which hence may affect reliability. Furthermore it is vital to ensure that migration does not unnecessarily disrupt active services through resource contention (e.g., CPU, network bandwidth) with the migrating OS.

The pre-copy approach in which the memory pages are iteratively copied from the source machine to the destination host without stopping the execution of the virtual machine being migrated. Memory page transferred level protection is used to ensure a consistent snapshot is transferred, and a rate-adaptive algorithm is used to control the impact of migration traffic on running services. The final phase pauses the virtual machine, copies any remaining pages to the destination, and resumes execution there. A 'pull' method which faults in missing pages across the network since this adds a residual dependency of arbitrarily long duration, as well as providing poor performance.

The weakness however, is the overhead of moving memory pages that are later modified, and hence must be transferred again. For many workloads there will be a small set of memory pages that are updated very repeatedly, and which it is not worth attempting to maintain coherently on the destination machine before stopping and copying the remainder of the VM. The fundamental question for iterative pre-copy migration is: how does one determine when it is time to stop the pre-copy phase because too much time and resource is being wasted?

Many solutions have been proposed to overcome post-copy migration method limitation as will be presented in the literature review (Chapter 2). All of these approaches add or modify some feature to achieve a lower data transfer or a lower downtime or even a lower total migration time as a tradeoff with other parameters like computation time in compression, or more delay time in memory management, or failure resiliency. A solution is therefore needed that can do a comprehensive management to enhance the live VM migration.

1.6 Objectives

A novel live VM migration model is proposed in this thesis based on distributed shared memory (DSM) which will alleviate the communication performance bottleneck and will provide better migration performance with lower downtime, with pre-copy failure resiliency. The idea of using DSM is to enhance the advantages of pre-copy approach with reduced movement of the dirty memory pages through the network with minimum cost and accepted amount of data transferred. Thesis proposed method uses the DSM data consistency update and exchange protocols that are applied to huge real cloud application such as data-intensive application framework (MapReduce, Vertex-centric, Relational query execution) to exchange data messages with multiple execution threads, in order to reduce the migration time.

The high tolerance and low latency of DSM model made it the best choice for enhancement of pre-copy live migration with minimum cost as a general metric measurement.

The goals of this thesis are therefore described as follows:

- Develop a novel live VM migration process based on software DSM application, to allow sharing the VM memory space between the DSM cluster computation nodes.
- Implement the virtualization infrastructure computation structure with the DSM High Performance Computing (HPC).
- Use Network File System protocol to deploy the shared storage between the virtualization structure and the DSM HPC computation cluster.
- Evaluate and compare the performance of the proposed approach with the pre-copy approach.

1.7 Thesis Organization

Chapter 2 presents a review of the literature of various techniques of live VM migration methods. It also discusses live VM migration metrics that are considered in research. Further this Chapter focuses on the primary migration technique, the pre-copy method, and how it works.

Chapter 3 presents the DSM history development in Section 3.1. Section 3.2 describes the DSM cluster models as a parallel computing cluster with distinct program execution techniques, sequential or parallel, then a DSM programming concept has been introduced.

Chapter 4 presents the proposed DSM live VM migration model architecture and its structural services and functions of each main service.

Chapter 5 presents the performance measurements of the proposed model with comparison to the default XenServer hypervisor live VM migration method, the pre-copy method.

Chapter 6 presents the conclusion of the thesis work. It also recommends some future activities for live VM migration.

1.8 Summary

In this chapter an overview on cloud computing model and live VM migration process has been provided. Furthermore motivation behind thesis problem statement has been defined so as to highlight the thesis research objectives. Eventually, thesis organization has been outlined to address the significance of each chapter. In the next chapter, literature review about live VM migration techniques and their issues are discussed.

Chapter 2

Literature Review

2.1 Introduction

Live VM migration techniques started by the idea of copying the running VM memory space and CPU state from the current physical machine to the target or destination machine. The first approach was proposed in 2005 by C. Clark [15] where pre-copy model is discussed. The majority of live VM migration process methods are based on the pre-copy approach where all the researchers focused on enhancing the moving of running VM memory space and vCPU state with minimal cost. In this chapter we study VM migration methods and categorize the work presented in the literature as hierarchal group models based on different criteria according to the way the live VM migration is done. Most of these techniques are based on memory transfer in an iterative push-pull mechanism. As a result a higher delay and longer downtime is needed to achieve the migration. On the other hand a very limited work focuses on live VM migration based on CPU logging and replay (CPU execution tracking) first proposed in 2007 by M. Xu [16]; however this method requires excessive synchronization and not much work has been done using this approach. Many optimization methods were proposed to enhance the way of transferring memory image and CPU state.

In general the metrics that must be taken into consideration when implementing any live VM migration are:

Total migration time: it is the time elapsed from starting the migration process on the source until finishing it on the destination. Clearly, the smaller the migration time the better performance will be achieved.

Down time: it is the time elapsed from suspending the CPU (vCPU) execution of the VM on the source Physical Machine (PM) until the vCPU is resumed in the destination PM. Similar to the total migration time, the value of this parameter should be minimized.

Data transferred size: The size of data moved through the network during the total live migration time. Obviously, the smaller amount of data transferred the better performance achieved because less network bandwidth will be consumed.

VMs application performance: The migrated VM application response time. Objective is to keep the application response and performance as much close to the native performance (VM application performance without migration).

2.2 Live VM Migration Techniques

The existing live VM migration techniques is classified into three main categories. The first category is based on the classic memory state transfer, the second category is based on optimizing memory state transfer, and the last one is based on CPU logging. Memory state transfer technique copies source VM memory state to the destination physical machine. The optimized technique has the same memory state transfer concept but with different improvements on the migration metrics. CPU logging and re-play technique is based on tracking the CPU execution in the source VM and replaying it in the destination server with the assumption that the VMs storage is shared between source PM and destination PM for all methods. Figure 2.2 at the end of this chapter depicts this proposed classification hierarchy tree model of Live VM Migration Process. These techniques are described as following:

2.2.1 Classic Memory State Transfer Methods

The classic methods are divided into three types as described in the following sections.

2.2.1.1 Pre-Copy Approach

Channel In [15] a classic pre-copy approach to move the memory state is proposed. This method first transfers all memory pages from source machine to target machine while the source VM continues to run. The modified memory pages are then transferred iteratively until a consistent memory image is obtained in the destination. The modified pages, named memory dirty pages, contain a set of pages highly modified during the migration time, called Writing Working Set (WWS). These special pages must be sent in stop and copy phase. After certain threshold of memory pages is transferred the VM register and vCPU state are transferred and resumed in the destination machine. Thereafter, the remaining pages are pulled by the destination machine. This

method has a shortcoming in the dirty pages limit where some applications load can produce intensive memory reference calls, which could not finish the migration process or may consume high network bandwidth and server resources due to the long migration process time. Many threshold criteria for iterative procedure are proposed, such as number of dirty pages sent, size of remaining memory, or the number of iteration exceeding a certain value.

2.2.1.2 Post-Copy Approach

In [17], a post-copy method is proposed where registers and vCPU state are transferred and resumed, then the destination physical machine starts pulling the demanding pages through the network. Network delay may reduce the performance of the application execution in the destination. This approach can reduce the total migration time, because it eliminates the iterative resending of the dirty pages as required in the pre-copy approach. However, a higher down time is needed and the application performance degradation will occur because of the pulled memory pages delay through the network.

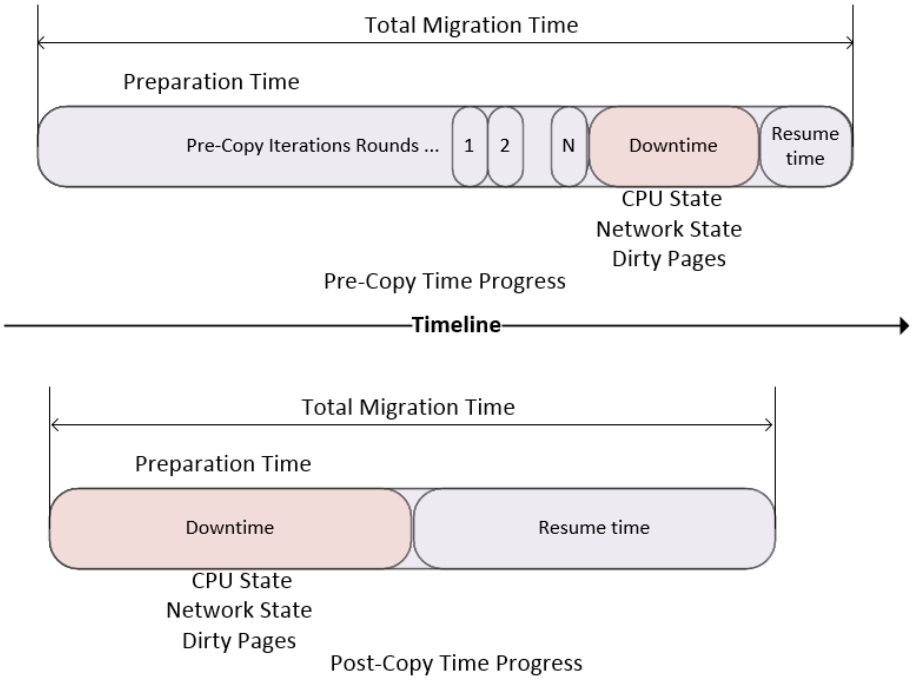


Figure 2.1: Difference between Pre- and Post-copy Methods [17]

In [17], the authors proposed two methods to reduce the number of dirty memory pages to send by starting to push the component of post-copy memory pages with adaptive preparing to reduce the

number of network pages faults, and they used the Dynamic Self-Ballooning (DSB) mechanism to reduce the guest kernel memory foot print that needs to install certain driver in the guest kernel OS. Figure 2.1 explains the time differences between pre and post copy operations methods.

2.2.1.3 Hybrid Approach

In [18] a hybrid approach is proposed where a working set of the entire memory state is sent in one pass (warm up phase) before sending the registers and vCPU state. These bundle of memory pages are considered as a subset of memory pages being used by the VM. After receiving the vCPU state, the destination runs its application, and starts pulling the faulty pages. A working set estimation provisioning and check-points is done with VM process fork technique to allow the destination machine to inherit same set of the working memory pages. This approach may reduce the down time and may keep the application performance accepted under certain condition. Although this method is a hybrid of pre-copy and post-copy, but its application performance is close to post-copy approach because the demanding pages are faulted and a very small set of WWS in the warm up phase is pushed.

2.2.2 Optimized Classic Methods

The optimized classic methods objectives are to improve the performance metrics. We classify these methods into the following groups of mechanisms:

2.2.2.1 Reduce Sending Dirty Pages

The goal of this group is to reduce sending dirty pages, which will reduce the total migration time, down time, size of data sent through the network and reduce the network bandwidth consumption. The following techniques belong to this group.

2.2.2.1.1 Compression Techniques

As these techniques reduce the size of the data sent by using different kinds of compression methods to compress the selected memory pages that will be sent. In [19, 20] an adaptive compression method (MECOM) is used based on memory page characteristics, with zero-aware compression algorithm to achieve performance balancing of VM migration. Memory pages are compressed in batches in the source machine and decompressed in the destination. The main idea of compression is to take advantage of the similarities of the memory data representation and the zero bytes repetition.

In [21] a suite of semantic compression techniques (MiG) is proposed, which is customized based on each memory page. The MiG technique categorizes memory pages based on free pages, code pages or heap pages. Based on the page type, it will use a compression technique that can overcome the Address Space Layout Randomization (ASLR) that is used in modern operating system to ensure security.

A delta compression approach is used in [22], which transfers only the update of dirty pages by encoding one dataset in term of another dataset. This difference-based compression method optimizes the reconstruction of the pages at the destination machine and the network bandwidth consumption. This method operates based on encode one dataset in terms of another, which will use less compression and decompression overhead.

2.2.2.1.2 CPU Scheduling Techniques

The idea of these techniques is to control the CPU execution during the live migration to reduce producing dirty pages or to recognize them and delay sending them until the last iteration. This method is based on weighting CPU schedule for the vCPU by allocating a time weight CPU slot. The weight scheduling can have three types: CPU percentage to each vCPU, weight factor to access the CPU or a credit schedule based on different task queues. In [23] credit scheduling, execution of the active vCPU of the migrated VM is slowed down while increasing the CPU executing to the transfer migration process. Authors in [24] reduce the destination vCPU working frequency while guiding the source vCPU to send the demanding dirty pages. This solution is similar to pause the vCPU process and resume it later in order to limit the VM update to memory pages that will reduce the update speed of the dirty pages production. This will reduce the VM migration time and cost with a trade off with application performance. The approach presented in [25] uses Context Based Prediction algorithm (CBP) that is based on historical statistics of dirty pages. The algorithm exploits Prediction by Partial Match (PPM) CPU scheduling to predict the dirty pages. These dirty pages are sent during the last iteration phase only. The prediction of dirty pages is based on up to date statistics of dirty pages to avoid sending them until last iteration, and this will reduce the data size transferred and total migration time. The data statistics of PPM is based on n-order Markov model.

2.2.2.1.3 Memory Management Techniques

Memory management based techniques identify the dirty pages that are mostly used. The unmodified pages are sent first, then the most dynamically changed pages are sent in the last round. Memory Exploration and Encoding (ME2) based on OS behavior (Black box) and application behavior (Gray Box) is proposed in [26]. The useful pages are identified and sent by running length encoding. White box approach used to probe and explore memory allocation mechanism, such that the migration service will decide which pages to send according to the VM bitmaps (using Xen hypervisor). The ME2 divide the memory pages into two groups: allocated and unallocated in order to avoid sending the unallocated memory pages.

A dynamic memory reallocation (self-ballooning) for unused pages to release and return it back from hypervisor to host operating system is proposed in [27]. The scheme aims to eliminate the transfer of free memory pages, which reduces the memory footprint. This method works with a pre-paging algorithm, an active push driver of memory pages where only the active memory pages will be sent.

A bitmap used to track dirty pages is presented in [28, 29] to utilize sending the least active memory pages first. The bitmap page marks the frequently updated memory pages until the last round to consider these pages as writing working set (WWS) pages to enhance the post-copy method. This will ensure sending dirty pages only once, in addition to one scan copy at the beginning of transferring the memory image.

A Least Recently Used (LRU) memory management algorithm with Splay tree algorithm to predict the most recently used memory pages is proposed in [30]. The memory pages that will be used in future is predicted using LRU and splay algorithm, to group the closely accessed pages during the migration. The grouped pages are sent later in the last iteration of the pre-copy method.

Historical statistics is used to identify high dirty pages in [31]. The identification of the high dirty pages is based on time series prediction technique in the past and future. In order to avoid transferring unnecessary pages, the high dirty pages are sent at the last iteration.

An application memory rate of change to control the migration is used in [32]. A helper thread is used to perform monitoring and tracking of dirty pages to avoid sending them during the migration

time until last round. Furthermore, a high speed network connection, InfiniBand network connection, is used to reduce migration time.

The Linux Memory Management Unit (MMU) integrated with pages fault detection mechanism to enhance page transfer using different threads is presented in [33]. The LRU algorithm is used to identify the dirty pages and update the bitmap array. The LRU reordering will reduce the number of resend pages.

Xen WSClock pages replacement algorithm as pre-processing phase is used in [34] to reduce the amount of transferred memory. WSClock algorithm collects the most recently used pages using LRU and splay algorithms to delay sending them until last round of post-copy method.

Two-phase migration process (a second chance strategy) is proposed in [35] to reduce the number of duplicate pages in each iteration. The first phase uses a second chance strategy to reduce the number of page duplicates across source and destination physical machines. In the second phase they utilize the Second Chance strategy (SC) to lower the frequency of sending duplicate page.

2.2.2.2 Using Shared Storage

In [36], memory state is loaded from secondary storage for utilizing modern operating system cache capabilities. Utilizing the modern OS requires secondary storage to cache the data of the physical memory in order to track the VM input and output operations to the shared attached storage between the physical hosted servers. Instead of transferring the memory pages, it uses a memory to disk mapping transfer data with the target machine. In order to eliminate sending duplicate data, the unloaded memory pages (mapped pages to storage) are sent at the last iteration phase.

A memory compaction technique based on disk cache memory snapshot is proposed in [37], with VM's downtime control using the WWS memory pages history. Based on using the modern OS caching scheme of hard disk to load memory pages as block of pages used to map between Page Frame Number (PFN) and the offset of duplicated data blocks inside the set of data block to avoid sending duplicated pages.

In [38], memory pages duplicated on non-volatile storage are loaded directly and in parallel from the attached storage. The memory state pages will be re-fetched in the target machine from the

shared storage once the migrated VM restarted at the destination host. The memory pages caching to shared storage in the source machine and loading from the shared storage operation runs in parallel which reduce the migration time.

2.2.2.3 Work Load Prediction

Black-Box and Gray-Box used to evaluate the operating system and application workload is proposed in [39]. A Sandpiper system model is proposed to support the black-box or gray-box (or a combination technique) that will be used to discover the OS and application memory intensity workload. This allows a proactive memory allocation and transfer from source and destination machines.

VM performance prediction is used in [40] to migrate the VMs using the post-copy method where the idle VMs are the candidates for VM migration because of their slow memory change.

A resource reservation method with workload-aware migration strategy is proposed in [41]. In this work, a live migration framework of multiple VMs with resources reservation and performance of workload to select the best VMs that can be migrated is presented.

Frames inconsistence (memory frames with invalid data) prediction strategy using AMP-LvMrt algorithm is proposed in [42], to predict number of inconsistent memory frames to be sent in duplicated way to avoid sending invalid data. Reconfiguration of memory space based on frame inconsistency prediction is performed to avoid sending duplicate dirty pages. The inconsistent frames are duplicated at stop and copy stage. CQNCR model is proposed to trigger the best order of massive live migration [43]. CQNCR is used to find an efficient migration sequence plan based on VM applications and workload.

2.2.2.4 Hypervisor Enhancement (Direct Physical Resources Access)

Direct pass through physical hardware using hot plug technology and Linux bounding driver, which allows assignment of physical PCI device to specific guest OS is proposed in [44]. Using para-virtualization driver that reduces the guest OS and hypervisor switching which leads to better performance. Enabling guest VM direct hardware access (like DMA memory remapping) can reduce the hypervisor intervention and achieve high throughput migration process.

In [45] authors describe a lightweight software mechanism with direct hardware access via shadow driver. Shadow driver is an agent in the guest OS kernel that captures and restores the state of device driver. It uses this information to configure a driver for the corresponding device on the destination machine.

Efficient resource management with aggressive VM relocation among physical servers, with a special character device driver allows transparent memory pages retrievals from source host to running VM at destination is presented in [46]. In order to minimize the time needed for switching of the execution host, special driver is used to allow transparent destination host server to access the VM memory pages for retrieving them directly.

2.2.2.5 Process State Based Migration Methods

In this technique, a time bound, Thread based Live Migration (TLM) mechanism to transfer machine state is employed. Overcommitted CPU used to minimize down time is proposed in [47]. Additional threads are added to the pre-copy live migration to achieve migration within bounded time period. This work using CPU overcommitted mechanism can minimize migration time and downtime.

A SnowFlock solution is proposed in [48] that forks VM to clone it instantaneously to multiple replicas running on different hosts. SnowFlock utilizes lazy state replication to minimize the amount of state propagation. Using VM fork will enable pattern replication and process inheritance which can reduce cloning and migration VMs.

2.2.2.6 Whole VM Images Cloning Using VM Abstraction and Replication

Whole VM images (the VM CPU state, Memory state and Storage) cloning using VM abstraction and replication is a VM incremental based migration by Three Phase Migration TPM algorithm with block bitmap tracking to the local desk is proposed in [49]. This method is used to transfer whole VM vCPU states, Memory image and disk storage by using incremental migration algorithm to reduce amount of data transferred.

An abstraction template of VM substrate (partial memory state of the VM memory image) to represent generic VM stored in memory with state-full activation is proposed in [50]. As a substrate clone and migrate the VMs memory pages, copy on write and on the fly resource configuration to save memory space size, to transfer less amount of data.

Shrinker, a common data aggregation used to perform migration, is proposed in [51]. It can improve WAN migration. Shrinker detects memory pages and disk blocks duplicated in virtual cluster to avoid sending the same contents multiple times through WAN network.

2.2.2.7 Mathematical Model Based Prediction

An analytical model that describes the cost function of live migration with respect to a threshold value of stop and copy state, α, under uniform and non-uniform dirtying rate; to find the best value of α to determine when to stop the iteration phase in the pre-copy method to achieve minimum time downtime is proposed in [52]. Three cases has been tested of the dirtying pages: uniform page dirtying, hot pages being copied during the pre-copy phase, and hot pages copied only during the VM downtimes.

Markov chain model is used to predict increasing speed of snapshots to migrate through WAN in [53]. A prediction based strategy optimizes cloud VM migration process over WAN network, with adjacent factor during the prediction to reduce the VM snapshot migration.

2.2.2.8 Combined Approach

In [22, 26, 33, and 37] authors combined different approaches to achieve best parameter performance of live migration such as compression, memory management and utilization of shared storage.

2.2.2.9 Avoiding TCP Overhead Using InfiniBand Network with Remote Direct Memory Access

Using Remote Direct Memory Access (RDMA) with InfiniBand network to avoid using TCP by OS bypass, which allows data communication to be directly initiated from the processor proposed in [54]. A dynamic adaptive algorithm to limit the transfer rate of the migration traffic is used in this model.

2.2.2.10 Parallelizing Live VM Migration Process

By parallelizing migration process a speed up of transferring pages and a reduction of total migration time and down time is achieved in [55]. Sending migration threads and receiving threads are employed to enhance the VM migration process by applying data parallelism and pipeline parallelism to most of the primitive operations.

2.2.3 CPU Execution State Logging and Replay

This category focuses on tracking the CPU execution with logging capabilities to transfer the logging rather than the memory pages. The CPU logs will be retraced or re-executed in the destination physical machine.

2.2.3.1 Retrace Model

In [16] a ReTrace model retraces the CPU execution using Store-Point that logs the source CPU execution and replays it in destination physical machine. A trace of collection tools based on a deterministic replay of the CPU execution is used by using two steps the ReTrace capturing to track the CPU execution and ReTrace expansion to replay the CPU log at destination physical machine.

2.2.3.2 Asynchronous Logging Model

Remus framework is proposed in [56], where it is used to asynchronously log source CPU and replay it at the destination using check-pointing and replay mechanism. Checkpoint is used on top of the Xen to copy the memory to the destination machine while the source physical machine continues to run the VM. Meanwhile CPU tracking and logging is progressing. After that, the log is sent to the destination to replay the execution at the destination machine. The execution in the source machine must be slowdown and the destination must be faster. A critical issue here is the synchronization between the source and destination to track the source execution. If the logging generation is faster than the tracing the migration will not finish.

2.3 Summary

In this Chapter a detailed description about live VM migration techniques proposed in literature is provided. These live VM migration methods arranged in categories based on the enhancement mechanisms. Figure 2.2 shows the classification of the live VM migration methods.

There is a need to overcome the limitation of the most of these methods impediments where each live VM migration technique can enhance one metric or two but a degradation occurs to other metrics. A mechanism that can be integrated with virtualization setup in easy and seamless way is needed to enhance live VM migration process, especially in the data centers environments where virtualization infrastructure is based on clustering computing model with shared storage resources.

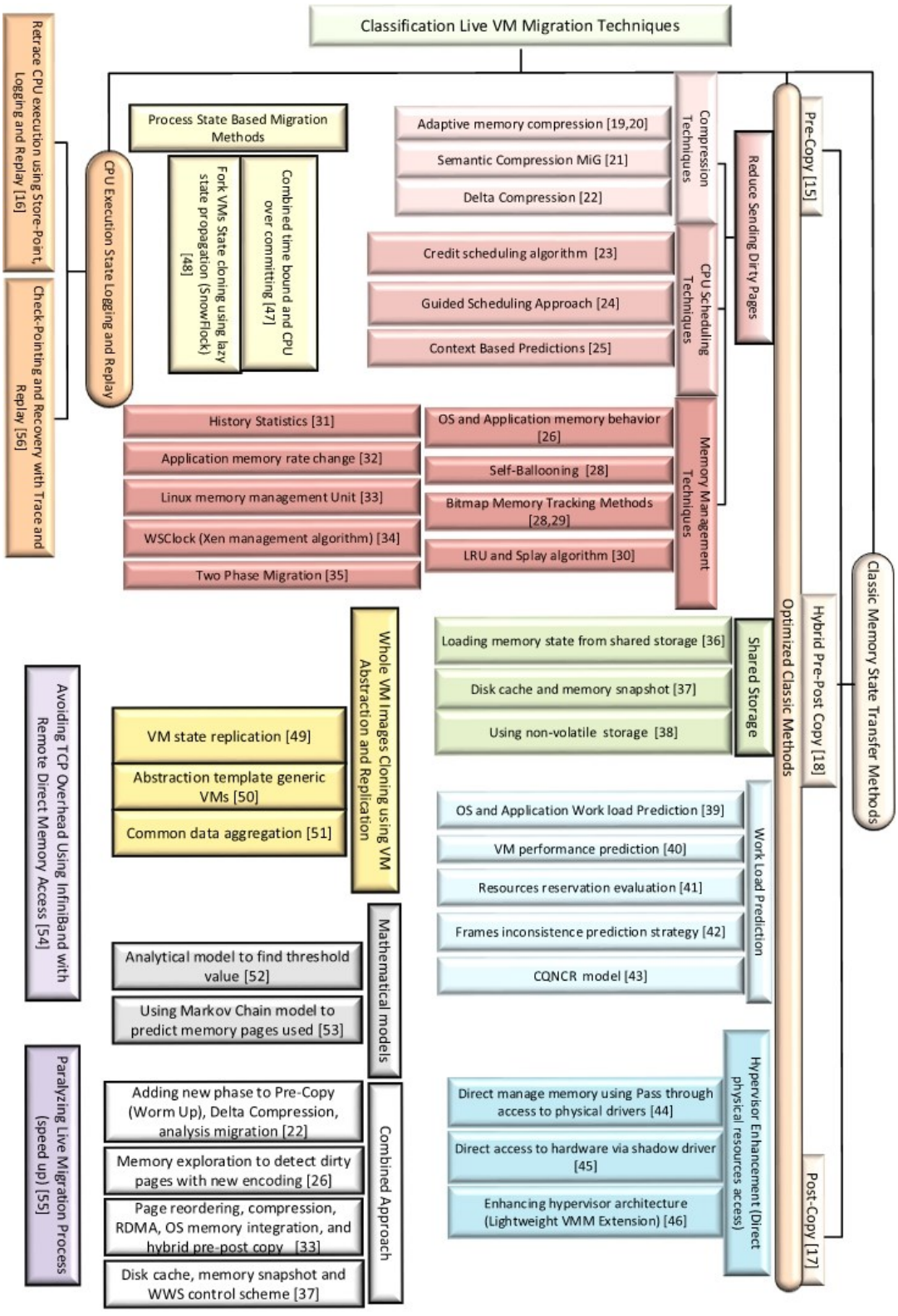


Figure 2.2: Live VM Migration Classification

Such a model is presented in Chapter 3 that makes use of salient features and functionalities of live VM migration more effectively by using Distributed Shared Memory based clustering capabilities to exchange memory data and updates, more effective and sending a lower number of memory dirty pages with high accuracy update, and with parallelization speedup using HPC computing model.

Chapter 3
Distributed Shared Memory (DSM)

In classic computing architecture, subsystem could be built to take advantage of idle memory in peer nodes on a local area network, namely, using remote memory as a paging device, instead of the local disk. The intuition behind that idea is the fact that networks are becoming faster. Therefore the access to remote memory may be faster than the access to an electromechanical local disk. We can exploit remote memories, software implementation of distributed shared memory, which makes computer cluster to appear like a shared memory machine. That is, create an operating system abstraction that provides an illusion of shared memory to the applications, even though the nodes in the local area network do not physically share memory. For this reason, distributed shared memory raises the question, whether the shared memory makes life simpler for application development in a multiprocessor, through providing that same abstraction in a distributed system, and making the cluster looks like a shared memory machine?

Distributed Shared Memory (DSM) implements a shared memory model on physical distributed memories across multiple computing nodes, in order to achieve ease of programming, cost-effectiveness, and scalability. This is accomplished using the private memories of the nodes by controlling access to pages of the shared memory and transferring data to and from private memories when necessary. Shared memory multiprocessors system, facilities the process migration by moving the process from the source processor queue to the destination processor in one single machine or multiple machines. Because of the process control block, code and stack segments will be in the same memory address space. In DSM, there is a shared virtual memory space between processors nodes. The shared virtual address space is a space shared by a number of processors and any processor can access all memory locations in the address space directly. A memory mapper driver is needed to run, maintain and update the changes between the private local memory of each node and the virtual shared memory address space as depicted in Figure 3.1. From

DSM point of view, cloud computing has numerous definitions, one of them is based on distributed shared memory, which implements the first idea of abstraction and resource sharing.

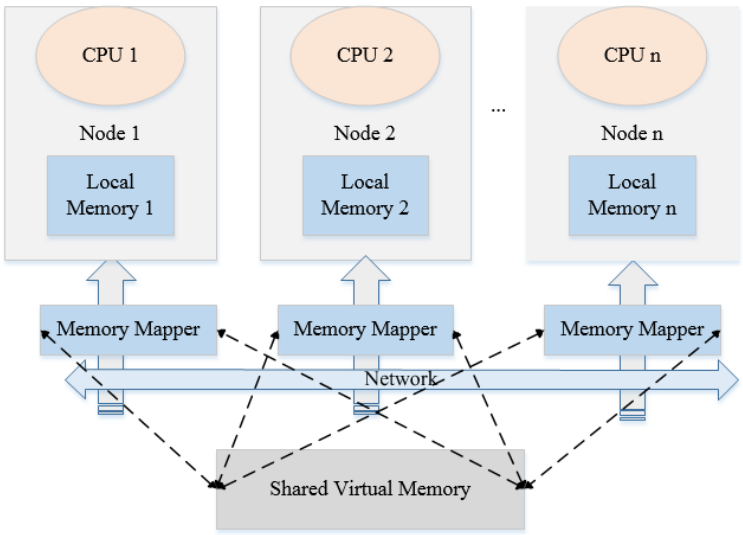


Figure 3.1: DSM General Design View

Cloud can be defined as an object based distributed operating system to provide a unified environment over distributed hardware. Location independence for data as well as processing, atomicity of distributed computation and fault tolerance are some of the research goals of cloud computing [57]. In general, cloud can exploit DSM management and implementation to incorporate operating systems with software, by implementing a set of primitive OS system calls on top of system kernel object-based OS.

The memory state transfer and update is a big challenge in all DSM related work, which needs to maintain the changes and update all hosts in the DSM cluster. DSM memory values coherency and update between processors nodes demand synchronization protocols, which have been developed to solve all shared memory update values issues. On top of that, this thesis uses the DSM to do the live VM migration to exploit the DSM protocols memory update, which speeds up the live VM migration while keeping the advantages of pre-copy live VM migration method.

3.1 DSM History Overview

A large number of DSM models have been proposed for memory consistency and coherency in order to achieve data freshness with minimum cost [57]. Now, having introduced distributed shared memory, this section provides a bird's eye view of the history of shared memory systems

over the last 20 plus years. The intent is not to go into the details of every one of these different systems, but just to give sort of the background about all the efforts that have gone on in building shared memory systems, both in hardware and in software. Software DSM was first thought of in the mid-80s. The IVY system [58] was built at Yale University by Kai Li and the Clouds Operating System was built at Georgia Tech and other similar systems built at UPenn. This, is the beginning of Software Distributed Shared Memory. Later on, in the early 90s, systems like Munin [59] and TreadMarks [60] were built. This would represent a second generation of Distributed Shared Memory systems. Moreover in the last half of the 90s, there were systems like Blizzard [57], Shasta [61], Cashmere [62] and Beehive [63] which are a distributed shared memory computing model that took some of the ideas from the early 90s even further. In parallel with the software DSM, there was also a completely different track that was being pursued. This track focused on providing structured objects in a cluster for programming. And systems such as Linda [64] and Orca [65], were done in the early 90s. Stampede [66] at Georgia Tech was done in concert with the Digital Equipment Corporation in the mid-90s and continued on, later on, into Stampede Rt and PTS, and Persistent Temporal Streams. This particular axis of development of structured distributed shared memory is attractive because it gives a higher level abstraction than just memory for computations that needs to be built on a cluster. The early hardware shared memory systems such as BBN Butterfly [57] and Sequent Symmetry [57] appeared in the market in the mid-80s and, synchronization method by Mellor-Crummey and Scott used BBN Butterfly and Sequent Symmetry as the experimental platform for the evaluation of the different synchronization algorithms. KSR-1 was another shared memory machine that was built in the early 90s. Alewife was a research prototype that was built at MIT, DASH [57] was a research prototype that was built at Stanford and both of them looked at how to scale up beyond an SMP, and build a truly distributed shared memory machine. SGI [57] silicon graphics built SGI origin 2000 as a scalable version of a distributed shared memory machine. SGI Altix [57] later on took it even further, thousands of processors exist in SGI Altix as a large-scale shared memory machine. IBM Bluegene [67] is another example. And today, if you look at what is going on in the space of high performance computing, it is clusters of symmetric multiprocessors (SMPs) which have become the work horses in data centers. Figure 3.2 summarizes the DSM evolution and types.

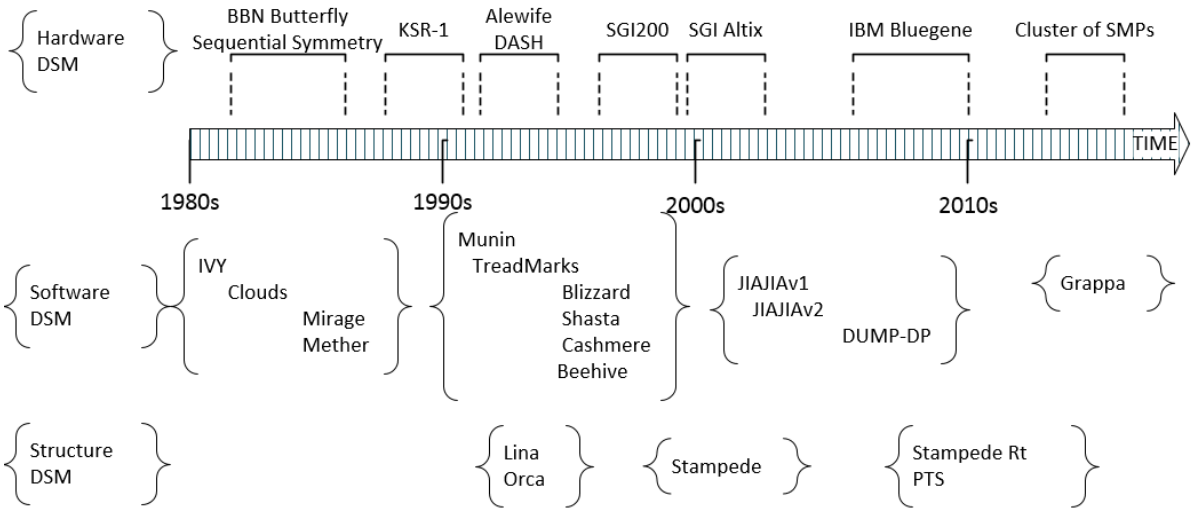


Figure 3.2: DSM evolution and types

The DSM system may consume a massive communication bandwidth to maintain the data values between processors, such as first DSM system IVY using sequential consistency, which propagates shared memory variables updates to all members. In addition to the coherence condition for the memory values, synchronization between processes is needed. The later DSM models proposed alleviated communication performance bottleneck by reducing network traffic messages and the shared memory values update. Majority of DSM proposed focused on memory consistency protocols that define the behavior of the shared memory in the DSM system and for the programmer as well. Lazy Release Consistency (LRC) in TreadMark DSM model and Scope Consistency (ScC) in JIAJIA V1.1 DSM model are examples of reducing data traffic through network while maintaining memory consistency. JIAJIA V2.1 proposed a home migration protocol, which minimizes data communication between processors to maintain memory coherency. Munin DSM utilizes the weak Eager Release Consistency (ERC) protocol to remove the propagation at acquire time. A weaker updating in consistency methods is used in Entry Consistency (EC) where the propagation update is reduced further. However an explicit programming statement must be stated for variables that need guaranteed update synchronization. JUMP-DP [68] is a DSM model that utilizes the best of the mentioned DSM models and memory consistency methods with network communication enhancement. JUMP-DP is a migration home protocol based on the page home location to be migrated from a processor when suffering page fault, a dynamic home will fit better than fixed home in many memory pattern applications. If the home processor never accesses the page, then the update will be from other processors during the

synchronization time. The mechanism of this scheme produces less time propagation update for messages in synchronization phase. In addition to the message interaction relaxation, a network communication enhancement is used to speed up the communication channel. In [70] the authors introduce a new low latency software tolerance distributed shared memory called Grappa, which is based on a high data-intensive application to run on commodity cluster computation setup. This model scale up the application (software) performance even if the application has poor memory locality or even for a high dependency application input. Grappa is optimal in loading the application in cluster computing, because its properties of exploiting application parallelism with low communication latency, high throughput and minimum messages passing. This way will overcome most of DSM limitations, like the poor application performance, high data exchanges and the high memory coherency update. This is due to the limited inter-node bandwidth, high internode latency, and the design decision of piggybacking on the virtual memory system for seamless global memory accesses. Grappa can work under different cluster models such as symmetric multiprocessors (SMPs), or unsymmetrical multiprocessors cluster nodes, which adopts the shared-memory, fine-grained parallel programming mindset from the Multi-Threaded Architecture (MTA). Grappa includes an overlay network that combines small messages together into larger physical network packets, which reducing number of exchange messages, thus maximizing the available bandwidth of networks. This communication layer is built in the user-space, utilizing modern programming language features to provide the global address space abstraction. Efficiencies come from supporting the sharing of a finer granularity than a page, avoiding the page-fault trap overhead, and enabling compiler optimizations on global memory accesses.

3.2 DSM Cluster Models

The DSM cluster can be based on various messages and computation schemes such as using message passing or shared memory as data exchanges, and to execute code in parallel or in sequential as following:

3.2.1 Cluster as A Parallel Machine Sequential Program

Suppose that we have a sequential program. Multiple considerations exist on the optimal way in which the cluster could be used along with the multi-processors environment in order to achieve application speed up or enhancement. One possibility is to do what is called automatic

parallelization. That is, instead of writing an explicitly parallel program, we write a sequential program. The heavy lifting task is left for somebody to determine the identification opportunities for parallelizing the program and map it to the underlying cluster. This procedure is known as implicit parallel program. There are opportunities for parallelism, but the program itself is not written as a parallel program. Therefore it is the onus of the tool, such as the automatic parallelizing compiler, to look at the sequential program and identify opportunities for parallelism and exploit that by using the resources that are available in the cluster. High-performance FORTRAN (HPF) is an example of a programming language that does automatic parallelization, but it is user-assisted parallelization in the sense that the user who is writing the sequential program is using directives for distribution of data and computation. Those directives are then used by this parallelizing compiler to say, these are opportunities for mapping these computations onto the resources of a cluster as Figure 3.3 shows. Therefore, it puts it on different nodes on the cluster and in that way, it exploits the parallelism that is there in the hardware, starting from the sequential program and doing the heavy lifting in terms of converting the sequential program to a parallel program to extract performance for this application. This kind of automatic parallelization, or implicitly parallel programming, works very well for certain classes of program called data parallel programs. In such programs, for the most part, the data accesses are fairly static, and it is determinable at compile time. In other words, there is limited potential for exploiting the available parallelism in the cluster if we resort to implicitly parallel programming.

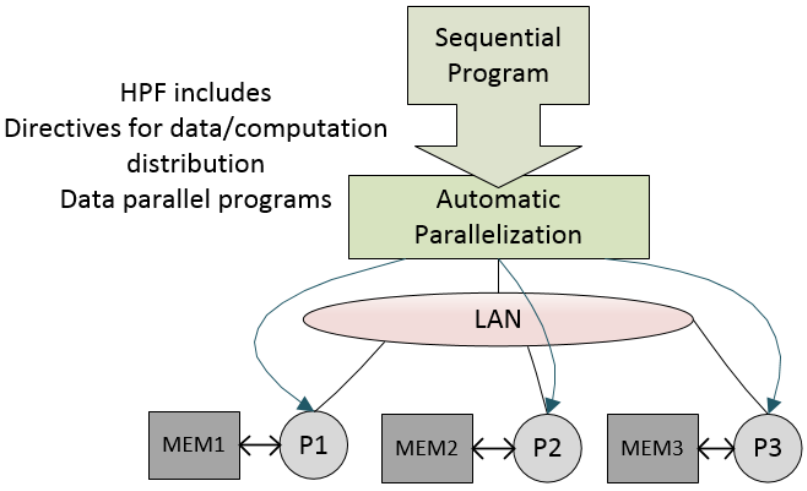


Figure 3.3: Cluster as a Parallel Machine Sequential Program

3.2.2 Cluster as A Parallel Machine Message Passing

A program is written as a truly parallel program, or in other words, the application programmer is going to think about his application and write the program as an explicitly parallel program. Actually, there are two styles of writing explicitly parallel programs. Correspondingly, there are system support for these two styles of explicitly parallelized programs. One is called message passing style of explicitly parallelized program and the other is shared memory space parallelism. In the message passing scheme, the run time system provides a message passing which has a good number of primitives for message passing interface. An application thread is used to send and receive from its peers that are executing on other nodes of the cluster. This message passing style of explicitly parallel program is true to the physical nature of the cluster, which can send and receive processing updates between the cluster nodes. Figure 3.4 shows the physical nature of the cluster, where the fact that every processor has its private memory. This memory is not shared across all the processors. Therefore the only way a processor can communicate with another processor is by sending a message through the network where the destination processor can receive. The sender processor cannot directly reaches the memory of the destination processor, because that is not the way a cluster is architected. So, the messaging passing library is a true fit to the physical nature of the cluster, that there is no physically shared memory. There exists several examples of message passing libraries that have been written to support explicit parallel programming. A cluster includes Message Passing Interface, MPI for short, PVM, and CLF from digital equipment corporations. These are all examples of message passing libraries that have been built with the intent of allowing application programmer to write explicitly parallel programs using this message passing style. Currently, many scientific applications running on large scale clusters in national labs like Lawrence Livermore, Argonne National Labs and so on, use this style of programming using MPI as the message passing fabric. The only downside to message passing style of programming is that it is difficult to program using this style.

The second style of writing explicitly parallel program, the shared memory space parallelization, is explained in the next Section.

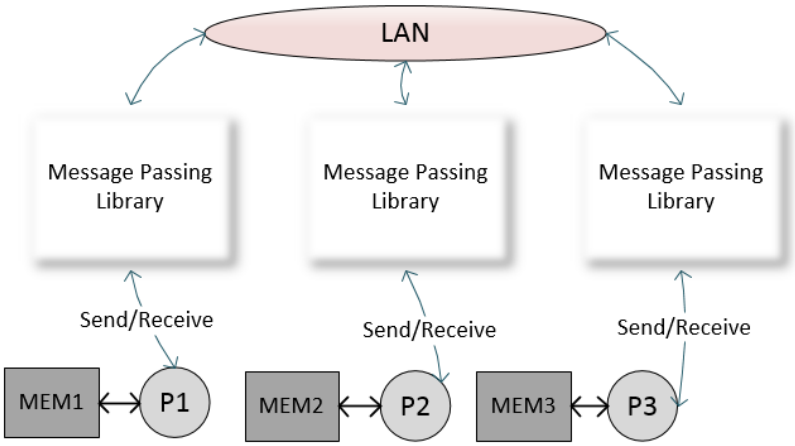


Figure 3.4 Cluster Message Passing Architecture

3.2.3 Cluster as a Parallel Machine DSM

It is easier for programmer who writes a sequential program to write an explicitly parallel program and to maintain the transitions communication using the shared memory model. Notion of shared memory is natural to think of shared data structures among different threads of an application. And that's the reason making the transition from sequential program to parallel programming, using for instance the POSIX thread library or SMP, is fairly intuitive and easy pathway. On the other hand, if the programmer has to think in terms of coordinating the activities on different processes by explicitly descending and stopping messages from their peers, then it is more comfortable to deal with programming variables and function calls through one accessible memory. That is calling for a fail radical change of thinking in terms of how to structure a program.

This was the motivation for coming up with this abstraction of distributed shared memory in a cluster. The idea is based on the illusion of giving the application programmer the ability to write an explicitly parallel program, where the entire cluster memory is shared. However, the memory is not physically shared; rather the DSM library is going to give the illusion to the threads running on each one of these processes that all of this memory is shared. Therefore the shared memory style have an easier transition path for instance, from going from a sequential program or going from a program that have been written on an SMP to a program that runs on the cluster, because programmer do not have to think in terms of message passing. However, they can think in terms of shared memory, sharing pointers across the entire cluster, and so on. Also, since a shared

memory semantic in the DSM library for the application program is provided, there is no need for marshalling and un-marshalling arguments that are being passed from one processor to another. All of that is being handled by the fact that there is shared memory, so when a procedure call is issued, and that procedure call is accessing some portion of memory that happens to be on a remote memory, that memory is going to magically become available to the thread that is making the procedure call. In other words, the DSM abstraction gives the same level of comfort to a programmer who used to program on a real shared memory machine when they moved to cluster. Because they can use same set of primitives, like locks and barriers for synchronization, and the Pthread style of creating threads that will run on different nodes of the cluster. This is the advantage of DSM style of writing an explicitly parallel program as Figure 3.5 explains.

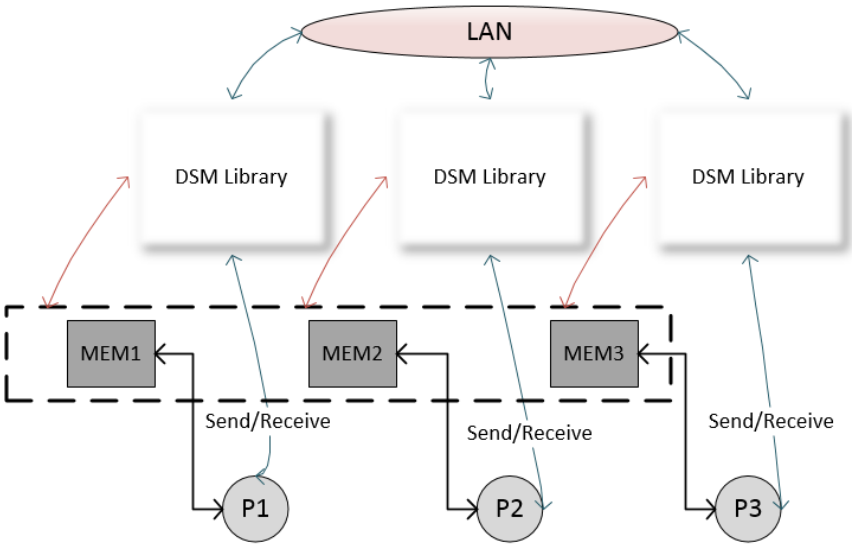


Figure 3.5: Cluster DSM Architecture

3.3 Shared Memory Programming

As discussed earlier, synchronization is considered a crucial aspect in shared memory operation. Lock is a primitive and particularly the mutual exclusion lock is a primitive that is used ubiquitously in writing shared memory parallel programs to protect data structure so that one thread can exclusively modify the data and release the lock so that another thread can access the data later on. Similarly, barriers are another synchronization primitives that are very popular in scientific programs about what the operating system has to do in order to have efficient implementation of locks as well as barriers. Now the upshot is, if a shared memory program is to

be written, two types of memory accesses are going to happen. One type of memory access is the normal reads and writes to shared data that is being manipulated by a particular thread. The second kind of memory access is going to be for synchronization variables that are used in implementing locks and barriers by the operating system itself. The support for these primitives may be using the operating system, or it could be by a user level threads library that is providing these mutual exclusion locks, or barrier primitives, but in implementing those synchronization primitives, those algorithms (the lock and barrier) are going to use reads and writes to shared memory, hence there are two types of shared memory accesses going on in the execution of a parallel program. One is access to normal shared data and the other is access to synchronization variables.

3.4 Summary

In this chapter, DSM system is discussed in details based on numerous clustering architecture where a DSM cluster model with parallelization of running code can be the best choice to deploy the live VM migration process.

Chapter 4
DSM Live VM Migration Infrastructure

In this thesis work the environment setup consists of one physical structure and three logical structures. The physical setup shows the real physical hardware that is used to build the DSM cluster virtualization environments. The three services that are needed to support live VM migration, which are virtualization, shared storage and DSM with HPC cluster. In next sections the physical structure and the logical structure are explained in detail.

4.1 Physical Infrastructure

In this work, two identical Dell workstations (named Worth and Unhand) with high speed processor 4 cores of Intel Xeon 3.6 GHz speed, connected by Linksys Ethernet switch with port speed 100Mbps are deployed as Figure 4.1 depicts. In Tables 4.1, 4.2 and 4.3 a summary of the number of hardware, their specifications and network physical and logical addresses is provided.

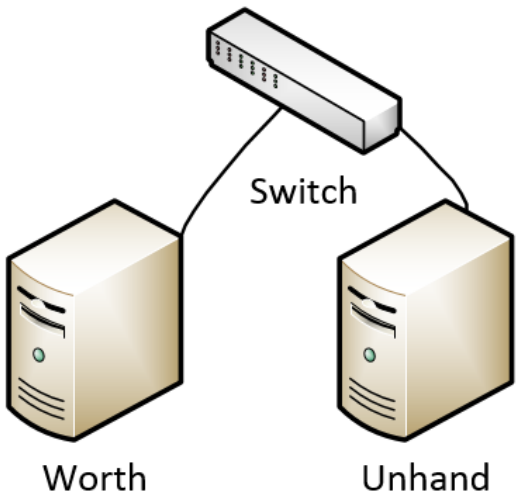


Figure 4.1: Physical Setup

Hardware Types	Number
Dell Precision T1700 (Worth and Unhand)	2
Linksys Etherfast cable/DSL Router with 8 Ports 100Mbits/s	1

Table 4.1: Physical Hardware Components

Dell Precision T1700 Specifications	
CPU	Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz (4 Cores)
Memory (RAM)	32G Byte Kingston 1600 MHz (0.6 ns)
Storage (Hard Disk)	1T Byte ATA Disk
Network	Intel Ethernet Connection I217-LM 10/100/1000Mbps
OS	Citrix Xenserver (Linux Centos 5.6 Custom)
Xen Kernel	2.6.32.43-0.4.1.xs1.8.0.835.170778xen

Table 4.2: Workstation Hardware Specification

WorkStation Name	Physical Address (MAC)	Logical Address (IP)
Worth	98:90:96:E1:DA:9B	132.205.19.14
Unhand	98:90:96:D9:C6:29	132.205.19.4

Table 0.3: Network Information

Each Dell server run custom version of Linux OS (CENTOS5.6) with support Xen kernel 2.6.32.43-0.4.1.xs1.8.0.835.170778xen. The network connection is limited by the switch speed capabilities, which is only 100Mbps, the two workstations adjust the network card speed to 100Mbps.

The other three logical setup of this model represents the deployment of the Xenserver hypervisor architecture, the shared storage server (NFS) and the high performance cluster with the DSM.

4.2 Logical Infrastructure

The logical setup services are working in a cooperative way to handle the live VM migration to implement the XenServer motion the name of live VM migration in an optimized way, the enhancement of the XenServer motion using the NFS shared storage and the DSM HPC cluster to speed up the XenServer VM motion. The building block architecture of the proposed live VM migration is composed of three services layers to facilitate the migration through running the hypervisor migration process as a job in the DSM HPC cluster computation. Figure 4.2 is viewing the conceptual architecture of the model blocks components and communication flows.

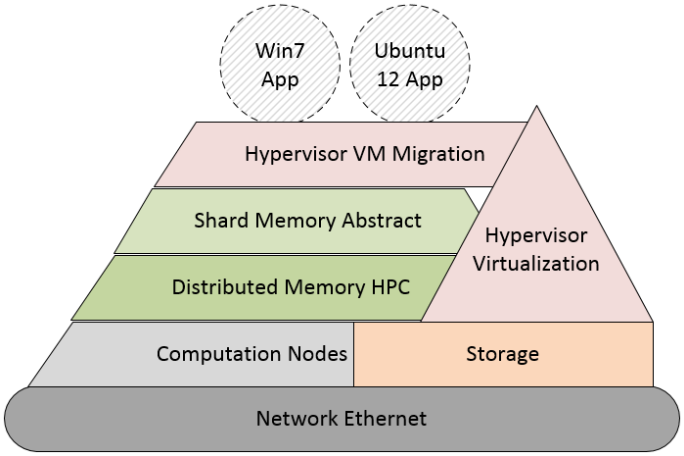


Figure 4.2: Proposed Block Model

4.2.1 Shared Storage Network File System (NFS) server

In datacenters architecture a unified storage that is accessible to common application services servers is used, to provide a shared storage pool. A very popular shared storage protocols is Storage Area Network (SAN) that is connected to servers using a private fiber link using fiber channel switches that guarantee free transmission errors. This kind of storage is used for enterprise services architecture for its reliability and speed. The reliability come from the storage array Redundant Array of Independent Disks (RAID) system, which supports zero time of recovery for hard disk failures such as RAID5 or RAID10 high data recovery converges of hardware fault. The SAN storage can support security using Logical Unit Number (LUN) for traffic isolation of services and to logically divide the SAN storage into logical storage segments to serve different kind of applications servers; for example one SAN can serve three servers farms (web server's farm, database server's farm, and mail server's farm). But this solution is costly and need dedicated hardware for storages with Internet Small Computer System Interface (iSCSI) connectors interface for the fiber channel switch and a fiber switch. In addition to a separate network infrastructure the private segment connects the servers to the SAN (fiber links) and the public or production network (Ethernet links) that connects users to servers.

A second shared storage solution, Network Attached Storage (NAS), tries to consolidate the network infrastructure so it can use the Ethernet Network to connect servers with storages without need to a dedicated fiber network. This technology uses Network File System with Common Internet File System (NFS/CIFS) protocol to allow servers to read from NAS or write to it and for

data synchronization to update any changes between servers and the data storage. Again this solution needs a dedicated hardware NAS server to serve the datacenter. Figure 4.3 shows the difference between SAN and NAS storage.

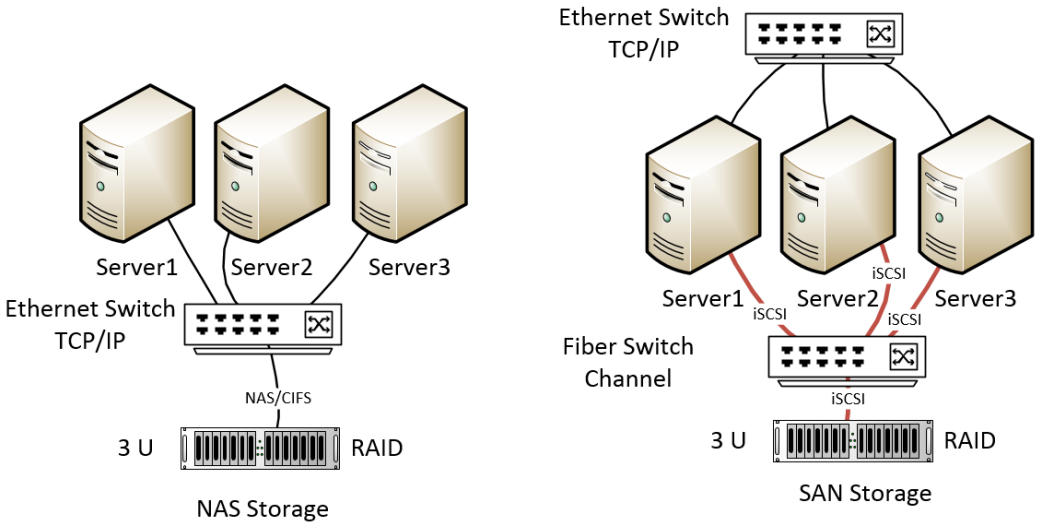


Figure 4.3: Difference between NSA and SAN Storages

The third shared storage solution is Network File System, which is a distributed file system protocol that allows NFS server to grant access to its storage for NFS clients directly as a shared disk for all clients' machines using Remote Procedure Calls (RPC). By using the NFS protocol any running server can share its local storage with the group server farms without any dedicated network or hardware servers or switches. The NFS service can mount segment of the storage using Logical Volume partitions for the hard disks which provide security and management to the storage in addition to the reliability support by applying the RAID array to the NFS server storage disks. A good benefit of using NFS protocol is the lower cost to deploy in datacenters and its ease of running it with IT services application. A mandatory disk configuration must be done before deploying the NFS by configuring the Logical Volume LVM partitioning to allow dynamic disk segmentation be exported and mounted by NFS server and clients. NFS protocol support authentication and authorization for clients support low level file or directory access.

NFS is a transparent protocol that allows the shared storage server update to be synchronized with all clients. In Figure 4.4, Worth server exports its local storage as a shared storage using the NFS Linux demand service. The NFSd needs to work a network ports map communication that must configure in NFS configuration file. A privilege and access policy must be defined in export file

of the Centos Linux export configuration file. The clients need to have the NFS client demon. It then must mount the NFS storage to local directory where the size of the mounted storage is equal to the size of the server shared storage.

The server configuration and client configuration is described below:

Server Configuration export file (/etc/exports) configured:

*/export/vdisk *(rw,no_root_squash,sync)*

*/export/viso *(rw,no_root_squash,sync)*

Clients mount configurations in Worth and Unhand workstations commands:

mount -F nfs worth:/export/vdisk /vdisk

mount -F nfs worth:/dsm /dsm

After configuration the NFSd service must be restarted "*service nfs restart*"

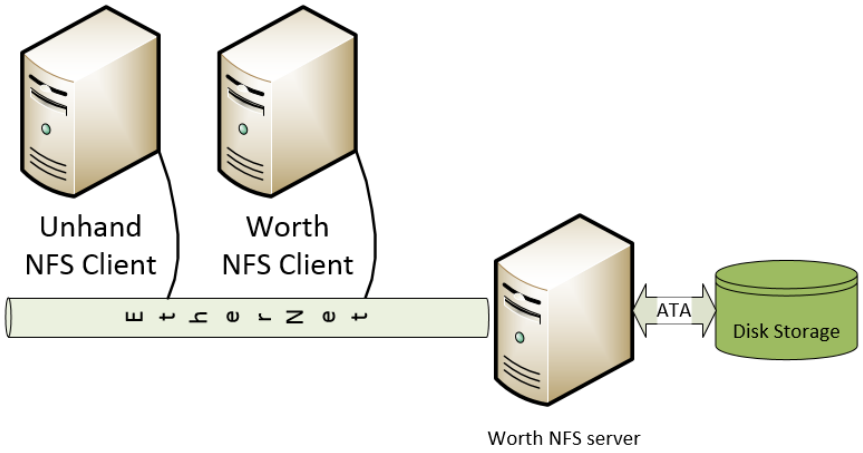


Figure 4.4: NFS Setup

The /vdisk directory is the shared storage to save the VM images, and the /dsm directory is used to save the HPC cluster and DSM common information.

4.2.2 Virtualization Infrastructure

In this thesis work Citrix XenServer version 6.2 is used to create the virtualization infrastructure, with CitrixCenter management console, which is a software for managing VMs and virtual machines templates.

4.2.2.1 Virtualization Objects

The virtualization resources used in this setup are two virtual machines, the first machine hosted windows7 named Flake and the second virtual machine hosted Flask. Figure 4.4 shows the XenCenter console of Flake. Table 4.4 summarize the Flake VM hardware specifications with its logical network address. Figure 4.5 shows the XenCenter console of Flask, which is Ubuntu12 Linux virtual machine and Table 4.5 explains the hardware specifications and network logical address.

To create the virtualization infrastructure a virtual server pool named (DSM pool) is created. This pool includes both the host servers, Worth and Unhand. Once both host servers join the virtualization pool, it must have a master node, which is the Worth host server. This server have all the virtualization configuration files and monitoring reports from physical host server and the VMs guests machines. The XenCenter, which is a management tool for managing the XenServer hypervisor in a graphical way. The XenCenter console directly connects to hypervisor using port TCP 5900 for management and TCP 443 (secure http 'https') for authentication and confidential communication. Also the XenCenter connects to the guest VM desktop using Virtual Network Computing (VNC) protocol, which is a simple protocol for remote access to Graphical User Interface (GUI). Figures 4.6 and 4.5 show the desktop of Windows and Linux OS.

In XenServer motion (the live VM migration) the pre-copy method is used to achieve the live VM migration. The two virtualization servers (hypervisor) share the same storage using the NFS server. Both hypervisors nodes are in the same Local Area Network (LAN) as a condition to run the live VM migration between the two physical servers. The VM must be in running state to do live VM migration. The XenServer can move the VMs in both directions form Worth to Unhand and from Unhand to Worth.

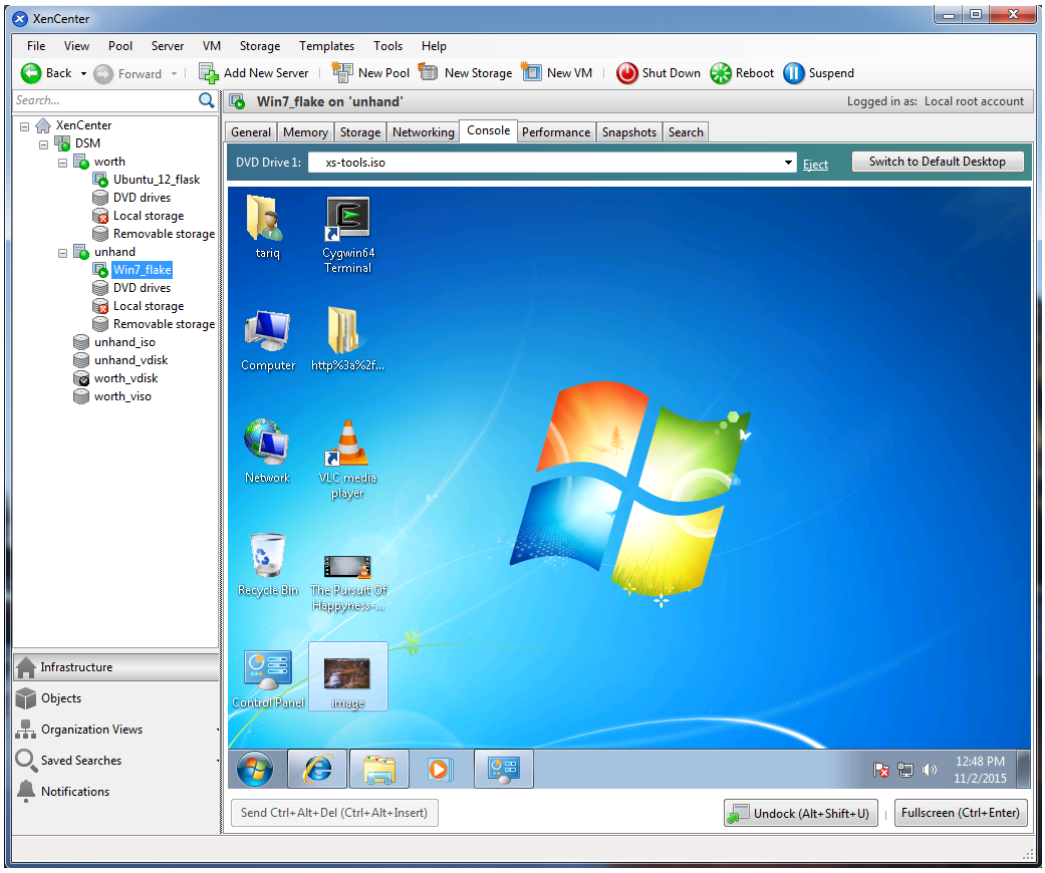


Figure 4.5: Flake VM XenCenter Console

Win7 Virtual Machine (Flake)	
CPU	Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz (1 Core)
Memory (RAM)	4 G Byte
Storage (Hard Disk)	80G Byte
IP address	132.205.19.12

Table 4.4: Windows VM Specification

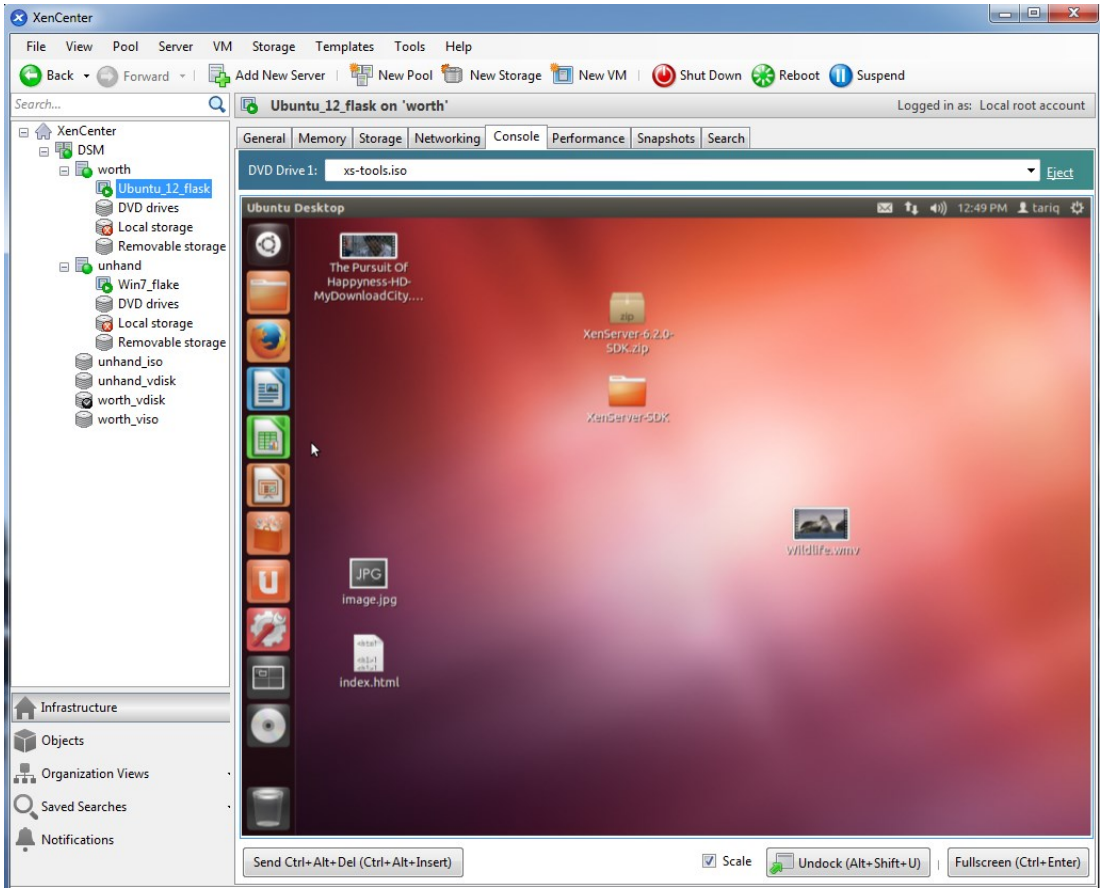


Figure 4.6: Flask VM XenCenter Console

Ubuntu12 Virtual Machine (Flask)	
CPU	Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz (1 Core)
Memory (RAM)	4 G Byte
Storage (Hard Disk)	80G Byte
IP address	132.205.19.13

Table 4.5: Ubuntu Linux VM Specification

XenCenter supports most of Xen hypervisor commands, to provision new virtualization objects and to control the VMs lifecycle: create, start, running, suspended, paused and power down states as Figure 4.7 shows. A resource monitoring tool run in hypervisor privilege domain Dom0 to

collect the virtualization objects and synchronize the update with the XenCenter tool with real time logs update of each object and sub object, like the VM NICs, CPU, Memory and Disk.

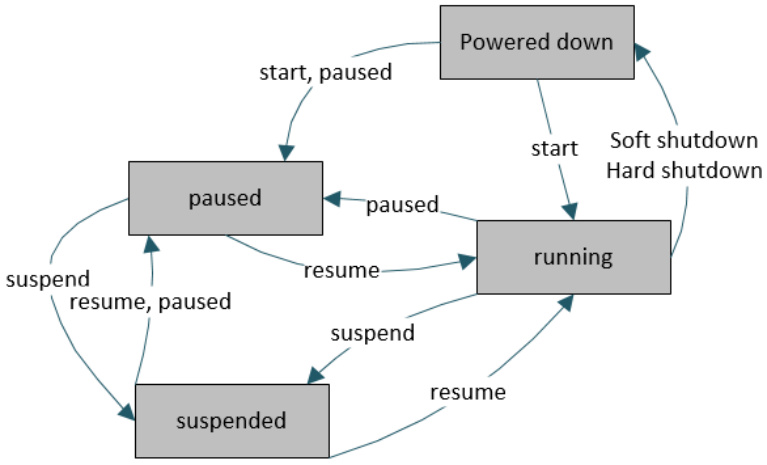


Figure 4.7: VM Lifecycle

For each virtualization infrastructure (host servers and guest VMs) there are 9 tabs that include different information about the server or VMs. The tabs information are described in Table 4.6. XenServer has a network virtualization switch, the vSwitch controller, which allows the guest VMs to communicate with outside network. In XenServer XAPI *xe network-list* can list network information about the hypervisor network bridge information of the host network server.

uuid (RO) : 1fb13301-d76d-8b57-3b54-7036fc89013d
name-label (RW) : Host internal management network
name-description (RW) : Network on which guests will be assigned a private link-local IP address which can be used to talk XenAPI
Bridge (RO) : xenapi
uuid (RO) : b6f253e3-8952-d819-e83c-dde62d5a74c0
name-label (RW) : Pool-wide network associated with eth0
name-description (RW) : bridge (RO): xenbr0

Tab Name	Tab Description
General Tab	General information about the server or VM like host name, IP address, memory size, CPU, XenServer version and uptime.
Memory Tab	Shows memory utilization like the free memory size and used memory size.
Storage Tab	The hard disk information, its size and locations, and the usage ratio in addition to the NFS mounted information.
Networking Tab	The Xen virtual switch and network connection information, and the Dynamic Host Configuration Protocol (DHCP) configuration.
NICs Tab	The network interface configuration and status.
Console Tab	Shows the remote desktop Graphic User Interface (GUI) console.
Performance Tab	The CPU, Network, Disk, Memory utilization.
Users Tab	Authorization and user management logging and accounting.
Search Tab	Tab to search virtualization objects in the whole virtualization infrastructure.

Table 4.6: XenCenter Tabs Information

4.2.2.2 *XenServer Virtualization Architecture*

XenServer is a bare-metal hypervisor type which is installed directly on the hardware as explained in Chapter 1, Section 1.2 on the hypervisor types. To install the XenServer safely a custom configuration must be set to change the Dom0 configuration to allow it to support HPC cluster configuration. The basic setting for Dom0 is to increase the hard disk size and memory size to handle the DSM and HPC compilation and installation, the python *constants.py* installation script, located in */opt/xensource/installer* directory. This file contains the default initial XenServer installation configuration parameters. The new values of *root_size* disk installation space of Dom0 is changed from 4096GB to 80GB, and the *swap_size* memory of Dom0 changed from 512MB to 1024MB. The Global Partition table (GPT) parameter *GPT_SUPPORT* must be disabled to get full access to Dom0 of Master Boot Record (MBR) partition tables. After installation of the two hypervisors on the two computation nodes (the host servers), the initial configuration is to create the virtualization resources pool that contains all virtualization objects. The virtualization objects are grouped by resource pool, this resource pool has full control to the created VMs and physical

server to construct the virtualization infrastructure platform, with full access to the NFS shared storage.

The created VMs are stored in the NFS shared storage */export/vdisk/16785d73-7317-1685-addb-09d0599428d6/* at worth NFS server side. The storage directory name is the pool ID as the *xe pool-list* can show the objects and sub objects, which are indexed by unique IDs automatically generated by the command below. The default Storage Repository SR with read and write privileges has the same value of the directory name of the VM virtual disks stored.

```
xe pool-list
uuid ( RO)                    : bb694502-00b7-dd85-25d5-76c56428b9f9
        name-label ( RW)      : DSM
    name-description ( RW):
        master ( RO)          : aa62801f-6a07-4c2a-a29d-5d6b7327fa44
        default-SR ( RW)      : 16785d73-7317-1685-addb-09d0599428d6
```

The live VM migration of the running VMs in XenServer named XenMotion, is done by the pre-copy method. The migration function can be called and controlled using the XenServer xe command line utility or through programming using the XAPI using C++ language. Figure 4.8 depicts the virtualization architecture that includes the two host servers Worth and Unhand, using the NFS shared storage server on Worth. The two virtualization servers (hypervisors) and the NFS shared storage must be in the same Local Area Network (LAN) as a condition to run the live VM migration between the two physical servers Work and Unhand, which host the two guest VMs Flask and Flake.

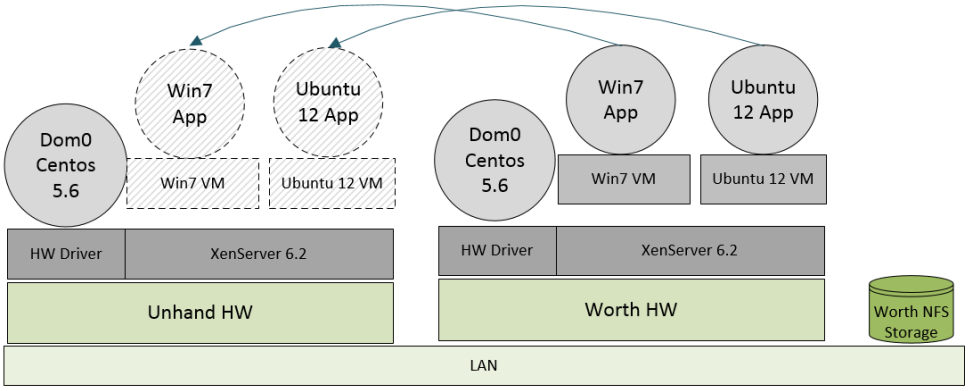


Figure 0.8: XenServer with NFS Server Setup

The control of live migration can use the XenCenter console or use the Xen XAPI calls (the *xe* utility commands), which gives more capabilities to control the hypervisor servers and VMs properties and states. This XAPI can work and be called from the virtual pool host server members, and be executed through the pool master node. Next section provides an explanation about the migration producer using the proposed custom migration code.

4.2.2.3 *Custom Live VM Migration Program*

In order to control the live VM migration it must be called using a programming system calls for XenServer. A standard development kit SDK is developed to allow Xen developers to modify the XenServer default function that includes the source code of the XenServer libraries in addition to binary libraries to support XEN API. In this work a custom program had been built to call the migration function of the XenServer and to integrate the migration function with the DSM cluster.

The proposed migration program named (*norm_vm_mig.c*) is divided into five phases as following:

Phase 1: the initialization state of the selected VM parameters, with source and destination servers' information.

Phase 2: new Xen session created to access the class of Xen VMs and its functions and attributes.

Phase 3: a virtualization object status check and live migration eligibility check for all running VMs in the virtualization infrastructure management node (Worth in this case).

Phase 4: a task object group data structure created to hold all VMs that have passed the migration test.

Phase 5: check the VM if it is a member of the task object group, then call the migration function to initialize the Xen migration API.

Figure 4.9 shows the application use case diagram scenario.

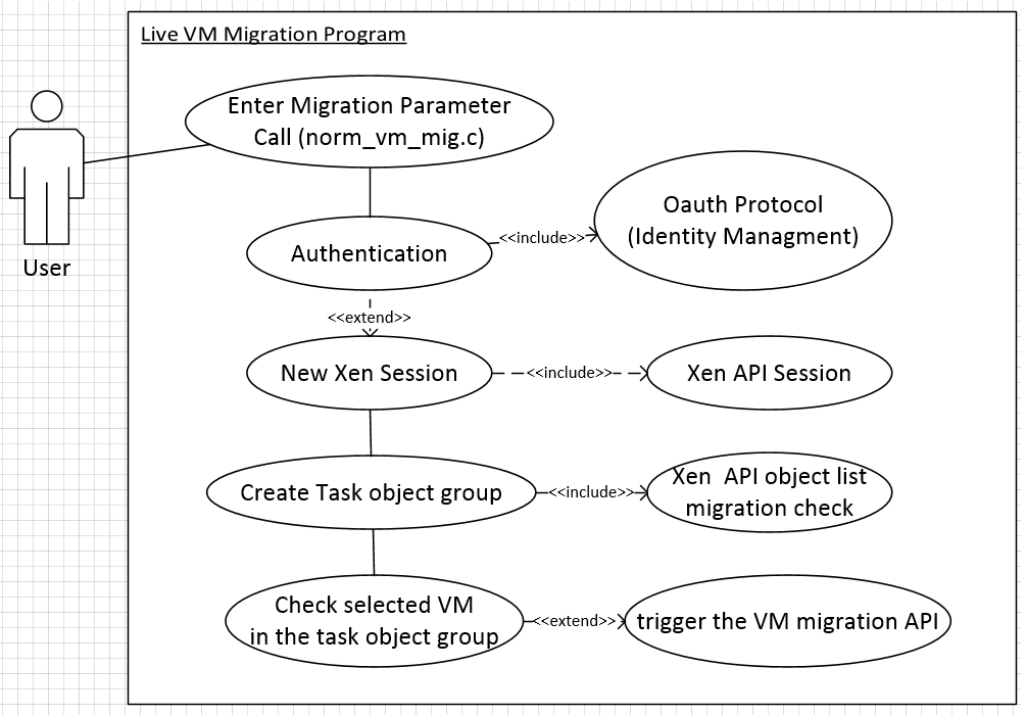


Figure 0.9: VM Migration Program Use Case Scenario

The Xen C development libraries are all included in one API header directive file, which is included in the program header:

#include <xen/api/xen_all.h>

The program *norm_vm_mig.c* is compiled using Makefile and an executable file is generated with same name of the migration program (*norm_vm_mig.out*). The program runs under Linux OS using normal Linux Command Line Interface (CLI) by calling the program with its parameters, (*./norm_vm_mig.out https://worth root password worth unhand Win7_Flak*). The program parameters are defined as following:

First argument (argv[1]): The virtualization master node URL, by using curl library to hold the master node Unified Resource Locator (URL) address (https://worth).

Second argument (argv[2]): the virtualization infrastructures username (root).

Third argument (argv[3]): the virtualization infrastructures password (password).

Fourth argument (argv[4]): the source physical machine (the machine that currently hosts the selected VM to migrate) Worth.

Fifth argument (argv[5]): the destination physical machine (the machine that will host the selected VM to migrate) Unhand.

Sixth argument (argv[6]): the migrated VM.

The *norm_vm_mig.c* program is equivalent to the XenServer migration utility command (*xe vm-migrate vm=Win7_flake host=worth --live*), which can run again using the Linux CLI terminal. This program is very important to allow the live VM migration to run on top of HPC cluster. Section 4.2.3 explains the HPC cluster architecture, deployment and how application can run through the cluster.

4.2.3 DSM HPC Cluster Migration Framework

The high performance computing cluster is built using Message Passing Interface (MPI) communication computing library, using distributed memory for cluster nodes communication as a computing node to distribute the tasks through the cluster members (Worth and Unhand).

4.2.3.1 HPC Cluster Authentication Setup

To create the HPC cluster a secure communication channel must be set up using the Secure Shell (SSH) protocol so that a public key exchange needs to be done to allow computation nodes to allow login access without using username and password. The SSH protocol can generate the public key by issuing *ssh-keygen* command to create the authentication public key security resulting in the following:

```
Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:
```

Your identification has been saved in /root/.ssh/id_rsa.

Your public key has been saved in /root/.ssh/id_rsa.pub.

The key fingerprint is:

ca:e7:b4:f1:2c:f8:e8:2a:a2:0a:3e:97:ba:2c:92:66 root@worth

The generated public key is stored in the "*/root/.ssh/id_rsa.pub"* file and needs to be copied into the "/*root/.ssh/authorized_keys.dsmadmin"* to grant access between the computation nodes; Worth must have Unhand Key and Unhand must have Worth key. The *known_hosts* file in */root/.ssh* directory holds the host's computation nodes *ssh-rsa* exchanged keys.

4.2.3.2 *HPC Cluster Distributed Memory with Message Passing*

In general HPC clustering the distributed memory concept is mandatory to support parallel programming, which requires the use of explicit message passing (MP), to allow processors to communicate. The processors parallel task load and programming is very tedious and complicated to implement using message passing by inter-processor communication libraries. But the enhancement of parallelization relies on the program architecture and it's threading numbers, most flexible parallelization method by message passing between processors or using shared memory space. Nowadays there is an established standard for message passing called MPI (Message Passing Interface) that is supported by all high performance computing vendors.

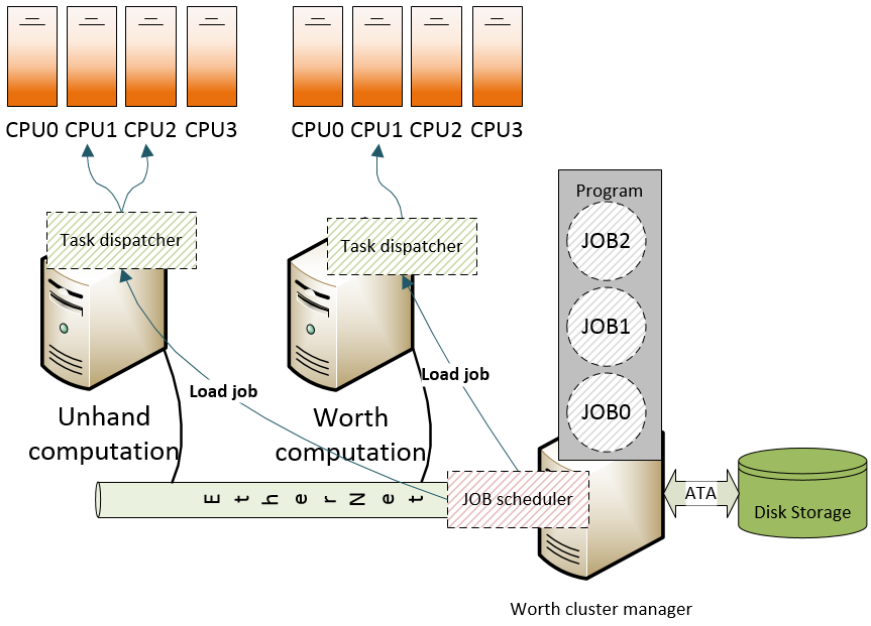


Figure 4.10: Scheduling of HPC Jobs

The basic inter-processor communication in the cluster nodes is to send messages between the computation CPUs which share the work load. The Worth host is used as HPC manager node in addition to its turn as a computation node. The cluster manager node controls the jobs assignment task, as depicted in Figure 4.10.

The HPC cluster provides parallelization service to the codes segments, based on the code attributes, where code can be run in parallel by using auto-parallelization capability of the OPENMPI library. If the code supports parallel computation it will be easier to run in HPC setup with more performance throughput. The cluster communication work based on the Message Passing Interface (MPI processors communication paradigm) between computing nodes to allow the inter-processors communication the message passing is controlled by standalone libraries like (MPICH, MVAPICH2, Intel MPI, and OPENMPI). These libraries provide the inter-processes communication within the cluster nodes and the cores of the same CPU in each computation node. The sequence flow diagram of the cluster job schedule is depicted in Figure 4.11, where all the communication between the computation nodes processes is done through the MPI messages to access the other process memory space or variable value. A sophisticated sending and receiving programming messages and program flow control blocking and non-blocking methods need be considered to implement inter-processes communication synchronization.

In the cluster task assignment model the tasks are loaded into cluster computation based on three running modes: serial program execution, parallel program execution in single node using OPENMPI and parallel program execution through multiple nodes using MPI. The job assignment is controlled by programs primitives to guide program portion execution and program segments home moving from one node or cores to others. In general, the best way to run any task under parallel execution cluster is to control the source code execution segments and to load each program portion and tie it with cluster node ID and core ID, but with sequential program in parallel cluster with automatic parallelization this happens in one computation node with binary parallelization for applicable program segment in the same computation node. The OPENMPI library is used to enhance serial program execution in the same single computation node using automatic (*#pragma omp parallel*) directive, which can scan the program flow dependency and optimize the execution time as parallel execution. The core assignment for program segments is based on cluster OS CPU scheduling with scheduling modification capability using parallelization primitives.

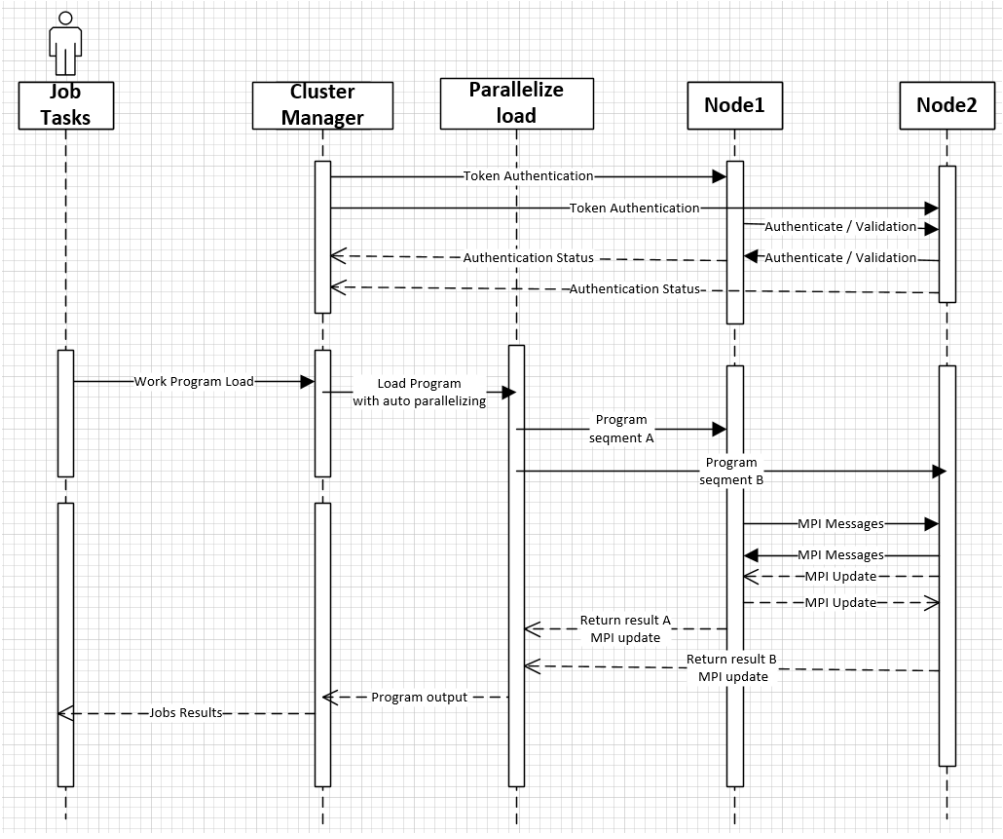


Figure 4.11: HPC MPI Job Sequence Diagram

4.2.3.2.1 HPC Cluster Configuration

The cluster architecture depends on the Linux OS distribution. In this proposed cluster the custom Linux CENTOS 5.6 runs as Dom0 the XenServer privileges domain. In clustering, a management cluster application must run to follow the resources and the mapped jobs. In the proposed model the cluster management and load control is implanted manually to run the programs tasks through the cluster processors. The first service used to setup the HPC cluster is to establish the connectivity and group the cluster CPUs as one computation farm. A *connectivity_loop.c* program which is proposed to start the cluster as following:

```
#include <mpi.h> // the message passing interface library
#include <omp.h> // the openmpi library for auto parallelization
Define Status flag of MPI message MPI_Status  s_status;
Signaling handler function( hand_sig)
Cluster Connectivity_test(p_rank)
{
While (Termination signal not received Signaling handler function() )
// Stop the connection program once a ctl+c signal is generated
{// heartbeat test
If Process_ID is Node A then {
 Node A sends hello update
 Node A sends received acknowledgment
Sleep();
}
Else {
 Node B sends hello update
 Node B sends received acknowledgment
Sleep();
}
} //end while loop
} //end function
main(int argc, char **argv)
```

```
{
   Set Maximum Processors number array PN[MPI_MAX_PROCESSOR_NAME+1];
   Initialize MPI  sockets MPI_Init(&argc, &argv);
   Set processors rank number MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);
   Set Processors number MPI_Comm_size(MPI_COMM_WORLD, &pn);
   Check the processors connectivity pthread_create(Connectivity_test());
// The job tasks is assigned to CPUs cluster farm by calling the function through CPU ID rank
// number and CPUs can communicate using  the block below
{
#pragma omp parallel
 // the automatic parallel directive trigger the code that is supposed to be parallelized
 norm_vm_mig( https://worth, root, password, worth ,unhand, Win7_Flak)
 // call the migration function using the HPC cluster
}
Message passing between processes
  MPI_Barrier(MPI_COMM_WORLD);
 // blocking status to finish the job for all CPUs members
   MPI_Finalize();
   }
```

To run the connectivity service through the cluster, the cluster nodes members are defined in *mpi_hosts* file which include the nodes IP address (Unhand and Worth). Then the compiled version of the *connectivity_loop.c* is executed under the *mpirun* command. Number of CPUs can be set through the *mpirun* command for each cluster node in the *hostfile* parameter.

mpirun -v -np 2 --hostfile mpi_hosts connectivity_loop.out

The computing paradigm through the distributed memory is based on the message passing technique, which is very tedious and complicated for programmers. Next section will use the shared memory approach for inter-processor communication, which provides an easy abstraction from work to run the application on top of the distributed memory and the program data are accessible for all processor member via the distributed shared memory computation approach.

4.2.3.3 HPC Cluster Distributed Shared Memory

The DSM model works with the HPC cluster to provide the shared memory accessibility for all cluster nodes and processors as shown in Figure 4.12. DSM changes the inter-processor communication based on shared memory communication paradigm, which allows all processors in the HPC cluster to access the same job memory space. It is easier to control the program execution in parallel computing infrastructure, other than that the memory state is shared between cluster computations nodes with an efficient mechanism used to control memory updates. In such a model the process migration only requires moving the process state from CPU scheduling ready queue on one computation node processor to the ready queue on the other node processor, since process control block PCB, code and stack are all in the same memory address space, and shared virtual memory is a single address space shared by number of processors. Any processor can access any memory location in the shared address space directly.

The DSM model used in this thesis is the software Grappa DSM. This DSM provides the shared memory abstraction for the HPC cluster. To call migration function using Grappa abstraction, the Grappa library source code must be included and initialized to be used by VM migration function as a pre-configure framework as following code sample:

```
#include <Grappa.hpp>
#include "graphlab.hpp"
using namespace Grappa;
run([=]{
    Metrics::reset_all_cores();
    Metrics::start_tracing();
    double start = walltime();
    GlobalAddress<Shared_mem> g_buffure = make_global(shared_mem);
// the DSM is running
   norm_vm_mig( https://worth, root, password, worth ,unhand, Win7_Flak)
 // calling migration function
    finish<&gce>([g_board,board]{
    });
```

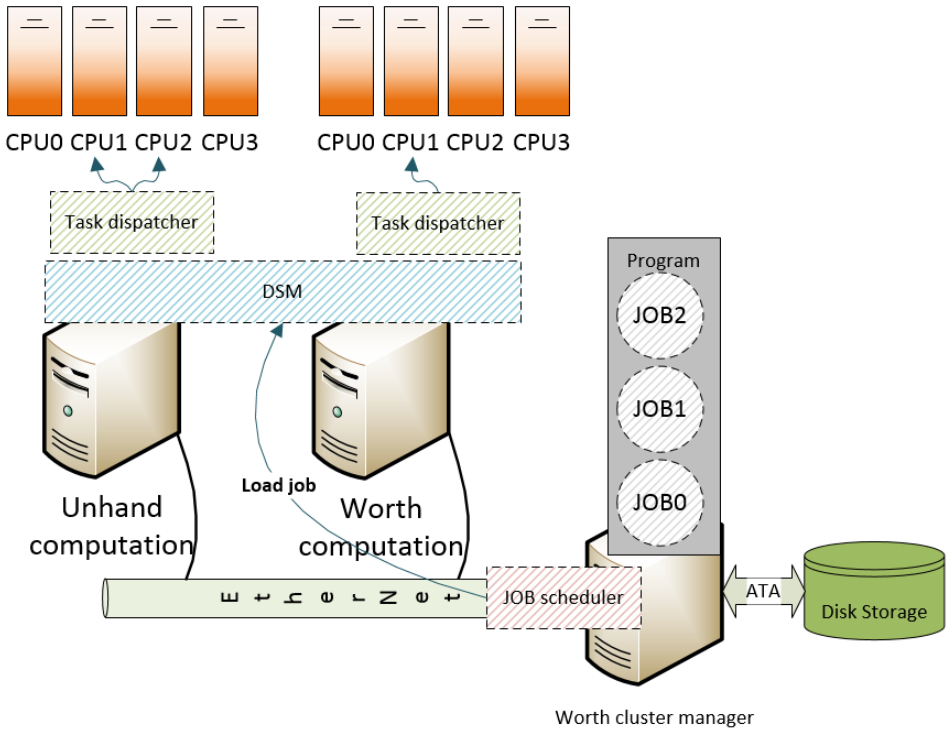


Figure 4.12: Scheduling of DSM HPC Jobs

A minor modification for the Grappa functions platform is considered in this work. In Grappa the message size communication is very small compared to the memory pages updates, which is about 4KB in the pre-copy approach. This is the first major bottleneck for moving block of memory pages. The second challenge in Grappa to load the live VM migration process is the variable locality management of the Grappa, where to achieve a lower number of communication messages the data accessed pattern with low locality is modified on its home processor rather than sending a copy of the code image to the destination core. On the other hand, it is more efficient for data accessed pattern with high locality to send the whole copy of the memory data to the requested core. In pre-copy the best is not to move the highly changed memory pages, but it's more efficient to send the low locality memory pages with lower memory dirtying pages rate to be sent first. The modification algorithm implemented in the DSM is presented in the following pseudocode.

Grappa DSM modification Pseudocode

Memory buffer = size of (dirty Page)

While (migrated memory not consistence in destination server)

Do

allocate current memory pages (buffer)

Update memory index flag table

Dirtiness = Check locality statistics (flag table)

If(Dirtiness)

Don't send {NO OPERATION}

Else

send memory(buffer)

End while

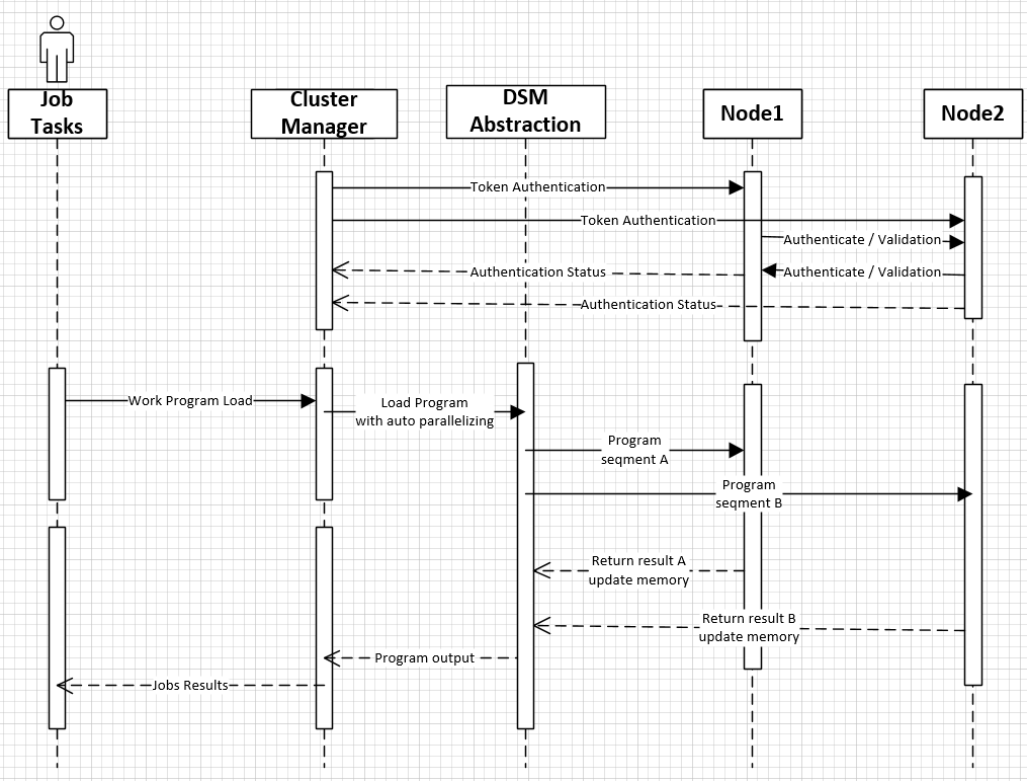


Figure 4.13: DSM Jobs Sequence Diagram

By utilizing memory coherency and consistency methods to implement Live VM migration it can achieve a very fast, stable and convenient migration process technique. Especially for memory pages updates, only one processor will execute at a time originally during migration. In this proposed approach based on DSM the hypervisor will migrate its VM on top of the DSM HPC cluster framework, which implements mapping between the local memory in source and

destination to the distributed shared virtual memory address space, with parallelizing the migration process to increase the speed up, Figure 4.13 depicts the sequence flow diagram of running application using DSM cluster.

4.2.3.4 Live VM Migration using DSM

Figure 4.14 depicts the abstract components of the whole model. In this example setup the source machine (Worth) runs two VMs (Flake and Flask); the guest OS (Win7) of Flake will be migrated. The process will start by triggering the migration function of the XenServer XAPI, which then starts the migration process to do the pre-copy live VM migration method explain in Chapter 1. DSM will provide a shared memory access to all cluster computation nodes that enable direct access to all memory pages for both source and destination virtualization servers.

The live VM migration process is localized by the source physical machine, the OPENMPI scan the serialized code to find any parallel segment to start the speed up. The DSM is used as a global memory space between the HPC cluster computation nodes that can help the pulling process migration of the VM. Meanwhile the source physical machine continues to push the virtual CPU state of the migrated VM and the Network state as a priority task, then the remaining dirty pages is moved as a lower priority task.

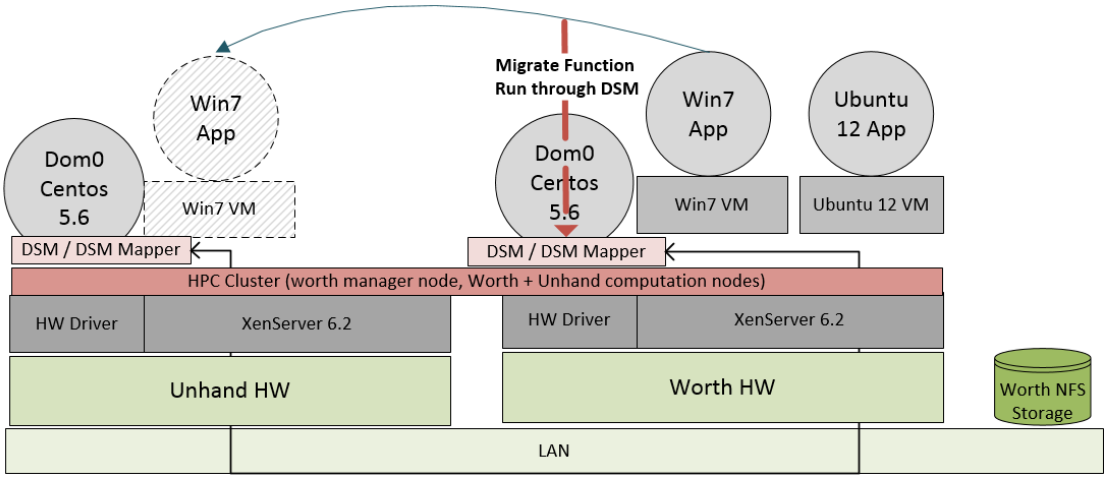


Figure 0.14: Full View Setup XenServer with Shared Storage and DSM HPC Cluster

The source and destination virtualization servers will be loaded with the migration task to start and terminate the live VM migration based on the destination VM memory image consistency threshold, which is bounded by three parameters: maximum number of iterations, less memory

pages changes (less dirty pages generation), and consistent memory image at destination. Figure 4.15 describes the pre-copy with DSM approach as time-space diagram.

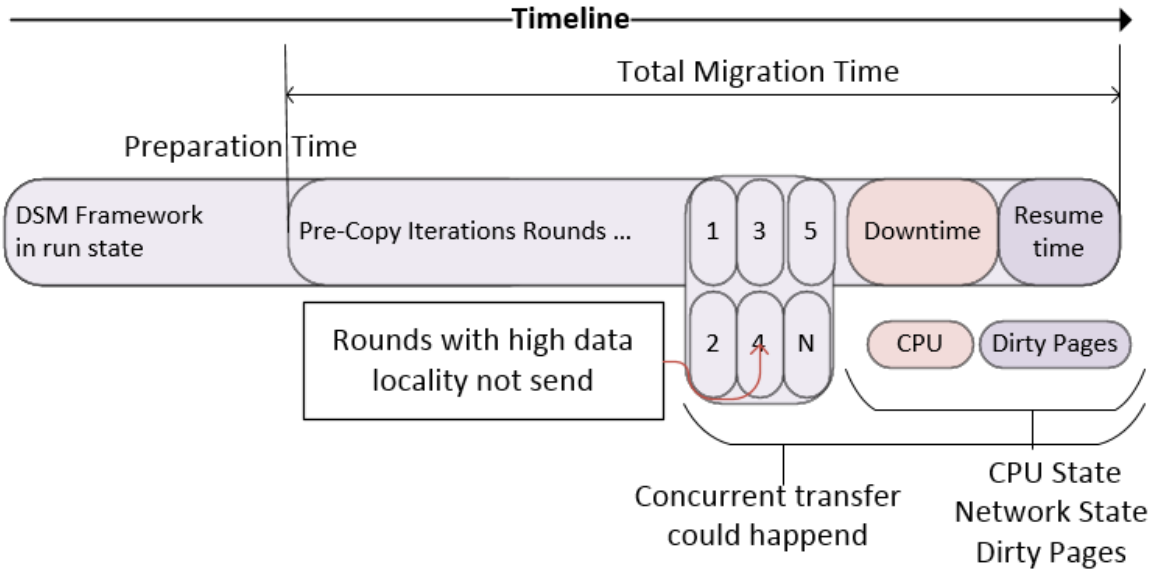


Figure 4.15: DSM Pre-Copy Time Progress

4.3 Summary

The proposed model is discussed in this Chapter that uses the DSM HPC cluster to run the live VM migration process in cloud infrastructure datacenters. The system architecture for DSM HPC cluster contains three main services: the shared storage service, DSM HPC cluster framework, and the virtualization environment. The three services are integrated together by running the hypervisor migration function on top of the DSM HPC cluster using the shared storage between the source and destination.

Chapter 5

Performance Measurements

The live VM migration is done for both guest OSs, the Windows7 VM Flake and the Ubuntu12 Linux Flask, between the two host servers Unhand and Worth. The experiment is run for each different case for six times to make sure the measured values do not contain any special errors, and then an average is calculated. The tools used in the measurement are described in Section 5.1 and the results are discussed in Section 5.2.

5.1 Measurement Tools

The tools that are used to measure the four migration metrics vary from the Linux shell script to dedicated XenServer utilities and resources monitoring, in addition to network utilities and traffic monitor tools.

5.1.1 RR2CSV XenServer Tool

The Round Robbin to Comma Separated Value (*RR2CSV*) tool is used to output XenCloudPlatform and XenServer RRd values in CSV format. It runs inside Dom0, and by default displays all active data sources that RRd maintains, but it is also possible to select data sources of interest as command-line arguments. Data Sources (or DS) are RRD concepts. They are objects that have measurements associated with them. They include a metric, a source from which the metric originate, and a consolidation function. For instance, if there are multiple VMs, it will have as many "*memory_target*" data sources as VMs number. Table 5.1 shows the virtualization objects information header and its sample value, which are gathered in sequence timestamp that is populated in fixed time intervals configured by the command parameters.

index	Table Header Name	Sample Value
1.	Timestamp	2015-10-05T02:28:45Z (time)
2.	AVERAGE:vm:Ubuntu12_flask:memory_target	4.29E+09 (Byte)
3.	AVERAGE:vm:Ubuntu12_flask:memory_internal_free	141912 (Byte)
4.	AVERAGE:vm:Ubuntu12_ flask:vif_0_rx	38.9698 (bits)

5.	AVERAGE:vm:Ubuntu12_ flask:vif_0_tx	17.3599 (bits)
6.	AVERAGE:vm:Ubuntu12_ flask:memory	4.29E+09 (Byte)
7.	AVERAGE:vm:Ubuntu12_ flask:vbd_hda_read	371.895 (Byte)
8.	AVERAGE:vm:Ubuntu12_ flask:vbd_hda_write	1146.87 (Byte)
9.	AVERAGE:vm:Ubuntu12_ flask:cpu0	0.0072 (Ratio)
10.	AVERAGE:vm:Win7_flake:memory_internal_free	2965256 (Byte)
11.	AVERAGE:vm:Win7_flake:memory_target	4.29E+09 (Byte)
12.	AVERAGE:vm:Win7_flake:vbd_hda_read	9253.352 (Byte)
13.	AVERAGE:vm:Win7_flake:vbd_hda_write	14694.33 (Byte)
14.	AVERAGE:vm:Win7_flake:vif_0_rx	38.9698 (bits)
15.	AVERAGE:vm:Win7_flake:vif_0_tx	1122.651(bits)
16.	AVERAGE:vm:Win7_flake:memory	4.29E+09 (Byte)
17.	AVERAGE:vm:Win7_flake:cpu0	0.0619 (Ratio)
18.	AVERAGE:vm:xen6.2:vif_0_rx	198.4373 (bits)
19.	AVERAGE:vm:xen6.2:vif_0_tx	623.9515 (bits)

Table 0.1: RR2CSV Table Headers

The RR2CSV tool is triggered before the migration process is started to allow gather initial values before migration and the result is redirected to be saved into a comma separated value file. The values are sampled and saved every 1 second (*rrd2csv -n -s 1 > log_info.csv).* This tool is the main tool used to get the data about the virtualization objects of the XenServer, as an authentication way compared with other measurement tools as in Sections 5.1.2 and 5.1.3.

5.1.2 IPERF Network Traffic Measurement

The *IPERF* is a tool for active measurements of the maximum reachable bandwidth on IP networks. It supports change of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). For each test it reports the bandwidth, and other network parameters. *IPERF* Linux command (*iperf -c localhost -u -p 5001 -t 600 -b 1M*) is used to measure the host server network traffic and these values are compared with Xen RR2CSV values which are found to be similar. In addition to the *IPERF* tool the ping tool is used to measure the downtime based on the network connectivity. The *ping* is configured to measure the response time in milli-second (*ping flake -t -l 1 -w 1*) with one response wait.

5.1.3 Linux Shell Script

The Linux time command to find the time execution of a script or program is used to obtain the migration execution time in order to complete the total migration time. A comparison with the timestamp of the RR2CSV results is done to check the results logic. A Linux script named (*monitor.sh*) is used to obtain the physical host servers resource measurements for the memory utilization, CPU utilization and the network traffic throughput. The results of the Linux script are also compared with the RR2CSV.

```
#!/bin/bash
i=1
 while [ $i -le 500 ]
do
now=$(date +"%N")
echo "Current time : $now"
echo " Current time : $now ----------">> network.txt
netstat -I >> network.txt
echo "------------------------">> network.txt
echo " Current time : $now ----------">> mem.txt
vmstat -s >> mem.txt
echo "------------------------">> mem.txt
echo " Current time : $now ----------">> pro.txt
mpstat -P ALL >> pro.txt
echo "------------------------">> pro.txt
echo " Current time : $now ----------">> hd.txt
df -a >> hd.txt
echo "------------------------">> hd.txt
i=$(( $i + 1 ))
sleep 60
done
```

5.2 VM Workload Benchmarks

The guest VM OS is loaded with four different workloads to test the live VM migration under different case scenarios, which load the migrated VM by applications that have variant execution behaviors as benchmark, as given in Table 5.2.

Workload	Benchmark
Idle OS	Run guest OS with idle state (for both OS types)
CPU intensive task	For Linux Compiling XEN source code For Windows Installing Cygwin Installation
Memory Intensive task	Playing video (for both OS types)
Network Intensive Task	Web server (for both OS types)

Table 5.2: Work Load Benchmark

5.2.1 OS Idle Workload

In idle OS the memory dirty pages generated have lowest time of generation during the VM migration without running any heavy workload. The OS load generation of memory changes in idle case is used to evaluate the DSM live VM migration model.

5.2.2 CPU Intensive Workload

Compiling source code is a heavy CPU load and in this case of workload the CPU context of compiling time is higher than other CPUs jobs. For the CPU intensive task a Xen hypervisor source code compilation is run during the live VM migration of the Linux guest OS, to test the migration during high CPU load. For windows an application installation workload is used as equivalent for code compilation CPU intensive usage. The Cygwin Linux emulator tool for windows is installed during the VM migration to load the CPU.

5.2.3 Memory Intensive Workload

The memory intensive task workload is achieved by running video during the live migration for both types of OSs, Windows and Linux VM, in addition to the disk intensive read load for the video from the hard disk. The video file used in the migration experiment is *MP4* format with size 700MB. The video is played in a frame rate of 24 frames per second with each frame resolution of

1280*546. The player used in the experiment is the VLC player, which works in both Windows and Linux environment. The playing video bit rate in windows VM is 757Kb/s and in Linux the bit rate speed is 737Kb/s. The VGA graphic card adapter is the Xen virtual Graphic Processing Unit (vGPU), the card driver for windows is the generic driver and for Ubuntu the graphic adapter is the basic driver.

5.2.4 Network Intensive Workload

The network workload is done by deploying a webserver in both guest VMs. For windows the IIS web server is installed and used. For Linux an apache web server is installed and used. The same home page is used to test the web server in both VMs where the web page contains some text and image with a reasonable web page size of a usual web page of about 16 MB.

To load the web server with web traffic a stress test web application is used that can emulate a web server client to request the web server's pages. 4000 users had been emulated with one http get function to load the web server. Figure 5.1 shows the web tester configuration and number of users. The request time delay between the user click is 20 seconds but the user has only one click which is the request click. The average click time of the URL is 63ms in the web concurrent web user's requests.

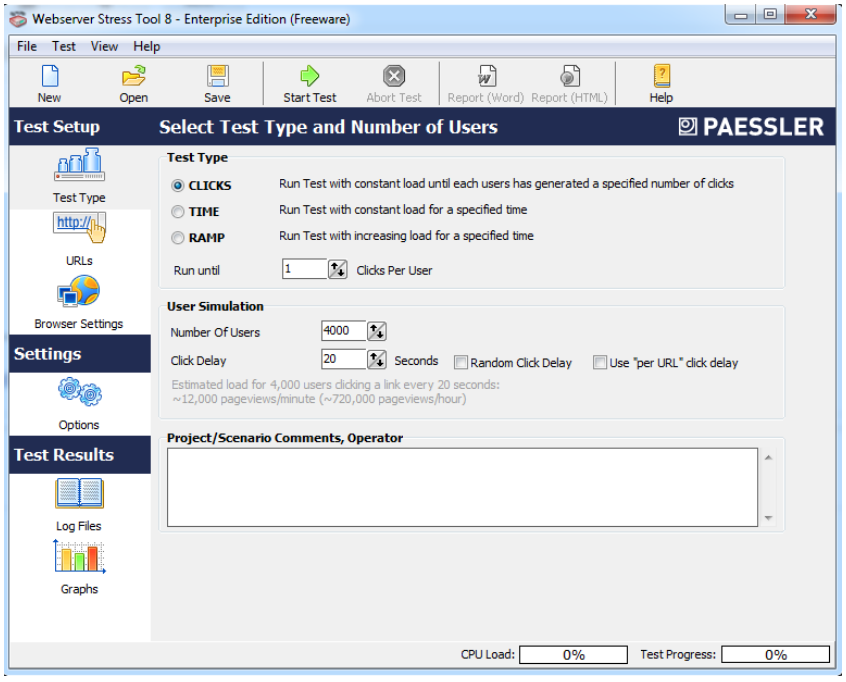


Figure 5.1: Web Stress Test Tool

5.3 Results and Discussion

The first measurement for the live VM migration is to find the total migration time for all cases of VM workloads. Figure 5.2 shows Windows total migration time with and without using the DSM. The total migration time is reduced in all cases but the best is with Cygwin installation and idle workloads, which is about 25% and 20% of enhancement respectively, whereas with web work load the enhancement is 6%. Figure 5.3 depicts the Linux VM total migration time. Again with DSM it is less with all workloads and the best ratio is for idle VM workload with 16.6% of enhancement reducing the total migration time.

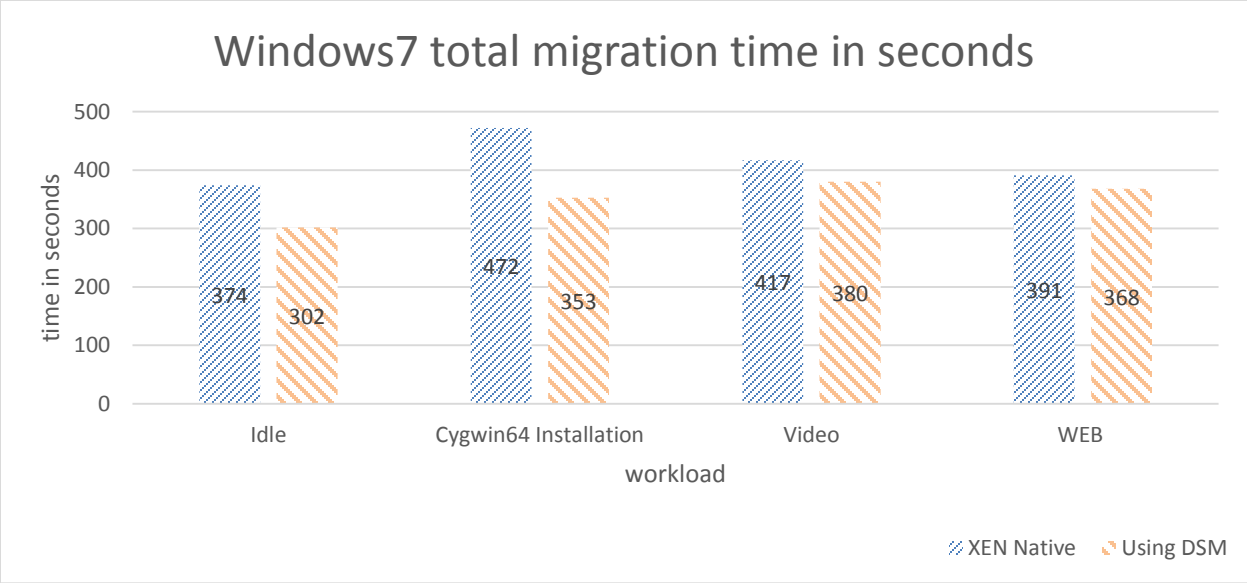


Figure 5.2: Windows Total Migration Time

The DSM HPC cluster framework provides the speed up for live VM migration by parallelizing the migration process using the source physical host server cores, with memory accessibility directly from the destination physical host server receive process. The DSM locality check during memory pages change tracking reduces sending the memory high dirty pages in the iterative phase of the pre-copy. The stop condition of the iterative phase is reached sooner than the normal XenServer pre-copy approach, which reduces the total migration time and the amount of data sent during the migration process.

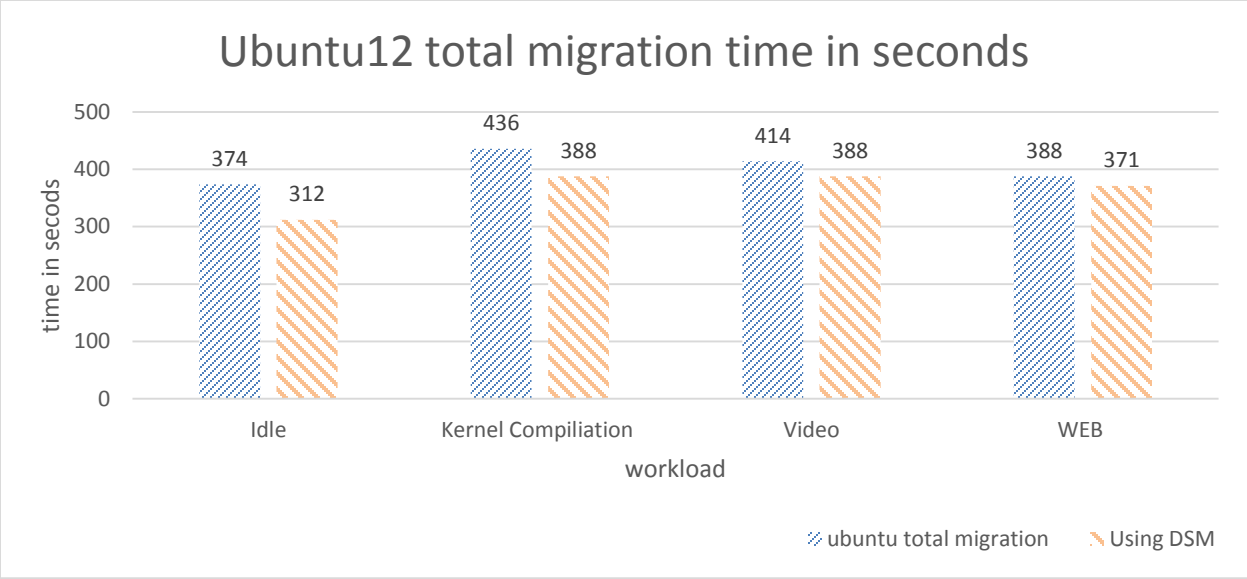


Figure 0.3: Linux Total Migration Time

In Figures 5.4 and 5.5, the downtime measured for Windows and Linux with DSM has a good reduction in all VM workload cases. It achieves about 42.8% of downtime reduction in the Web work load of windows VM and 42.8% in Linux VM. In windows video playing there is no enhancement because GPU is working with the memory load and video images marshalling. But with Linux it achieves 10% of enhancement knowing that with Linux the video does not stream in a smooth way, because of the basic graphic driver work behavior. However with windows the video marshalling is better and efficient than Linux.

In XenServer migration, after the iterative phase finishes and the stop and copy phase is started to move the CPU state, network state and the remaining dirty pages to the destination host server, the network maximum transfer bandwidth is increased to reduce the time of moving CPU state and network state, the downtime period. With DSM migration the downtime is enhanced by setting the CPU state as a global accessible value for the destination server with parallelization of sending function in push phase and pull phase. In general, the parallelization in this model is not guaranteed because the migration function is a sequential or serial in execution by default, but by using the automatic parallelization method the OPENMIP library will look ahead the execution and figure out the parallel binary segments and run these independent segments in parallel with the main program flow.

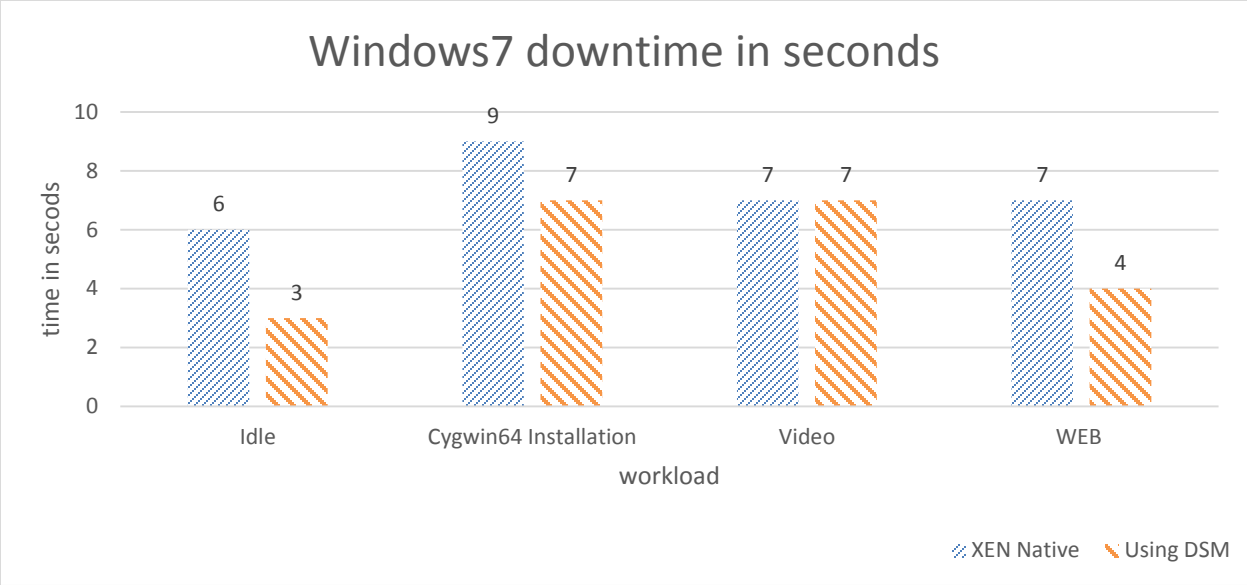


Figure 5.4: Windows Downtime

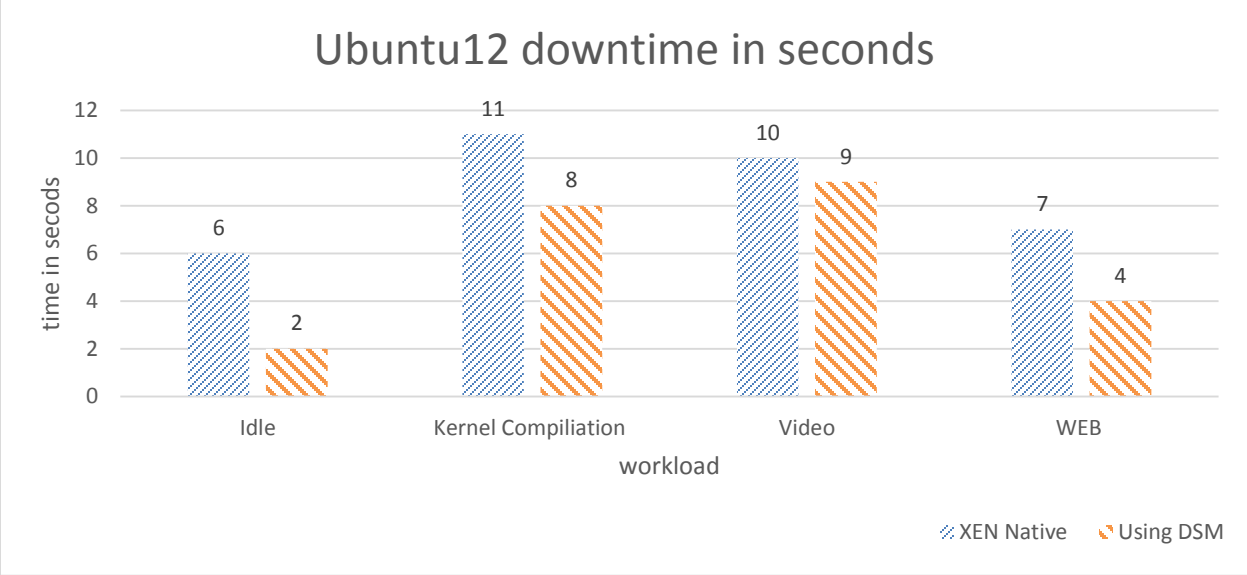


Figure 5.5: Linux Downtime

Figures 5.6 to 5.13 depict the behavior of the network traffic transferred during the VM migration and the VM CPU performance. In general the sample is generalized as a timestamp sequence, because the migration had been done during different time periods. The traffic figures show that the network traffic is limited to a fixed maximum data rate because the XenServer network configuration is bounded during the iteration pre-copy phase to protect the network bandwidth. This limitation of maximum traffic is removed in the stop and copy phase of pre-copy method to

reduce the downtime as much as possible. This is shown as a peak in the traffic figures during the last round of iteration phase, which is with starting the stop and copy phase.

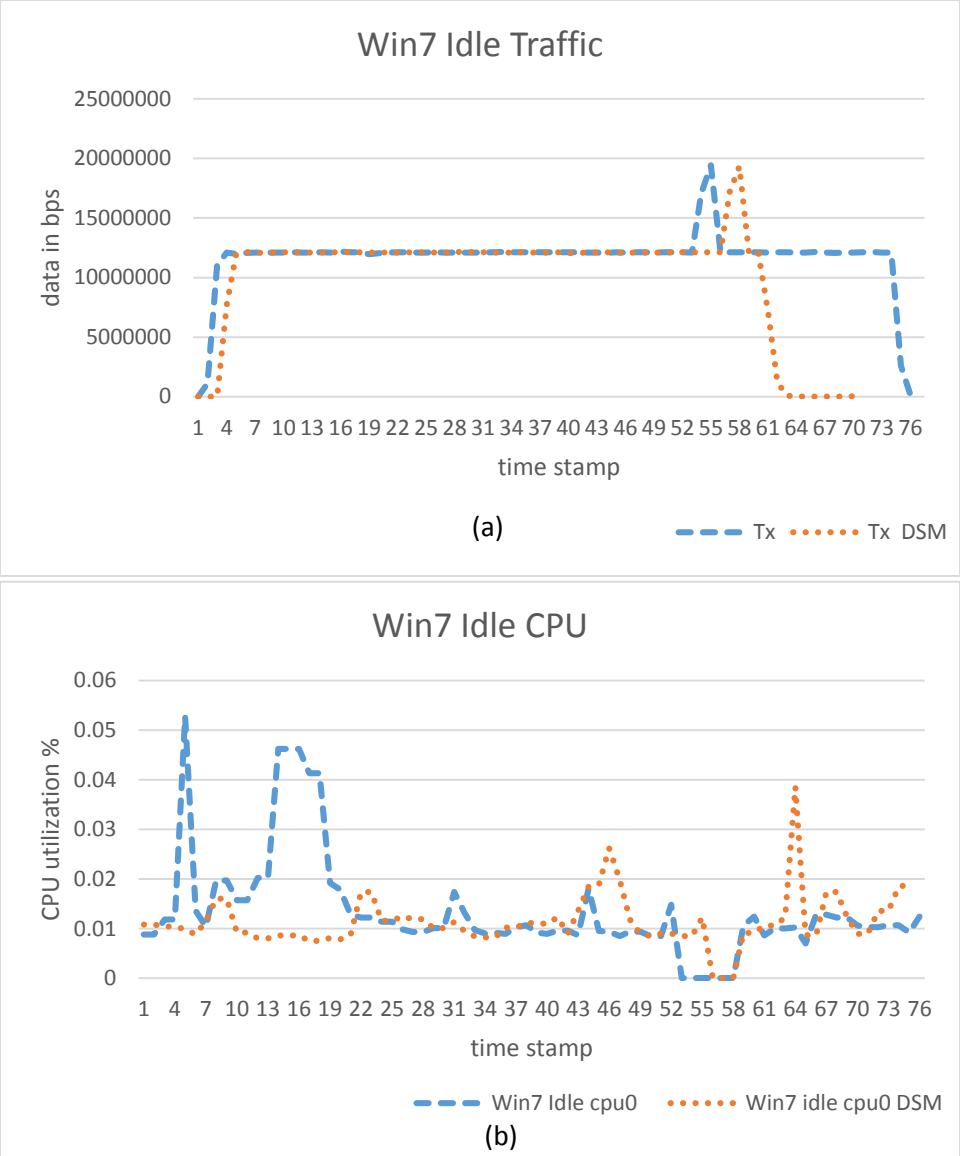


Figure 5.6: Windows Idle Workload (a) Traffic Size (b) CPU Performance

Figure 5.6 (a) shows the traffic rate sent during the migration of windows VM with Idle work load. The CPU utilization in Figure 5.6 (b) depicts the performance of the CPU during the migration with same behavior of normal XenServer migration. The down time period is very clear in the time stamp when the CPU utilization is 0%.

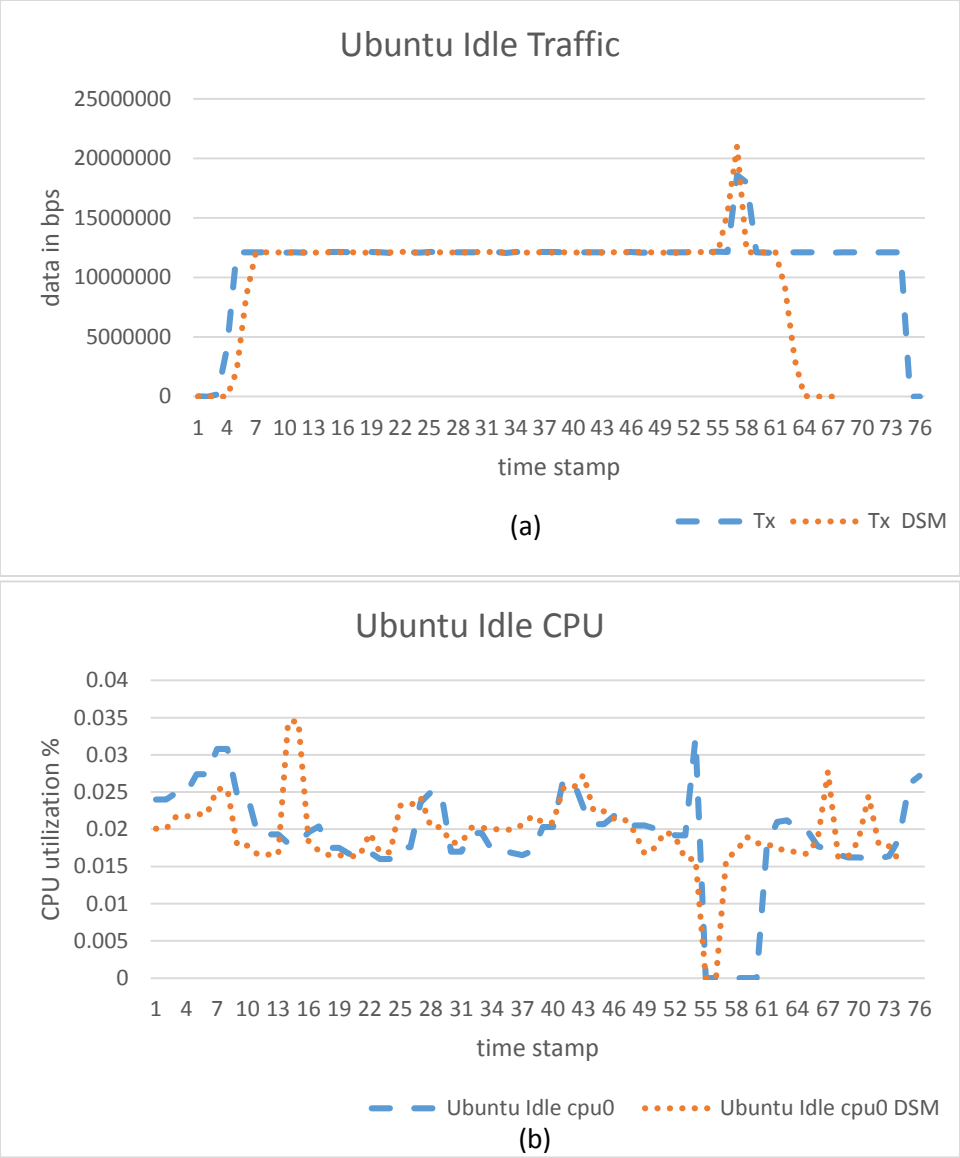


Figure 0.7: Linux Idle Workload (a) Traffic Size (b) CPU Performance

Figure 5.7 (a) and (b) shows the idle load of Ubuntu12 Linux OS live migration of the VM. In (a) the network traffic rate is shown for both live VM migration models, the normal Xen migration and with the DSM cluster model.

The size of data transferred in both VMs idle workload is reduced, because the migration is finished earlier as shown in Figures 5.6(a) and 5.7(a), because the condition of stopping the iteration phase is achieved earlier than the Xen pre-copy method without DSM. CPU and network states in our DSM model are also transferred in a lower time due to parallelization and shared space abstraction, as shown in Figures 5.6(b) and 5.7(b).

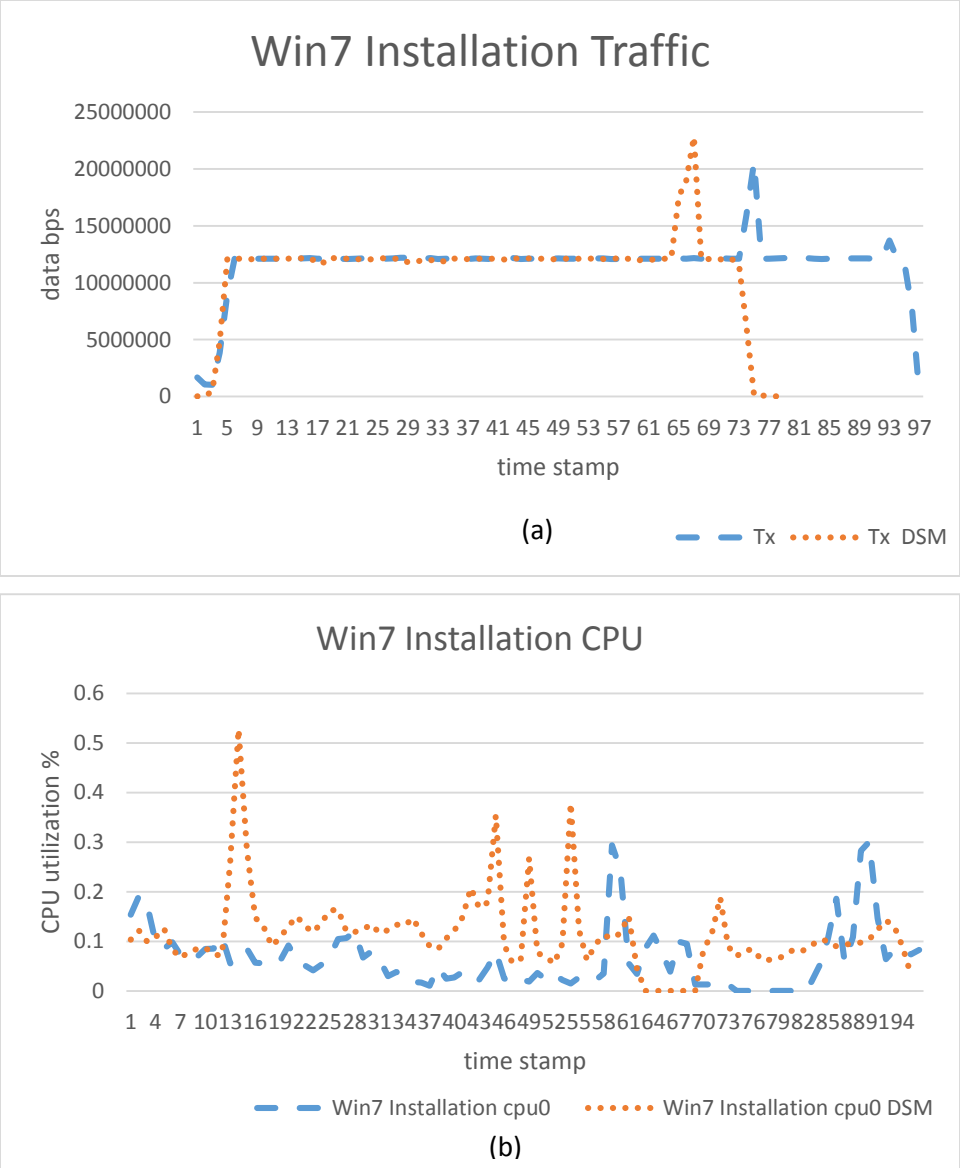
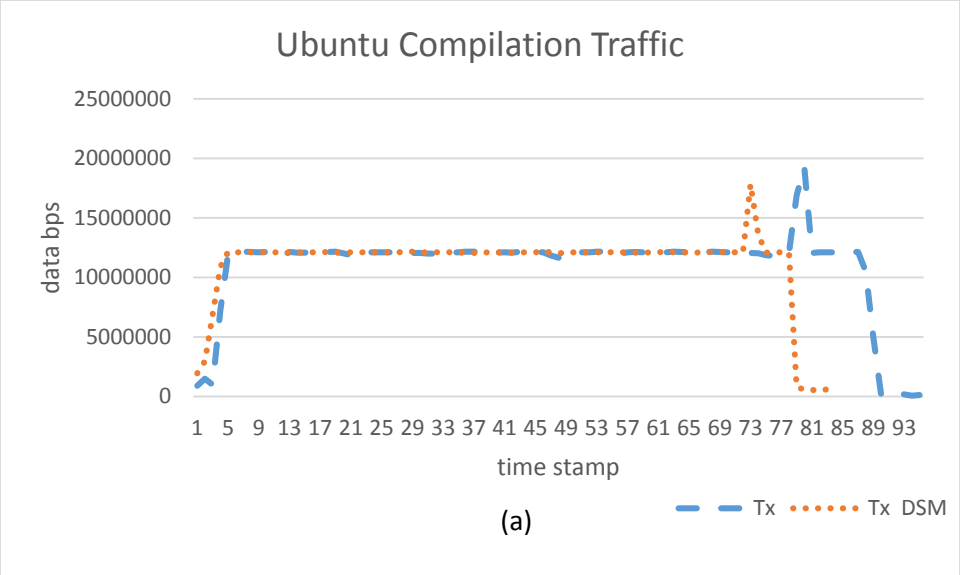


Figure 0.8: Windows Installation Workload (a) Traffic Size (b) CPU Performance

In Figure 5.8, the live VM migration for windows is shown. An application installation is running that produces higher CPU state than the idle OS workload. With DSM model it takes less time to finish the CPU state and network than normal work load, and a less down time is needed with the new proposed model. The pulse in the traffic bandwidth indicates the starting of moving of the CPU state but the traffic control is bounded back again after the stop and copy started directly.



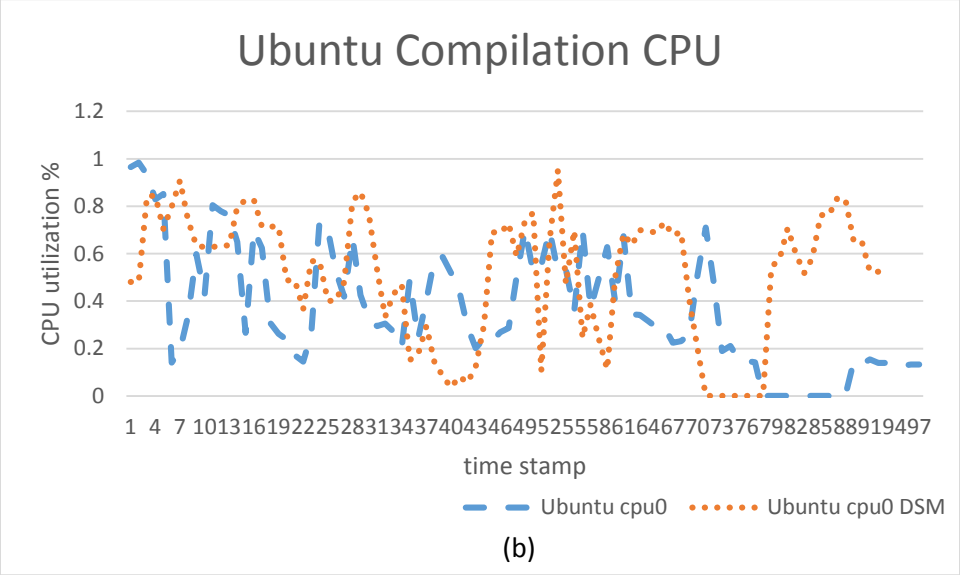
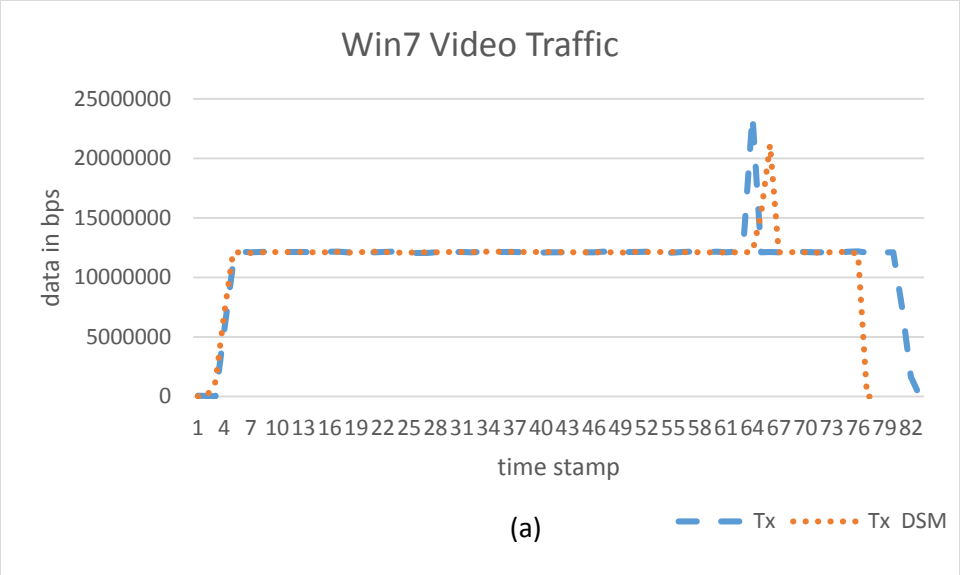


Figure 5.9: Linux Xen Compilation Workload (a) Traffic Size (b) CPU Performance

For Xen compilation workload the CPU state is higher than the idle CPU workload. Figure 5.9 explains the migration traffic (a) and the CPU performance in (b).

In Windows and Linux CPU intensive workload, number of memory pages transferred is reduced because the iterative phase and the stop and copy phase finish earlier in our DSM model, which reduces the amount of total data transferred.



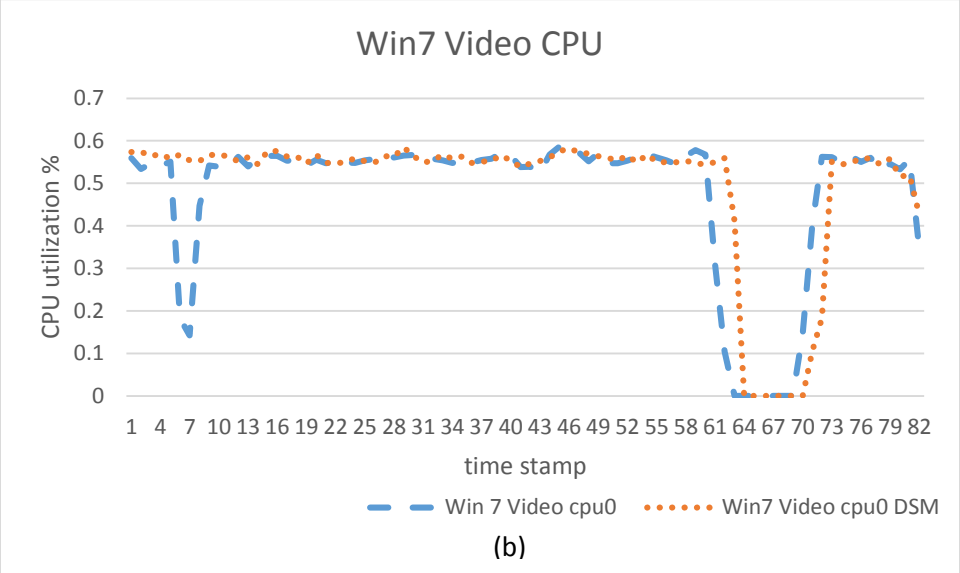
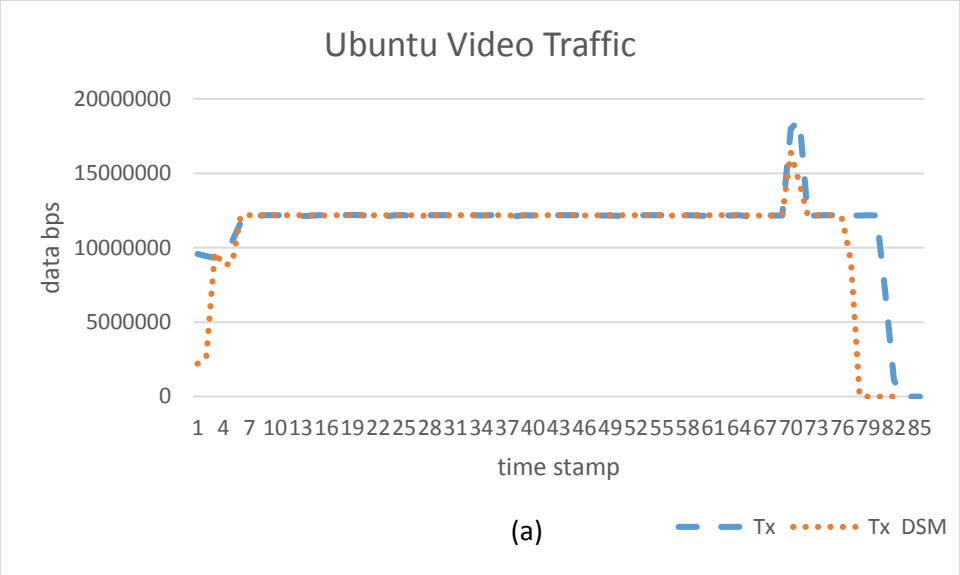


Figure 5.10: Windows Video Workload (a) Traffic Size (b) CPU Performance

Video workload migration test is depicted in Figure 5.10. The video is loaded from the hard disk in data chunks to be played in frames sequence using the VM virtual graphic card that uses the GPU processor that off loads the CPU to do the job.



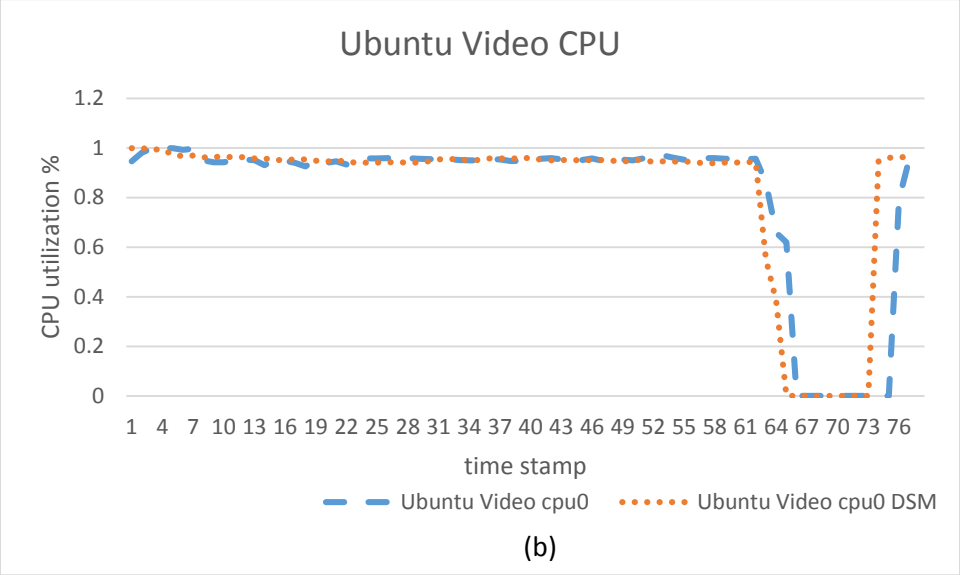
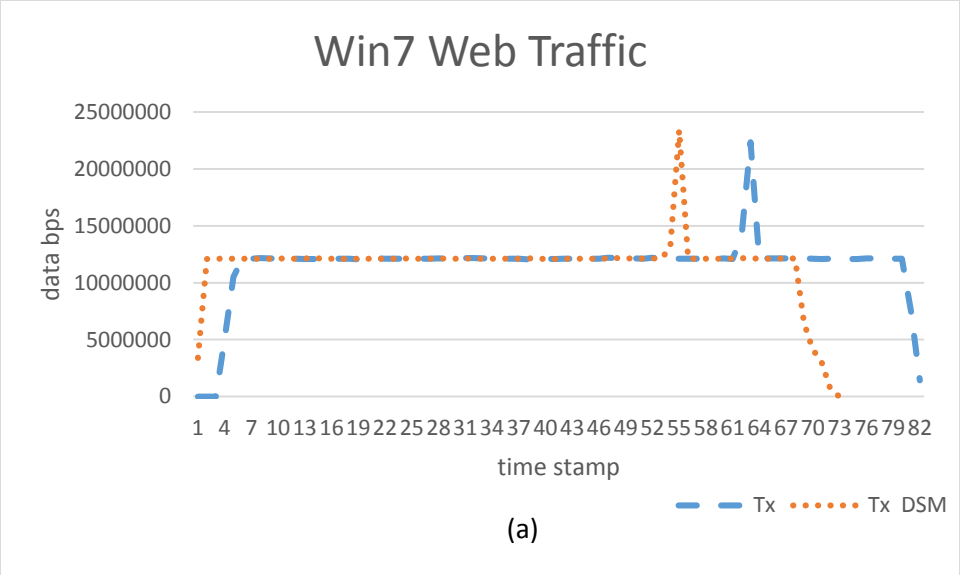


Figure 5.11: Linux Video Workload (a) Traffic Size (b) CPU Performance

For the Linux and due to the driver limitation of the virtual graphic card that increases the load to the CPU to load and play the video. Figure 5.11 (b) shows that the CPU utilization for Linux is higher than in case of Windows (Figure 5.12 (b)), which means in the Linux OS the video is played with high dependency on CPU. The video workload increases the dirtiness pages production with low locality aspect that causes a minor reduction in total migration time, and due to the virtual GPU which produces a higher CPU data processing causing not to reduce further the amount of data sent.



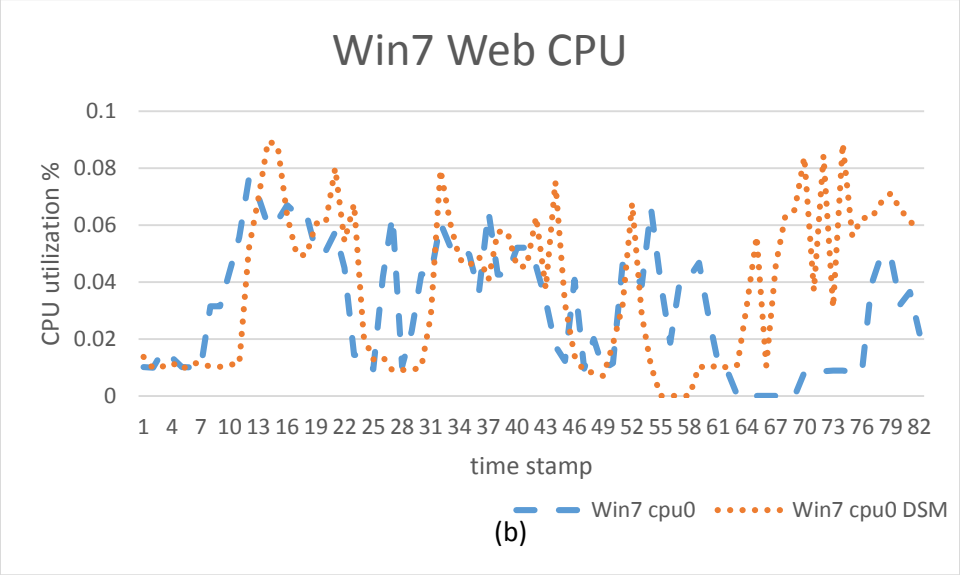
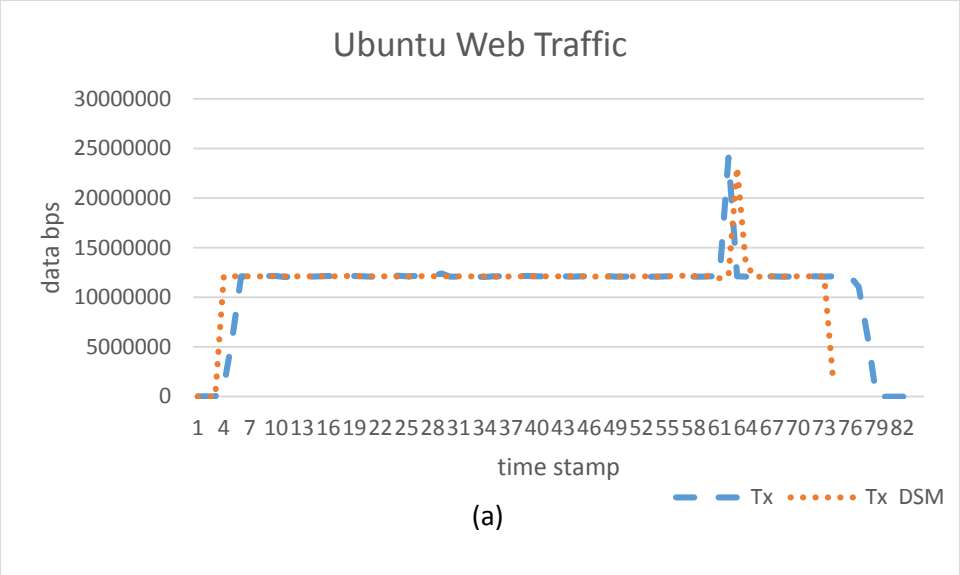


Figure 5.12: Windows Web Workload (a) Traffic Size (b) CPU Performance

For network intensive task Figure 5.12 (a) describes the traffic bandwidth of moving web server VM. The network state of each client connection to the web server must be maintained until the client terminates the connection. Figure 5.12 (b) shows the CPU utilization of Windows VM.



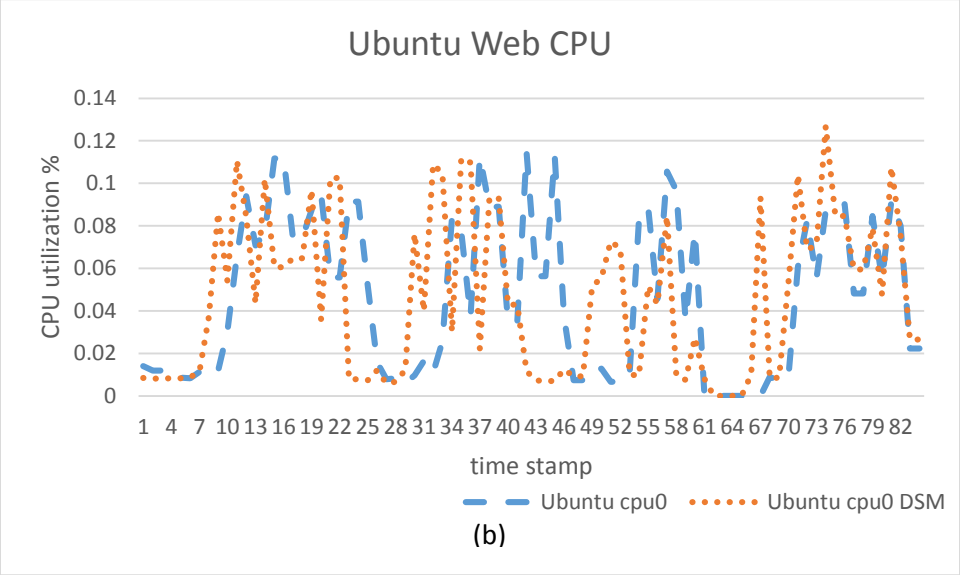


Figure 0.13: Linux Web Workload (a) Traffic Size (b) CPU Performance

For apache web server in the Linux VM the network state is transferred during the stop and copy phase at timestamp 63 as Figure 5.13 (a) depicts. The CPU utilization is shown in Figure 5.13 (b).

The data transferred in web workload mainly has a big network state to save the web server clients connection which increases memory foot print with higher locality attribute. Due to the increased higher locality data the amount of data transferred is reduced.

5.4 Summary

In this Chapter the results of live VM migration with using DSM HPC cluster is compared to the XenServer normal live VM migration process. The experiment is run for each kind of VM with the four types of work load tested six times and the average value used. The results show the thesis proposed model achieves a 50% reduction in downtime in idle work load for windows VM and 66% reduction in downtime of the Linux VM. In general, enhancement is provided both to the downtime and total migration time for other workloads. The proposed model also supports a reduction in the amount of data transferred with no effect on the CPU performance, which is also an indication of the VM application performance. Our DSM model works well with workloads that have higher temporal locality attribute, such as web workload and CPU intensive workload. On the other hand, workloads such as video streaming that have spatial locality have a lower improved performance.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The live VM migration is an important issue in cloud services continuity plan and service level agreement commitment. In datacenters cloud infrastructure responds to resources requests demands dynamically, which mean virtualization resources (Virtual Machines running on Hypervisors) must be moved between the datacenter physical servers. The live VM migration process is used to achieve cloud resources provision agility, which depends on four metrics: total migration time, down time, amount of sending data and the VM application performance.

In general downtime in live migration process is considered the most critical metric in the migration process because the service is down during that time period. In the pre-copy method, the downtime is minimized by importing the updated memory state as much as possible during the iterative phase, which makes the downtime depend on moving the CPU state and network state during the stop and copy phase.

In this thesis a novel method is proposed to run the live VM migration in the cloud data centers using a distributed shared memory high performance computing cluster. In this way, memory of each computation nodes shared and accessible to all nodes CPUs with high abstraction for nodes local memory. In the DSM model the same pre-copy method is used, so the iteration phase provides the same memory stable state with minimum transferring of dirty pages, because of the proposed DSM locality attributes, which does not send any highly changed memory page. Furthermore, the speed up is achieved by using automatic parallelization of applicable and memory accessibility mapping between source machine and destination machine in sending memory pages or CPU and network state. The proposed model is built and integrated with virtualization architecture using the share storage. DSM HPC structure setup the live VM migration by using the DSM memory access abstract which allows an inter-process communication directly.

Eventually, in this thesis, the four live VM migration metrics are reduced, the down time reduced by a factor of 66.6% and 50% for both Windows and Linux idle work load, respectively. For CPU

intensive workload a 22.2% in windows and 27.3% in Linux down time reduction is achieved. In general a clear reduction in total migration of the live VM migration is achieved, without any degradation in the CPU performance for all migration cases. The amount of data transferred is reduced in all experiment case scenarios.

6.2 Future Work

For future work, a wide task is to integrate the DSM computation model as a part of the virtualization architecture not as a standalone solution. The cloud management platform needs more reliable infrastructure construction by implementing the proposed model with the virtualization architecture to achieve a fast and realistic migration of VMs between physical servers in datacenters. Cloud management layer control the virtualization infrastructure layer, cloud management decide which VM will be migrated, when the VM will be migrated and where the VM will be migrated. Cloud manager need to have a real time statistic about the physical servers and VMs state to decide the optimal action within the cloud infrastructure. A cross layer integration for the DSM cluster with cloud manager will be implemented in future work.

References

[1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” Version 15, 10-7-09. National Institute of Standards and Technology, Information Technology Laboratory, September 2011, pp. 1-7.

[2] M. A. Vouk, "Cloud computing — issues, research and implementations," in Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on, 2008, pp. 31-40.

[3] B. Furht, A. Escalante, “Cloud Computing Fundamentals, “A Chapter in a book titled Handbook of Cloud Computing , Springer New York Dordrecht Heidelberg London, e-ISBN 978-1-4419-6524-0, 2010, pp. 3-20.

[4] N. Mirzaei, “Cloud Computing”. [Online]. Available: http://grids.ucs.indiana.edu/ptliupages/publications/ReportNarimanMirzaeiJan09.pdf

[5] M. Pearce, S. Zeadally and R. Hunt, "Virtualization: issues, security threats, and solutions," ACM Computing Surveys, vol. 45, pp. 17 (39 pp.), 02, 2013.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, “Xen and the art of virtualization,” in Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York, U. S. A., Oct. 2003, pp. 164–177.

[7] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. Shin, “Performance evaluation of virtualization technologies for server consolidation,” HP Laboratories, Tech. Rep., 2007, pp. 1-14.

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux virtual machine monitor,” in Proceedings of the Linux Symposium, vol. 1, 2007, pp. 225–230.

[9] J. Watson, “Virtualbox: bits and bytes masquerading as machines,” Linux Journal, vol. 2008, no. 166, p. 1, 2008.

[10] D. Bartholomew, “Qemu: a multihost, multitarget emulator,” Linux Journal, vol. 2006, no. 145, p. 3, 2006.

[11] D. Leinenbach and T. Santen, “Verifying the Microsoft Hyper-V Hypervisor with VCC,” FM 2009: Formal Methods, 2009, pp. 806–809.

[12] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, “Quantifying the performance isolation properties of virtualization systems,” in Proceedings of the 2007 workshop on Experimental computer science, ser. ExpCS ’07. New York, NY, USA: ACM, 2007, pp. 1–9.

[13] I. Parallels, “An introduction to OS virtualization and parallel virtuozzo containers,” Parallels, Inc, Tech.Rep., 2010. [Online]. Available: http://www.parallels.com/r/pdf/wp/pvc/Parallels Virtuozzo Containers WP an introduction to os EN.pdf

[14] Oracle, “Performance evaluation of oracle vm server virtualization software,” Oracle, Whitepaper, 2008. [Online]. Available: http: //www.oracle.com/us/technologies/virtualization/oraclevm/026997.pdf

[15] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield, "Live migration of virtual machines," Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, pp. 273-286, 2005.

[16] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman and V. Inc, "ReTrace: Collecting execution trace with virtual machine deterministic replay," Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, 2007.

[17] M. R. Hines, U. Deshpande and K. Gopalan, "Post-copy live migration of virtual machines," Operating Systems Review, vol. 43, pp.14-26, 2009.

[18] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), pp.1-5, 2012.

[19] H. Jin, L. Deng, S. Wu, X. Shi and X. Pan, "Live virtual machine migration with adaptive, memory compression," IEEE International Conference on Cluster Computing and Workshops (CLUSTER), pp. 1-10, 2009.

[20] H. Jin, L. Deng, S. Wu, X. Shi, H. Chen and X. Pan, "MECOM: Live migration of virtual machines by adaptively compressing memory pages," Future Generation Comput. Syst., vol. 38, pp. 23-35, 2014.

[21] A. Rai, R. Ramjee, A. Anand, V. N. Padmanabhan and G. Varghese. “MiG: Efficient migration of desktop VMs using semantic compression,” USENIX Annual Technical Conference, pp. 25-36, 2013.

[22] S. Hacking, "Improving the live migration process of large enterprise applications,” 3rd International Workshop on Virtualization Technologies in Distributed Computing, VTDC'09, 2009, pp. 51-58.

[23] A. Amani and K. Zamanifar, "Improving the time of live migration virtual machine by optimized algorithm scheduler credit," 4th International Conference on Computer and Knowledge Engineering (ICCKE), pp. 346-51,2014.

[24] H. Jin, W. Gao, S. Wu, X. Shi, X. Wu and F. Zhou, "Optimizing the live migration of virtual machine by CPU scheduling," Journal of Network and Computer Applications, vol. 34, pp. 1088-96, 2011.

[25] Yong Cui, Yusong Lin, Yi Guo, Runzhi Li,Zongmin Wang, “Optimizing live migration of virtual machines with context based prediction algorithm,” International Workshop on Cloud Computing and Information Security (CCIS 2013) 2013.

[26] Y. Ma, H. Wang, J. Dong, Y. Li and S. Cheng, "ME2: Efficient live migration of virtual machine with memory exploration and encoding," IEEE International Conference on Cluster Computing, pp. 610-13, 2012.

[27] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using pre-paging and dynamic self-ballooning," ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE'09, pp. 51-60, 2009.

[28] F. Ma, F. Liu and Z. Liu. “Live virtual machine migration based on improved pre-copy approach”. Software Engineering and Service Sciences (ICSESS) 2010, pp. 230-233.

[29] F. Yin, W. Liu and J. Song, "Live virtual machine migration with optimized three-stage memory copy," 8th FTRA International Conference on Future Information Technology, FutureTech 2013.

[30] E. P. Zaw and N. L. Thein. “Improved live VM migration using LRU and splay tree algorithm,” International Journal of Computer Science and Telecommunications (3), pp. 1-7, 2012.

[31] B. Hu, Z. Lei, Y. Lei, D. Xu and J. Li, "A time-series based precopy approach for live migration of virtual machines," IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS 2011), pp. 947-52, 2011.

[32] K. Z. Ibrahim, S. Hofmeyr, C. Iancu and E. Roman, "Optimized pre-copy live migration for memory intensive applications, " International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011), 2011.

[33] A. Shribman and B. Hudzia, "Pre-copy and post-copy VM live migration for memory intensive applications," BDMC, CGWS, HeteroPar, HiBB, OMHI, Paraphrase, PROPER, Resilience, UCHPC, VHPC, pp. 539-47, 2013.

[34] C. Sagana, M. Geetha and R. C. Suganthe, "Performance enhancement in live migration for cloud computing environments," International Conference on Information Communication and Embedded Systems (ICICES 2013, pp. 361-6), 2013.

[35] C. Lin, Y. Huang and Z. Jian. “A two-phase iterative pre-copy strategy for live migration of virtual machines,” 8th International Conference On Computing Technology and Information Management (ICCM), 2012.

[36] Changyeon Jo, E. Gustafsson, Jeongseok Son and B. Egger. “Efficient live migration of virtual machines using shared storage,” ACM SIGPLAN.(USA), pp. 41-51, 2013.

[37] G. Piao, Y. Oh, B. Sung and C. Park. “Efficient pre-copy live migration with memory compaction and adaptive VM downtime control,” IEEE Fourth International Conference On Big Data and Cloud Computing (BdCloud), pp. 85-90, 2014.

[38] C. Jo and B. Egger, "Optimizing live migration for virtual desktop clouds," IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 104-11, 2013.

[39] T. Wood, P. Shenoy, A. Venkataramani and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation, Cambridge, MA, pp. 17-17, 2007.

[40] T. Hirofuchi, H. Nakada, S. Itoh and S. Sekiguchi, "Reactive consolidation of virtual machines enabled by postcopy live migration," 5th International Workshop on Virtualization Technologies in Distributed Computing, VTDC'11, pp. 11-18, 2011.

[41] Kejiang Ye, Xiaohong Jiang, Dawei Huang, Jianhai Chen and Bei Wang. "Live migration of multiple virtual machines with resource reservation in cloud computing environments," IEEE International Conference On Cloud Computing (CLOUD), pp. 267-274, 2011.

[42] C. Lin, Z. Jian and C. Hsu. "A strategy of service quality optimization for live virtual machine migration," IEEE 17th International conference on Computational Science and Engineering (CSE), 2014.

[43] M. F. Bari, M. F. Zhani, Qi Zhang, R. Ahmed and R. Boutaba. "CQNCR: Optimal VM migration planning in cloud data centers," Networking Conference IFIP, pp. 1-9, 2014.

[44] E. Zhai, G. D. Cummings and Y. Dong. "Live migration with pass-through device for linux VM," The Ottawa Linux Symposium, pp. 361-268, 2008.

[45] A. Kadav and M. M. Swift. "Live migration of direct-access devices," ACM SIGOPS Operating Systems (USA) 43(3), pp. 95-104, 2009.

[46] T. Hirofuchi, H. Nakada, S. Itoh and S. Sekiguchi, "Enabling instantaneous relocation of virtual machines with a lightweight VMM extension," Cluster, Cloud and Grid Computing (CCGrid), 2010.

[47] K. Chanchio and P. Thaenkaew, "Time-bound, thread-based live migration of virtual machines," 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2014.

[48] [H. Lagar-Cavilla, J. A. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno and M. Satyanarayanan, "SnowFlock: Rapid virtual machine cloning for cloud computing," 4th ACM European Conference on Computer Systems, pp. 1-12, 2009.

[49] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun and Haogang Chen, "Live and incremental whole-system migration of virtual machines using block-bitmap," IEEE International Conference On Cluster Computing, pp. 99-106, 2008.

[50] K. Wang, J. Rao and Cheng-Zhong Xu, "Rethink the Virtual Machine Template," SIGPLAN Notices, vol. 46, pp. 39-49, 07, 2011.

[51] P. Riteau, C. Morin and T. Priol. “Shrinker: Improving live migration of virtual clusters over WANs with distributed data deduplication and content-based addressing,” 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011).

[52] A. Aldhalaan and D. A. Menasce, "Analytic performance modeling and optimization of live VM migration," 10th European Workshop, EPEW 2013, pp. 28-42, 2013.

[53] C. Chen and J. Cao, "Prediction-based optimization of live virtual machine migration,"Network and Parallel Computing LNCS, Vol. 8707, pp. 347-56, 2014.

[54] W. Huang, Q. Gao, J. Liu and D. K. P. “High performance virtual machine migration with RDMA over modern interconnects,” IEEE International Conference on Cluster Computing, pp. 11-20, 2007.

[55] X. Song, J. Shi, R. Liu, J. Yang and H. Chen, "Parallelizing live migration of virtual machines," 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE13, pp. 85-95, 2013.

[56] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson and A. Warfield. “Remus: High availability via asynchronous virtual machine replication,” 5th USENIX Symposium on Networked Systems Design and Implementation. NSDI, pp. 161-174, 2008.

[57] J. Protic, M. Tomasevic and V. Milutinovic. “Distributed Shared Memory: Concepts and Systems,” John Wiley & Sons, 1998 - Computers.

[58] Li, Kai. "IVY: A Shared Virtual Memory System for Parallel Computing." ICPP (2) 88 (1988): 94.

[59] Carter, John B. "Design of the Munin distributed shared memory system."Journal of Parallel and Distributed Computing 29, no. 2 (1995): 219-227.

[60] P. Keleher, S. Dwarkadas, A. L. Cox and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proc. of the winter 1994 USENIX Conference, pages 115-131, January 1994.

[61] Scales, Daniel J., and Kourosh Gharachorloo. "Shasta: A System for Supporting Fine-Grain Shared Memory Across Clusters." In PPSC. 1997.

[62] Dwarkadas, Sandhya, Nikos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. "Cashmere-VLM: Remote memory paging for software distributed shared memory." In Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings, pp. 153-159. IEEE, 1999.

[63] Singla, Aman, and Umakishore Ramachandran. "The Beehive Cluster System."

[64] N. Carriero and D. Gelernter. Linda in context. Commun. ACM, 32(4):444–458, April 1989

[65] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M.F. Kaashoek, ‘‘Portability in the Orca Shared Object System,’’ Technical Report, Vrije Universiteit, Amsterdam, The Netherlands (Sept. 1997).

[66] Nikhil, Rishiyur S., Umakishore Ramachandran, James M. Rehg, Robert H. Halstead Jr, Christopher F. Joerg, and Leonidas Kontothanassis. Stampede A Programming System for Emerging Scalable Interactive Multimedia Applications. Springer Berlin Heidelberg, 1999.

[67] Almasi, Gheorghe, Sameh Asaad, Ralph E. Bellofatto, H. Randall Bickford, Matthias A. Blumrich, Bernard Brezzo, Arthur A. Bright et al. "Overview of the IBM Blue Gene/P project." IBM Journal of Research and Development 52, no. 1-2 (2008): 199-220.

[68] B. W. Cheung, C. Wang and K. Hwang. “JUMP-DP: A software DSM system with low-latency communication support,” International Conference on Protocol for Software Distributed Shared Memory Parallel and Distributed Processing Techniques and Applications (PDPTA’2000), Las Vegas. 2000.

[70] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan and M. Oskin. Latency-tolerant software distributed shared memory. Presented at 2015 USENIX Annual Technical Conference (USENIX ATC 15). 2015.