Memory Sharing in Cloud Servers

SUBMITTED IN FULFILMENT OF THE REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Yaozhong Ge

BINFOTECH(HONS)

SCHOOL OF COMPUTER SCIENCE

FACULTY OF SCIENCE

QUEENSLAND UNIVERSITY OF TECHNOLOGY

2023

Dedicated to my parents for their immeasurable patience, support, and encouragement.

Abstract

Over-committing computing resources is a widely adopted strategy in Infrastructure as a Service (IaaS) cloud data centres for increased cluster utilization. In IaaS cloud data centres, virtual machines (VMs) do not always fully utilize their provisioned resources. The existence of the gap between the provisioned and actual utilized resources of VMs gives cloud service providers an opportunity to over-commit resources while meeting their service-level agreements (SLAs). Appropriate resource over-commitment enables the use of fewer physical machines (PMs) and consequently improves the energy efficiency of a cloud data centre, which can significantly reduce its operating cost with only a low risk of violating Quality of Service (QoS) requirements. In principle, risks of over-committing memory resources can be hedged if each VM on PMs consumes only a small portion of the requested memory resource. If such a hedge fails, one of the consequences is memory overload. To handle memory overload on PMs, cloud service providers must live migrate VMs from memory overloaded PMs to underutilized PMs. However, in over-committed cloud data centres, live VM migration can cause cascading overloads in the worst case.

This thesis presents systematic memory sharing frameworks for handling memory overload of PMs in over-committed cloud data centres. There are three progressive levels of memory sharing, one-to-one memory sharing, one-to-many memory sharing, and many-to-many memory sharing.

One-to-one memory sharing is detailed as a fundamental memory sharing system between two machines. In the one-to-one memory sharing system, the underutilized PM is defined as the lender PM, which can feasibly lend its spare memory resource to another PM. The memory overload PM is defined as the borrower PM, which borrows memory resource from a remote PM. The one-to-one memory sharing system involves a unified control algorithm for a PM to automatically borrow memory if memory overload occurs or to lend spare memory if it is

feasible. Moreover, the procedure for memory sharing negotiation and establishment is also described in this memory sharing system.

One-to-many memory sharing is proposed for a lender PM to lend its spare memory resource to many borrower PMs at the same time. By expanding one-to-many memory sharing, many-to-many memory sharing is proposed for many lender PMs to share their spare memory resource to many borrower PMs. In contrast to one-to-many memory sharing, in which a borrower PM can borrow the memory resource from only one lender PM, many-to-many memory sharing allows a borrower PM to borrow the memory resource from many lender PMs. Since one-to-many and many-to-many memory sharing have the same requirements, memory sharing frameworks are proposed from two perspectives: instant processing and block processing.

Instant processing of a memory sharing framework is designed for instantly pairing the lender PM to the borrower PM for both one-to-many and many-to-many memory sharing. It involves a clustering-based method for lender PM selection, through taking advantage of applying machine learning to statistical information gained from resource usage trace data, in order to reduce overhead and the risk of memory overload of lender PMs during the memory sharing. The instant processing of a memory sharing framework also includes a mean-based First-Fit Decreasing (FFD) method to select the lender PM.

Block processing of a memory sharing framework is also designed for both one-to-many and many-to-many memory sharing. In the block processing of a memory sharing framework, multiple memory borrowing requests are processed at the same time instead of being processed one by one. This allows formulation of an optimization problem. Hence, a heuristic procedure, Genetic Algorithm (GA), is used for lender PM selection in the block processing of the memory sharing framework.

Experimental studies are conducted to evaluate the proposed system and frameworks. The one-to-one memory sharing system is physically implemented and is fully functional. One-to-many and many-to-many memory sharing frameworks are evaluated by simulation of Alibaba's cluster. Instant processing of a memory sharing framework provides a reasonable improvement of memory resource utilization, while block processing of memory sharing further improves memory utilization of the cluster in both one-to-many and many-to-many memory sharing.

Keywords

Cloud computing services, data centre, memory overload, memory sharing, resource over-committing.

Acknowledgments

I would like to express my deep gratitude to the most supportive person during my PhD candidature, my principal supervisor, Prof. Yu-Chu Tian. He has continuously guided me through every challenge in my PhD journey. Without his valuable criticisms and insightful advice, I would not have been possible to complete my PhD research.

I also express appreciation to my associate supervisor, Dr. Maolin Tang for his numerous supports and valuable suggestions throughout my PhD journey.

Finally, I would like to thank my fellow students and postdocs in Prof. Tian's group. They give me great encouragement and valuable advice when I experience difficulties during my PhD research.

Table of Contents

Nomenclature

Abbreviations

ACO	Ant Colony Optimization
API	Application Programming Interface
AWS	Amazon Web Services
B	Buffer transmitted through RDMA operation
CNN	Convolutional Neural Networks
CQE	Completion Queue Element
DRAM	Dynamic Random-Access Memory
FCFS	First Come First Serve
FFD	First-Fit-Decreasing
GA	Genetic Algorithm
GCP	Google Cloud Platform
GNX	Generalized N-point Crossover
IaaS	Infrastructure as a Service
iSCSI	Internet Small Computer Systems Interface
KVM	Kernel-based Virtual Machine
MR	Memory Region
NoF	NVMe over Fabric
NVMe	Non-Volatile Memory Express
NVRAM	Non-Volatile Random-Access Memory
OpenMP	Open Multi-Processing

PaaS	Platform as a Service
PCIe	Peripheral Component Interconnect Express
PM	Physical Machine
PMs	Physical Machines
QoS	Quality of Service
POSIX	Portable Operating System Interface
PSO	Particle Swarm Optimization
QP	Queue Pair which is composed by a SQ and a RQ
RC	Reliable Connection
RDMA	Remote Direct Memory Access
RQ	Receive Queue
RR	Round Robing
SaaS	Software as a Service
SDN	Software-Defined Networking
SHC	Stochastic Hill Climbing
SLAs	Service-Level Agreements
SQ	Send Queue
SSD	Solid-State Drive
SSI	Single System Image
SWIM	System Wide Information Management
UC	Unreliable Connection
UD	Unreliable Datagram
VM	Virtual Machine
VMs	Virtual Machines
WQE	Work Queue Element

Symbols **Chapter**

$\mathbb{F}$	Total distance between threshold and each lender PM candidature	Ch5
M_b	Memory used by kernel buffers	Ch5
M_c	Memory used by page cache and slabs	Ch5
M_d	Distance between used memory and threshold	Ch5
M_f	Memory free	Ch5
M_r	Memory reserved for other applications	Ch3
M_{req}	Memory requested by borrower PM	Ch3,4,5
M_s	Memory shareable	Ch3,4,5
M_{shared}	Memory resource has been shared out	Ch4,5
M_t	Memory total	Ch3,4,5
M_u	Memory used	Ch3,4,5
$M_u^{(mshr)}$	Memory used for a memory sharing (mshr) enabled PM	Ch5
N_b	Total number of borrower PMs	Ch5
N_l	Total number of lender PM candidatures	Ch5
N_m	Total number of PMs	Ch4,5
R	Set of memory borrowing Requests	Ch5
S_t	Swap total	Ch3
S_u	Swap used	Ch3
T	Total time points	Ch4
T_{lower}	Lower threshold to trigger memory overload detection	Ch3,4,5
T_{upper}	Upper threshold to trigger memory overload detection	Ch3

Greek Letters

$\boldsymbol{\mu}$	Mean profile	Ch4

Superscripts

$\boldsymbol{X}$	Set of data patterns represent memory usage data	Ch4

Subscripts

d	Distance of two PMs	Ch4
i	The first index of PM	Ch4
j	The second index of PM	Ch4
k	Distance factor	Ch5
n	The index of PM	Ch5
$\boldsymbol{x}_i$	Memory usage data set for PM with index i	Ch4

List of Figures

List of Tables

Chapter 1

Introduction

In the era of data deluge, data are being generated and disseminated by countless digital devices at an unprecedented speed then collected and consumed by industry. Internet related companies produce various intelligent computing services based on big data for informing better decisions or providing product services. These intelligent computing services are commonly served through virtual machine (VM) centred cloud services, such as Infrastructure as a Service (IaaS).

Over-committing computing resources is a widely adopted strategy for increased cluster utilization in IaaS cloud data centres. In a cloud computing service of IaaS, computation tasks are run in virtual machines (VMs), where VMs are deployed on physical machines (PMs). A physical machine (PM) has to accept more VMs than its capacity, if the strategy of over-committing computing resources is enabled. Such a strategy becomes feasible if most of the VMs have not fully utilized their requested computing resource. However, a potential consequence of over-committing computing resources is memory overload of PMs. Memory overload occurs if memory usage exceeds a defined alarm threshold, exposing running computation tasks at a risk of being terminated by the operating system.

In an IaaS cloud data centre, live migration of VMs is a prevailing measure to handle a memory overloaded PM. In the live migration of VMs, one or more VMs on memory overloaded PMs are migrated to underutilized PMs on the fly. However, this not only consumes network bandwidth, CPU, and other resources, but also causes a temporary unavailability of the VMs being migrated.

The main purpose of this research project is to develop an approach to sharing memory resources among cloud servers. Sharing memory resources means the computer memory resource

of a PM can be used by other PMs in operating system level. It tends to increase computing resource utilization and the computation reliability of Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). Moreover, sharing memory resources enables the ability to deploy memory-intensive applications beyond the capacity of physical computing resources.

1.1 Research Background

This section describes the background information on this research, including a general background on cloud computing and a discussion of the computing resource management challenge of cloud computing.

1.1.1 Services of Cloud Computing

There are three mainstream computing service models for clouding computing: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [Kavis, 2014].

a Infrastructure as a Service (IaaS)

IaaS is a cloud computing offering in which a cloud service vendor provides customers access to computing resources such as servers, storage and networking [IBM, 2019]. In IaaS, each resource is offered as a separate service component. Thus, customers just rent necessary service components for their business, rather than buying their own physical infrastructure [Microsoft Azure, 2019a]. Moreover, IaaS reduces the problems for customers of managing their own physical infrastructure and allows service components to be conveniently scaled up and down according to demand.

IaaS allows customers to deploy their own platforms and applications on the service providers' infrastructures, and to conveniently scale up and down to meet demand, because the computing resources are managed by the service provider.

b Platform as a Service (PaaS)

PaaS includes not only infrastructure resources, but also middleware, such as development tools, database management systems and other underlying applications, in order to support a complete web application lifecycle: building, testing, deploying, managing, and updating [Microsoft Azure, 2019b]. Customers just need to focus on developed applications and services, while everything else is typically managed by the cloud service vendor [Microsoft Azure, 2019b].

c Software as a Service (SaaS)

SaaS delivers software applications over a network [Cusumano, 2010]. With SaaS products, software providers can deploy their software hosted on IaaS or PaaS, and grant customers access to the software in the cloud environment. Software providers are responsible for managing customer access, account creation, resource provisioning, account management in the software, and so on.

1.1.2 Virtual Machines

The virtual machine is one of the computing resources in IaaS and PaaS. Usually, multiple VMs are deployed on one PM, and these VMs are managed by a hypervisor, such as Kernel-based Virtual Machine (KVM). This PM is named the host system. The Hypervisor is an operating system component running on the host system. It manages and monitors resource utilization of VMs, and allocates VMs a part of hardware resources, such as CPUs and memory resources. In the cloud environment, hypervisors from multiple host systems work integrally for managing very large numbers of VMs, and scaling resources up and down as requested by customers. Moreover, in IaaS, customers rent VMs directly from cloud service vendors, and self-maintain these VMs. In contrast, cloud service vendors maintain VMs in PaaS. They deploy services on VM clusters and make services available to their customers.

1.1.3 Over-committing Computing Resource in Cloud Computing

Over-committing computing resources is a widely adopted strategy in IaaS cloud data centres for increased cluster utilization. In IaaS cloud data centres, VMs do not always fully utilize

their provisioned resources. The existence of a gap between the provisioned and actual utilized resources gives cloud service providers an opportunity to over-commit resources while meeting their Service-Level Agreements (SLAs). Appropriate resource over-commitment enables the use of fewer PMs and consequently improves the energy efficiency of a cloud data centre. This will significantly reduce the operating costs of the data centre, with only a low risk of violating Quality of Service (QoS) requirements. While over-committing resources has impacts on the performance of VMs, it shows limited performance degradation in general for the end users of cloud services except for memory overload.

1.1.4 Memory Overload

Memory overload occurs when a PM has memory pressure over a threshold for more than five minutes [Baset et al., 2012]. It risks the running computation tasks of being terminated by the operating system. If the memory usage of a PM reaches its total capacity, the latest process that requests non-existent memory resources will be terminated. In this case, the SLA and QoS requirements may be violated.

1.1.5 Virtual Machine Live Migration

Live VM migration is a prevailing measure to handle memory overload in PMs. If a PM is considered as being overloaded, one or more VMs running on the overloaded PM could be migrated to other available PMs in order to avoid any violation of QoS requirements. However, live VM migration introduces overheads to all involved entities and processes, such as the source and destination PMs, the VM being migrated, and the VMs co-located on these two PMs [Zhang et al., 2018]. The overheads can be a short unavailability of the VM being migrated, extra costs of shared infrastructure resources (e.g., network bandwidth), and/or extra costs of CPU resources on the source and destination PMs. In addition, the CPU of the source PM could be underutilized after the VM migration. Moreover, live VM migration also affects resource management related jobs for the cloud infrastructure, such as VM placement and VM consolidation.

1.1.6 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a method of reading data from, and writing data to, the memory of a remote machine. It neither involves any network software stack or kernel nor consumes any CPU time for data transfers with the remote memory [Guo et al., 2016]. In contrast, transmitting a message in a traditional socket-based model requires copy operations through the kernel space. Therefore, RDMA eliminates context switch, intermediate data copies in various stack, and protocol processing in contrast to the traditional model. It is one of the capabilities of InfiniBand-networked PMs. InfiniBand is a special networking technology that provides extremely high data throughput and very low latency, particularly in high-performance computing, in-network computing, and data-intensive computing environments [Mauch et al., 2013, Wu et al., 2019, Xue and Zhu, 2021]. Moreover, efficiency of VM migration can be significantly improved by using RDMA [Huang et al., 2007].

1.2 Statement of Research Problem

Over the years, the rate of resource utilization has been enhanced in cloud data centres through the virtualization-based over-commitment strategy and server consolidation technologies. However, there is still much room for its further improvement. For example, Alibaba's large-scale cloud systems show that over-committing and under-committing problems can coexist in cloud data centres [Everman et al., 2021].

The challenge is how to further increase the memory utilization of PMs without worrying about memory overload or frequent live VM migration operations. Firstly, over-committed PMs should not be assigned extra computation tasks with a higher possibility of memory overload. Secondly, under-committed PMs are kept online, even if an over-commitment strategy and server consolidation operations are applied.

The research problem is how to share memory resources between cloud servers. Firstly, there is no efficient approach to run applications beyond the memory capacity of the host PM. Secondly, there is no efficient approach to utilize memory resource of the under-committed PMs for handling memory overload occurred on the over-committed PMs. Thirdly, there is no efficient technique to improve overall computing resource utilization of the cloud servers, meanwhile keep computing availability.

1.3 Motivation of the Research

Both IaaS and PaaS are commonly designed to run PMs in high or full utilization and avoid idling VMs and PMs, in order to minimize cost of holding resources and save energy consumption. However, this design may lead to a lack of computing resource in critical situations, especially result in memory overload of the PM. There are several examples of the scenario which can be benefited from memory sharing among cloud servers.

a Underutilized Resource Provisioning

An issue in IaaS is that VMs do not always fully utilize their provisioned resources. The issue between provisioned and utilized resources gives cloud providers an opportunity to overbook their infrastructures, by deploying more virtual resources than the physical resources available [Tsitsipas et al., 2017].

b Memory Overload

Overbooking can leverage underutilized capacity in the cloud and make service more cost efficient than before. However, memory overload occurs if resource utilization of VMs (virtual resources) is higher than the capacity of physical resources. CPU resource overload might slow down the computation, while memory resource overload would cause computing resource unavailability.

A prevailing measure to handle memory overload is live VM migration. However, the migration procedure may take longer than the duration of memory overload. With high memory over-committing ratios, the duration of memory overload is transient in most cases. For realistic web workloads in a data centre, the majority of memory overloads take up to two minutes, whereas only one third of them last 10 seconds or shorter [Williams et al., 2011].

Memory sharing can be a complementary measure to live VM migration. When handling transient memory overload, memory sharing can be a better measure than live VM migration by temporally sharing the spare memory resource of a PM to another PM with memory overload. It is more lightweight than live VM migration by applying RDMA to relieve the computing pressure in order to achieve high resource utilization with low cost and increase computation

reliability. RDMA allows sharing memory temporarily among PMs with speed similar as ordinary memory speed.

It is worth mentioning that the memory sharing presented in this research is not designed to replace VM live migration. Rather, it will effectively reduce the number of live VM migrations induced by memory overload. The memory sharing is suitable for the management of transient memory overload that lasts for a short period of time. It can be pre-activated or activated instantly. However, for sustained memory overload, memory sharing is not recommended and VM live migration will be more suitable. After successful migration, local memory will be used instead of remote memory. It is noted that live migration requires to transfer the memory pages of the VM from one PM to another PM and will typically cause unavailability of the VM during the migration. Depending on the size of the VM, live VM migration may take seconds or longer. It may even use RDMA of InfiniBand for the required transmission.

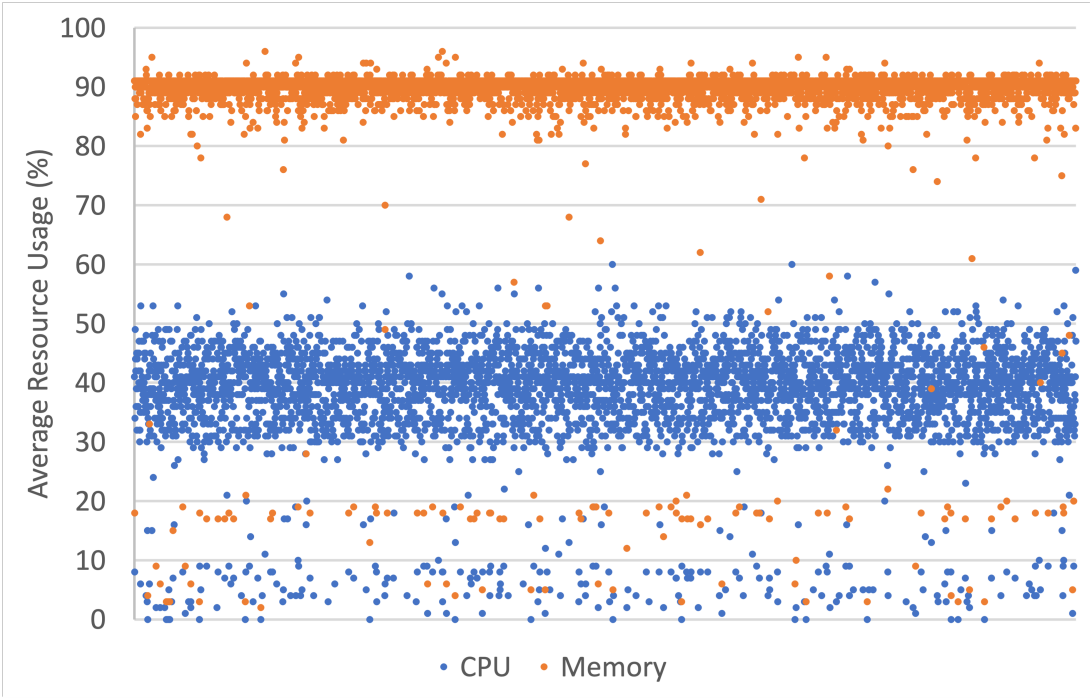
c Insufficient Physical Machine Capacity

Cloud vendors might find it hard to deploy memory-intensive VM instances in peak hours due to lack of memory resource. Memory-intensive types of VM requires huge amounts of memory resource. During peak hours, the amount of available memory resource in PMs might be limited. This is not large enough for deploying memory-intensive VMs. Cloud vendors must delay new deployment or perform VM consolidation in order to make enough memory resource available for new deployments. In addition, cloud vendors have limitations on deploying a memory-intensive VM type which requires more memory than PM capacity. However, these limitations can be eliminated by enabling memory sharing.

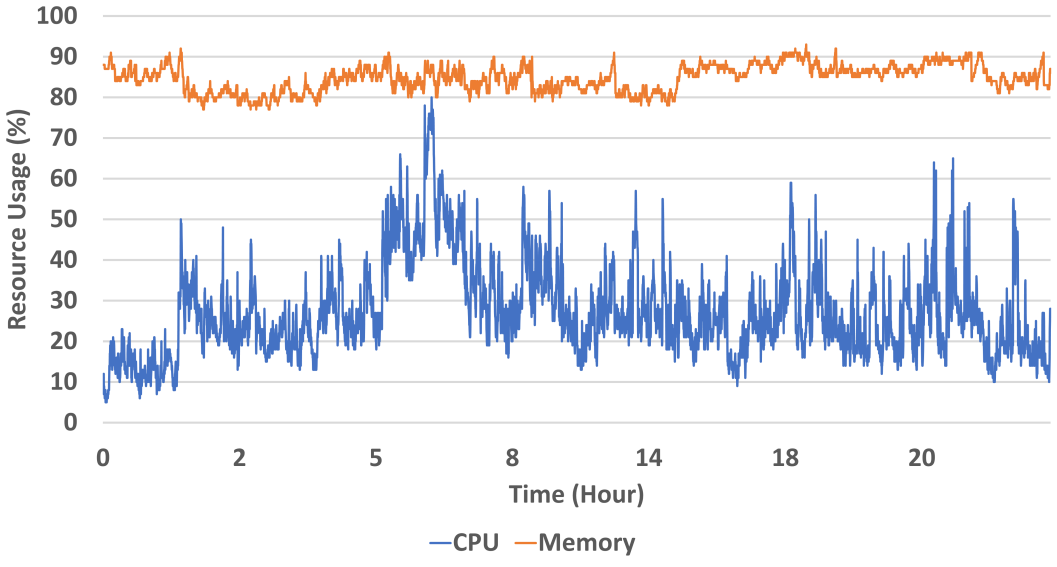
d A Motivational Example

Memory sharing allows a physical machine to remotely access other physical machines' memory resources. Since the main factor in the overload issue is the lack of local memory resources, remote memory resources could act as extra memory resources for the overloaded physical machine.

A motivational example is Alibaba's 8-day trace data [Guo et al., 2019] [Alibaba Open Source, 2018], which shows that the cluster's average memory usage is 88%, while average



(a) Average Resource Usage of $4,023$ PMS Over 8 Days



(b) Resource Usage of a Typical PM Over 24 Hours

Figure 1.1: Alibaba's Cluster Trace Data

CPU usage is 38%, as shown in Figure 1.1a. Thus, memory resource is the bottleneck of further increment of cluster utilization.

Memory resource sharing provides an opportunity to further raise the overall memory utilization of a cluster. In Alibaba's cluster trace data, memory usage over 95% occurs 8.2% of record. Memory usage over 90% occurs 55.7% of record. It means there is still a plenty of room for raising overall memory utilization of the cluster.

To analyse in depth, a PM is randomly selected as an example. Figure 1.1b exhibits resource usage of a typical PM in Alibaba's cluster. In most time, memory usage fluctuates between 80% and 90%, while CPU usage is relatively low compared with memory. If extra computation tasks are scheduled to this PM for increment of 10% memory utilization rate, this PM would occur several times of memory overloads. It is unacceptable in existing systems. However, if memory overload is handled by memory sharing through temporarily utilizing the spare memory resource of a remote PM, such increments on memory utilization will be possible.

1.4 Technical Gaps

Efforts to solve these problems focus on improving traditional strategies, such as VM live migration and threshold alternatives to trigger scaling procedures.

VM live migration must suspend computation tasks in VMs. Data transmission from a PM to another requires CPU resource on both PMs for packet encapsulation and decapsulation and occupies a part of network bandwidth to transmit persistent data and memory data of VMs. In addition, VM live migration solves the memory resource overload problem, but brings another problem. Migrating a VM not only relieves memory resource overload, but also reduces CPU resource utilization. If the CPU resource is not overloaded before migration, the CPU resource will be underutilized after migration.

Threshold alternatives focus on figuring out when to trigger a scaling procedure, which is a sweet point between the cost and the computation reliability included in the SLA.

Memory balancing and remote memory are two major candidate approaches for handling memory overload of PMs in cloud data centres. However, memory balancing requires the coexistence of over- and under-committed computation tasks in the same PM. The total memory requested from all VMs on the PM is capped by the memory capacity of the PM. In comparison,

remote memory for PMs does not have such a limitation. However, existing efforts in remote memory for PMs require a pre-configuration of disaggregated memory on dedicated PMs. Furthermore, multiple or hybrid disaggregated memory designs are not essential for handling memory overload for PMs in cloud centres.

The discussion above outlines the technical gaps that exist regarding our memory sharing requirements for PMs in cloud data centres. The requirements of memory sharing are: 1) a PM can access remote memory resources when it becomes memory overloaded; and 2) when a PM has the spare memory resource, it can share out its memory resource to a remote PM. This motivates the research and development of this paper on memory sharing for handling memory overload on PMs in cloud data centres.

1.5 Research Questions and Objectives

This research project aims to develop a systematic memory sharing framework for handling the memory overload issue for PMs in over-committed cloud data centres. It is essential to deal with the inefficient memory resource utilization issue among multiple PMs since over-provisioned VM causes lack of memory resource on the VM's host system. The proposed framework improves resource utilization by introducing a new method, memory sharing, instead of solving research problems from optimization of an existing solution, live VM migration. There are three levels of memory sharing: one-to-one, one-to-many, and many-to-many, and the research questions are formed according to these levels of memory sharing. In particular, the following research questions are addressed in this thesis:

1. One-to-one Memory sharing: How to establish a memory sharing connection between the two PMs?

   (a) One-to-one memory sharing refers to one underutilized PM that can share its spare memory resource with one memory overloaded PM.

   (b) Connection procedure between the two PMs should be designed. A PM should automatically establish a memory sharing connection and use a remote memory resource when memory overloading. In contrast, a PM should keep waiting to share its spare memory resource when feasible.

(c) A fundamental memory sharing system needs to be designed and implemented to answer this question.

2. One-to-many Memory Sharing: How to decide which underutilized PM is selected for memory sharing?

   (a) One-to-many memory sharing refers to one underutilized PM able to concurrently share its spare memory resource with many PMs with memory overload.

   (b) This question seeks a memory sharing framework with a control or schedule algorithm to select an underutilized PM for sharing its spare memory resource with a memory overloaded PM because there can be many underutilized PMs and many memory-overload PMs appearing at the same time in an over-committed cloud data centre.

   (c) The solution needs to be designed so that the same underutilized PM can be selected multiple times in case the spare memory resource of an underutilized PM is large enough for sharing with multiple PMs with memory overload.

3. Many-to-many Memory Sharing: How to decide which underutilized PMs are selected for memory sharing?

   (a) Many-to-many memory sharing refers to many underutilized PMs concurrently sharing their spare memory resource with many memory-overloaded PMs.

   (b) How to select multiple underutilized PMs for a memory overloaded PM, answers this question. This design is targeted to the situation, where no one of the underutilized PMs can satisfy the remote memory requirements of a memory overloaded PM although multiple underutilized PMs have the spare memory resource to share. Many-to-many memory sharing, as the last level, is also designed to have better memory balancing among PMs than one-to-many memory sharing.

   (c) The proposed memory sharing framework needs to be expanded in order to select multiple underutilized PMs for a single memory overload PM.

This research develops a comprehensive memory sharing framework for handling memory overload issue in the cloud data centre. To tackle these challenging research questions, three progressive objectives have been designed for achieving the research goal, with consideration

that virtualization environment on PM cluster is quite complicated for directly designing and implementing a brand-new framework:

1. Design and implement a one-to-one memory sharing system that allows a PM itself to understand its memory usage status and automatically switch its role between an underutilized PM role and a memory overloaded PM role. In the role of memory overloaded, the PM itself can automatically establish memory sharing connection to an underutilized PM. In the role of underutilized, the PM can accept a memory sharing connection for sharing out its spare memory resource.

2. Design a one-to-many memory sharing framework that allows multiple memory-overloaded PMs to use spare memory resource from one underutilized PM. Depending on the number of memory-overloaded PMs, multiple underutilized PMs can be used.

3. Design a many-to-many memory sharing framework that allows many PMs with memory overload to use spare memory resources from many underutilized PMs.

1.6 Major Contributions and Innovations

There are three major contributions in this thesis: a one-to-one memory sharing system, and two different approaches for both one-to-many and many-to-many memory sharing. One-to-many and many-to-many memory sharing are jointly solved because they have very many similarities. Regarding this, instant processing and block processing approaches to memory sharing are proposed. The major contributions of this thesis are as follows:

1) A one-to-one memory sharing system is presented and implemented. A unified control algorithm is designed for a PM to automatically use a remote memory resource when memory overloading, and for a PM to automatically share out its spare memory when it is feasible.

2) Instant processing of memory sharing is presented for both one-to-many and many-to-many memory sharing. Memory sharing requested by memory-overloaded PMs is processed one by one. It is similar to the first-in, first-out method, thus memory sharing request can be processed immediately. Instant processing of memory sharing contains a

clustering-based algorithm with tolerance of missing values for filtering PM candidates as underutilized PMs, and a mean-based first-fit-decreasing (FFD) algorithm for instant processing of memory sharing.

3) Block processing of memory sharing is presented for both one-to-many and many-to-many memory sharing as well. One block of memory sharing requested by memory overloaded PMs is processed at a time. Block processing of memory sharing utilizes a heuristic algorithm, genetic algorithm (GA), to make a memory sharing plan for selecting underutilized PMs.

There are four major innovations in this thesis, as follows:

1) The memory sharing system is a novel solution for handling memory overloaded of PMs, while the existed prevailing measure to handle the memory overload problem is to migrate out VMs running on memory overloaded PMs.

2) A new clustering method is presented for time-series data with missing values. Existed clustering methods usually use interpolation to compensate for missing data in time-series datasets, while our proposed clustering method can get results without the need of interpolation for compensation.

3) The instant processing framework of memory sharing is presented for fast response to memory sharing requirements. It can be an extra measure before performing the live migration of VMs, because the instant processing is preferred for handling transient overload, while the live VM migration is more suitable to handle sustained overload.

4) The block processing framework of memory sharing is presented as a complementary solution to the instant processing framework for handling the memory overload problem to over-committed cloud data centres. It can handle more memory overloaded PMs than instant processing framework by enabling heuristic algorithm, genetic algorithm (GA).

1.7 Thesis Organization

The overall structure of this thesis is shown in Figure 1.2. It should be noted that algorithms designed for instant processing of memory sharing and algorithms designed for block processing of memory sharing can be used for both one-to-many and many-to-many memory sharing.

Thus, Chapter 4 presents instant processing of memory sharing, while Chapter 5 presents block processing of memory sharing. Frameworks in both chapters involve two variants of memory sharing framework for one-to-many and many-to-many memory sharing.

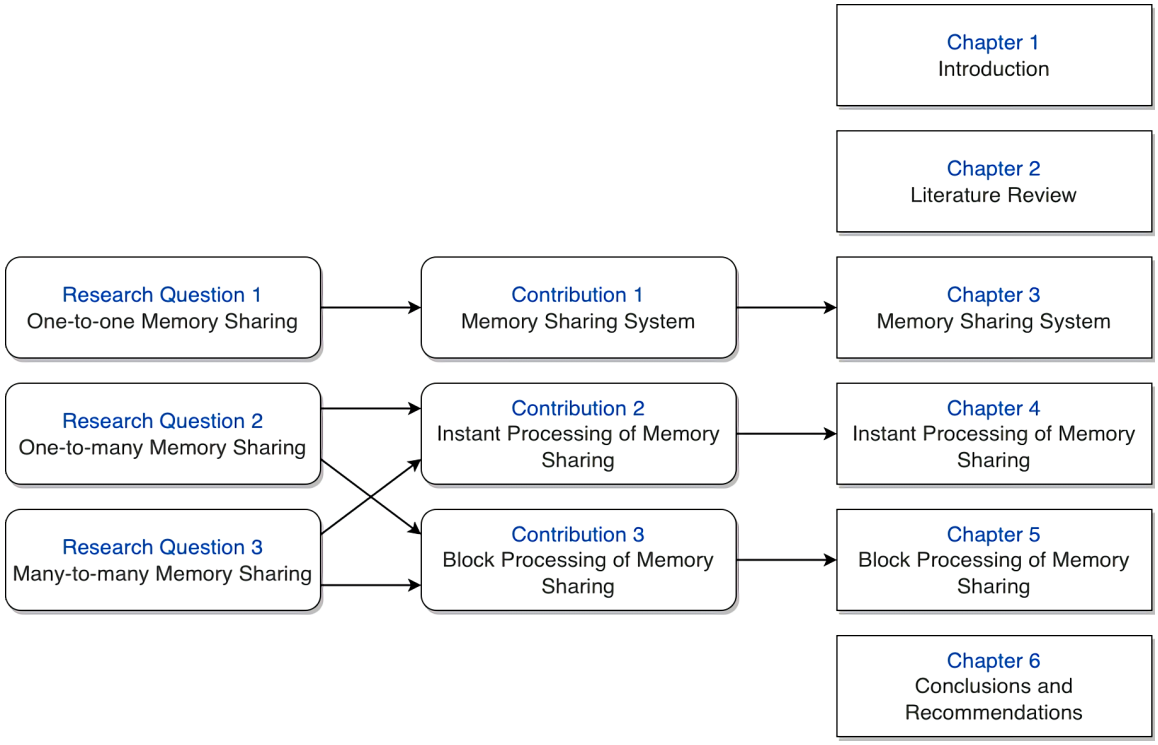


Figure 1.2: Thesis Organizational Structure

The remainder of the thesis is organized as follows:

Chapter 2 presents the literature review of studies relevant to the handling of memory overload issue, including VM migration, distributed shared memory, memory sharing, and memory resource management. It also investigates issues and technical gaps in existing efforts.

Chapter 3 develops a memory sharing system for one-to-one memory sharing. With this system, a PM can automatically share its spare memory resource when it is feasible, or access remote memory resource as secondary memory when memory is overloaded.

Chapter 4 proposes an instant processing framework of memory sharing to process memory sharing one-by-one. It includes two variants of the framework for both one-to-many memory sharing and many-to-many memory sharing. A clustering method and a mean-based FFD for memory sharing is also proposed in this chapter.

Chapter 5 proposes a block processing framework of memory sharing to process multiple

memory sharing requests all at once. It also contains two variants of a framework for one-to-many memory sharing and many-to-many memory sharing. A GA-based algorithm is proposed for maximizing the rate of successful paring a memory overloaded PM with one or more PMs with spare memory resources.

Chapter 6 concludes the thesis by summarizing the findings and contributions. Suggestions and potential research directions for future work are also presented in this chapter.

1.8 List of Publications

The research outcomes derive a conference paper and three journal articles. The conference paper has been presented in international symposium. One of the journal articles has been submitted. Others are in preparation. These articles are listed below:

1) Ge, Y., Ding, Z., Tang, M., & Tian, Y. C. (2019). Resource provisioning for MapReduce computation in cloud container environment. In *IEEE 18th international symposium on network computing and applications*, pp. 1-4. Cambridge, MA, USA. IEEE

2) Ge, Y., Tian, Y. C., *et al.* (2023). Memory sharing for handling memory overload on physical machines in cloud data centres. Accepted for publication in the *Journal of Cloud Computing*. Relevant to Chapter 3

3) Ge, Y., Hu, S., Tian, Y. C., *et al.* Clustering-based memory sharing framework for handling memory overload in over-committed cloud. In preparation for submission to a journal. Relevant to Chapter 4.

4) Ge, Y., Tian, Y. C., *et al.* Genetic-algorithm based memory sharing framework for handling memory overload in over-committed cloud. In preparation for submission to a journal. Relevant to Chapter 5.

Chapter 2

Literature Review

In the public cloud, especially cloud computing of IaaS, there is an issue that VMs do not always fully utilize their provisioned resources. The issue between provisioned and utilized resources gives cloud providers an opportunity to overbook resources while meeting their SLAs. Therefore, admission control and scheduling mechanisms can be utilized to overbook physical resources in order to improve resource utilization and admit more applications concurrently [Tomás and Tordsson, 2013]. Overbooking can leverage underutilized capacity in the cloud and make its services profitable. However, overbooking may have impacts on server performance, and even violate SLAs [Hoeflin and Reeser, 2012]. Most of the time, performance impacts are hard for customers to observe. However, memory overload occurs when a physical host server has memory pressure over a threshold for more than five minutes [Baset et al., 2012]. The overload could lead to SLA violations and service reliability issues. Hence, preventing or solving memory overload issue would make overbooking unimpeded and beneficial to cloud providers.

This chapter reviews published literature related to the research issue investigated in this study. The rest of the chapter is organized as follows: Section 2.1 discusses VM migration technology for avoiding or handling memory overloading of PMs in the data centre. Section 2.2 investigates distributed shared memory inspired by Single System Image (SSI) cluster, which is a cluster of machines that operates as one single system. Section 2.3 describes what is Remote Direct Memory Access (RDMA). Section 2.4 discusses various technologies that allow a PM to be beneficial by using the memory resource of a remote PM. Section 2.5 investigates methods to manage memory resources for clusters in a cloud data centre, especially for the

cloud environment which has the capability for sharing spare memory resources among PMs in the cluster. Section 2.6 discusses memory sharing management in the optimization perspective. Finally, Section 2.7 concludes the chapter by summarizing the existing efforts related to the research problems and gaps found in related works.

2.1 Virtual Machine Migration

Live VM migration is the process of moving power on VMs from one compute resource or storage location to another [VMware, 2019]. It is a common scheme for solving overload issues. There are various methodologies for live VM migration. Svärd et al. [2015] introduced pre-copy, post-copy, and hybrid live migration. They shared the same procedure that included creating a new paused VM in the destination PM, transferring memory pages and CPU state from an existing VM to the new VM, resuming a new VM and destroying the existing VM. Pre-copy and post-copy aimed to decide when to transfer memory data to the destination. Pre-copy transferred memory pages before transferring the CPU state, while post-copy pulled memory pages after resuming the new VM. The Hybrid method took advantages from both pre-copy and post-copy in order to minimize the time and cost of migration. Mishra et al. [2012] mentioned persistent storage could be excluded from migration because it was usually in other locations and attached to the host via the network.

Apart from the migration procedure, there are three important questions to address: how to detect the overloaded condition, which VM to be migrated, and which PM as the destination.

For detecting the overloaded condition, Wood et al. [2007] introduced black-box and grey-box strategies to monitor the resource usage. Black-box monitors CPU, network, and memory resource from the PM; while grey-box has access to information, such as application-level statistics, inside the VM. Mishra et al. [2012] did a similar work to the black-box strategy.

A multi-object monitor system developed by Cao et al. [2021] combined black-box and grey-box strategies, but its grey-box only collected OS-level statistics. Although information from the grey-box might be quite helpful for solving other important questions, the grey-box requires additional modifications on each VM while the black-box does not. In addition, collecting application-level statistics could be more complicated than collecting information in the black-box strategy.

There are three common approaches to selecting a candidate VM for migration. Wood et al. [2007] selected VMs where resource requirements could not be locally fulfilled. However, this approach had the possibility of triggering cascading overloads which caused the destination PM to be in an overloaded condition after migration.

Xiao et al. [2013], Cao et al. [2021], and Ji et al. [2018] developed holistic approaches to solve this issue. they generated a holistic view of resource requirements and availability for all managed VMs and PMs, and then rebalanced the resource utilization in order to figure out which VM to be migrated. However, balancing resource utilization implies there are underutilized resources which go against the goal of overbooking.

The last approach was affinity based which incorporated other objectives in addition to resource requirements [Mishra et al., 2012]. Zhang et al. [2012] developed an affinity-based approach with the objective of reducing potential overload. In order to achieve this objective, a scattered migration algorithm was developed to scatter VMs with high utilization correlation onto different PMs. Reducing the potential overload could reduce the migration cost in the long term. However, resource utilization of cloud servers fluctuates over time, so this approach might not bring expected results.

Selecting a destination PM usually depends on the available resource capacity or affinity of the candidate VM migration. Available resource capacity-based selection considers only if availability of resources at the destination is enough [Mishra et al., 2012]. If the holistic approach is utilized to select a candidate VM for migration, the rebalancing plan will also describe where to migrate candidate VMs. In addition, leveraging the affinity between VMs can also be utilized to identify a suitable host destination PM. Zhang's work, as mentioned above, considered that VMs with a high utilization correlation were not favourable sharing partner in a single PM.

The capability of VM live migration causes even impacts on the system's performance. A short unavailability is inevitable to candidate VMs for migration. Meanwhile, a live migration procedure might have negative effects on performance of shared resources, such as network bandwidth. Voorsluys et al. [2009] evaluated the effects of a VM live migration on the performance of applications running on Xen platform and showed that the migration overhead was acceptable in most cases but could not be disregarded. Thus, the challenge is how to further reduce the migration overhead while increasing resource utilization.

2.2 Distributed Shared Memory

Memory resource sharing allows a PM to remotely access other PMs' memory resources. Since the main factor of memory overload issue is lack of memory resources, remote memory resources could act as extra memory resources to the overloaded PM. Thus, this potentially avoids the need for VM migration and improves the efficiency of memory resource utilization.

Ding [2018] designed a distributed shared memory system for the virtualization environment. It utilized the ivy protocol [Li, 1988] and the network feature of RDMA for high-speed data transfer. The distributed shared memory system abstracted memory resources from multiple PMs to a memory resource pool in order to support a cluster of machines that appeared to be one single system. Although Ding's work was not directly designed for this scenario, it proved the feasibility of sharing memory resources in a virtualization environment.

Another shared memory system was developed by Ahn et al. [2018]. It reserved some PMs for memory sharing purposes and attached the memory resources of these PMs to machines which required extra memory resources. The memory attachment approach allowed access to memory resource remotely. However, those memory sharing purpose machines wasted other resources, such as the CPU resource.

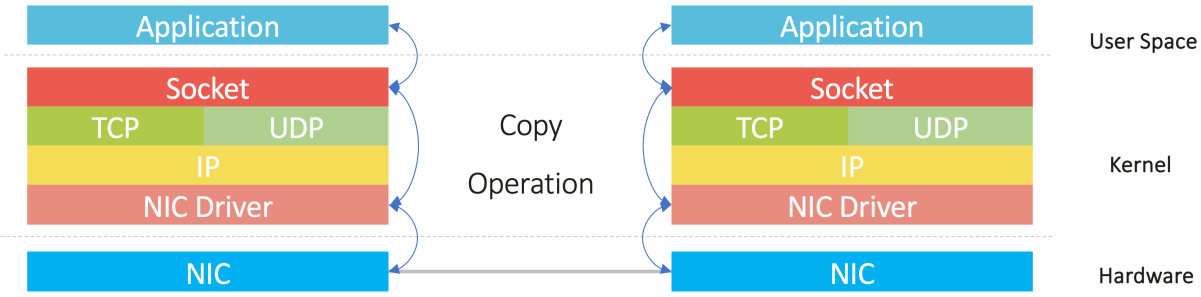
Jiang et al. [2018] developed a user-level utility for managing the memory distributed. It linked memory page to local disk and utilized the disk address to represent the memory address in order to access the memory resource remotely. However, the data transfer speed could not be fast enough for the native system memory purpose.

Seshadri et al. [2015] developed a virtual memory framework. It introduced overlaying memory resources which appended a virtual memory to the existing physical memory resource. Virtual memory was memory resource linked to remote PMs via RDMA. It could be completely transparent to both the hypervisor and the VM. However, dynamically resizing the overlay memory resource was not possible in this work.

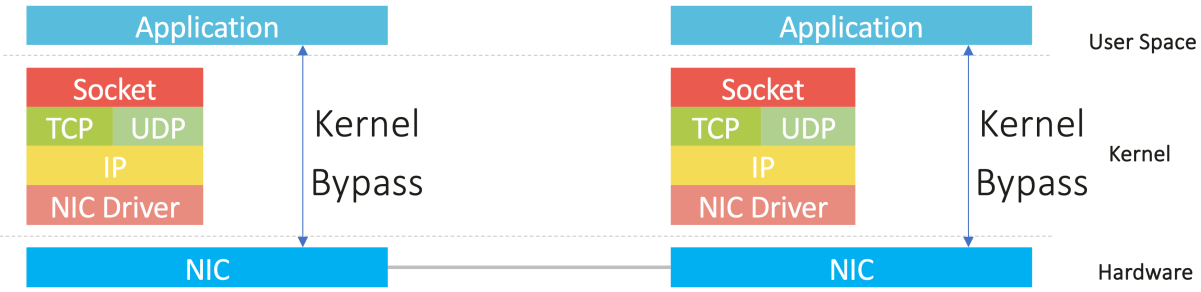
Overall, memory resource sharing among PMs is possible where RDMA is the key hardware feature to support it. Nevertheless, there are still two challenges: how to quickly attach/detach remote memory resource, and how to resize an attached remote memory resource.

2.3 Remote Direct Memory Access

There are two directions for implementing memory sharing: distributed shared memory and memory management based on Remote Direct Memory Access (RDMA). Distributed shared memory (DSM) has been proposed and developed in 1980s and 1990s, and became a part of single-system image (SSI) clustering system. It is a form of memory architecture where all physical machines (PMs) use one logical memory space which is comprised of physically separated memories. However, since modern data centres do not use SSI model to manage their clusters and node PMs of the clusters, such a direction is not considered in this research project. On the other hand, most of the modern data centres have an InfiniBand-enabled network for super-fast data transfer with low latency. A networking adapter which provides support of InfiniBand also provides a feature to access (read, write) memory on a remote physical machine (PM) without interrupting the processing of CPU(s) on that remote PM. Hence, this research starts from understanding how RDMA works in order to be able to do memory sharing based on RDMA.



(a) Traditional Messaging [Cardona, 2019]



(b) RDMA Messaging [Cardona, 2019]

Figure 2.1: Traditional Messaging Versus RDMA Messaging

Remote Direct Memory Access (RDMA) is an ability to read and write data on memory from a remote machine. According to Mellanox [Mellanox Technologies, 2019], RDMA performs data transfers without involving network software stack and kernel, nor consuming any

CPU time in the remote machine, as shown in Figure 2.1b. In contrast, transmitting a message in a traditional socket-based model requires copy operation through kernel space, as shown in Figure 2.1a. To sum up, RDMA eliminates context switch, intermediate data copies in various stacks, and protocol processing in contrast to the traditional model. Thus, RDMA is widely used for improving the performance of transferring data through a network; for example, improving performance of MPI-based parallel computing [Liu et al., 2004].

RDMA based applications can be implemented on both Ethernet and InfiniBand. Ethernet based implementations are based on TCP or UDP, which are either quite heavy weight (TCP-based) or require a lossless network (UDP-based). InfiniBand is a special stand for high-performance computing which provides extremely high data throughput and quite low latency. Thus, most RDMA based applications are based on InfiniBand.

2.4 Memory Sharing

Two main directions to mitigate memory pressure on machines are dynamic memory balancing and remote memory paging.

2.4.1 Dynamic Memory Balancing

In general, dynamic memory balancing can be adopted to re-balance provisioned memory resources for VMs running on a single PM. Several memory management mechanisms and policies are designed to dynamically re-allocate memory resources of VMs through the memory ballooning technique. The ballooning technique can reclaim the memory pages considered as the least valuable by the operating system running in a VM.

Moltó et al. [2016] proposed a memory over-subscription framework for over-committing the memory resource of PMs in cloud by providing automatic vertical elasticity to adapt the memory size of the VMs to their current memory consumption, where the ballooning technique is used for memory reallocation and VM live migration is used to prevent memory overload of the PM.

Another method to balance memory usage among VMs in a single PM is based on idle memory tax. The idle memory tax rate will increase if the ratio of idle memory to active

memory for the VM rises. Waldspurger [2003] introduced the concepts of idle memory tax, content-based page sharing, and hot I/O page remapping, for VMware ESX Servers. The idle memory tax was proposed to achieve efficient memory utilization while maintaining performance isolation. Moreover, content-based page sharing, and hot I/O page remapping can take advantage of transparent page remapping in order to eliminate redundancy while reducing the overheads of memory page copying.

Zhang et al. [2017b] also designed an automatic memory control system based on idle memory tax for the Xen hypervisor. The control system was based on the ballooning driver of the Xen hypervisor. However, the continuous calculation of the idle memory tax for each VM causes a high CPU overhead.

In addition, memory can be balanced among VMs in a single PM through a prediction-based strategy with periodical memory monitoring, prediction, and reallocation. A memory predictor was developed to estimate the amount of re-claimable memory and additional memory required for reducing VM paging penalty [Zhao et al., 2009]. It consists of two effective memory predictors. One predictor is used to estimate the amount of memory which is available for reclaiming without a notable decrement on performance.

Another predictor is to estimate additional memory which is required for reducing the VM paging penalty. Wang et al. [2016] studied how to predict memory usage based on the working set size of VMs. They used a simulation tool to represent a sequential workload, and an offline profiling tool which can incur huge overhead if used online. Such predictions could be difficult to build in an environment with time-varying workloads. However, an accurate prediction requires computerized profiling, which is difficult to build in an environment with a time-varying workload.

As an alternative to the dynamical re-allocation of memory to the VMs on a single PM, using the memory resources of the host PM as a swap space of the VMs can also implement memory balancing among the VMs. Zhang et al. [2017a] proposed a shared memory swapper framework to improve VM swapping performance, which provided just-in-time performance recovery for memory intensive applications to quickly regain their runtime momentum instead of relying on costly page faults. However, it is not as efficient as using host memory via ballooning.

Apart from the VM-orientated memory balancing mechanisms described above, there are a few application-oriented methods, which measures memory usage for application in order

to precisely allocate sufficient memory resource. Zhou et al. [2004] used the dynamic miss-ratio curve that plots the application's page miss-ratios against varying amounts of memory to measure an application's memory requirement. Hines et al. [2011] introduced a policy framework that determines the application's memory assignments by linear optimization.

Other approaches include improvement of the swap subsystem and joint provisioning. Amit et al. [2014] solved various types of superfluous swap operations, decayed swap file sequentially, and ineffective prefetching decision for the swap subsystem of the KVM hypervisor. Meng et al. [2010] designed a resource provisioning strategy to seek favourable VMs combinations and estimate the size of jointly provisioning VMs.

2.4.2 Disaggregated Memory and Remote Memory Paging

Memory balancing among PMs is more complicated than among VMs in a single PM because each single PM has a fixed amount of physical memory. As the memory resource of a PM cannot be physically moved to another PM, moving memory pages between the main memory and secondary memory is implemented in operating systems to overcome the limitation of the physical memory in PMs. Since a PM's memory resource cannot be reallocated, the mainstream approach is the method of moving memory pages between the main memory and secondary storage. A memory disaggregated PM contains only a limited amount of local memory, while it is able to utilize free memory elsewhere in the same local network over a network connection [Ruan et al., 2020]. This is also called remote memory paging.

Existing efforts have been made toward disaggregated memory for various targets, including the PM, hypervisor, and computation. Extending the concept of secondary memory, the remote memory paging technique has also been developed. It enables a memory disaggregated PM to use free memory of other PMs in the same local network over a network connection [Ruan et al., 2020].

a Physical Machine

To mitigate the memory load of a PM, a partial memory resource from remote PMs can be abstracted as a block device and then used as a local swap device.

Choi et al. [2017] evaluated the performance of using a remote block device as a swap through TCP/IP and RDMA. They found using RDMA can provide an increase in speed of 20 times in read operation and 6 times in write operation faster than using TCP/IP. RDMA-based remote block device had 2 times the speedup compared to HDD for quick-sorting. This study confirmed only the feasibility of using remote block device as swap space.

Srinuan et al. [2020] designed a mechanism of remote memory tracking and swap space management for the use of memory resources from remote PMs. Their approach establishes a memory pool across node PMs in a cluster and lets the process's page table track remote memory page frames, in order to enhance memory aggregation from connected PMs. However, it is designed for the connected server cluster, involving compute nodes and memory nodes, implying that the CPU resources of the memory nodes are wasted.

In addition to one back-end swap device, several frameworks are designed in the form of hybrid swap. For example, the Gu et al. [2017], and Liang et al. [2005] utilized remote memory and hard disk together for enhanced reliability. However, constantly reading/writing on a disk slows down the performance of other I/O operations and consumes network bandwidth if the disk is connected via a network such as iSCSI.

Cao and Liu [2020] designed a hierarchical disaggregated memory orchestration system. The system enables inactive page compression and swap-out optimization for a reduced size of the page. Also, it is embedded with a selection mechanism to choose the most suitable back-end device. The hybrid swap architecture is further enhanced by Newhall et al. [2016] through allowing node RAM, disk, flash SSD, PCM, and network storage devices as swap devices.

All these approaches not only have I/O performance impact, but also demand other resources and consequently increase extra operation costs. For example, writing inactive pages to the local device, such as disk and SSD, can impact I/O performance, while choosing the most suitable back-end device can produce computation overhead.

Such design of multi-back-end devices for swap space is similar to hybrid memory. Tan et al. [2020] presented an adaptive data migration approach for hybrid Non-Volatile Random Access Memory (NVRAM) systems. It involves two components, where one eliminates invalid page migrations and another component only rewrites dirty data back when the page is swapped back to NVRAM.

Liu et al. [2020] proposed object-level memory allocation and a migration mechanism of a hybrid memory system. Their work requires a profiling tool to characterize memory access patterns of objects in all execution phases of applications, and a performance model for static memory allocation at start up and dynamic object migration at runtime. However, application source codes are required to be transformed via static code instrumentation in order to work with the object-level memory allocation and migration mechanism.

Ye et al. [2008] presented an analysis of hypothetical hybrid main memory, which was comprised of a first-level Dynamic Random-Access Memory (DRAM) and a 10-100x slower second-level memory. They found workloads had large performance degradation when run on hybrid main memory systems.

Moreover, Oura et al. [2017] designed multi-thread page-swap protocols to accelerate remote memory enabled swap operations. Based on that, they developed distributed large memory for processing large-size data that exceeds the main memory capacity of the PM. Their work allows the running of multi-thread programs written in OpenMP and POSIX threads for large-scale problems in which the problem data size is larger than the memory resource capacity of the PM for computation. However, the design is for large memory and requires modifications to user programs, increasing the difficulty for end users.

Compress memory [Zhou et al., 2018] [Li et al., 2021] [Roy et al., 2001] [Pekhimenko et al., 2013] [Gupta et al., 2010] can be also utilized to extend PM memory capacity so that the memory load of a PM is mitigated. However, this is not discussed in this literature review because memory compression is not involved in this research.

b Hypervisor

Remote memory paging works not only on multiple PMs, but also on multiple VMs on a single PM. Koh et al. [2019] and Lim et al. [2012] designed hypervisor-oriented frameworks to swap inactive memory pages from VMs to remote memory connected to the host PM. Koh et al. [2019] designed a hypervisor-integrated disaggregated memory system, which could dynamically select optimal block size for each memory region depending on the degrees of spatial locality for different regions of memory in a VM. Lim et al. [2012] extended the Xen hypervisor to emulate the disaggregated memory, where remote pages are swapped into local memory on-demand upon access.

Cao and Liu [2018] further added page compression during memory paging from a VM to the host PM. They presented a shared-memory based memory paging service to improve VM memory swapping performance through redirecting the VM swap traffic to a compressed shared memory swap area; meanwhile the swap page table is compressed to form an efficient index structure in order to improve utilization of the shared memory swap area.

Kocharyan et al. [2020] presented a system to monitor the working sets of VMs and reclaim unused memory of VMs through Xen balloon driver for paging memory from memory overloaded VMs to VMs on other PMs.

Deshpande et al. [2010], Hines and Gopalan [2007] developed a fully transparent distributed system within the Xen hypervisor to coordinate the use of cluster-wide memory resources for large memory and I/O intensive workloads.

In addition, Williams et al. [2011] indicated that using remote memory for VMs requiring extra memory for a sustained period of time was inefficient. They found that it is inefficient to use remote memory for VMs that need extra memory resources for a long duration. For such VMs, they pointed out that live VM migration could be better than remote memory.

Overall, a single PM usually has multiple running VMs, and managing a remote swap device for VMs is more complicated than for PMs.

c Computation

It is worth mentioning that swapping inactive memory pages may not be the most efficient method for some memory intensive computations [Kissel and Swany, 2016]. Distributed in-memory key-value store can be used for applying remote memory resources to a program [Dragojević et al., 2014]. The memory load of a PM can be mitigated if an application stores its runtime data in remote memory.

SpongeFile [Elmeleegy et al., 2014] is a logical byte array comprised of large chunks that can be stored in other PMs. It is designed specifically for distributed MapReduce computation. However, this and other similar methods are designed for special purposes, and thus are not a general solution to the mitigation of the memory load of PMs cloud data centres.

Ahn et al. [2018] designed a virtual shared memory framework, which provided parameter sharing via shared memory between distributed processes in order to improve communication

performance.

Overall, in past two decades, efforts have focused on balancing VM memory on a single PM, and how to design and implement a remote memory paging system. With the emerge of high-performance storage devices, the performance of remote memory paging systems has not yet been compared with NVMe SSD. In addition, there is lack of research into how remote memory paging helps with memory overload challenge.

2.5 Memory Resource Management

In data centres, available memory resources come from multiple cloud servers, and the amount of available memory resource changes over the time. Thus, memory resource management based on memory resource sharing is required for data centres. Memory resource management needs to address two important questions: when to acquire extra memory resource, and which machine should provide how much memory resource to which machine.

In determining when to acquire extra memory resource, Beloglazov and Buyya [2013], Yadav and Rama Krishna [2019], and Patel and Patel [2018] developed threshold-based overload detection mechanisms. A threshold-based mechanism monitors memory usage in real-time. If memory usage goes over a given threshold, it would trigger extra memory resource allocation. However, a threshold-based mechanism might trigger allocation too late if the given threshold was quite close to the capacity. On the other hand, if the given threshold was quite low, it might trigger allocation too early and causing an underutilized resource. In addition, Beloglazov's work required knowing of stationary workload and state configuration, which would not work on unknown workloads that cloud servers usually have. Therefore, an improved threshold-based mechanism or a completely new mechanism is required to determine when to acquire extra memory resource.

In order to determine which machine should provide how much memory resource to which machine, Zhang et al. [2017b] introduced idle memory tax which is adopted from economic theory. The high idle memory tax applied to VMs which had low memory utilization. In contrast, low idle memory tax applied to VMs which had high memory utilization. Thus, balancing idle memory tax determined which should provide memory resource and which could acquire the extra memory resource. Zhao et al. [2009] developed another memory resource

balancer. In addition to monitor memory resource usage, it utilized prediction techniques to predict memory needs, and rebalance memory resource periodically. However, the memory needs of cloud servers could be unpredictable because running processes might be unknown and could not be profiled. In addition, it might be complicated to calculate a rebalancing plan for a huge quantity of machines because these works were designed for single PM which only has a few VMs.

Dynamic bin packing algorithm might be another approach to determine which machine should provide how much memory resource to which machine. In a bin packing problem, items in different sizes must be packed into a limited number of bins while minimizing the number of bins used. Furthermore, in a dynamic bin packing problem, items arrive and depart dynamically. Memory resource management in data centres has a similar problem. In memory resource management, item represents requested remote memory resource, and bin represents available memory resource located in PM. The basic goal of memory resource management is allocating available memory resource to machines which request remote memory resource. Thus, memory resource management could be considered as the dynamic bin packing problem, but the size of bin dynamically changes. A fully dynamic bin packing algorithm might be beneficial for this challenge. Ivkovic and Lloyd [1998], Feldkord et al. [2018], Berndt et al. [2020], and Seiden [2002] designed dynamic bin packing algorithms (e.g., Myopic packing and Harmonic++). These algorithms had similar performance results. However, they all required a few existed bins to be repacked. Johnson [1974] and Li et al. [2016] designed dynamic bin packing algorithms with the goal of minimizing computation time cost. Compared to the work discussed above, their algorithms had a lower cost of computation time but had worse performance. To sum up, an existing dynamic bin packing algorithm could be modified in order to solve the issue. The choice of existing dynamic bin packing algorithms could be according to the focus: faster computation speed or better computation result.

2.6 Optimization for Memory Sharing Management

The occurrence of memory overload is a consequence of an uneven distribution of overall work on cloud resources, which can be solved from various perspectives. A conventional approach to address memory sharing problem is scheduling algorithms, such as the memory scheduling algorithm and the process scheduling algorithm. However, scheduling algorithms may not

achieve the "best" design relative to the memory sharing problem, although they can solve the problem. In fact, memory sharing can also be managed from the optimal perspective. The memory management problem for memory sharing is a discrete optimization problem, in which an object such as an integer, permutation or graph can be found from a countable set.

2.6.1 Bin Packing Problem

The optimization problem for memory sharing management can be considered as the bin packing problem. Memory overloaded PMs require additional memory resources to overcome the memory overload problem. These additional memory resources are remotely shared from spare memory resources of other PMs. Hence, it can be considered as the required additional memory resources are packed into bins, which are spare memory resources of other PMs.

Ant Colony Optimization (ACO) [Dorigo et al., 2006] is one of the most recent techniques for approximate optimization, which is inspired by the ants' foraging behaviour. Ants communicate indirectly by means of chemical pheromone trails, which allows them to find shortest paths between their nest and food sources. This characteristic of the ant colony is exploited in ACO for solving discrete optimization problems.

Brugger et al. [2004] presented a metaheuristic solution method based on ACO for solving one-dimensional bin packing problem. In the proposed metaheuristic solution method, a preprocessing procedure is proposed to reduce the actual problem size faced by the ants for improving the efficiency of the algorithm. A new pheromone decoding scheme and a new pheromone update strategy are also proposed to further improve the performance of the proposed algorithm.

Levine and Ducatelle [2004] presented a pure ACO method for the bin packing problem, and a hybrid approach which contains a local search algorithm for augmentation of the proposed pure ACO method. They found the hybrid approach can have better performance in certain problem classes than the pure ACO method and other existing evolutionary methods.

Particle Swarm Optimization (PSO) [Kennedy and Eberhart, 1995] is a population-based algorithm. It is initialized with a population of candidate solutions. A particle represents as a candidate solution which is associated a randomized velocity and moves through the search space. Each particle keeps track of its coordinates in the search space, which are associated with its best solution or the overall best value all particles have achieved so far. The PSO is

robust and fast in solving nonlinear, non-differentiable, multimodal optimization problems.

Liu et al. [2008] proposed a multi-objective evolutionary PSO algorithm as a solution for the bin packing problem. The solution incorporates the concept of Pareto's optimality to evolve a series of solutions along the trade-off surface. Thus, it does not need a combination of both objectives into a composite scalar weighting function.

Abdul-Minaam et al. [2020] proposed an adaptive procedure to solve the one-dimensional bin packing problem, which involves using of PSO algorithm and fitness-dependent optimizer. The proposed algorithm uses a modified first fit heuristic approach to generate a feasible initial population and adjusts the parameters to search for the final optimal solution.

Genetic algorithm (GA) [Goldberg and Holland, 1988] is a probabilistic search algorithm inspired by the mechanics of natural selection and natural genetics. It starts with a set of solutions called population, where each solution is represented by a chromosome. The population size is preserved throughout all generations. In each generation, fitness of each chromosome is evaluated. According to the fitness values, chromosomes for the next generation are probabilistically selected and randomly mate and produce offspring, where crossover and mutation randomly occurs during produce of the offspring. Since chromosomes with a high fitness value are highly likely to be selected, chromosomes from the new generation may have a higher average fitness value than those from the old generation. The process of evolution is repeated until a given end condition is satisfied.

Dokeroglu and Cosar [2014] proposed scalable hybrid parallel algorithms for solving one-dimensional bin packing problems. The proposed hybrid parallel algorithms combine parallel computation techniques, evolutionary grouping GA metaheuristics, and bin-oriented heuristics to obtain a solution in polynomial running times.

Kröger [1995] presented a sequential and a parallel GA for solving the constrained two-dimensional bin packing problem. The concept of meta-rectangles is proposed and incorporated into the algorithm for temporarily fixing hyperplanes of existing solutions. So that the hierarchical structure of packing schemes with guillotine constraint is exploited to reduce the complexity of the problem without impacting the quality of the solutions generated.

Sridhar et al. [2017] presented a combinational of heuristic GA for solving three-dimensional

bin packing optimization problem, where each bin is arbitrary sized rectangular prismatic container. The optimization goal of the proposed method is to minimize the empty volume inside the container instead of satisfying various practical constraints like box orientation, stack priority, container stability, weight constraint, overlapping constraint, shipment placement constraint.

There are several other optimization algorithms for the bin packing problems, such as whale optimization, simulated annealing, grasshopper optimization, and so on.

Abdel-Basset et al. [2018] presented an improved Lévy-based whale optimization algorithm, as a new variant of the whale optimization algorithm. The proposed algorithm is suitable to search the combinatorial search space of the bin packing problem.

Rao and Iyengar [1994] proposed a bin packing method based on simulated annealing method for several specific scenarios, such as a static task allocation in process scheduling and batch processing. The authors found the quality of the solutions obtained by the stochastic method are consistently stable, while the quality of the solutions obtained by the heuristic methods are problematic-instance dependent which is erratic.

The grasshopper optimization algorithm is a new metaheuristic algorithm for solving the bin packing problem. It is widely used in a variety of industrial scenarios because of easy deployment and high accuracy. However, it has some shortcomings, such as unbalanced processes of exploration and exploitation, unstable convergence speed, and easy to fall into the local optimum. Feng et al. [2020] enhanced grasshopper optimization algorithm by using a nonlinear convergence parameter, niche mechanism, and the β-hill climbing technique to overcome the shortcomings of the grasshopper optimization algorithm.

Martello et al. [2000] defined an exact branch-and-bound algorithm for the three-dimensional bin packing problem by introducing an exact algorithm with incorporation of original approximation algorithms for filling a single bin.

Gupta and Ho [1999] proposed a heuristic algorithm to solve the one-dimensional bin packing problem. The proposed heuristic algorithm is useful for solving the problem which requires most of the bins to be exactly filled. It means the proposed algorithm cannot give an optimal result if the sum of requirements of items is higher than twice the bin capacity.

Overall, the bin packing problem is an NP-hard optimization problem, and many metaheuristic algorithms have been proposed to solve it. The optimization problem for memory sharing management is similar as the one-dimensional bin packing problem, while existing metaheuristic algorithms in solving multi-dimensional bin packing problem may not be necessary for our problem. In addition, Haouari and Serairi [2009] investigated six metaheuristic optimization algorithms for solving the one-dimensional bin packing problem, and found that the GA performed very well and required short CPU times.

2.6.2 Load Balancing

In addition to manage memory sharing by bin packing algorithms, the memory sharing can also be considered as to rebalance overall memory resources among PMs in cloud data centres. Load balancing can ensure that no PM is memory overloaded. It also helps to improve the efficiency of the utilization of resources. Load balancing and better use of resources is considered as the optimization problem and are ensured by numerous available algorithms, such as ACO, PSO, and GA.

Li and Wu [2019] presented a task scheduling algorithm based on ACO for solving the load imbalance problem in System Wide Information Management (SWIM) task scheduling. By using the hardware performance and quality index and load standard deviation function of SWIM resource nodes to update the pheromone for ACO, the proposed algorithm can reduce the task execution time and improve the utilization of SWIM system resources with a more load balanced state.

Another ACO-based task scheduling algorithm for cloud computing was presented by Li et al. [2011]. With consideration of first come first served policy, the proposed algorithm dynamically adapts the scheduling strategy to the changing environment and the types of tasks, so that it can balance the entire system load and minimize the makespan of a given task set.

Xu et al. [2018] presented a VM allocation algorithm based on ACO to achieve multi-dimensional resource load balancing of all PMs in cloud data centres. The proposed algorithm customizes the ACO in the context of VM allocation and introduces an improved PM selection strategy to the basic ACO in order to prevent the premature convergence or falling into the local optima.

Balanced load distribution of workloads among nodes in the cloud environment was presented by Nishant et al. [2012] and Gao and Wu [2015]. The approach proposed by Nishant et al. [2012] detects overloaded and underloaded nodes and thereby performs operations based on the identified nodes. In different from the original ACO approach, which each ant builds their own individual result set and it is then built into a complete solution, the proposed approach lets the ants continuously update a single result set rather than updating their own result set. Gao and Wu [2015]focused more on minimizing the time cost of the dynamic load balancing strategy based on the ACO. To quickly find out the candidate nodes for load balancing, the moving probability of ants, whether the forward ant meets the backward ant in the neighbour node, is defined by the combination of task execution prediction.

Ramezani et al. [2014] and Pan and Chen [2015] proposed task-based system load balancing methods based on PSO to achieve system load balancing in the cloud environment. The method proposed by Ramezani et al. [2014] only transfers extra tasks from an overloaded VM instead of migrating the entire overloaded VM to the new host PM. Pan and Chen [2015] proposed an improved PSO algorithm to establish a corresponding resource-task allocation model with consideration of the characteristics of complex networks because of uncertainty of resource nodes in the cloud computing environment.

PSO can be combined with other metaheuristic algorithms. Visalakshi and Sivanandam [2009] presented a hybrid PSO method, which combines a simulated annealing method, for solving the task assignment problem. The tasks are independent and non-preemptive in nature. The proposed algorithm dynamically schedules heterogeneous tasks on to heterogeneous processors in a distributed setup in order to have a balanced load among the processors. Golchi et al. [2019] presented a hybrid of firefly algorithm and improved PSO algorithms for reaching the better average load and improving the important metrics, such as effective resource utilization and the response time of tasks.

Dasgupta et al. [2013] and Makasarwala and Hazari [2016] proposed GA-based load balancing strategies for balancing the load of workloads among PMs in cloud data centres. They found the GA-based load balancing strategy outperformed the existing approaches, such as First Come First Serve (FCFS), Round Robing (RR) and a local search algorithm Stochastic Hill Climbing (SHC).

Yadav et al. [2021] used a GA to manage incoming traffic distribution over various VMs

for resource allocations. They found that the GA provided better performance in terms of optimizing the resource utilization, while minimizing the average response time and preventing overload on VMs.

Chou et al. [2014] presented a GA-based load balancing system for efficiently distributing large data from clients to different servers, in Software-Defined Networking (SDN) environment. They found GA-based algorithm had better performance than load-based, random, and RR.

An improved GA was proposed by Liu et al. [2017] for balancing the load of VMs. The optimization accuracy of the GA was improved by the improvement of selection and crossover operators in the genetic process. Their improved GA had better load balancing performance than traditional GA and RR.

Parallel GA is an algorithm that uses multiple GAs to solve a single task. Wang and Li [2016] proposed a multi-population GA for task scheduling. They used the min-min and max-min algorithm to initialize the population in order to boost the search efficiency. Metropolis criterion was used to screen the offspring for accepting poor individuals with a certain probability, in order to maintain population diversity and avoid the local optimum. Ashouraei et al. [2018] proposed another parallel GA-based method for scheduling tasks with priorities in order to achieve better SLA. The Parallel GA presented by Effatparvar and Garshasbi [2014] is also for tasks scheduling and load balancing to reduce the total response time and increase the system utilization.

GA can also be combined with other optimization algorithms. Xue et al. [2019] proposed a dynamic load balancing scheme which integrates GA with ACO for further enhancing the performance of software-defined networking. It capitalizes the merit of fast global search of GA and efficient search of an optimal solution of ACO to assign the network traffic to the resources in such a way that no one resource is overloaded and therefore the overall performance is maximized.

In addition to pure metaheuristic optimization algorithms, machine learning and deep learning are also used for optimization of load balancing challenges.

Talaat et al. [2020] presented a dynamic resource allocation method based on reinforcement learning and GA for load balancing of resource utilization in fog computing. The proposed

dynamic resource allocation method can handle the incoming requests, and distributes them between the available servers equally, according to continuous monitored information of network traffic and server load.

Kaur et al. [2020] presented a deep learning-based framework for workflow execution in the cloud environment. Convolutional Neural Networks (CNN) are used to make effective and accurate decisions for resource allocation to the incoming requests, by choosing the most suitable resource to complete the incoming requests. By enabling CNN, the proposed framework uses a deep learning-based technique to obtain the optimal schedule for dynamically provisioning resources for VMs and balancing the load of VMs.

Overall, the GA is initially a discrete technique that is also suitable for combinatorial problems, while the PSO is a continuous technique that is very poorly suited to combinatorial problems if there is no improvement nor combination with other metaheuristics. Moreover, the ACO has been found that it falls into the local optima quickly. Hence, the GA is possible to be used for balancing memory resources among PMs in cloud data centres. It is also proven that having a better performance than RR, especially for complex cases.

2.7 Literature Review Summary

Cloud vendors are now more and more interested in over-committing their computing resources to maximize the resource utilization and minimize operating costs. While over-committing resources has impacts on the performance of VMs, it shows limited performance degradation in general for the end users of cloud services except for memory overload. This chapter presented a literature review investigating handling memory overload PM from various perspectives, including improvements on live VM migration, dynamic memory balancing among VMs hosted on a PM, disaggregated memory, remote memory paging, and memory resource management.

In order to handle memory overload, there are several important issues for live VM migration: how to live migrate a VM, how to predict or detect the memory overload condition, which VM should be migrated, and which PM should accept VM migration. Although there are lots of existing well-designed approaches to address these issues, these can produce additional overhead and memory overloaded PMs. In fact, memory overload lasts for a short duration. If the memory overload occurs for quite a long time, it can be considered as the failure of

VM placement or the over-committing strategy. Thus, a lightweight approach is required for handling memory overload.

Distributed shared memory shows solving the research problem from a completely different perspective. Distributed shared memory addresses physically separated memories as a single shared address space. However, implementation of distributed shared memory is complex and requires modification of existing systems.

Memory sharing becomes the mainstream approach for handling memory overload of VMs by dynamic memory balancing, or handling memory overloaded PMs by disaggregated memory and remote memory paging. Dynamic memory balancing re-allocates memory resources of VMs on the fly. Provisioned memory resource will be reduced if a VM uses a small portion of allocated memory resource, and vice versa. Disaggregated memory and remote memory paging allow a PM to use a memory resource located in a remote PM. Such designs can help to answer the research problem by implementing memory sharing through a method of remote memory paging. However, most of the efforts require dedicated PMs to act as a memory resource provider, meanwhile the space of remote memory paging cannot be dynamically attached or detached.

Literature related to memory resource management shows several directions for designing algorithms for memory sharing. When to trigger the memory sharing can be inspired by a threshold-based mechanism. Moreover, how to select a memory underutilized PM for a memory overloaded PM can be inspired by existing algorithms for solving bin packing problems. In addition, another perspective can be balancing memory utilization among PMs in cloud data centres by memory sharing. The literature shows the GA can provide a better performance than the RR.

Chapter 3

Memory Sharing System

Memory overload is a critical issue in over-committed clouds, which impacts computation workload and service stability. It occurs when a Physical Machine (PM) has memory pressure over a threshold for more than a given duration, such as five minutes [Baset et al., 2012]. Computation workloads on low priority of Quality of Service (QoS) will be paused or terminated to renounce their memory resource, so that workloads with strict QoS requirement can be maintained. Therefore, a memory disaggregation mechanism is demanded to deal with the memory overload problem for over-committing the memory resources of PMs in cloud data centres.

A prevailing measure to handle memory overload PM is the live virtual machine (VM) migration. However, live VM migration is a heavyweight solution, which introduces overheads to both source and destination PMs and migrated VMs. It also occupies network bandwidth and CPU time during migration. In addition, the duration of doing live VM migration may be longer than the duration of memory overload.

In most cases, the duration of memory overload is transient, even at high memory over-committing ratios. For realistic web workloads in a data centre, about 88% of overload events take less than 2 minutes, while 30.6% of the overload events only last 10 seconds or shorter [Williams et al., 2011]. Thus, memory overload could be classified into two types: transient overload and sustained overload. VM live migration is suitable for sustained overload, while it is too heavyweight for transient overload. As most memory overload events are transient, a lightweight approach with more flexible use and reuse of existing hardware memory resources is preferable for handling memory overload of PMs.

With the emerge of high-speed network devices and technologies, such as Remote Direct Memory Access (RDMA) and InfiniBand, a PM can reduce its memory load by remote memory paging with another PM. Transmitting a message in the traditional socket-based model requires copy operations through the kernel space and consumes CPU time for data transfers. In contrast, RDMA is a method of reading data from, and writing data to, the memory device of a remote machine without involving the network software stack and kernel, which eliminates context switch, intermediate data copies in various stack, and protocol processing in contrast to the traditional model [Guo et al., 2016]. It is carried by InfiniBand, which is a networking technology that provides extremely high data throughput and low latency, particularly in high-performance and in-network computing environments [Xue and Zhu, 2021]. Recent research shows RDMA and InfiniBand networking have been utilized for disaggregated memory in over-committed clouds. A memory overloaded PM can swap memory pages out of its physical memory to the physical memory of a dedicated or underutilized PM.

Memory sharing, as a complementary solution to live VM migration, is also called memory disaggregation. The principle of memory sharing is to offload part of memory pressure to a remote PM. The memory sharing system is designed for handling memory overload of physical machines (PMs) in the cloud data centre. The feasibility of swap-based memory sharing is disscussed in the previous chapter. However, how to establish a connection between the two PMs is still a challenge, especially when to connect and disconnect a remote in-memory block device and how two PMs communicate to establish such connection.

In this chapter, a memory sharing system, which integrates a mechanism of threshold-based memory overload detection, is presented for handling memory overload of InfiniBand networked PMs in data centres. It enables a PM with memory overload to automatically borrow memory from a remote PM with spare memory, thus handling the memory overload problem. Overall, the main contributions of this chapter include:

1) A memory sharing system architecture is presented for handling memory overload of PMs in cloud data centres;

2) A unified control algorithm is designed for a PM to automatically borrow memory when memory overloading and lend spare memory when feasible; and

3) The memory sharing system is physically implemented and experimentally evaluated in

terms of its functionality and read/write performance.

The measured performance of our memory sharing system in read/write speed is similar to that of accessing a local Non-Volatile Memory Express Solid-State Drive (NVMe SSD).

The rest of the chapter is organized as follows: Section 3.1 describes the technical gaps and motivation. Then, Section 3.2 exhibits how RDMA operates. Section 3.3 shows the designs of the memory sharing procedure and feasibility examination. The architecture of our memory sharing system is presented in Section 3.4. Section 3.5 shows the design of the control algorithm for memory borrowing and lending. This is followed by system implementation in Section 3.6. Section 3.7 describes the conduct of experiments for performance evaluation. Finally, Section 3.8 concludes the chapter.

3.1 Technical Gaps and Motivation

Memory balancing and remote memory are two major candidate approaches for handling memory overload of PMs in cloud data centres. However, memory balancing requires the coexistence of over- and under-committed computation tasks in the same PM. The total memory requested from all VMs on the PM is capped by the memory capacity of the PM. In comparison, remote memory for PMs does not have such a limitation. However, existing efforts in remote memory for PMs require a pre-configuration of disaggregated memory on dedicated PMs. Furthermore, multiple or hybrid disaggregated memory designs are not essential for handling memory overload for PMs in cloud data centres.

Therefore, technical gaps exist regarding our memory sharing requirements for PMs in cloud data centres, i.e., 1) a PM can access remote memory resources when it becomes memory overload; and 2) when a PM has the spare memory resource, it can dynamically share out its memory resource to a remote PM. This motivates the research and the development of this chapter on memory sharing for handling memory overload on PMs in cloud data centres.

3.2 Operations of Remote Direct Memory Access

RDMA over InfiniBand uses Queue Pair (QP) to transfer data messages. Each communication endpoint needs to create an QP in order to talk to each other. Each QP is constrained by a Send

Queue (SQ) which handles outbound data transfer and a Receive Queue (RQ) which handles inbound data transfer.

RDMA over InfiniBand supports three operation modes: send/receive operation, read/write operation, and atomic operations. The send/receive operation is a bilateral operation, where the receiver must be involved in order to complete the operation. In practice, the send/receive operation is mostly used to transmit control messages, while data messages are mostly transmitted through read/write operation. The read/write operation, on the other hand, is a one side operation, where the receiver is not involved during the operation. Moreover, atomic operations are atomic extensions to RDMA operations. These operations include atomic fetch/add and compare/swap. The atomic fetch/add operation increments value of a specified virtual address with a given value. The atomic compare/swap operation compares value of a specified virtual address with a given value, and swap value if they are not equal.

Mellanox mentions in the user manual [Mellanox Technologies, 2015] that Mellanox InfiniBand supports three RDMA transport modes: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD), as shown in Table 3.1. The Reliable Datagram is not supported.

Table 3.1: Transport Modes of Queue Pair

Operation	Unreliable Datagram (UD)	Unreliable Connection (UC)	Reliable Connection (RC)
Send/Receive	✓	✓	✓
RDMA Write		✓	✓
RDMA Read			✓
Atomic operations			✓
Reliability			✓
Multicast	✓		
Max message size	MTU	1 GB	1 GB

In these three transport modes, RC supports all operation types which include send/receive, write/read, and atomic operations. RC is a reliable connection while the other two modes are not. UC supports send/receive and RDMA write, while UD only supports send/receive. In addition, UD is a multicast communication with a maximum message size the same as a maximum transmission unit. UC and RC have one gigabyte of maximum message size limit.

3.2.1 Common Prerequisites for RDMA Operations

There are a few prerequisites for transmitting data through RDMA. Firstly, both server and client side are required to register a Memory Region (MR) which tells the kernel that a segment of the memory address is reserved for RDMA communication and creates a channel from InfiniBand to MR. Secondly, the server and the client need to resolve addresses of each other. Address resolution methods include assigning IP addresses to server and client and connecting via IP address or discovering and connecting via InfiniBand specific protocols.

3.2.2 Send/Receive Procedure

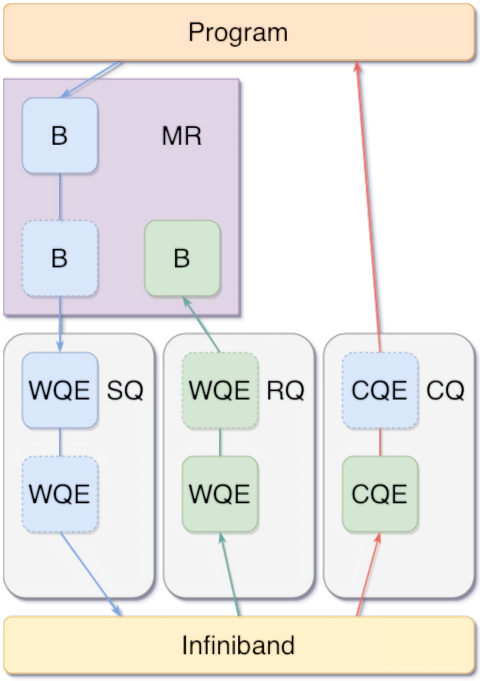


Figure 3.1: RDMA Send/Receive

Figure 3.1 shows the procedure of RDMA send/receive. Both server and client need to establish a QP and a Complete Queue (CQ). QP is required to handle inbound/outbound data flow by InfiniBand. CQ is used to inform the program when a send/receive operation is completed.

The Work Queue Element (WQE) is placed in the send queue and receive queue. It describes types of operation (either send or receive) and the memory address of the buffer. A WQE placed in the sender's send queue contains the memory address of the buffer that needs to be sent from the MR to the client. Whereas a WQE placed in the receiver's receive queue contains the memory address of arriving buffer in MR. Once a transaction (either send or receive) is

completed, a Completion Queue Element (CQE) will be placed on CQ.

3.2.3 Write/Read Procedure

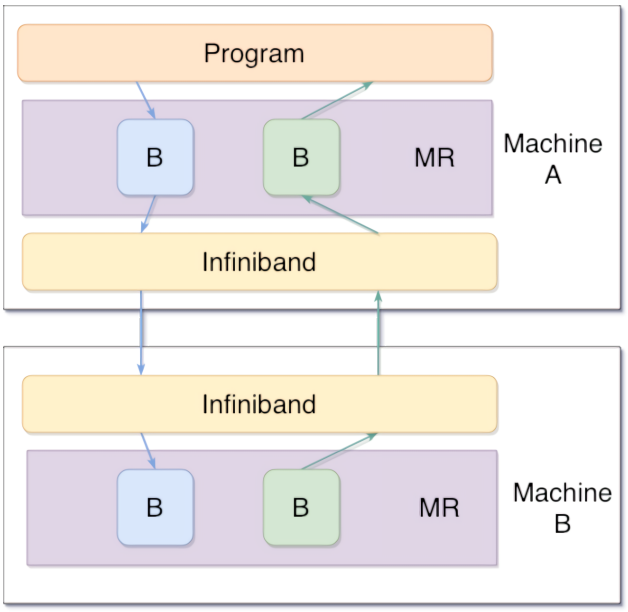


Figure 3.2: RDMA Write/Read

The Write/Read Procedure, on the other hand, is straightforward. The program just needs to ask InfiniBand to copy a buffer located in a machine to another machine. Figure 3.2 describes RDMA Write/Read Procedure where Machine A is the sender and Machine B is the receiver. Performing the RDMA write operation requires the program to tell the following information to InfiniBand in Machine A: operation type (either write or read), buffer address in Machine A, buffer address in Machine B, and key of MR in Machine B. Key of MR is used to identify a specific MR because there might be more than one MR in Machine B. If it is a read operation, InfiniBand will copy the buffer from the given memory address of a given MR in Machine B to give a buffer address in Machine A. Whereas, the buffer in Machine A will be copied to a given location in Machine B if it is RDMA write operation.

In practice, the buffer address and key of MR in the receiver machine is not fixed. RDMA send/receive is usually used for exchanging the buffer address and the key between sender and receiver in order to make the RDMA write/read be operable.

3.3 Design of Memory Sharing Procedure and Feasibility Study

The operations of RDMA are validated in our existing workstations by implementing and evaluating a fundamental memory sharing model. Based on several key findings from validation of RDMA operations, a procedure of one-to-one memory sharing is designed, and its feasibility is investigated in this section.

3.3.1 Fundamental Memory Sharing

A fundamental memory sharing model with various RDMA workloads is implemented for feasibility examination in order to confirm whether InfiniBand can manage remote memory resources and is the same as what it is expected. Potential issues have been investigated by the feasibility examination and are considered in the algorithm and mechanism design introduced in the rest chapters of this thesis.

In the feasibility examination, it is confirmed that the buffer address and key of the MR can be exchanged between a server and a client through RDMA send/receive operation, and the RDMA write/read operation can be performed to exchange given data. The examination starts from registering MR with a given size and establish QP in both server and client. Then, the server listens on a random port in order to be connected with a client. When a client connects to the server, they send both the buffer address and key of MR to another through RDMA send/receive, followed by writing some data to remote memory. After that, they both read and print received data in standard output.

Several of RDMA operation's behaviours in various unusual situations are also examined. Four key points are found, which need to be attended to in algorithm and mechanism design, as shown below:

1) A WQE must be placed in RQ before placing a WQE in SQ. It means RDMA receive on receiver side must be performed prior to RDMA send on sender side. Otherwise, the RDMA send would fail.

2) RDMA write/read allows the local MR and remote MR to have different sizes. It cannot transmit a buffer which has a larger size than the local MR or remote MR.

3) One machine can register multiple MRs.

4) Registering MRs takes time. The larger size of MR results in longer duration of MR registration. Moreover, the duration of releasing MR is not affected by size of MR.

3.3.2 One-to-One Memory Sharing

Feasibility examination of one-to-one memory sharing confirms that a PM can use the memory resource of a remote PM as its swap space in order to handle memory overloading, while the fundamental memory sharing described in the above section only confirms how RDMA sends and reads data.

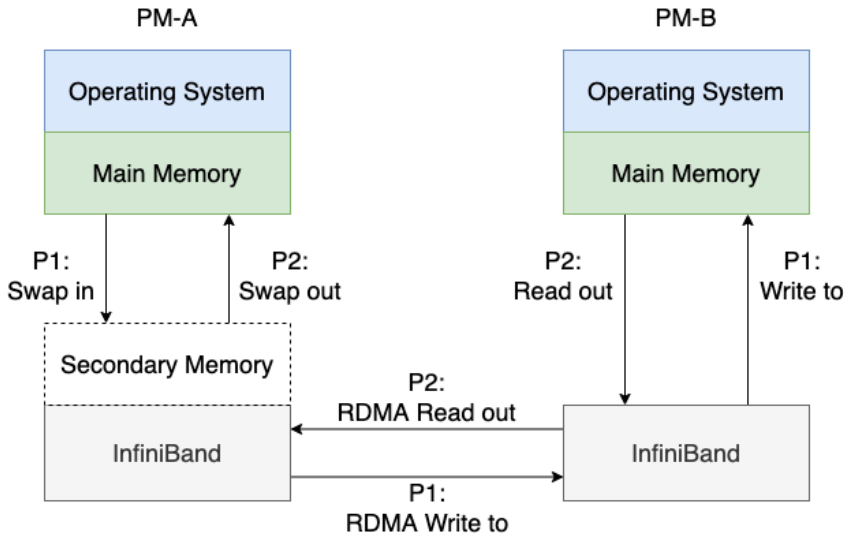


Figure 3.3: One-to-One Memory Sharing

One-to-one memory sharing examination is shown in Figure 3.3. PM-A represents the PM which requires memory resource from another PM. PM-B represents the PM which shares out its spare memory resource.

The data transfer protocol for exchanging data between PM-A and PM-B is based on NVMe over Fabric (NoF). NoF is a protocol for transferring storage commands of Non-Volatile Memory Express (NVMe) block device between client nodes and target nodes over InfiniBand or Ethernet networks through RDMA [Guz et al., 2017]. It standardizes the wired data transfer process and hardware drivers for efficient access over RDMA-capable networks with minimal processing required by the target node.

The procedure of one-to-one memory sharing is described below: At first, a ram disk block

device provided by Linux kernel is created on PM-B. Then, the NoF subsystem is set up on PM-A, where its backing store block device is a ram disk block device. Now PM-A can connect the subsystems on PM-B through NoF. A new block device will show up on $/dev$ path. Finally, PM-A activates the new block device that just showed up as a secondary memory device, which is swap space.

PM-A can use the memory resource of PM-B as its swap space now. Figure 3.3 shows how a memory page P1 is swapped from PM-A into PM-B, and memory page P2 from PM-B to PM-A. In addition, examination of the behaviours of secondary memory indicates that the operating system may take a quite long time to swap in memory pages from a remote PM to the main memory. By default, the operating system's kernel iterates over the swapped pages for searching every swapped out page, which can take a quite long time depends on the number of swapped pages. It is recommended to iterate over processes in order to recall swapped pages to the main memory.

3.4 System Architecture of One-to-one Memory Sharing

This section describes our memory sharing system architecture. Depending on the memory utilization, a PM may require additional memory from other PMs, or has spare memory to share out to other PMs. Therefore, each PM may play a role at same time as either a memory borrower or a memory lender. A PM that is using remote memory resources for handling memory overload is a borrower PM. In contrast, a PM that is providing its memory resource for a remote PM to use is referred to as a lender PM. In general, a PM may change its role from a borrower to lender or vice versa over time.

Our memory sharing system is shown in Figure 3.4. It is composed of three main components, that is, a controller, a virtual block device, and a data transfer protocol:

1) The controller is implemented as a user space application. It decides when remote memory is required.

2) The virtual block device is embedded in a kernel space module. It stores inactive memory pages exchanged from the memory device.

3) The data transfer protocol is designed for command execution and data transfer. It defines

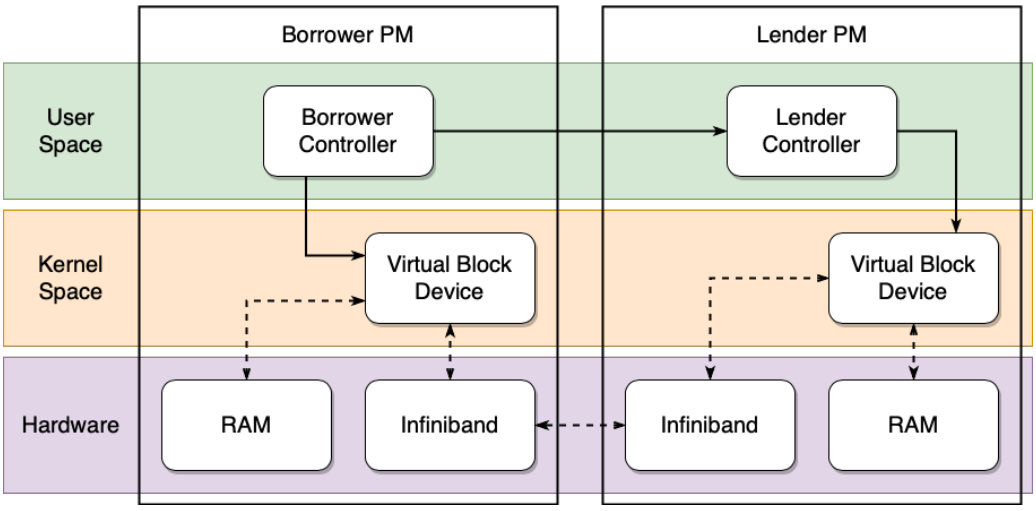


Figure 3.4: System Architecture of Memory Sharing

how the memory pages are transferred between the borrower and lender PMs.

As shown in Figure 3.4, the memory sharing requires a cooperation within the user space application (Controller), the kernel space module (Virtual Block Device), and the data transfer protocol. The solid arrows in the figure represent program command execution, while the dotted line arrows indicate data transfer routes. The hardware directly involved in the system operation includes the memory on each PM and InfiniBand-internetworked PMs.

3.4.1 Controller

A controller is designed for each PM that participates in memory sharing. It monitors the system memory information and manages the virtual block device of the PM. Logically, a PM may borrow memory from, or lend memory to, another PM. Therefore, the controller on the PM acts as either a borrower controller or a lender controller, but not both, at a time. If a PM is required to borrow memory from a remote PM that has spare memory to share out, the controller on this PM becomes a borrower controller, while the controller on the remote PM becomes a lender controller. With the change of the memory status of the PM, the controller on a PM may change its role from a borrower controller to a lender controller or vice versa.

For a borrower controller running on a borrower PM, if the used memory of the PM reaches a given threshold, the borrower controller communicates and exchanges memory sharing information with a lender PM for establishing memory sharing, followed by activating the virtual block device, which is linked to a lender PM through the data transfer protocol. Then, it enables

swap space so that the kernel exchanges inactive memory pages from the physical memory to the virtual block device. If the borrower PM is no longer considered as being overloaded, the borrower controller disables swap and detaches the virtual block device of remote memory.

The lender controller is activated passively by a memory sharing request. When handling a request for sharing memory from a borrower PM, it creates a virtual block device and reserves sufficient memory resources for sharing. If the virtual block device is disconnected from the borrower PM, the lender controller will clean up the virtual block device and free up the memory space that was shared out previously.

3.4.2 Virtual Block Device

The virtual block device is a logical storage interface on a borrower PM or a physical storage disk on a lender PM. While logically it is used like a hard disk, it behaves differently on borrower and lender PMs.

On a borrower PM, the virtual block device abstracts a hard disk to the operating system. It performs as a logical interface to accept block I/O requests. These block I/O requests are then passed through to a lender PM via InfiniBand networking.

On a lender PM, the virtual block device abstracts an in-memory hard disk that stores sectors in RAM. As memory is allocated only when requested, the amount of memory the virtual block device occupies changes over time depending on how much data needs to be stored. For example, on a virtual block device of 2 GiB, only 1 GiB memory will be consumed if only 1 GiB data needs to be stored. When the stored data is deleted from the virtual block device, the virtual block device will free up this previously occupied block of memory.

3.4.3 Data Transfer Protocol

The data transfer protocol for memory sharing is developed based on NVMe over Fabric (NoF) from existing systems. NVMe is designed to work over a Peripheral Component Interconnect Express (PCIe) bus. Legacy storage stacks for accessing a storage device over the network could be used to operate NVMe devices. However, the requirements of synchronizations and command translation largely offset the benefits of NVMe devices for remote access.

NoF is a protocol for transferring NVMe storage commands between client nodes and target

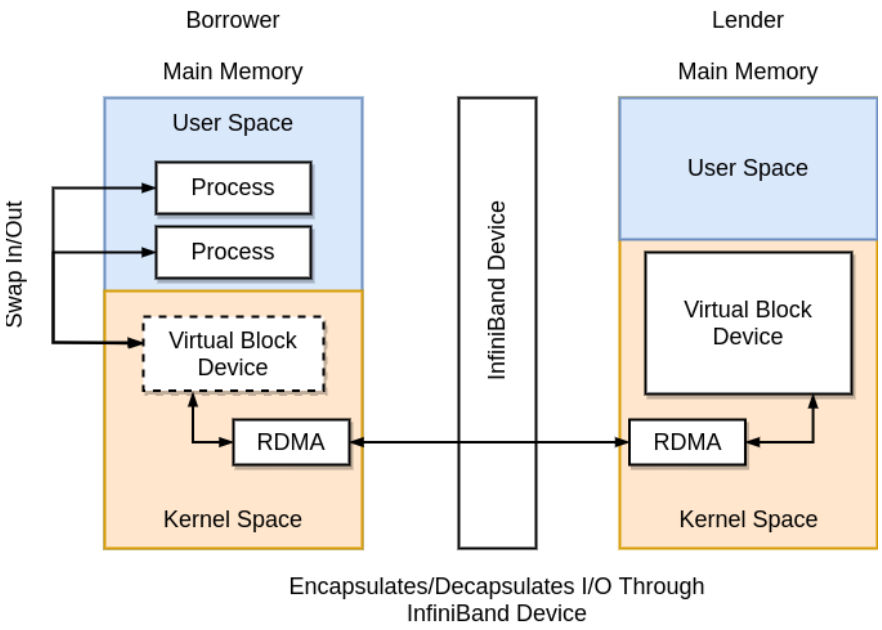


Figure 3.5: System Swap Diagram in Our Memory Sharing System

nodes over InfiniBand or Ethernet networks through RDMA [Guz et al., 2017]. It standardizes the wired data transfer process and hardware drivers for efficient access over RDMA-capable networks with minimal processing required by the target node.

The data transfer protocol together with the virtual block device in our memory sharing system is shown in the logical diagram of Figure 3.5. It encapsulates a block I/O request, sends the request to the target node through RDMA, decapsulates the block I/O request, and finally passes the block I/O request to the storage virtual block device. This enables efficient data transfer from a borrower PM to a lender PM and vice versa.

3.4.4 State Machine

With the design of the dual controller role described previously in section 3.4.1, the controller has three states: borrower, lender, and neutral. It stays in the Neutral state when the PM neither borrows nor lends memory resource. Transitions of these three states are demonstrated in state machine shown in Figure 3.6. They are discussed below.

When a PM boot up, it runs under a low level of workload. The controller stays at the Neutral state with no need to borrow memory from other PMs. Meanwhile, the PM has not shared out any spare memory resource yet to other PMs.

Then, the controller may stay at the Neutral state if it does not need to borrow memory from

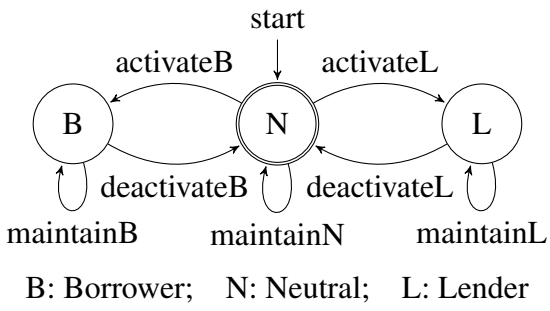


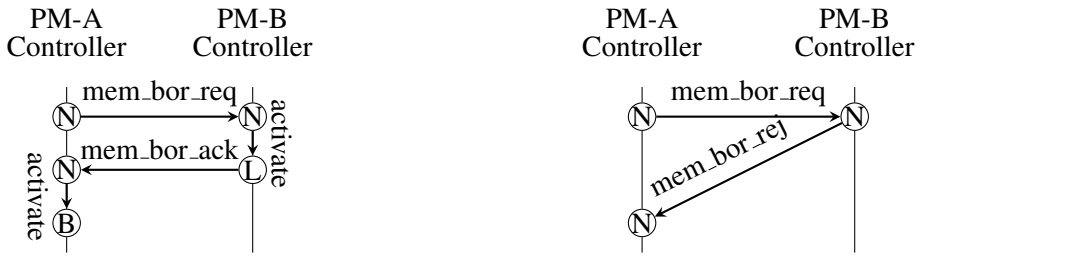
Figure 3.6: State Machine of the Memory Sharing System

other PMs or there is nowhere to borrow memory from. The controller may transit from the Neutral state to the Borrower state if it requires memory from other PMs, or to the Lender state if it is requested memory by other PMs and has spare memory to share out.

For the Borrower state, there are only two possible transitions: to either stay as itself or back to the Neutral state. If memory borrowing is maintained, the controller stays in the Borrower state. Otherwise, if memory borrowing is no longer needed or the lending PM cancels its memory sharing, the controller deactivates the memory borrowing and switches back to the Neutral state.

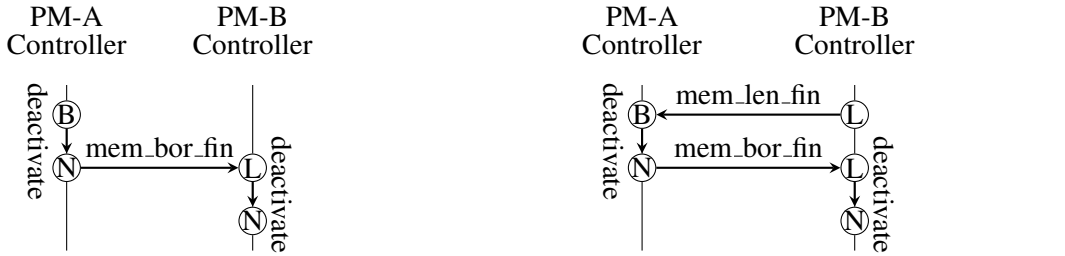
Similar to the Borrower state, the Lender state also has two possible transitions: to either stay as itself or back to the Neutral state. If the memory lending is maintained, the controller stays in the Lender state. Otherwise, if memory lending becomes infeasible due to the lack of spare memory or the borrowing PM terminates its memory borrowing, the lender controller deactivates memory lending and transits back to the Neutral state.

In the state machine of Figure 3.6, the transitions of the states occur periodically rather than instantly. They require the operations of all components in the system architecture on the same PM. They also need the cooperation of the controllers on both borrower PM and lender PM for memory sharing between the two PMs. This will be achieved through the design of signalling, as will be discussed later. Moreover, it is seen from the state machine Figure 3.6) that there are no direct transitions between the Borrower state and Lender state.



(a) Activation Successful (b) Activation Failed: No Enough Spare Memory

Figure 3.7: Signalling for Memory Activation



(a) Deactivation Success: Called by Borrower (b) Deactivation Success: Called by Lender

Figure 3.8: Signalling for Memory Deactivation

3.4.5 Signalling of Events

For cooperation of various components in a controller on a single PM or two controllers on two PMs, signalling events become important in the architectural design of the memory sharing system. In terms of the intended functions, there are generally two types of signalling: signalling for memory sharing activation and signalling for memory sharing deactivation.

There are two scenarios of memory sharing activation, as shown in Figure 3.7. Normally, lender PM-B has to prepare its lender virtual block device and change its state to lender before creating the borrower's virtual block device and establishing a memory sharing connection on borrower PM-A (Figure 3.7a). However, lender PM-B may reject memory sharing if it does not have enough spare memory resource for lending (Figure 3.7b).

Memory deactivation can be triggered by either borrower PM-A or lender PM-B. If memory overloading no longer exists, borrower PM-A will disconnect the remote memory resource before notifying lender PM-B for cleaning up of the lender virtual block device and reverting to Neutral state (Figure 3.8a). In a similar way, if lender PM-B becomes memory overloaded, and no longer has spare memory resource for sharing, it will request borrower PM-A to stop borrowing memory (Figure 3.8b).

Algorithm 1: Controller Algorithm

```
   Initialization: Set StateFlag ← Neutral
1  for Every monitoring period do
2      if StateFlag is Neutral then
3          Execute Algorithm 2 for Neural state;
4      else if StateFlag is Borrower then
5          Execute Algorithm 3 for Borrower state;
6      else
           // StateFlag is Lender
7          Execute Algorithm 4 for Lender state;
8  return
```

3.5 Memory Sharing Control Algorithms

This section describes how the controller on a borrower PM determines to activate memory sharing and how the controller on a lender PM responds. The pseudocode of controller algorithm is shown in Algorithm 1.

A PM needs to have the ability to dynamically switch its role between a borrower PM and a lender PM, as well as keep in the Neutral state. Algorithm 1 describes a unified flag-based controller for all memory sharing operations. In each period cycle, the controller decides the next state flag depending on its memory utilization and memory sharing request.

There are three flags, Neutral, Borrower, and Lender, which correspond with the three states described in Section 3.4.4. The controller with the Neutral flag decides if the PM can maintain Neutral flag or needs transition to the other flag: either the borrower controller or lender controller flag (lines 2-3). For the Borrower and Lender state, the controller decides if the PM can maintain the current state or needs revert back to the Neutral state.

Activation or Deactivation of the memory borrower and lender occurs on state transition. When transiting from neutral to other states, the memory borrower or lender is activated. On the other hand, the memory borrower or lender is deactivated when transiting to Neutral. This is decided by a controller with the borrower flag (lines 4-5) and lender flag (lines 6-7). Hence, the Borrower and Lender state must be transited to Neutral state prior to switching the lender or borrower.

Algorithm 2: Controller Algorithm for Neutral State

Data : $StateFlag$, M_{req}, M_s, M_t, M_u, S_t, T_{upper}
Output : $StateFlag$
Initialization: Set $StateFlag \leftarrow Neutral$

```
1  if Transition to Neutral itself, i.e., no request or request infeasible (M_req > M_s), then
2      if request infeasible, i.e., M_req > M_s, then
3          Reject it by sending response mem_bor_rej;
4      if memory borrowing needed, i.e., M_u > T_upper × M_t, then
5          Send memory request mem_bor_req;
6  else if Transition to Borrower state, i.e., mem_bor_ack received, then
7      Activate memory Borrower;
8      StateFlag ← Borrower;
9  else
       // Transiiton to Lender state
10     Activate memory Lender;
11     Send response mem_bor_ack;
12     StateFlag ← Lender;
13 return StateFlag;
```

3.5.1 Algorithm for Neutral state

The controller algorithm for the Neutral state is shown in Algorithm 2. The default flag (maintain Neutral state) will be returned if there is no Application Programming Interface (API) request received or the requested remote memory resource cannot be fulfilled (lines 1-5). During the stay of Neutral, current memory usage is examined against the threshold in order to detect memory overloading (lines 4-5). A request for borrowing memory resource mem_bor_req is sent if the threshold is reached.

State transitions are described in lines 6-12. Receiving mem_bor_ack results in memory borrower activation (line 6-8). Memory lender is activated only if requested memory resource m_{req} is smaller than memory reserved for sharing m_s (lines 9-12). Otherwise, borrower's request is rejected (lines 2-3). Calculation of m_s will be explained in Section 3.5.3.

3.5.2 Algorithm for Borrower Controller

Activation and deactivation of the borrower controller is determined by a dual threshold mechanism in order to avoid faulty or frequent activation of memory sharing. The first threshold is an alarm threshold T_{upper}. In our memory sharing system, a PM is considered as memory

Algorithm 3: Controller Algorithm for Borrower State

Data : $StateFlag$, M_u, M_t,S_u, T_{lower}
Output : $StateFlag$
Initialization: Set $StateFlag \leftarrow Borrower$
1 **if** *memory borrowing still needed (i.e.,* $M_u + S_u >= T_{lower} \times M_t$*)* ***and*** *feasible (i.e., no* mem_len_fin *received)* **then**
2 Do nothing (i.e., maintain Borrower stater);
3 **else**
4 Deactivate memory borrower;
5 Send mem_bor_fin;
6 $StateFlag \leftarrow Neutral$;
7 **return** $StateFlag$;

Algorithm 4: Controller Algorithm for Lender State

Data : M_t, M_u, $StateFlag$, T_{upper}
Output : $StateFlag$
Initialization: Set $StateFlag \leftarrow Lender$
1 **if** *memory lending not terminated by Borrower, i.e., no* mem_bor_fin *received,* **then**
`// Maintain Lender state`
2 **if** *lending likely becomes infeasible (*$M_u > T_{upper} \times M_t$*)* **then**
3 Send mem_len_fin to prepare deactivation;
4 **else**
5 Deactivate memory Lender;
6 $StateFlag \leftarrow Neutral$;
7 **return** $StateFlag$;

overloaded if its memory usage is beyond the alarm threshold T_{upper}. As realistic memory usage may fluctuate around T_{upper}, another threshold T_{lower} is set to decide whether memory sharing can be deactivated.

The algorithm for the borrower controller decides whether the memory borrower should be maintained. It only involves the second threshold part of the dual threshold mechanism, while the first threshold part is involved in line 4 of Algorithm 3. Borrower controller is maintained if memory usage is still higher than threshold T_{lower} and no termination request mem_len_fin is received (line 1).

Deactivation of memory borrower requires the sum of used memory M_u and used swap space S_u be smaller than threshold T_{lower} because memory pages in swap space will be fetched back to main memory during deactivation of the memory borrower. In addition, request mem_lend_fin may be sent from the lender PM to indicate that the lender PM is unable to keep sharing memory remotely. After deactivation of the memory borrower, it is necessary to notify the memory

lender by sending a mem_bor_fin request.

3.5.3 Algorithm for Lender Controller

This section describes how to calculate the size of the shareable memory resource and when the memory lender can be deactivated.

The remote memory resource can be reserved via a pre-configuration or dynamical configuration. The capacity of the shareable memory resource, M_s, can be calculated from either a pre-configured method or a dynamic method. The pre-configured method derives M_s from the total memory M_t less the reserved memory for all other applications, M_r, i.e.,

$$M_s = M_t - M_r \tag{3.1}$$

The dynamical configuration requires M_u to be known in advance. The dynamic method calculates M_s from the total memory M_t less used memory M_u, i.e.,

$$M_s = M_t - M_u \tag{3.2}$$

With this method, M_s needs to be updated periodically with Eq. (3.2) because M_u changes over time.

The shareable memory resource M_s is examined against the requested memory resource in order to determine whether the lender PM has sufficient spare memory resource for sharing. It is different from how much memory has been shared out. When the lender PM shares its memory resource to a borrower PM, it only reserves and shares out the requested amount of the memory resource sent by the borrower PM.

Deactivation of the memory lender depends on whether request mem_bor_fin is received (line 1), as shown in Algorithm 4. Request mem_bor_fin sent by borrower PM triggers deactivation of the memory lender. However, the controller checks whether the lender PM becomes memory overloaded in the process of maintaining the Lender state (line 2). If the lender PM becomes memory overloaded, the lender controller will send a request mem_len_fin to the borrower PM for termination of memory sharing. The borrower PM has to suspend or migrate running workloads and pull back swapped memory pages in order to complete the

termination of memory sharing.

3.6 Memory Sharing System Implementation

All components of the proposed memory sharing system are deployed on each of the PMs in a cloud data centre though not all of them need to execute at the same time. Their implementations are shown in Fig. 3.9 from the programming perspective. Overall, the controller of our memory sharing system is implemented as a user space program because it requires communications between PMs. Network communications must be managed by a kernel based Netfilter. The virtual block device is implemented as a kernel module interacting with other kernel functions. Moreover, our memory sharing system is implemented in Go and C languages, where the controller is implemented in Go and the virtual block device in implement in C.

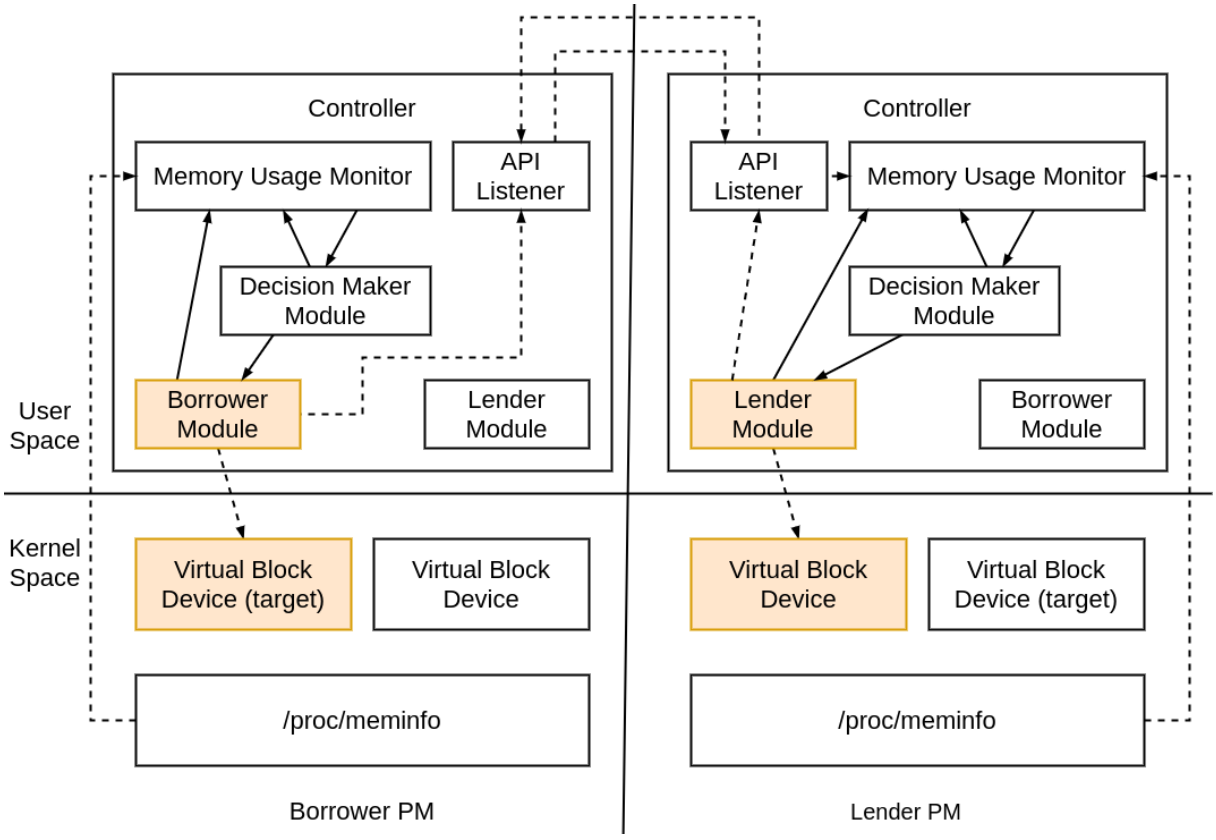


Figure 3.9: Implementation of memory sharing in user and kernel spaces on each PM. Solid arrows represent state transition between four modules: memory usage monitor, decision maker module, borrower module, and lender module. Dashed arrows represent input for triggering state transition.

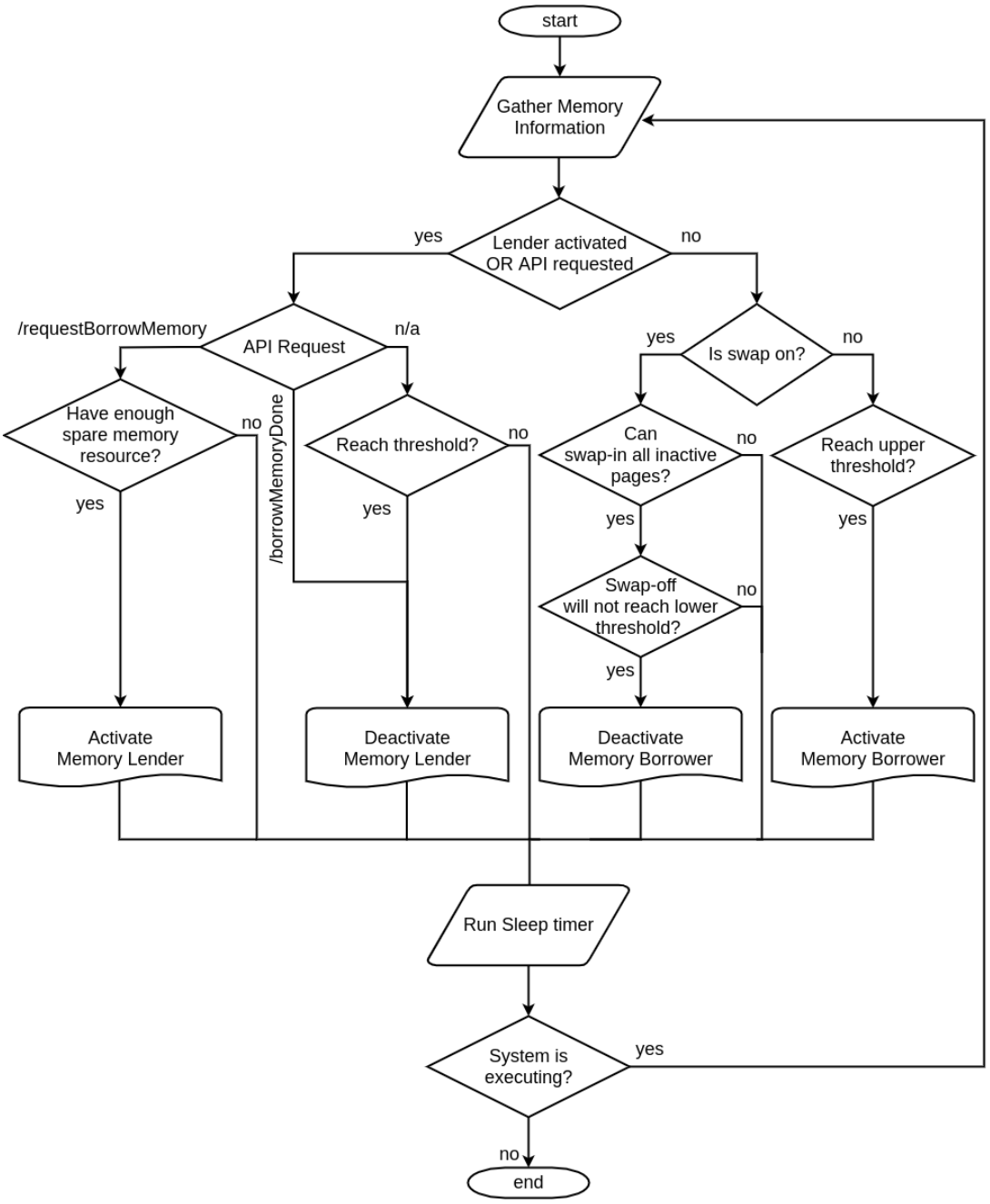


Figure 3.10: Controller Flowchart

3.6.1 Controller Implementation

The controller is implemented in a modular program with several bundled modules including a memory usage monitor, an API listener, a decision maker module, a borrower module, and a lender module. The memory usage monitor and API listener run concurrently in separate lightweight threads managed by the Go runtime.

They dynamically activate/deactivate the borrower module or lender module, as shown in the orange blocks in Figure 3.9. The solid arrow represents the state transition between

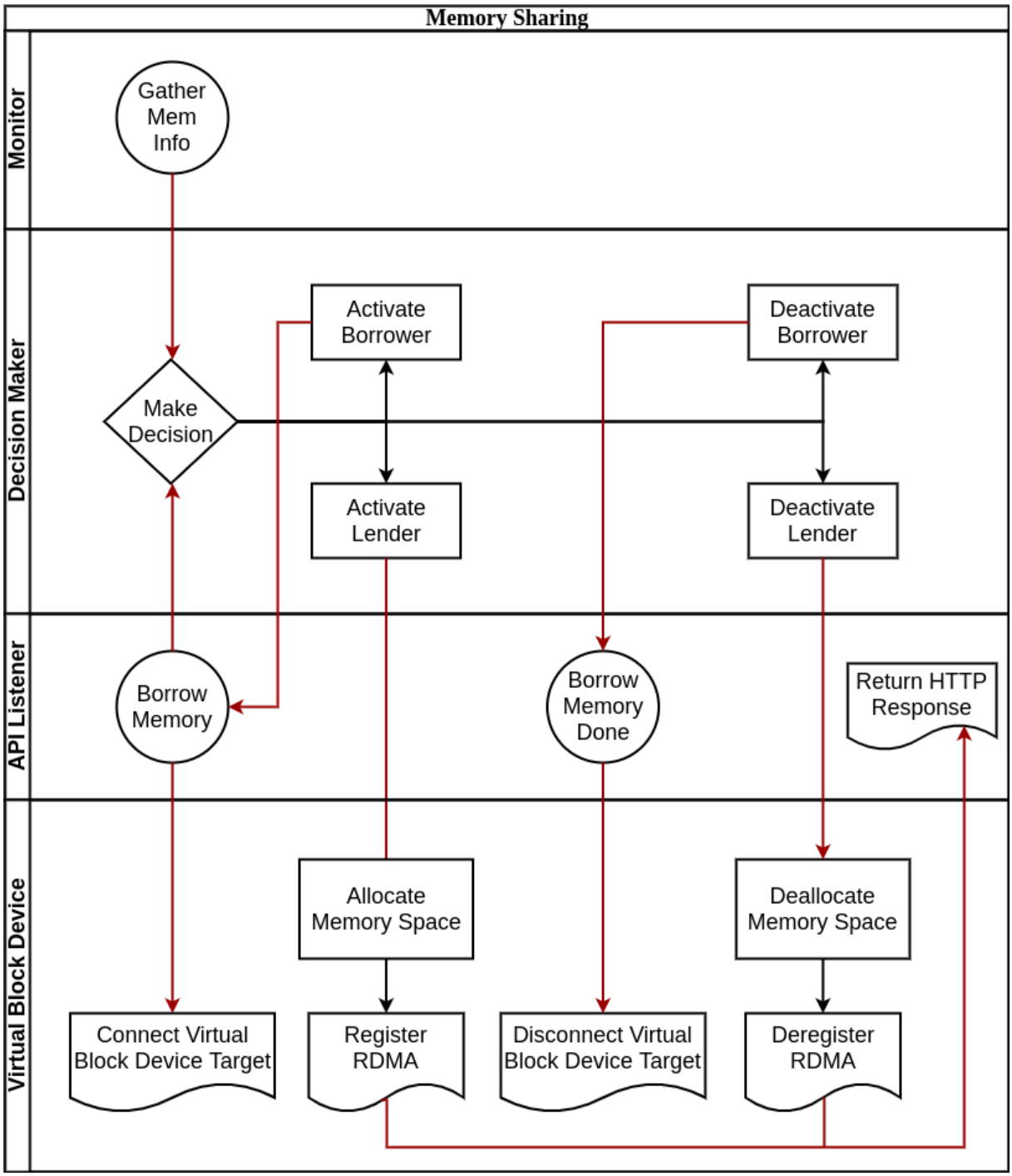


Figure 3.11: System Logic in Swim-Lane Diagram

four modules: memory usage monitor, decision maker module, borrower module, and lender module. The dashed arrow represents input for triggering the state transition. After collecting memory information, the state is transited from memory usage monitor to decision maker module for deciding if the borrower or lender needs to be activated or do nothing. Such logic is presented as a flowchart in Figure 3.10, where the memory usage monitor keeps periodically gathering memory information until the controller exits. Moreover, activation of the lender

module depends on events from the API listener. It is activated only if the decision maker module finds the lender PM can fulfill the requirement of the memory sharing request sent by the borrower PM.

More detailed operations of the controller and the relationships of the controller modules and Virtual Block Device are depicted in a system logic diagram in Figure 3.11. The red line arrow represents the final action of each system component. Apart from the controller, implementation of other system components will be described in subsections below.

Memory Usage Monitor. This module tracks the system memory usage and passing the information to decision maker module. It obtains the memory information of the PM periodically via a low-level system call to *kernel/proc/meminfo*.

API Listener. This module listens on controller APIs which are served and requested through HTTP methods. There are five API methods: *mem_bor_req* and *mem_bor_fin* are sent by the borrower PM; *mem_bor_ack*, *mem_bor_rej*, and *mem_len_fin* are sent by the lender PM.

Decision Maker Module. The dual-threshold mechanism is implemented in this module to decide if the borrower module or lender module needs to be activated. Activation of the borrower module is put into account if the passed memory information comes from the memory usage monitor only, while the activation decision for the lender module is made if the passed memory information additionally contains the requested memory resource M_{req} received from API listener.

Borrower Module. The commands *mem_bor_req* and *mem_bor_fin* are sent out by the borrower module for borrow operations of the remote memory resource. The command *mem_bor_req* contains a parameter M_{req}, which defines how much remote memory resource is required. The lender's return (*mem_bor_ack*) has a parameter NoF Qualified Name (NQN), which is used to identify the virtual block device on the lender PM. Once NQN is obtained from the lender PM, the borrower module will use the NVMe-cli tool with NQN to connect the remote virtual block device, and then set up and enable the swap device. If memory overload no longer exists, the borrower module will remove the swap device and then issue a *mem_bor_fin* command to notify lender PM. In addition, if *mem_len_fin* is received, the borrower module will move remote paging to local swap device and then issue a *mem_bor_fin* command.

Lender Module. When this module is activated, it configures the virtual block device with

the size of remote memory, sets the virtual block device address and NQN for NoF subsystem, and returns *mem_bor_ack* with NQN. *mem_bor_rej* may be returned for being unable to share memory. In response to *mem_bor_fin*, the lender module asks the virtual block device module to clean up the memory previously shared out. In addition, the lender module communicates with the virtual block device through Netlink. Netlink (*AF_NETLINK*) is used to transfer information between the kernel and user-space processes. It consists of a standard sockets-based interface for user space processes and internal kernel APIs for kernel modules. Moreover, if the lender PM becomes memory overloaded, the lender module will send a *mem_len_fin* command to the borrower PM to request a halt to memory sharing.

3.6.2 Implementation of Virtual Block Device

In the implementation of our memory sharing system, the virtual block device as a logical interface on a borrower PM is generated by NoF. On a lender PM, the virtual block device is implemented as a physical storage. Its logical structure is shown in Figure 3.12. In Figure 3.12, the yellow-shaded blocks represent three main data structures of the virtual block device: *netlink_kernel_cfg* (netlink), *gendisk*, and *radix_tree_root* (radix tree). They are discussed below.

The data structure netlink_kernel_cfg (Netlink). Netlink is served for listening requests from the lender module. The request in a Netlink message is to either configure gendisk (*set_capacity()*, *add_disk()*, etc.) or clean up the radix tree (*radix_tree_delete()*).

Data structure gendisk. A block device is defined by gendisk, which describes how to handle block I/O requests. *block_device_operation* only needs the following structure data of the block device: heads, sectors, and cylinders. An alternative *make_request()* function for the block device is needed to define for *request_queue*. *make_request()* performs data read and write operations through calling *radix_tree_lookup()* and *radix_tree_insert()*, respectively.

Data structure radix_tree_root. Blocks are stored in the data structure of the radix tree provided by the Linux kernel library <*linux/radix-tree.h*>.

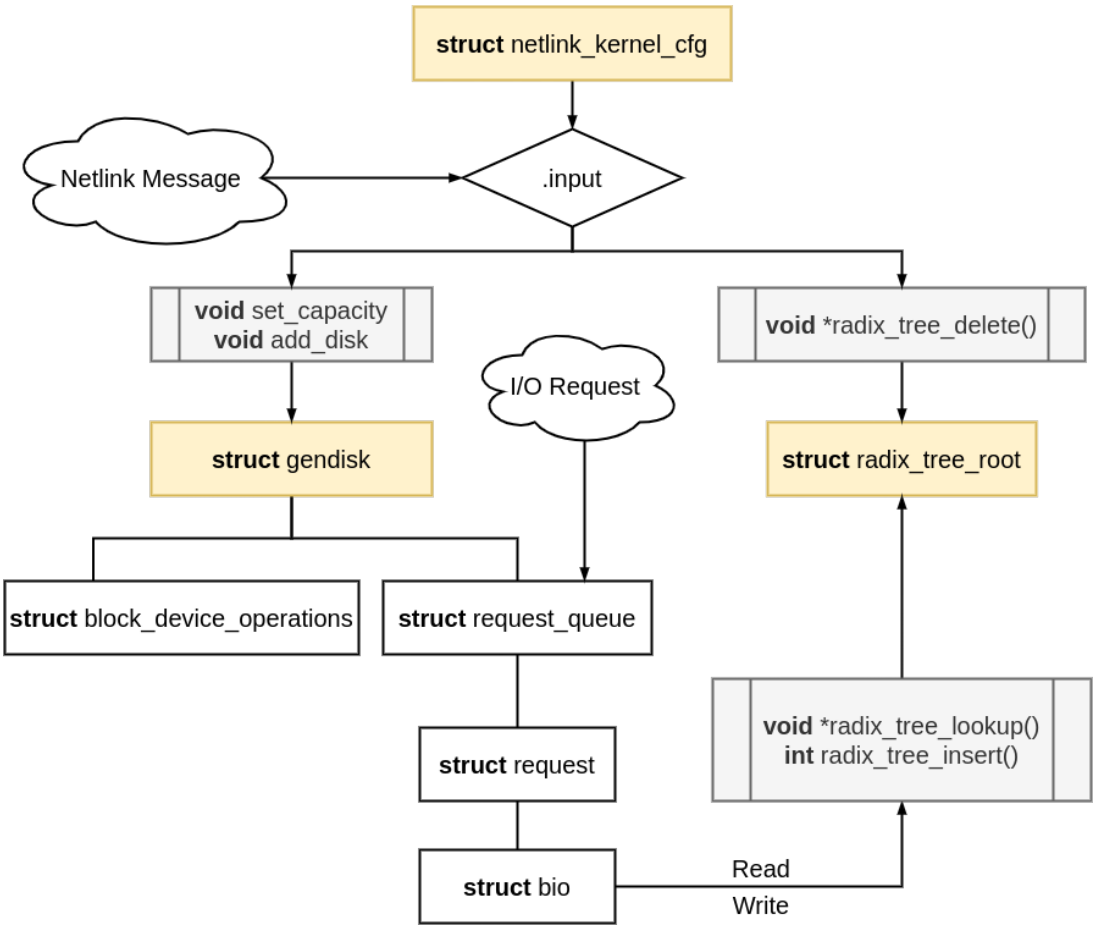


Figure 3.12: The Kernel Module of a Virtual Block Device as a Storage on a Lender PM

3.7 Experimental Evaluation

This section reports the experiments to verify the feasibility of the memory sharing system presented in this chapter for handling memory overload. It will show that remote memory can be attached and detached dynamically on the fly. It will also demonstrate that the read/write performance of remote memory operations is acceptable. The experiments aim to verify that memory sharing could have reasonable performance by comparing the performance of remote memory with the fastest possible local storage device under the simulation of memory page swapping.

3.7.1 Feature Comparisons with Existing Solutions

There are two essential feature requirements for memory sharing in cloud data centres. Firstly, memory sharing should be transparent to process, which allows a PM, or a running process to use a remote memory resource without modification to the program source code. Otherwise,

Table 3.2: Comparisons of Our Method in This Chapter With Existing Solutions

	Virtual block device Based	Transparent to process	Dynamic PM role
AIFM Ruan et al. [2020]			
DLM Oura et al. [2017]			
COMEX Srinuan et al. [2020]		✓	
XMemPod Cao and Liu [2020]		✓	
SMB Ahn et al. [2018]	✓		
RMBD Choi et al. [2017]	✓	✓	
Infiniswap Gu et al. [2017]	✓	✓	
HPBD Liang et al. [2005]	✓	✓	
Nswap2L Newhall et al. [2016]	✓	✓	
Our Framework	✓	✓	✓

the virtualization environment, such as VM and container, cannot benefit from memory sharing. Secondly, a PM should be able to dynamically share out its memory resource or use a remote memory resource because any PM in cloud data centres can be memory overloading or have spare memory resource.

Feature comparison with existing solutions is conducted based on how the solution is implemented, and the two requirements described above. Result is shown in Table 3.2. Most of the solutions are virtual block device based and are transparent to process by design. However, all of them are statically memory sharing, which pre-configures the PM for sharing out memory and the PM for accessing memory resource on a remote PM.

3.7.2 Memory Sharing Behaviour

We monitor memory information changes from start to end of PM overloading. A mock application is implemented and deployed to overload the PM. It keeps requesting memory allocation and filling up the allocated memory with random characters, until 1.5 times of memory resource capacity is used. Figure 3.13 exhibits the lender and borrower system in runtime, where y-axis represents the size of memory usage and x-axis represents time. It is conducted by running a mock application to overload the memory resource of PM A and PM B sequentially. The memory shortfall (swap) on the PM with Borrower role is filled up by the same amount of RAM from the PM with Lender role. The memory borrowing procedure is triggered at around

20 seconds, at which time swap is enabled for utilization. The memory usage growth from Figure 3.13b confirms the usage of remote memory in the lender system. The growth trend of swap in the borrower PM (i.e., the pink area in Figure 3.13a) is the same as the growth trend of RAM in the lender PM (i.e., the blue area in Figure 3.13b). This means that the memory shortfall on the borrower PM is filled up by the same amount of RAM from the lender PM though our memory sharing system.

The PM roles change after 2 minutes of this experiment. Borrower PM no longer has memory overload at the 124 second, while lender PM is pressed by the mock application for memory overloading. Lender PM, shown in Figure 3.13b, becomes memory overloaded and starts borrowing memory at the 147 second. It borrows memory resource from PM of Figure 3.13a. At this time, the PM roles of lender and borrower are switched.

3.7.3 Benchmark Applications

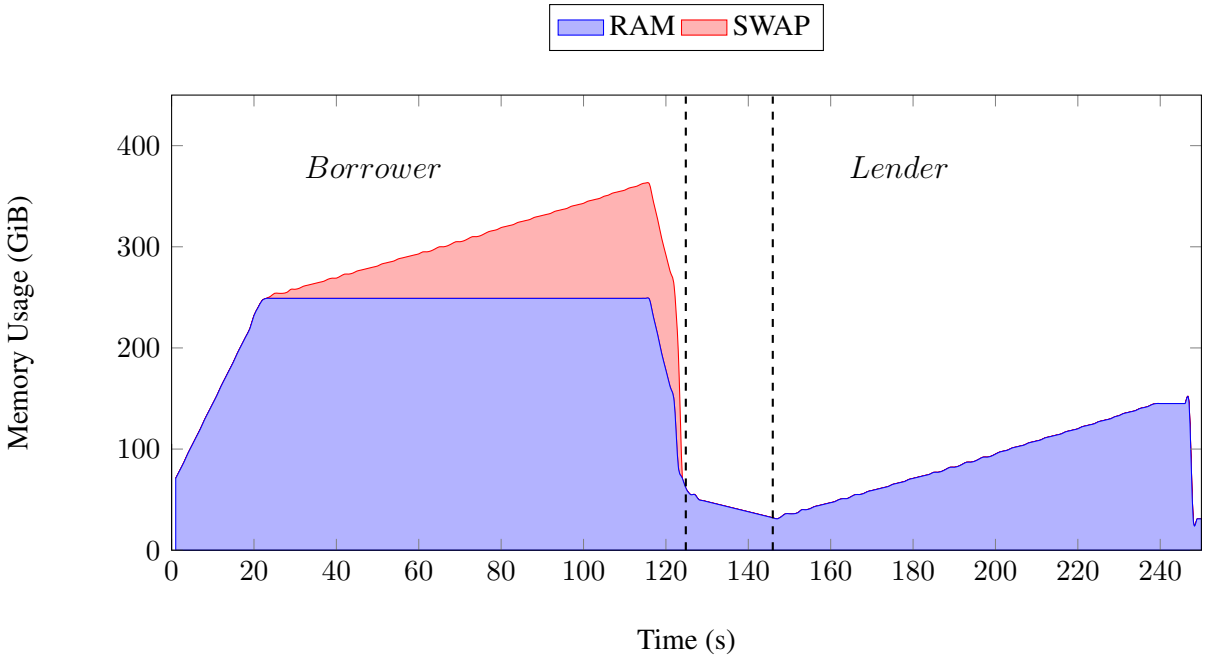
To evaluate the performance of our memory sharing system, we adopt two types of benchmarks: IOzone and DaCapo. These benchmarks are described in the following:

Benchmark IOzone. IOzone (version 3.490) is a filesystem benchmark tool, which generates and measures a variety of file operations [Che et al., 2008]. In our tests, 4 KiB buffer random read and write operations are selected for gaining theoretical performance on exchanging inactive memory pages.

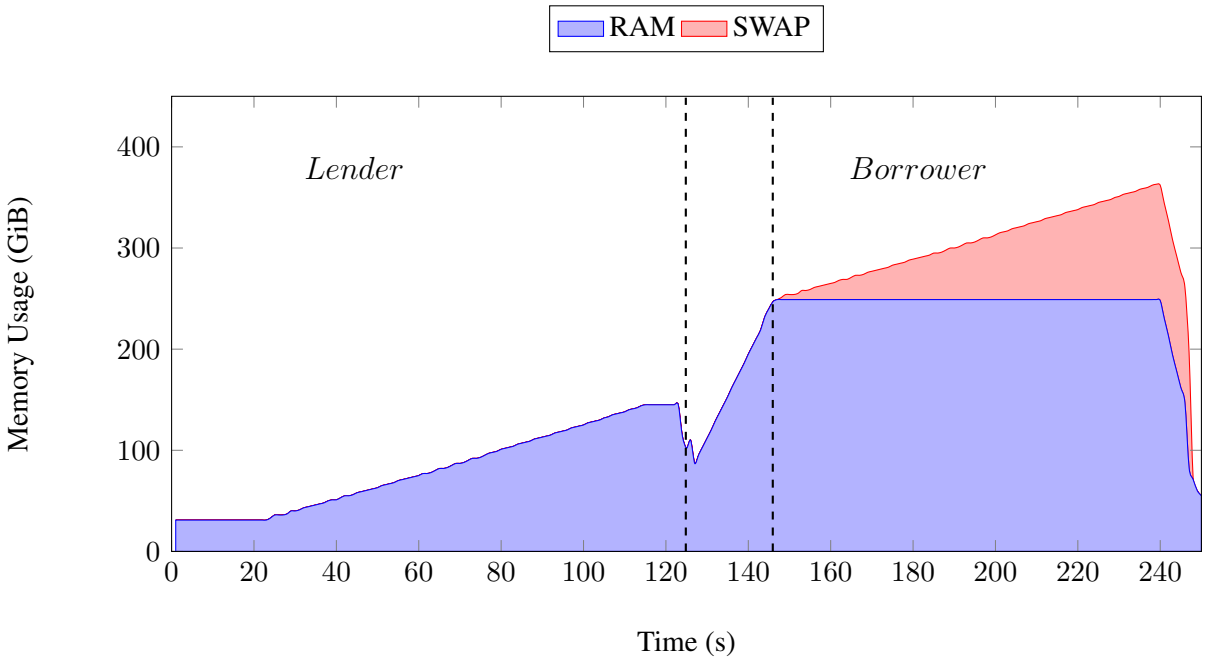
Benchmark DaCapo. DaCapo (version 9.12) [Blackburn et al., 2006] is a Java benchmark suite used to measure the performance of memory management and computer architecture communities. It consists of a set of 14 open-source, real-world applications with non-trivial memory loads. For our tests, we select three memory-intensive applications from DaCapo applications, i.e., h2, tradebeans, and tradesoap.

3.7.4 Experimental Setup for Performance Evaluation

In our experiments, two InfiniBand-networked workstations are used: one as a borrower PM and the other as a lender PM. Each of them has a 24-core Intel Xeon 5118 processor, 256 GiB 2666MHz Hynix Memory, Samsung NVMe SSD PM981 (270,000 IOPS Max on 4 KiB random read, 420,000 IOPS Max on 4 KiB random write), and Mellanox ConnectX-3 Pro



(a) PM A



(b) PM B

Figure 3.13: Experimental Results of Memory Usage Over Time

56GbE adapter card that supports RDMA. The OS used is Red Hat Enterprise Linux 7.7 64-bit, whose kernel is Linux 3.10.0.

Table 3.3: Experimental Setup for Performance Evaluation

	Bare-metal	Docker	KVM/QEMU VM
Application	IOzone	DaCapo (tradesoap, tradebeans, h2)	
Swap Source	Remote	NVMe, Disabled, Remote	

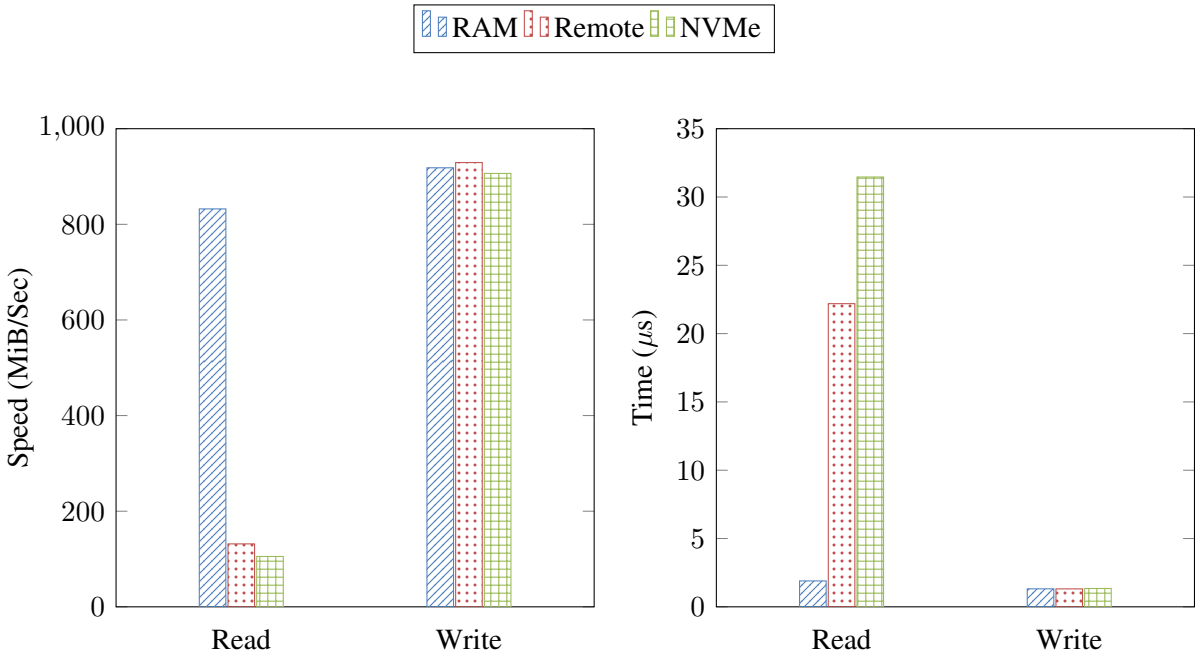
The setup for the theoretical performance experiments and practical experiments are detailed in Table 3.3, where Remote represents using our memory sharing system, NVMe represents using a local NVMe SSD as a swap device, and Disabled represents using system memory only. IOzone is used to test theoretical performance, while three applications from the DaCapo suite are used to test practical performance on two popular virtualization environments, Docker and KVM-based VM.

The three applications selected from the DaCapo benchmark applications are executed in container and VM environments, respectively. Thus, Docker (version 19.03.8) and Kernel-based Virtual Machine (KVM) are deployed in the two workstations. The performance of our memory sharing system is measured for both the Docker-based and KVM-based system configurations. Moreover, performance evaluation is conducted by comparing our memory sharing system with local NVMe SSD based swap device and local RAM only.

Docker is utilized for emulating memory-critical scenarios where available memory resources are limited to each application and thus the application is forced to use remote memory in most of its running time. The Docker image is Red Hat Universal Base Images (version 8.1), which involves a Java SE Runtime Environment (build 14). Each container instance can access 512 MiB memory and all PM swap space.

KVM is utilized for emulating memory-flexible scenarios in a VM-based cloud environment, which in our design will use remote memory only when the hosting PM becomes memory-overloaded. Two VMs are deployed on a PM, where one VM runs benchmark suite applications, and the other runs mock applications that will overload the PM. The guest OS of the VMs is Fedora 33 64-bit, whose kernel is Linux 5.10.19. It involves OpenJDK Runtime Environment 20.9 (build 15.0.2). Each of the VMs is assigned 12 CPUs and 250 GiB memory because each KVM-based VM functions as a Linux process where the Linux kernel of the host PM allocates

memory only when requested.



(a) Throughput (b) Latency

Figure 3.14: The performance of random read and write operations on the 4K-buffer block I/O benchmark.

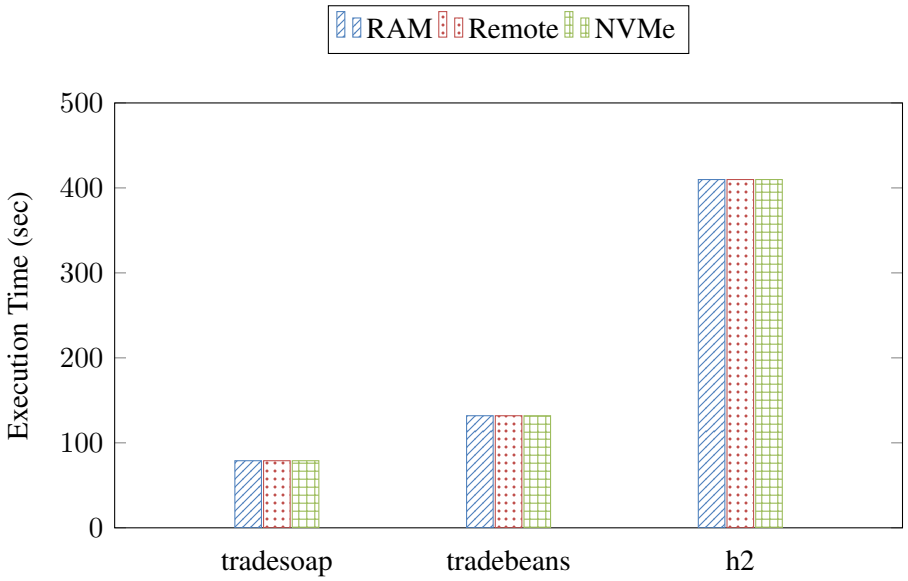
3.7.5 Theoretical Performance

We aim to verify that memory sharing could have a reasonable performance by using IOzone benchmark tool to test remote memory as a dedicated filesystem. In our experiments, remote memory is manually attached as virtual block device. 4 KiB buffer random read and write operations are tested on the attached block device. Figure 3.14 shows the throughput and latency results of 4 KiB buffer random read and write operations.

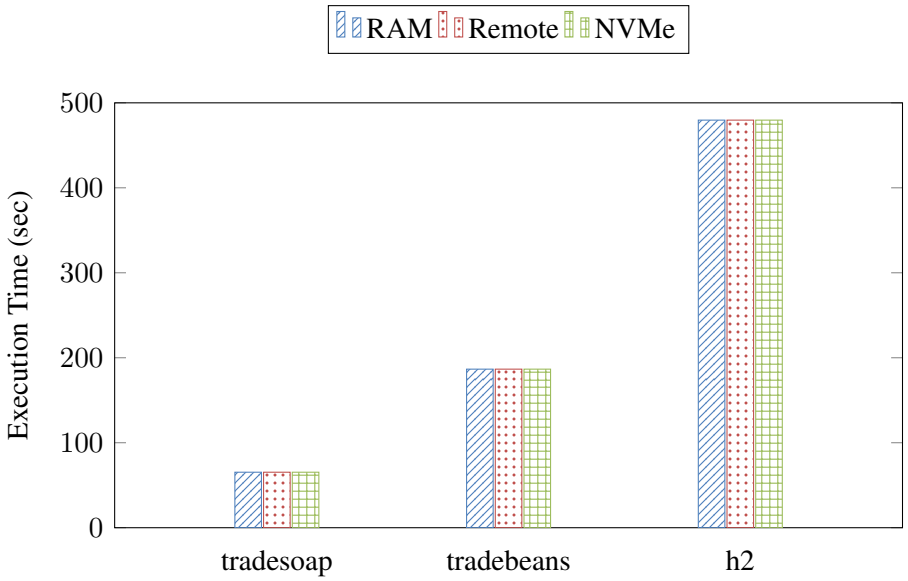
Our experiments show that remote memory always outperforms local NVMe SSD in both throughput and latency. In comparison with RAM, remote memory has a slightly higher speed than RAM on random write but is much slower than RAM on random read. This is also confirmed on operation latency. Random read on remote memory is slow because read operations end when a buffer has been put in a given destination memory address. However, write operations on remote memory are marked as being completed when the buffer is sitting in the RDMA working queue for directly writing to remote memory.

3.7.6 Memory Sharing on The DaCapo Benchmark Suite

Three applications, tradesoap, tradebeans, and h2 are selected from the DaCapo benchmark suite to examine the performance of remote memory. It is expected that remote memory could achieve a similar performance as the local NVMe SSD which is currently one of the fastest storage devices available.



(a) DaCapo Applications in Docker Environment



(b) DaCapo Applications in KVM/QEMU Environment

Figure 3.15: Comparisons of the execution time performance averaged from 10 runs for DaCapo applications (tradesoap, tradebeans, h2) in Docker and KVM/QEMU environments.

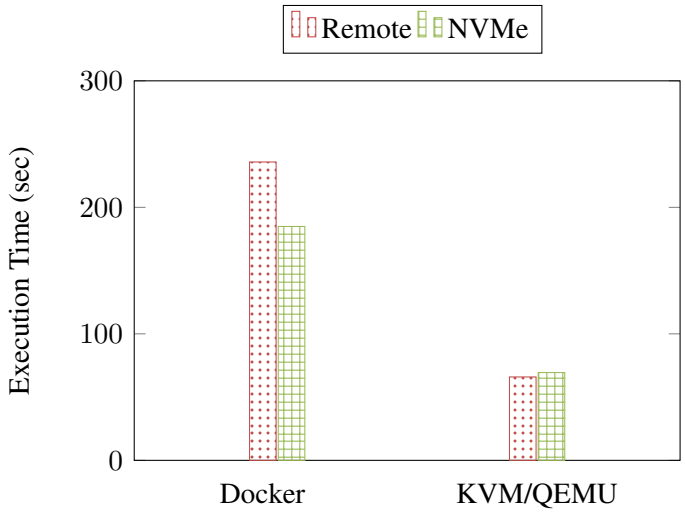
Figure 3.15 shows the results of running selected applications without memory overload. In this case, neither remote memory nor NVMe SSD based swap space is used because the

host system is not yet out of memory. Thus, remote memory is not triggered, and application performance is identical.

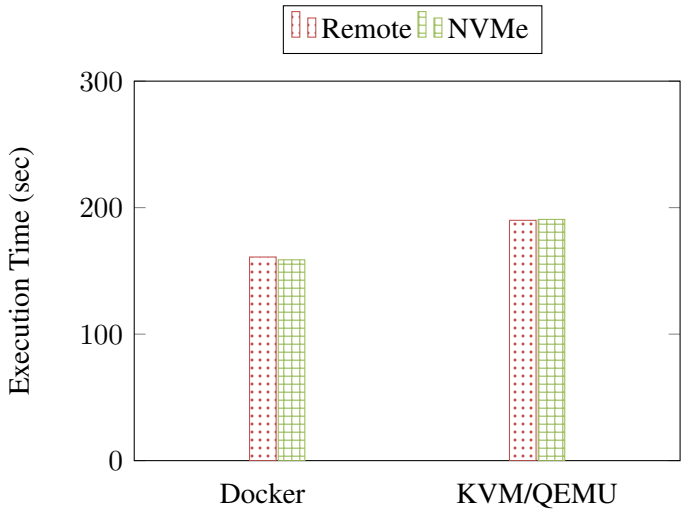
Figure 3.16 shows the results of running selected applications during system overloading. Docker represents running applications in container instance which simulates critical situation, while KVM/QEMU represents running applications in KVM-based VM which simulates practical situation. In simulation of a critical situation, container instance has limitation of 512 MiB memory and no limitation on swap space. Thus, most benchmark applications' data are exchanged to remote memory. In a practical situation, there is no limitation for VMs. PM is put into overloading status before running selected applications. Therefore, inactive memory pages of other system processes may be exchanged to remote memory in VM experiments. In addition, applications are killed (out-of-memory) by the operating system kernel if the remote memory or NVMe SSD based swap is disabled. Thus, there is no performance result for RAM only case.

The results indicate that remote memory has similar performance as local NVMe SSD based swap. Remote memory occasionally has better or worse performance than local NVMe SSD based swap, depending on how an application manages its data in memory. The NVMe SSD based swap has advantages on exchanging large consistent buffers by design, since it can read and write buffers from a clump of blocks. On the other hand, the remote memory has to read or write buffers one by one because InfiniBand sequentially handles the RDMA operations. Moreover, the remote memory has slight advantages on exchanging tiny and scattered buffers, as shown in Figure 3.14, because RAM is truly random access.

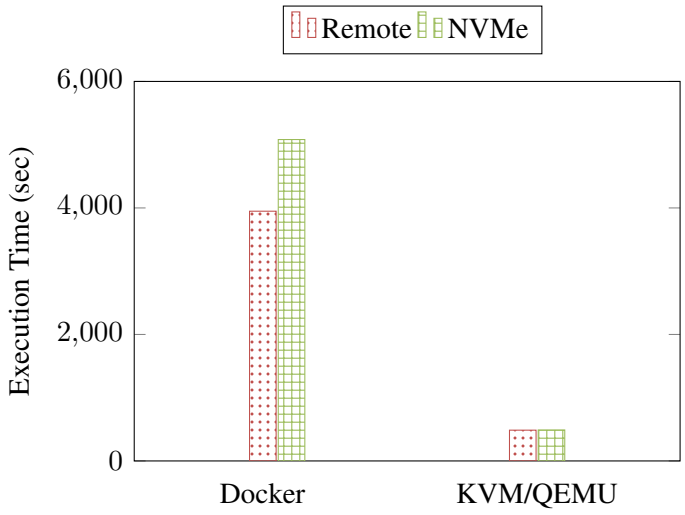
In cloud data centres, the remote memory can have both cost and performance advantages over swapping to local disks. Memory sharing reuses spare memory resources located in the remote PM, whilst swapping to local disks requires a dedicated disk space for swapping. In a typical setting of cloud data centers, disks are remotely attached via NAS or SANs, which physically are not a part of the PMs, although they are transparent to the PMs [Choudhary et al., 2017, Mulahuwaish et al., 2022]. Swapping to such remotely attached disks cannot bring the same performance as swapping to real local disks attached to PMs. If swapping to SATA-based SSD or HDD, the performance will be much worse than remote memory because these types of disk devices are generally much slower than NVMe SSD.



(a) tradesoap



(b) tradebeans



(c) h2

Figure 3.16: Execution time performance averaged from 10 runs for DaCapo applications (tradesoap, tradebeans, and h2) in Docker and KVM/QEMU environments.

3.8 Summary of Chapter

In this chapter, we have presented a memory sharing system with which a memory overloaded PM automatically borrows memory resources from a remote PM with spare memory to share out. Our system allows a PM to be over-committed through provisioning more memory resources than its capacity. Unlike traditional methods, such as VM live migration, our system can handle memory overload without interrupting or suspending running applications. In addition, memory sharing is transparent to system processes, including applications, VMs, and container instances. We have also designed a unified control algorithm. The borrower PM automatically borrows memory based on a dual-threshold trigger, while the lender reserves sufficient spare memory resources for memory sharing. Experimental studies have been conducted on InfiniBand-networked PMs to demonstrate that the memory sharing system is fully functioning as designed. The overall performance of the memory sharing system in the speed and execution time for remote memory access has been shown experimentally to be similar to that for accessing a local NVMe SSD as a swap space. Depending on how an application manages its data in memory, remote memory can occasionally have better or worse performance than local NVMe SSD based swap operations.

This chapter has shown the feasibility and full functionality of memory sharing in cloud data centres. Extending the work presented in this chapter, the work of memory sharing with multiple borrowers and multiple lenders is shown in following chapters. This requires planning, scheduling, and optimization of memory resources from numerous PMs in a data centre.

Chapter 4

Instant Processing of Memory Sharing

The previous chapter addresses how to automatically share memory resources between two physical machines (PMs). The PM which shares out its spare memory resource is defined as a lender PM, while the PM which utilizes another PM's memory resource as swap space is defined as a borrower PM. In the memory sharing system for memory sharing between two PMs, the role of lender PM and borrower PM can be transferred according to the instant memory usage of that PM. Moreover, a PM is in the neutral state if it is not memory overloading nor lending its spare memory resource.

Despite the progress in reusing the spare memory resource of lender PMs for borrower PMs, it is still lacking with regard to dynamically pairing borrower PMs and lender PMs because cloud data centres have numerous PMs which are grouped as clusters other than just two PMs, as assumed in the previous chapter. In cloud data centres, the borrower PM needs an ability to find a suitable lender PM for its memory disaggregation. Since the role of a PM dynamically changes between borrower, lender, and neutral state, it is required that the proposed framework can instantly find a lender PM for a borrower PM. In addition, the feasibility of many borrower PMs using the spare memory resource of many lender PMs has been confirmed in Chapter 3. It suggests that the proposed framework may find one or more lender PMs for a borrower PM.

The memory overload problem can be dealt with by temporary memory paging from a memory overloaded PM to a memory underutilized PM on demand. The two PMs should be reverted to their original state once memory overload no longer exists. This is expected to maximize the success of PMs pairing, while minimizing the possibility of memory overload of the PMs whose spare memory resource is used by other PMs.

This chapter proposes a memory sharing framework for handling memory overload occurring on multiple PMs at the same time. Memory resource is dynamically and temporarily shared from lender PM to borrower PM. According to the discussion above, two variants of the instant processing of memory sharing framework are proposed in this chapter. In the first variant, a borrower PM can use memory resource from only one lender PM. The second variant of the framework handles this using a different approach, which allows a borrower PM to use memory resource from multiple lender PMs. Furthermore, a lender PM can share its spare memory resource to multiple borrower PMs in both variants of the framework. Experiments in the proposed framework are conducted and evaluated in this chapter to investigate performance difference.

The goal of the instant processing framework of memory sharing is to maximize the success of PMs pairing, while minimizing the possibility of memory overload on lender PMs and the overhead of managing connections between the borrower PM and lander PM. To minimize the overhead of managing memory sharing connections, a profile-guided clustering algorithm is proposed to find lender candidates for instant processing of memory sharing.

Overall, this chapter makes following main contributions:

1) A profile-guided clustering algorithm with tolerance of missing values for filtering PM candidates to share memory;

2) A memory sharing framework is designed for sharing memory resource between many borrower PMs and one lender PM (one-to-many);

3) A variant of the instant processing framework of memory sharing, which supports sharing memory resource between many borrower PMs and many lender PMs (many-to-many);

4) The instant processing framework of memory sharing is experimentally evaluated in terms of its simulated improvement on resource utilization of cloud data centre.

The rest of the chapter is organized as follows: Section 4.1 discusses the background problem and research motivation for this chapter. Section 4.2 describes designs of one-to-many and many-to-many memory sharing and their feasibility examinations. Section 4.3 presents the architecture of the instant processing framework of memory sharing. Section 4.4 designs the one-to-many and many-to-many control algorithm for global controller, and the clustering

algorithm for the lender PM selection is detailed in Section 4.5. Experimental studies are conducted in Section 4.6 to validate the performance of our system. Finally, Section 4.7 concludes the chapter.

4.1 Problem Analysis and Motivation

Over the years, an over-commitment strategy has been widely adopted for enhancing resource utilization of cloud data centres. However, computation tasks with low priority would be terminated if the host PM becomes memory overloaded.

4.1.1 Case Study

Alibaba's large-scale cloud system is a cloud data centre that matches the problem scenario. Its open trace data shows that over-subscription and under-subscription problems can coexist in cloud data centers [Everman et al., 2021]. Such cloud systems involve both online services (a.k.a. long running applications) and batch workloads co-located in every machine in the Alibaba's cluster, where tasks may be terminated due to insufficient computing resources.

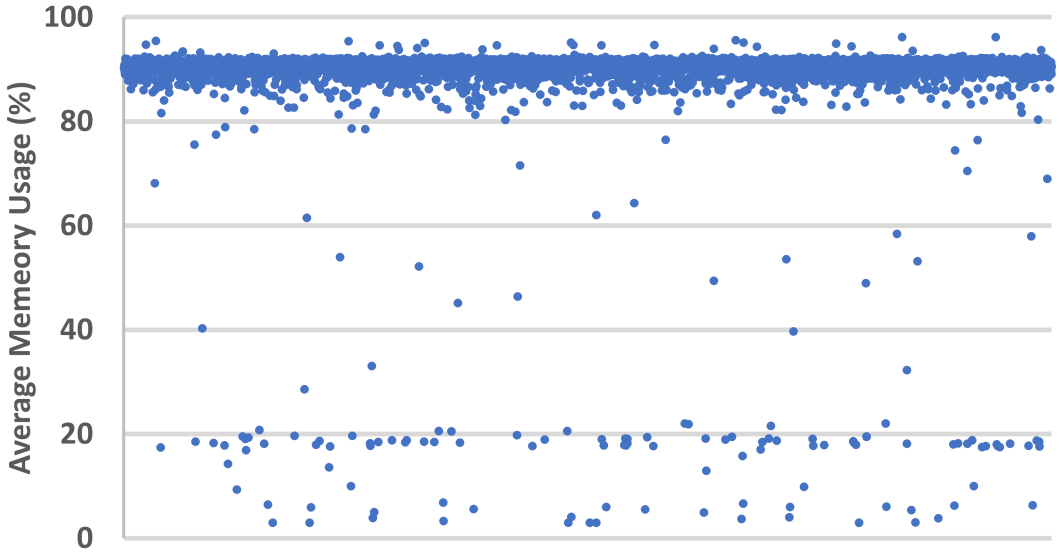


Figure 4.1: Average Memory Usage Over 8 Days in the Alibaba Data Centre

An 8-day trace data [Alibaba Open Source, 2018, Guo et al., 2019] shows that the cluster's average memory usage is 88%, as shown in Figure 4.1. A majority of PMs have a memory usage over 80%. However, there are still numerous PMs that have an average memory usage

around and under 20%, which gives the opportunity of balancing memory usage among PMs by an over-committing strategy and memory sharing between underutilized PMs and memory overloaded PMs.

The challenge is how to allow setting a higher level of over-commitment than the existing one without worrying about task termination caused by memory overload. Firstly, the scheduler should not assign extra computation tasks to high memory utilization PMs, as this increases the possibility of memory overload. Secondly, low memory utilization PMs and under-subscribed PMs should be kept online.

4.1.2 Technical Gaps and Motivation

Existing efforts have proved that the memory overloading issue of a PM can be handled by using memory resources on remote PMs. Most of them require a pre-configuration of disaggregated memory on dedicated PMs. Moreover, some efforts design multiple and hybrid disaggregated memory for handling memory overload of PMs in cloud data centres. However, there is a lack of effort focusing on how to benefit cloud data centres by applying memory sharing.

Therefore, technical gaps exist regarding our memory sharing requirements for PMs in cloud data centres, i.e., 1) a memory overloaded PM can find and choose a PM that has spare memory resource in a short time with low overhead; and 2) when a PM has shared its spare memory resource, it has a low possibility of becoming memory overloaded.

Since cloud data centres trace the resource usage of each machine for analysis, a machine learning method can be applied for finding a part of PMs with a similar pattern. A clustering method can be applied for grouping lender PM candidates based on statistical information or knowledge gained from resource usage trace data. By selecting feasible PMs from grouped lender PM candidates, the overhead of selecting the feasible underutilized PM can be reduced. This can also reduce the possibility of becoming memory overloaded when sharing out spare memory resource. In addition, this research is also motivated by the demand on a clustering method with tolerance of missing value because it is found that Alibaba's cluster data, as shown in the case study in Section 4.6.1, has massive missing values among all recorded PMs.

4.2 Design of Memory Sharing and Feasibility Study

By extending the principle of one-to-one memory sharing described in Section 3.3, this section illustrates the one-to-many memory sharing procedure and many-to-many memory sharing procedure. The feasibility of these procedures is examined and explained in this section.

4.2.1 One-to-Many Memory Sharing

One-to-many memory sharing demonstrates that one PM can share its spare memory resource to many PMs. The verification of its feasibility utilizes similar methods as one-to-one memory sharing. However, there are three PMs as shown in Figure 4.2. PM-A1 and PM-A2 are PMs which use the remote memory resource, while PM-B is the PM which shares out its spare memory resource.

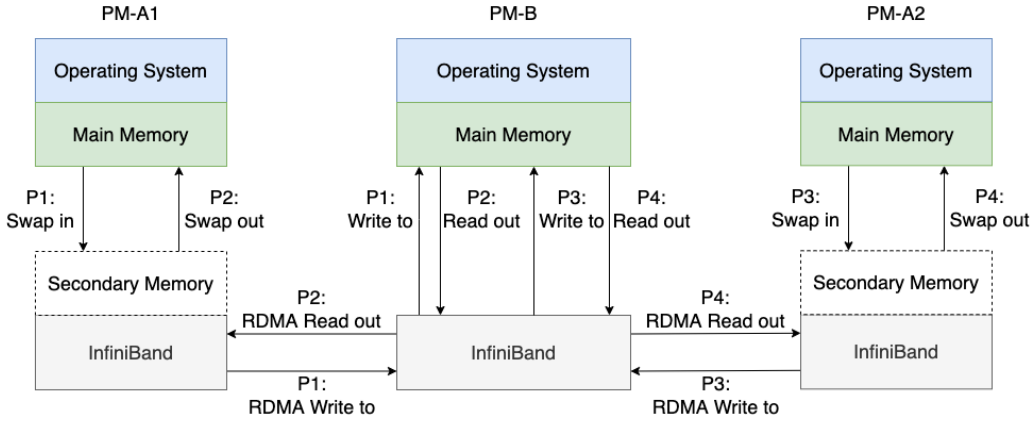


Figure 4.2: One-to-Many Memory Sharing

In this feasibility verification, PM-B creates two ram disk block devices, and sets up these block devices as two NoF subsystems for sharing memory to PM-A1 and PM-A2. Then, PM-A1 and PM-A2 can connect a unique NoF subsystem and write memory pages to the memory space of PM-B through RDMA.

An example with four memory paging operations is illustrated in Figure 4.2. PM-B is located in the middle of the figure. It shares its spare memory resource to other two PMs, PM-A1 and PM-A2, which are located on both sides of the figure. Memory page P1 and P3 are swapped into PM-B. Memory page P2 is swapped out from PM-B to PM-A1, while memory page P4 is swapped out from PM-B to PM-A2.

4.2.2 Many-to-Many Memory Sharing

Many-to-Many Memory Sharing demonstrates a complex scenario. There are many PMs which use remote memory resources, and many PMs which provide their spare memory resources for sharing to other PMs. In this case, multiple secondary memory devices are required to link multiple NoF subsystems published by different PMs. Multiple secondary devices can be directly handled by the swap feature of the operating system. By default, the operating system swaps memory pages to multiple swap spaces by a round-robin scheduler to balance the usage of each swap space.

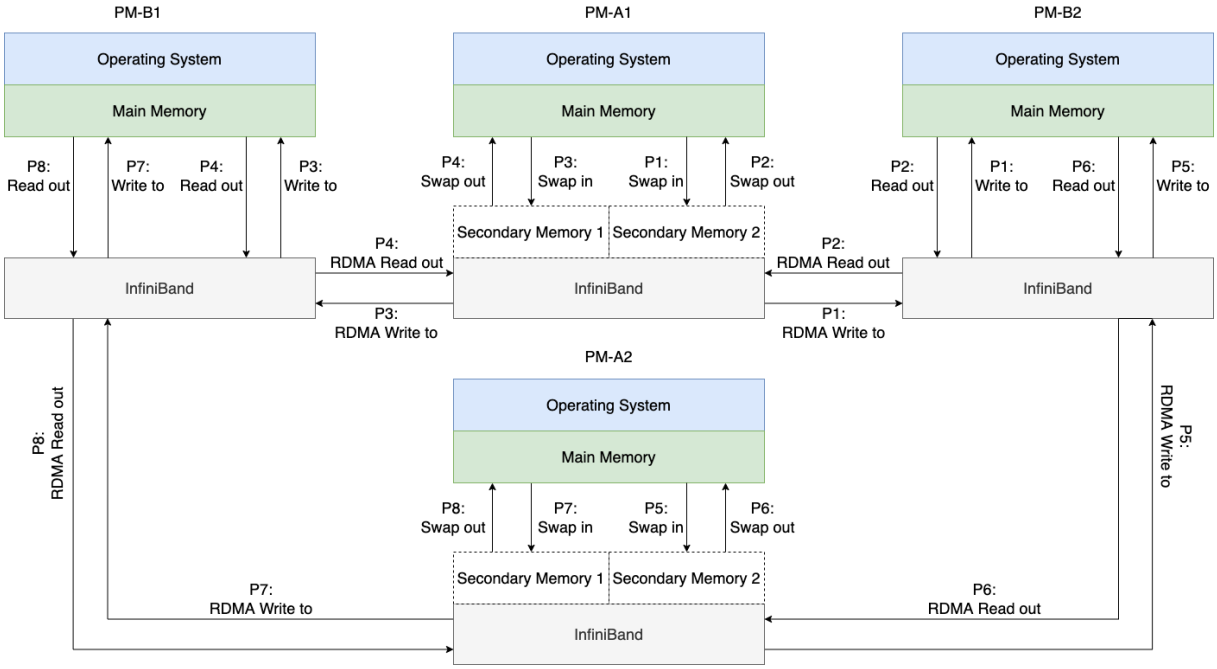


Figure 4.3: Many-to-Many Memory Sharing

In the feasibility verification of many-to-many memory sharing, as shown in Figure 4.3, PM-B1 and PM-B2 share out their spare memory for using by PM-A1 and PM-A2. Both PM-B1 and PM-B2 prepare two ram disk block devices and NoF subsystems for memory sharing. Each of PM-A1 and PM-A2 connects two NoF subsystems, where one NoF subsystem is from PM-B1 and another is from PM-B2. The swap activation is performed twice in order to activate two back-end devices for swap space as secondary memory. The operating system automatically utilizes both swap devices with balancing scheduling.

The feasibility study of this model also shows multiple swap devices can be deactivated in a different order than activation. This means it is feasible to convert from many-to-many memory sharing to one-to-many or one-to-one memory sharing on the fly whenever it is necessary.

During deactivation of the secondary memory device, memory pages are transferred from the deactivating NoF subsystem to existing online secondary memory devices.

4.3 Architecture of Instant Processing Framework

Our instant processing framework of memory sharing is proposed with one global controller and multiple local controllers for memory sharing among a cluster of PMs. The global controller runs on the management PM of the cluster. It can be a standalone PM or the PM which plays the role of cluster hypervisor manager. The local controller runs on every PM which has required borrowing remote memory resource or lending memory resource.

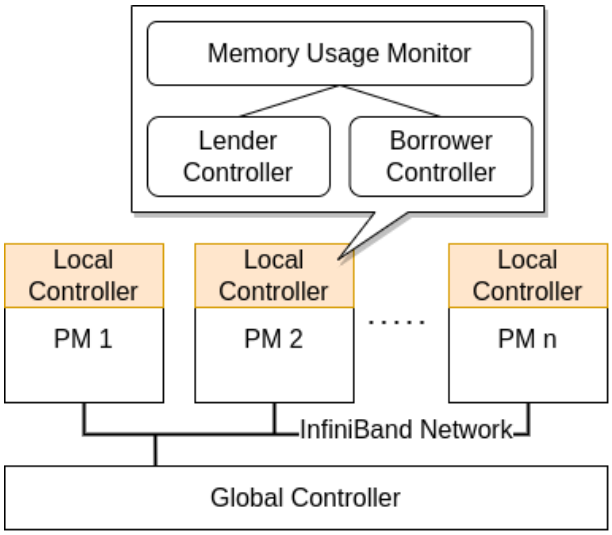


Figure 4.4: Brief Architecture of One-to-Many Memory Sharing System

Figure 4.4 shows the broad architecture of the instant processing of memory sharing framework. Each PM is connected through the InfiniBand-enabled network for memory sharing and management communication. The local controller running on each PM consists of three main modules: memory usage monitor, lender controller, and borrower controller. The memory usage monitor observes system memory usage only, with no prior knowledge of running workloads. The lender and borrower controller are used for management connection of memory sharing. More details of these modules have been well explained in Chapter 3. In addition, the memory usage monitor is slightly different from what is explained in Chapter 3. It not only monitors memory usage of its PM, but also reports observed memory usage to the global controller.

The global controller periodically collects instant memory usage of PMs. It also makes the decision for the borrower PM to select the lender PM according to collected memory usage trace

data from each PM of the cluster. Modules in the global controller and how they are operated with modules in the local controller are displayed in Figure 4.5. Depending on how the lender PM is selected, there are three or four modules in the global controller.

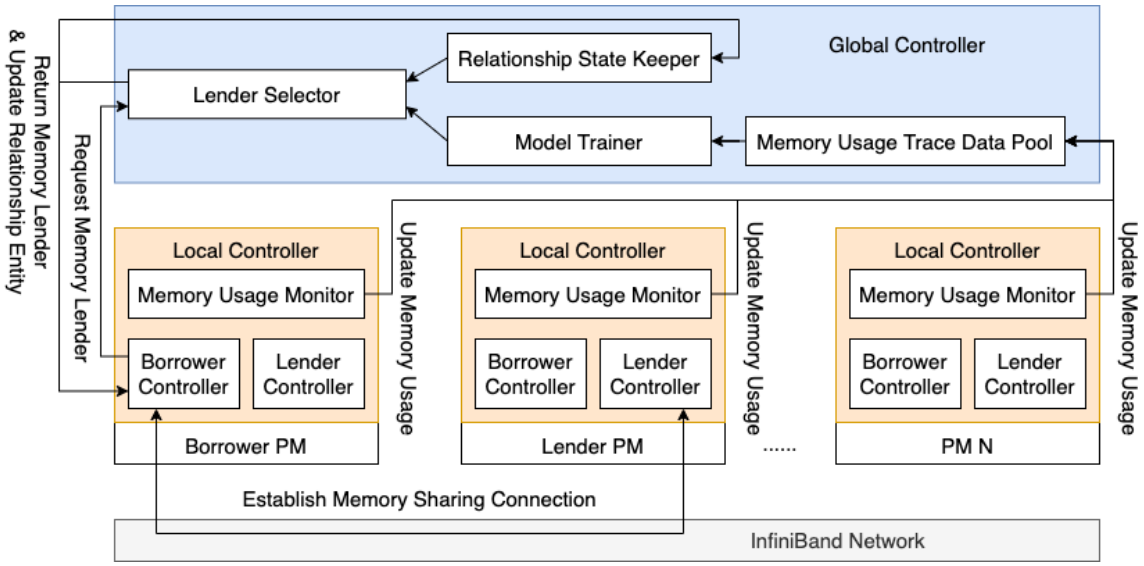


Figure 4.5: Architecture of One-to-Many Memory Sharing System

Memory Usage Trace Data Pool tracks memory usage of all PMs in the cluster. It receives usage data sent by the memory usage monitor from each PM. Such memory usage trace data is used by the model trainer to generate the lender PM candidate model.

Model Trainer generates the lender PM candidate model. The algorithm of how to generate such model is described in Section 4.5. It updates in each service cycle.

Relationship State Keeper stores relationships of the lender PM and borrower PM. Each relationship entry contains information of the lender PM ID, borrower PM ID and size of memory shared from the lender PM to the borrower PM. The relationship entry is created by the lender selector once a lender PM is selected for sharing memory resource to a borrower PM. However, deletion of the relationship entry is done by the borrower PM, after successful disconnection of the remote memory resource. The deletion of the relationship entry does not require notifying to the lender selector.

Lender Selector handles the request asking for the memory lender sent by the borrower controller of the local controller. Depending on the global controller algorithm, it can instantly make multiple borrower PMs and a lender PM as a pair or make multiple borrower PMs and multiple lender PMs as a pair. Lender PM candidates are filtered and selected by the model trainer. The paired information is stored as relationship entries in the relationship state keeper.

4.4 Global Controller Algorithms

The global controller algorithm defines how lender PMs are selected for borrower PMs to handle their memory overloading by memory sharing. There are two types of global controller: one-to-many memory sharing controller and many-to-many memory sharing controller, where one-to-many and many-to-many represent the type of lender-borrower pair. One-to-many represents a memory sharing relationship where one lender PM shares its spare memory resource to many borrower PMs. Many-to-many represents a memory sharing relationship that many lender PMs concurrently share their spare memory resource to many borrower PMs.

In both one-to-many and many-to-many memory sharing control, whether a lender PM candidate can share its spare memory resource to a specific borrower PM depends on evaluation of shareable memory. Memory sharing becomes feasible only if the lender PM has the positive result of shareable memory.

$$M_s = M_t \times T_{lower} - M_{req} - M_u - M_{shared} \tag{4.1}$$

The shareable memory size is calculated by Equation (4.1). The PM running in normal state means the sum of memory used M_u and memory resource which has been shared M_{shared} is smaller than the lower threshold. Thus, M_s is positive in normal state. By subtracting requested memory resource M_{req}, M_s may be still positive, which indicates memory sharing for the requester borrower PM is feasible. On the other hand, it is infeasible to share a memory resource if M_s is negative after subtracting requested memory resource M_{req}.

4.4.1 One-to-many Memory Sharing Controller

In instant processing of memory sharing, the one-to-many memory sharing controller finds a lender PM for borrower PMs. It finds one lender PM for one borrower PM at a time. If more than one borrower PM asks for a lender PM at the same time, it handles the requests one by one.

Figure 4.6 exhibits how the one-to-many memory sharing controller handles the memory borrowing request sent by the borrower PM. There are two threads of works running concurrently: memory usage trace data pool and lender selector.

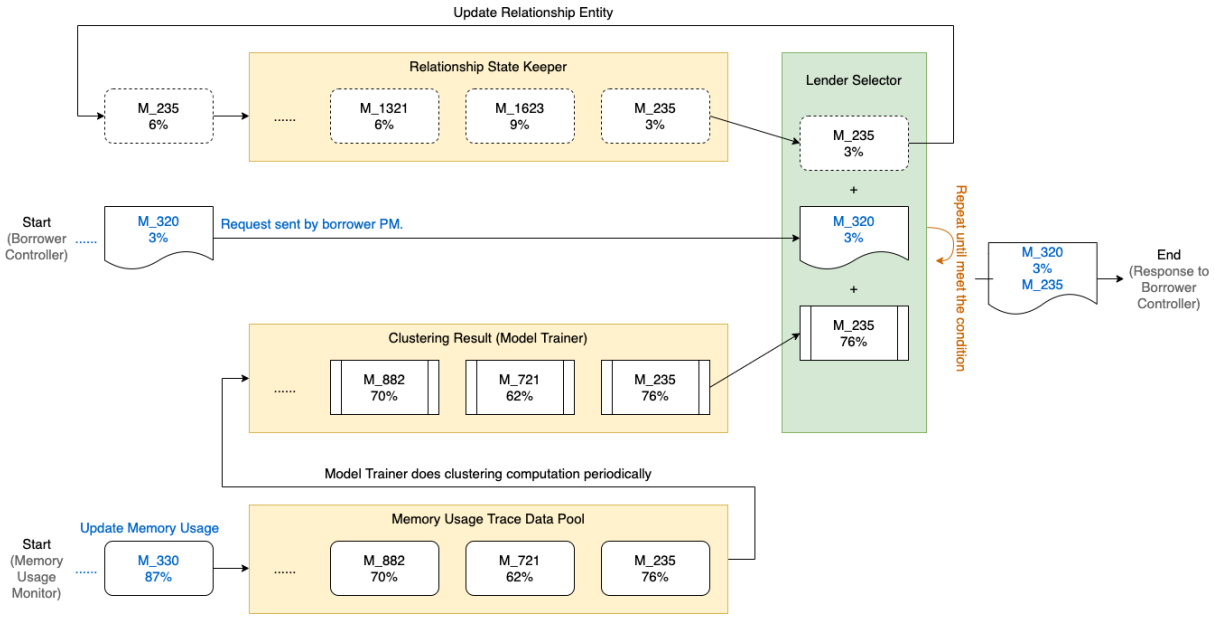


Figure 4.6: Process of One-to-Many Memory Sharing Controller in Instant Processing

Memory usage trace data pool stores the most recent memory usage data sent by the memory usage monitor in each local controller of node PMs. The memory usage data is then used by model trainer to filter and sort node PMs in order to generate a list of lender PM candidates. The filter and sort computation is clustering-based, which will be detailed in Section 4.5.

To respond to a memory borrowing request sent by borrower PM, the lender selector finds a feasible lender PM picked from lender PM candidates. This process is shown in the green rectangle of Figure 4.6. The lender selector picks a lender PM candidate M_235 from the clustering result. Since M_235 has a borrower-lender relationship, the lender selector also obtains the borrower-lender relationship entry of M_235 from the relationship state keeper. It then examines the feasibility of lending memory resource by M_235, according to current memory usage and the size of memory shared. If it is feasible, the lender selector will update the relationship entry in the relationship state keeper followed by responding to the borrower PM the result. On the other hand, this process is repeated if the examination fails.

The control algorithm for the lender selector is shown in Algorithm 5. It examines lender PM candidates in a dual loop, which walks through all PMs (line 2) in every group of the clustering result (line 1). The examination varies according to the borrower-lender relationship of the examined PM (line 3). The size of the memory resource which has been shared will be taken into account if the examined PM has an existing borrower-lender relationship entry (lines 3-6). Otherwise, only the used memory and requested memory are computed against

Algorithm 5: One-to-Many Control Algorithm for Lender Selection (Clustering)

```
   Input  : Clustering computation result.
   Output : Selected Lender PM
1  for each group in clustering result do
2      for each PM in PMs of this group do
3          if this PM has borrower-lender relationship entry then
4              Calculate M_s by Equation (4.1);
5              if M_s is positive then
6                  Set this PM as selected PM;
7                  Stop the loops;
8          else
9              if sum of M_u and m_req is smaller than threshold then
10                 Set this PM as selected PM;
11                 Stop the loops;
12 Update borrower-lender relationship entry to Relationship State Keeper;
13 Return selected lender PM;
```

the threshold in order to determine the feasibility of sharing out the memory resource from the examined PM (lines 8-9).

Algorithm 6: One-to-Many Control Algorithm for Lender Selection (Mean-based FFD)

```
   Input  : Average memory usage of all PMs
   Prerequisite: List of PMs is sorted by average memory usage in increment order.
   Output : Selected Lender PM
1  for each PM in list of PMs with increment order on average memory usage do
2      if this PM has borrower-lender relationship entry then
3          Calculate M_s by Equation (4.1);
4          if M_s is positive then
5              Set this PM as selected PM;
6              Stop the loops;
7      else
8          if sum of M_u and m_req is smaller than threshold then
9              Set this PM as selected PM;
10             Stop the loops;
11 Update borrower-lender relationship entry to Relationship State Keeper;
12 Return selected lender PM;
```

Another control algorithm, Mean-based First-Fit-Decreasing (Mean-based FFD), for lender selector is shown in Algorithm 6. It examines lender PM candidates in a single loop, which walks through all PMs (line 1) to find a lender PM which is feasible to accept the memory

borrowing request. The examination part (lines 2-10) is the same as the clustering-based control algorithm.

4.4.2 Many-to-many Memory Sharing Controller

The many-to-many memory sharing controller allows a borrower PM to use memory resource from multiple lender PMs at the same time. It has three differences from the one-to-many memory sharing controller. Firstly, the many-to-many memory sharing controller requires an extra process prior to memory selection, which is splitting off the memory borrowing request. Secondly, the memory selection algorithm is required to run multiple times for completing a memory borrowing request. Thirdly, selection results are grouped to form a single response to the borrower PM. These are detailed in Figure 4.7.

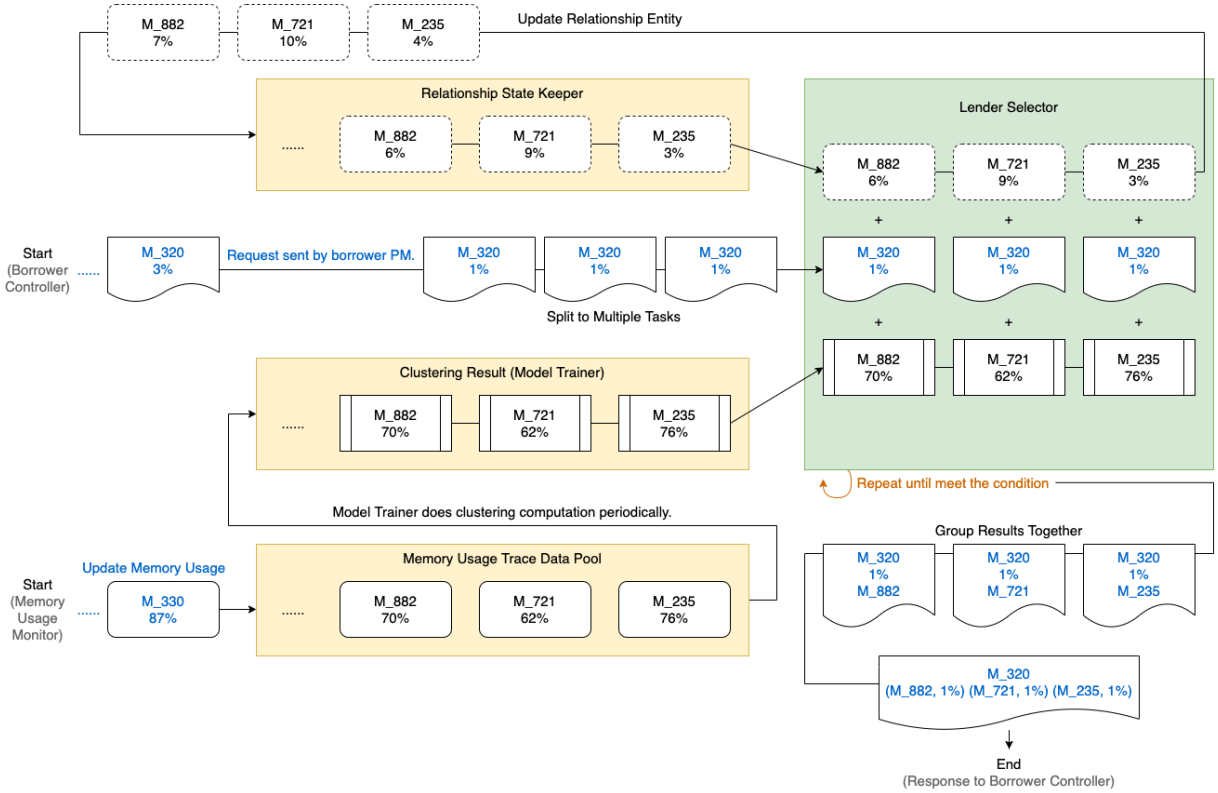


Figure 4.7: Process of Many-to-Many Memory Sharing Controller in Instant Processing

Splitting off the memory borrowing request is executed ahead of lender selection. An example is demonstrated in Figure 4.7, by assuming that all the PMs have the same memory capacity.. Borrower PM M_320 requests a remote memory resource with normalized size of 3%. Such a size of remote memory resource is split into three sub-requests as tasks to the lender selector. In addition, each task has equal size of requested memory, while the normalized

minimum size of requested memory for the task is 1%.

The lender selector runs the lender selection algorithm multiple times to find the lender PM for each task. As shown in the green rectangle, it selects lender PM M_882, M_721, and M_235 as lenders to borrower PM M_320. It then updates the lender PMs' borrower-lender relationship entries, followed by returning a grouped result to borrower PM M_320.

Algorithm 7: Many-to-Many Control Algorithm for Lender Selection (Clustering)

```
   Input  : Clustering computation result.
   Output : Selected Lender PMs
 1 Split M_req to multiple lender selection tasks;
 2 for each task in lender selection tasks do
 3     Do Algorithm 5 for this task;
 4     if has returned value (lender PM is found) then
 5         Add to selected lender PMs;
 6         Update borrower-lender relationship entry to Relationship State Keeper;
 7     else
 8         Revert updates of borrower-lender relationship entry executed in this loop;
 9         Stop the loop;
10 Return selected lender PMs;
```

The whole process of many-to-many memory sharing controller is also detailed in Algorithm 7. The memory borrowing request is split as first step (line 1). Algorithm 5 is performed for selecting the lender PM for each split task (lines 2-3). In each lender selection task, if a lender PM is selected, the borrower-lender relationship entry will be updated (line 6) in order to reflect this lender PM reservation for further runs of Algorithm 7. However, if Algorithm 7 returns no lender is selected, it will be necessary to revert updates of borrower-lender relationship entry (line 8). In addition, the number of returned lender PMs from Algorithm 7 needs to be checked against the number of split lender selection tasks. The many-to-many lender selection is successful only if these numbers are equal. Otherwise, it fails to find lender PMs for requester borrower PM.

Many-to-many memory sharing control algorithm for mean-based FFD, as shown in Figure 8, is slightly different from clustering-based. In mean-based FFD, sorted lender PM candidates are looped first (line 1). In each loop, memory borrowing requests are looped (line 2) to assign multiple memory borrowing requests to the lender PM candidate selected in the first loop when it is feasible.

Algorithm 8: Many-to-Many Control Algorithm for Lender Selection (Mean-based FFD)

Input : Average memory usage of all PMs
Prerequisite: List of PMs is sorted by average memory usage in increment order.
Output : Selected Lender PM

```
for each PM in list of PMs with increment order on average memory usage do
    for each task in lender selection tasks do
        if this PM has borrower-lender relationship entry then
            Calculate M_s by Equation (4.1);
            if M_s is positive then
                Set this PM as selected PM;
                Stop the loops;
        else
            if sum of M_u and m_req is smaller than threshold then
                Set this PM as selected PM;
                Stop the loops;
Update borrower-lender relationship entry to Relationship State Keeper;
Return selected lender PM;
```

4.4.3 Capability and Feasibility of Global Controller

It is assumed there is always a solution for the global controller. There is certainly at least one feasible PM for sharing memory, if the cluster runs under its capacity. However, if the overall memory usage of the cluster runs near or over the limit of the memory resource, there will be no solution for the global controller, which will result in the global controller unable to find any feasible lender PM for memory sharing. Such case can be considered as the failure on resource provisioning of the cluster. To avoid occurrence of this case, it is assumed one or more additional PMs is automatically booted up in order to prevent the cluster running over its capacity. Memory sharing becomes feasible with these additional PMs.

According to the assumption above, the global controller can always find a solution. The FFD can always guarantee a feasible solution by its nature. Similar to the FFD, our proposed algorithm orders the candidate lender PMs by descending size, and then and then call the first-fit bin packing.

4.5 PM Clustering Algorithm

The clustering algorithm is designed to group PMs with similar trend of memory usage changes. Traditional time series clustering methods cannot be applied because they cannot handle the situation of missing data in time series. Since memory usage data is tracked in cloud data centres, it is common that the data of one or more PMs is missed on a time point. The reasons could be such PMs are suspended for energy saving purpose or such PMs are temporarily unresponsive due to various errors.

4.5.1 Mean Profile

The principle of our clustering algorithm is to create a new line, called 'mean profile', based on the average value of each time point. Then, PMs can be clustered according to the distance between them and the mean profile.

Given a set of data patterns $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ...\boldsymbol{x}_M)$, where $\boldsymbol{x}_i = (x_{i1}, x_{i2}, ..., x_{iT})$ is the vector, $i = 1, ..., N_m$, N_m is the total number of machines and T is the total time points.

The similarity (distance) of two machines, x_i and x_j, is represented by

$$d(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(x_{it} - x_{jt})^2} \tag{4.2}$$

Mean profile can be calculated by

$$\boldsymbol{\mu} = (\mu_1, ..., \mu_T) = (\frac{1}{N_m}\sum_{i=1}^{N_m} x_{i1}, ..., \frac{1}{N_m}\sum_{i=1}^{N_m} x_{iT}). \tag{4.3}$$

The distance between each machine and the mean profile is

$$d(\boldsymbol{x}_i, \boldsymbol{\mu}) = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(x_{it} - \mu_t)^2}. \tag{4.4}$$

4.5.2 Clustering Method

The clustering method is run on the model trainer in order to filter and narrow down lender candidates for lender selector. It is designed to allow the borrower PM to ask limited PMs for the lender candidate instead of all PMs, meanwhile reducing the risk of overloading the lender PM.

There are several steps to do the clustering computation:

1) Calculate mean profile for filtered PMs by Equation (4.3).

2) Calculate distance to mean profile for each PM from filtered PMs, by Equation (4.4).

3) Group result of distance to mean profile, and form 10 groups because memory usage of each PM is normalized which is between 0 and 100. It means that distance to mean profile is between 0 and 100 as well.

4) Sort PMs in each group by their standard deviation.

The clustering computation is represented as a clustering algorithm, as shown in Algorithm 9, for one-to-many and many-to-many memory sharing frameworks. Prior to clustering, PMs under any of the following conditions are filtered out (lines 2-7): PM with memory usage higher than cluster's average memory usage and PM with more than a certain value of trace data missed. For Alibaba's trace data [Alibaba Open Source, 2018], such value can be 20%, as shown in Figure 4.8. Then, the mean profile and distance between mean profile and PMs are calculated in lines 8-9 and 10-11 respectively. Lines 12-15 describes how to group the clustering results. The remainder of division by ten is used to group clustering results based on the distance to mean profile in multiples of ten. Finally, the PMs of each clustering group are sorted (lines 16-17).

4.6 Experimental Studies and Evaluation

This section analyses cluster trace data published by Alibaba Group and describes the conduct of experiments to investigate improvement on memory resource utilization of the modern cloud data centre. The experiment is conducted in the form of simulation, using cluster trace data published by Alibaba. The simulation is implemented in Go programming language.

Algorithm 9: Clustering Method for Filtering and Sorting Node PMS to Lender PM Candidates

Input : Memory usage trace data pool.
Output: Lender PM candidates in form of PM clustering groups.

Calculate mean of memory usage of the whole cluster;
for *each PM in memory usage trace data pool* **do**
    Calculate mean of memory usages of this PM;
    Calculate standard deviation of memory usages of this PM;
    Count how many time points have no memory usage data for this PM;
    if *the mean is under the mean of the cluster* ***and*** *percentage of missing data is under 20%* **then**
        Add to lender PM candidates;
for *each PM in lender PM candidates* **do**
    Calculate mean profile by Equation (4.3);
for *each PM* i *in lender PM candidates* **do**
    $D_i \leftarrow$ Calculate distance between this PM and mean profile by Equation (4.4);
Generate 10 empty groups;
for *each PM* i *in lender PM candidates* **do**
    $g \leftarrow$ the least integer value greater than or equal to $\frac{100-D_i}{10}$;
    Add this PM to clustering group g;
for *each group of generated groups* **do**
    Sort PMs of this group by their standard deviation in increasing order;
Return PM clustering groups in decreasing order;

4.6.1 Alibaba Cluster Trace

The cluster trace data is sampled from one of Alibaba's production clusters [Alibaba Open Source, 2018], which has both online services (a.k.a. long-running applications) and batch workloads co-located in every PM of the cluster.

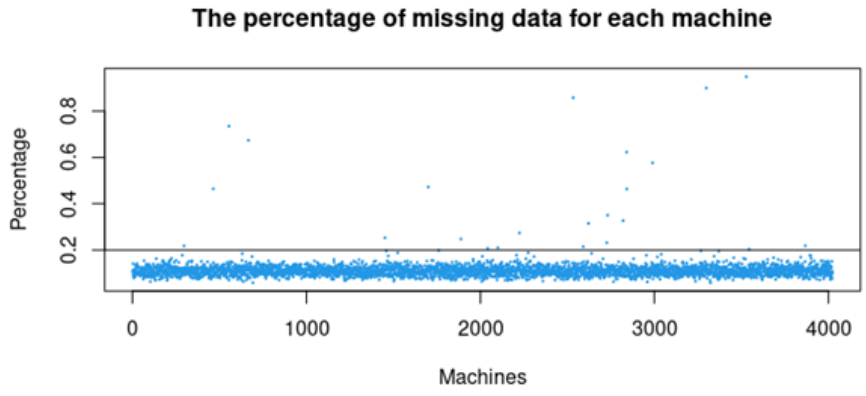


Figure 4.8: The Percentage of Missing Data for Each PM

Resource usage of 4023 PMs in a period of 8 days is recorded in cluster data. Resource usage is observed every ten seconds. Thus, there are 69120 observed time points for each PM. However, every PM has more than one time point missed. The percentage of missing data for each machine is presented in Figure 4.8. The PM m_3330 has the smallest proportion, 5.90%, of missing value; while m_3539 has the largest proportion of missing values, 94.84%. Thus, PMs whose percentage of missing data over 20% are removed prior to clustering modelling. This results in 4000 PMs that are left for clustering modelling.

4.6.2 Clustering Modeling

Firstly, the average memory usage for each PM in 69120 time points is calculated. Average memory usage for the whole cluster is also calculated, which is 88%. PMs with average memory usage higher than the cluster's average memory usage are filtered out. PMs with missed trace data in more than 20% of time points are also filtered out. This results in 469 PMs being left for clustering.

Secondly, the mean profile is calculated, which means the mean of 469 PMs for each time point. The range of the mean profile is between 57.85 and 93. The total length of the mean profile is 69120. However, there are 1878 points missed from the profile mean because there are no observed values in these time points for all PMs.

The next stage is to calculate distance between each machine and mean profile in each time point, followed by grouping the clustering result into ten groups. In order to clearly exhibits trend of each group, the mean profile of each group is calculated and shown in Figure 4.9 and Table 4.1. Only seven groups are displayed because there is not any PM whose distance away from the mean profile is larger than 70.

Finally, PMs in each group of clustering result are sorted based on their standard deviation.

4.6.3 Experimental Setup

The experiment is conducted by adding additional workload to each PM in the cluster. The memory usage data of each machine is recorded every ten seconds and is normalized to be shown in range from 0 to 100, which represents the percentage of memory usage to the capacity of the PM. Additional workload is added directly to the normalized memory usage. For

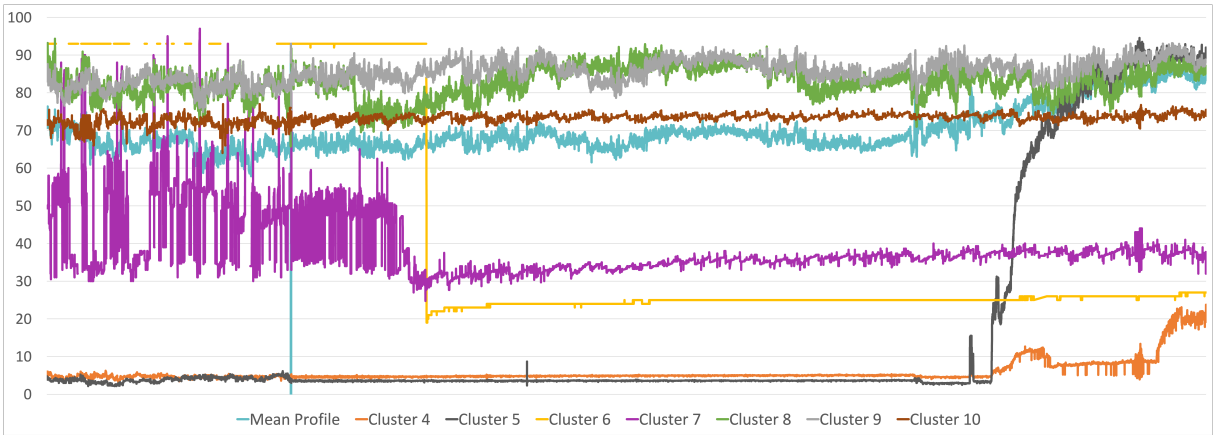


Figure 4.9: Clustering Results: Mean Profile and Clustering Result

Table 4.1: Clustering Results: Mean Profile of Clusters

Cluster	PM Count	Mean	SD	Min	Max
4	28	6.03	4.67	1	98
5	73	18.44	30.50	1	98
6	1	40.29	28.52	19	93
7	4	38.48	7.39	17	98
8	42	82.89	13.37	2	98
9	314	85.76	6.72	2	98
10	7	73.34	1.45	62	82

example, if the recorded memory usage is 95%, it will be shown as 98, in the experiment with additional 3% workload. Memory usage in the experiment can be higher than 100% after adding additional workload. Such a case is treated as memory overload.

The experiment is aimed to verify if: 1) the proposed memory sharing framework can improve memory resource utilization of cloud data centres; 2) clustering can reduce the number of lender candidates, while maintaining a reasonable performance.

The publicly accessible data used for simulation in the experiment is Alibaba's 8-day trace data [Alibaba Open Source, 2018]. A certain workload is added to each PM in Alibaba's cluster in order to simulate memory overload based on real data. During the experiment, memory usage over 98% is considered as memory overload. In addition, the experiment is conducted with four levels of additional workload; 2%, 3%, 4%, and 5% of normalized memory capacity of the PM.

To evaluate the performance, the clustering-based FFD (Algorithm 5 and Algorithm 7) and mean-based FFD (Algorithm 6 and Algorithm 8) approach are compared with standard deviation based FFD (SD). SD is similar as mean-based FFD. However, in SD, the lender PM

candidates are sorted based on standard deviation instead of the mean of 8-day memory usage in incremental order.

4.6.4 Experiment Condition

a Memory Overload of PMs Caused by Additional Workload

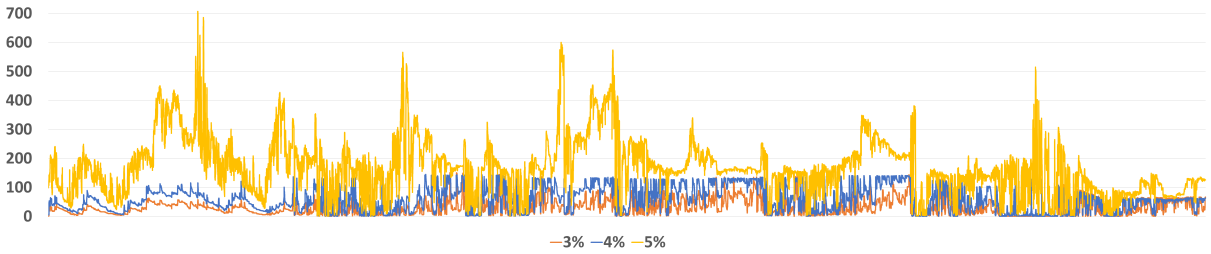


Figure 4.10: Count of Memory Overloading PMS Over the Time

Figure 4.10 shows how many PMs are under memory overloading in each time stamp after applying additional workloads on each PM. Since the average memory usage of overall cluster is around 88%, 5% additional workloads result in much more memory overloaded PMs than with 3% and 4% additional workloads. The number of memory overloaded PMs is up to 151 and 145 at a time point, for 3% and 4% additional workload respectively. However, such number increases to 707 with 5% additional workloads.

b Comparison of Lender PM Candidate

Table 4.2: Number of Lender PM Candidates

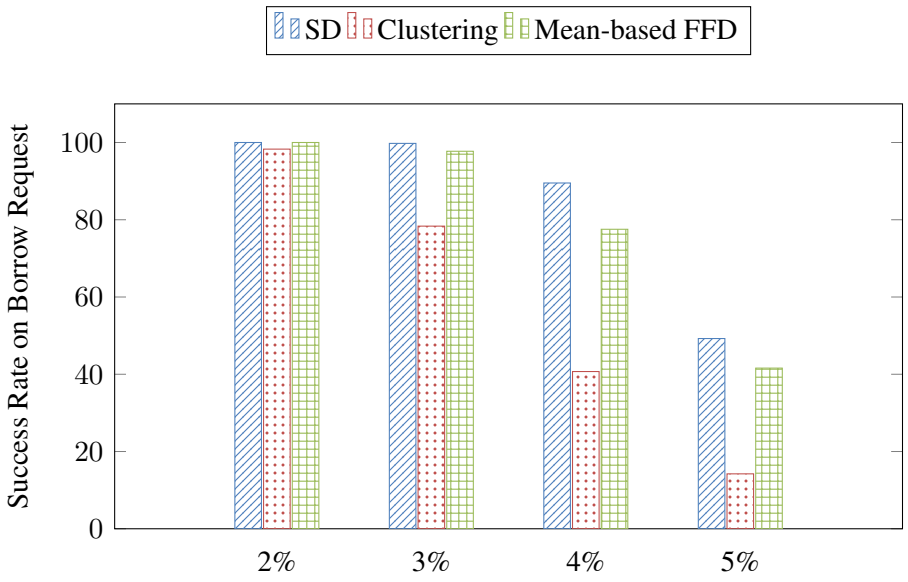
Additional Workload	SD	Clustering	Mean-based FFD
2%	4023	469	4023
3%	4023	469	4023
4%	4023	469	4023
5%	4023	469	4023

All 4023 PMs are considered as lender PM candidates, as shown in Table 4.2, while the proposed clustering-based method (Clustering) has 469 lender PM candidates scattered in 7 groups.

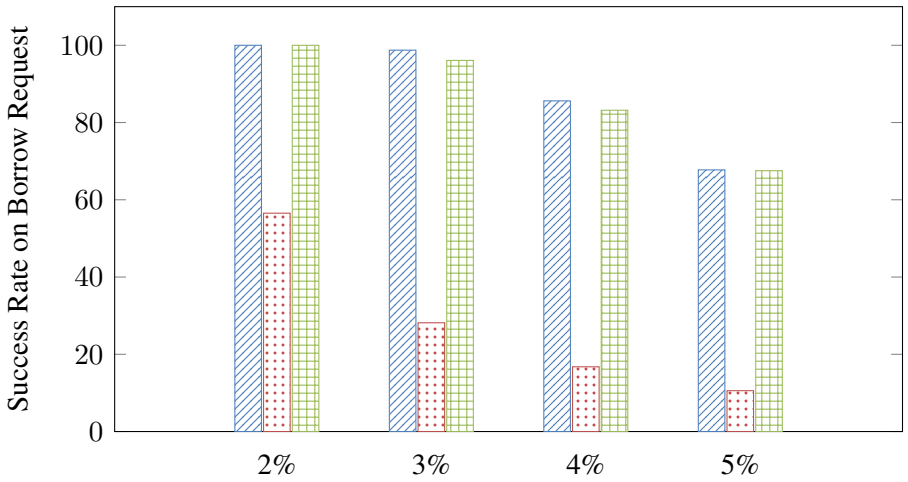
4.6.5 Experimental Evaluation

Experiments are done for both one-to-many and many-to-many memory sharing. The following are recorded for analysis: numbers about memory sharing requested by borrower PMs, numbers about maintaining memory sharing connection between the borrower PM and lender PM, and the number of distinct lender PM used in the experiment.

a Success Rate on Pairing Lender PM to Borrower PM



(a) Percentage of Successful Pairing Lender PM With Borrower PM in One-to-Many Memory Sharing



(b) Percentage of Successful Pairing Lender PM With Borrower PM in Many-to-Many Memory Sharing

Figure 4.11: Percentage of Successful Pairing Lender PM With Borrower PM in Instant Processing of Memory Sharing

The success rate on pairing the lender PM with the borrower PM in many-to-many memory sharing is calculated through dividing the count of successful pairing by the total count of memory borrowing requested by the borrower PM. Failure of pairing lender PM with borrower PM represents extra PMs are required to boot up in order to allow the global controller to find feasible lender PMs to memory borrowing requests. Figure 4.11a shows the success rate for one-to-many memory sharing, while many-to-many is shown in Figure 4.11b. Each figure is composed by results on four levels of additional workloads with three different control algorithms to select the lender PM.

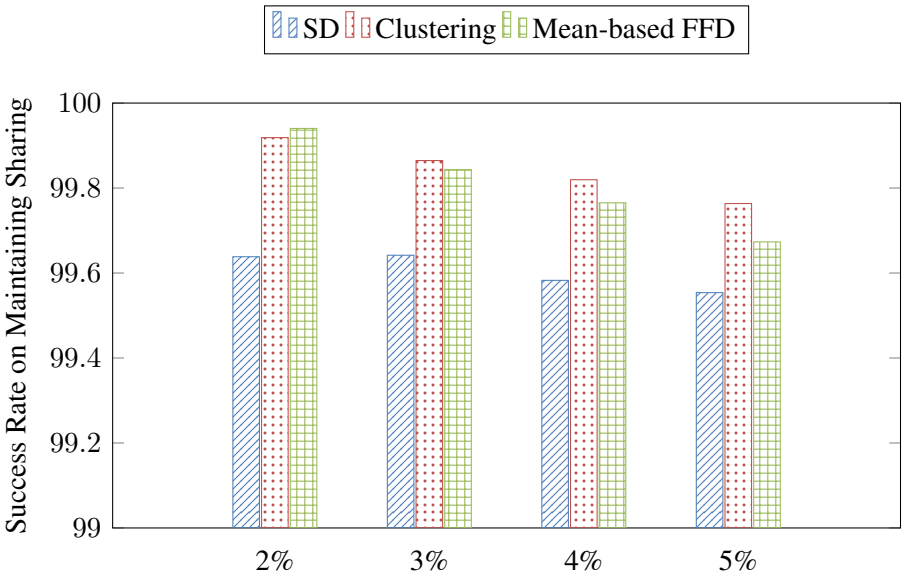
Mean-based FFD and SD have similar performance, which are higher than clustering. SD has better result on 4% and 5% additional workload than mean-based FFD, while its performance is slightly better than mean-based FFD in other scenarios. In addition, the drop in success rate over the increment of additional workload is reasonable. Success rate on 5% additional workload declines steeply because the number of memory borrowing request increases dramatically in such additional workload, as shown in Figure 4.10.

In contrast, clustering has quite low success rates on 5% additional workload performed for one-to-many memory sharing and all experiments for many-to-many memory sharing, although it is not designed for achieving a high success rate of pairing lender PM with the borrower PM. The reason for such a low success rate is that clustering only has around one tenth of lender PM candidates compared with others.

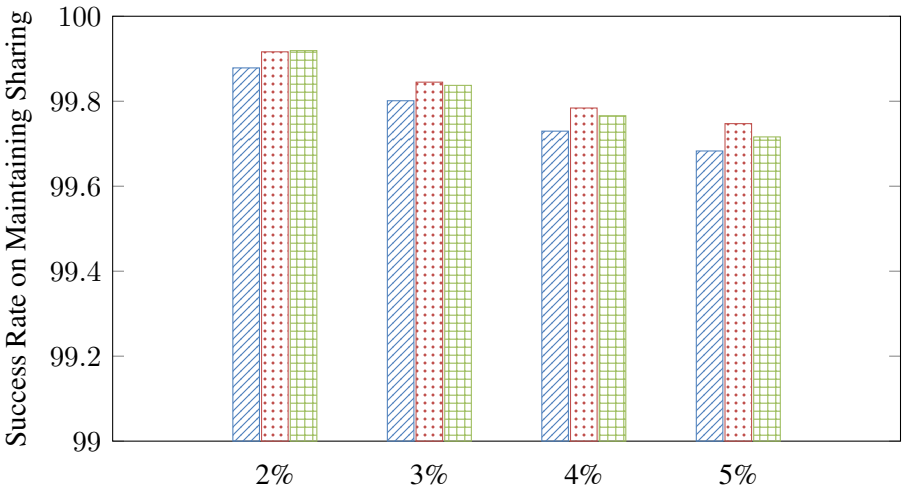
b Success Rate for Maintaining Memory Sharing Connection

The success rate of keeping memory sharing connection is calculated on the percentage of duration for which the lender PM can keep fulfilling the memory sharing requirement. Such duration is interrupted if the lender PM becomes memory overloaded during the connection of memory sharing with the borrower PM. All approaches have acceptable performance, where all of them have a result over 99.6%, as shown in Figure 4.12a (one-to-many memory sharing) and Figure 4.12b (many-to-many memory sharing). The clustering has the best performance in all experiments except for experiments with 2% additional workloads. Mean-based FFD has a similar performance as clustering, while SD has the worst performance.

In addition, the performance of clustering is analysed deeply to verify that small number of



(a) Percentage of Keeping Memory Sharing Connections in One-to-Many Memory Sharing



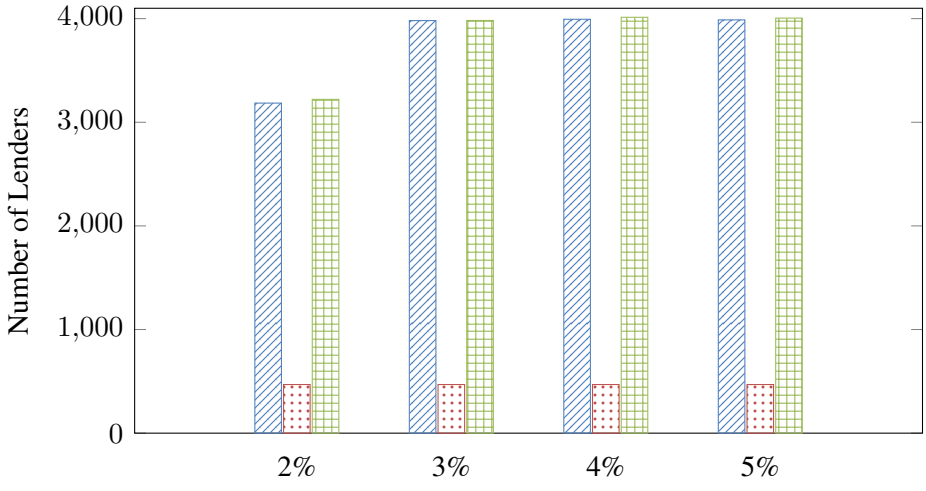
(b) Percentage of Keeping Memory Sharing Connections in Many-to-Many Memory Sharing

Figure 4.12: Percentage of Keeping Memory Sharing Connections in Instant Processing Framework of Memory Sharing

existing lender PMs for memory sharing does not increase success rate. A few more experiments are conducted by limiting the lender PM candidates for mean-based FFD. It is found that mean-based FFD still has lower rate than clustering, even when the lender PM candidates are limited to the same number as clustering.

c Distinct Lender Count

(a) Distinct Lender Count in One-to-Many Memory Sharing



(b) Distinct Lender Count in Many-to-Many Memory Sharing

Figure 4.13: Distinct Lender Count in Instant Processing Framework of Memory Sharing

Figure 4.13a (one-to-many memory sharing) and Figure 4.13b (many-to-many memory sharing) illustrate the distinct lender count which represents how many node PMs have played the role of lender PM during the experiment, where the same PM behaving as the lender PM

multiple times is not counted repeatedly. Among cloud servers, a higher number of lender PMs indicates higher overhead on managing memory sharing between lender PMs and borrower PMs.

The experiment results of clustering shows that the majority of lender PM candidates are used during experiments. 2% additional workload of one-to-many memory sharing uses 468 lender PM candidates, while all 469 are used in other experiments. It indicates that the number of lender PM candidates is not enough for additional workloads higher than 2%. This can also explain why the success rate of pairing the lender PM and the borrower PM is low by applying the clustering approach.

The other two approaches, SD and mean-based FFD, use many more lender PMs than clustering. SD occasionally uses a greater number of lender PMs than mean-based FFD in the experiment on one-to-many memory sharing, while it uses basically a similar number of lender PMs as mean-based FFD in the experiment of many-to-many memory sharing.

4.7 Summary of Chapter

In this chapter, an instant processing framework of memory sharing is presented for handling memory overload occuring on multiple PMs at the same time. The memory resource is dynamically and temporally shared from lender to borrower. The framework allows a lender to share its spare memory resource for multiple borrowers (one-to-many memory sharing), and allows multiple lenders to share their spare memory resources to multiple borrowers (many-to-many memory sharing). A profile-guided clustering algorithm is used for finding lender candidates and selecting the lender for the borrower. An additional selection algorithm for lender and borrower is mean-based FFD algorithm.

Experimental studies are conducted in simulation of Alibaba's public trace data for evaluating how much improvement our instant processing of memory sharing framework can bring to real world data centres. With a 2% increment of memory usage on each PM, our system can handle 98.3% of memory overload situations by utilizing spare memory resource of 11.7% PMs selected by clustering. In the experiment, only one tenth of PMs are considered as lender PM candidates for the clustering-based approach, while the mean-based FFD approach considers all PMs of the cluster as lender PM candidates. Experimental results show that for a typical

3% additional workload, the clustering-based approach can handle 78.2% memory overload situations in one-to-many memory sharing, while it handles 28.1% memory overload situations in many-to-many memory sharing. The mean-based FFD approach in a typical 3% additional workload can handle 97.6% of memory overload situations in one-to-many memory sharing, while it handles 95.9% of memory overload situations in many-to-many memory sharing. In addition, the clustering-based approach has a higher non-interrupt rate of maintaining memory sharing connection than the mean-based FFD most the time.

This chapter has shown the benefits of using memory sharing in cloud data centres. Although the clustering-based approach has less management overhead, which manages less lender PM candidates than mean-based FFD approach, the performance has huge room for improvement. This issue is investigated and tackled in Chapter 5.

Chapter 5

Block Processing of Memory Sharing

Improvement of memory utilization of a cluster by enabling instant processing of memory sharing has been proven in the previous chapter. However, proposed approaches and benchmark approaches are far away from maximum improvement. For clustering-based instant processing of memory sharing, the rate of successful pairing lender PM and borrower drops under 80% when adding extra 3% or more additional workloads in the one-to-many memory sharing model. Such a rate is even lower in the many-to-many memory sharing model, which is always under 60% from experiments with extra 2% to 5% workloads. In contrast, in the benchmark approach, mean-based FFD has reasonable performance. The performance results with mean-based FFD shows that such a rate drops under 80% only on extra 4% workload and extra 5% workload, no matter whether running on a one-to-many memory sharing model or a many-to-many model. Although the clustering-based approach has a higher rate of avoiding memory overload of the lender PM during memory sharing than mean-based FFD most of the time, while keeping a low overhead on management, there is still a large room for performance improvement. Therefore, another memory sharing framework is demanded for achieving such improvement.

The block processing framework of memory sharing with an optimization algorithm is introduced in this chapter. Instead of processing each memory sharing request one-by-one like instant processing of memory sharing, the block processing framework processes multiple of memory sharing requests at a time. A high rate of success in pairing the lender PM and the borrower PM under medium and high extra workloads PM is expected for block processing of memory sharing. Moreover, the rate of non-interrupt in maintaining memory sharing between lender PM and borrower PM is expected to be fair.

In the block processing of memory sharing, it demands minimizing memory usage of lender PM candidates which need to share out memory with known sizes while not exceeding the capacity of each lender PM candidate. This problem is known as NP-hard; hence numerous heuristic procedures for its resolution have been suggested. In this chapter, the optimization algorithm used for block processing of memory sharing is an evolutionary algorithm, genetic algorithm (GA). Genetic algorithms are randomized research algorithms which have been developed with the aim of mimicking the mechanics of natural selection and natural genetics. Genetic algorithms function on string structures, which evolve over time, according to the survival rule of the most capable using a random but structured information exchange.

Overall, this chapter makes the following main contributions:

1) A memory sharing framework is designed for block processing of sharing memory resources between many borrower PMs and one lender PM (one-to-many);

2) A variant of the block processing framework of memory sharing, which supports sharing memory resources between many borrower PMs and many lender PMs (many-to-many);

3) A genetic-algorithm based control algorithm for planning multiple lender PMs and multiple borrower PMs as a whole;

4) The block processing framework of memory sharing is experimentally evaluated in terms of its simulated improvement on resource utilization of the cloud data centre.

The rest of the chapter is organized as follows: Section 5.2 illustrates the architecture of the block processing framework of memory sharing, where the global controller is the most important component in the architecture. The algorithm of the global controller is detailed in Section 5.3. Section 5.4 shows the design of the lender PM planning algorithm which is GA-based. The experiments are evaluated and discussed in Section 5.5. Finally, Section 5.6 concludes the chapter.

5.1 Problem Formulation

The optimization goal of the proposed memory sharing framework is to raise the rate of successful pairing of the lender PM and the borrower PM, while keeping a high rate of successfully

maintaining memory sharing. It means that memory usage of each PM in the cluster is expected to be as low as possible with memory sharing enabled. This section formulates a memory usage model to calculate used memory of a PM during memory sharing and formulates an evaluation model for block processing. The evaluation model indicates overall memory usage of the cluster and is used to evaluate results of the optimization problem.

5.1.1 Memory Usage Model in Memory Sharing

Memory information is retrieved via a low-level system call to *kernel/proc/meminfo*, which provides the total amount of usable memory M_t, the amount of memory unused by the system M_f, the amount of cache memory M_c, and the amount of memory used for file buffers M_b. Thus, the amount of used memory M_u is calculated by the following equation:

$$M_u = M_t - M_f - M_c - M_b \tag{5.1}$$

In the proposed memory sharing framework, the memory resource of a PM can be shared out in order to be used by a remote PM. Thus, Equation (5.1) is extended for memory usage calculation in the memory sharing framework. The amount of used memory for a memory sharing enabled PM $M_u^{(mshr)}$ is used to check if a PM has sufficient memory resource for fulfilling the memory sharing request by sharing out its spare memory resource. It is calculated by following equation:

$$M_u^{(mshr)} = M_u + M_{shared} + M_{req} \tag{5.2}$$

In Equation (5.2), M_u represents size of memory the PM itself used, which is calculated by Equation (5.1). M_{shared} represents size of memory which has already been shared out. It can be zero value if the PM has not shared out its spare memory resource. M_{req} represents the size of the memory resource that will be shared out in this process of memory sharing. If a PM has shared out a large amount of its memory resource (M_{shared}) and has a request to share out further memory resource (M_{req}), the $M_u^{(mshr)}$ may be larger than M_t. It means further memory sharing on this PM is not feasible.

5.1.2 Quantification Model for Impact of Memory Sharing

An impact quantification model is designed to evaluate the fitness of the block processing result. In the block processing of memory sharing, multiple memory borrowing requests are processed at a time. Thus, there is a list of $M_u^{(mshr)}$ to represent memory usages for all lender PM candidates.

In the quantification model for the impact of memory sharing, the results of block processing are evaluated based on distance between used memory $M_u^{(mshr)}$ and the lower threshold T_{lower}. The distance is calculated by following equation:

$$M_d = \frac{M_u^{(mshr)}}{M_t} - T_{lower} \tag{5.3}$$

M_d is calculated by using normalized used memory after memory sharing minus the lower threshold because $M_u^{(mshr)}$ represents the amount of memory while threshold represents a percentage. A constraint of this model is that M_d should not be positive. Memory overload will likely occur after sharing out the memory resource, if M_d is positive. Therefore, a distance factor k is introduced for adding penalty to calculation. Calculation of k is shown in Algorithm 10.

The total distance for evaluating the result of block processing is formulated as follows:

$$\mathbb{F} = \sum_{n=1}^{N_l} M_{nd} \times k \tag{5.4}$$

In Equation (5.4), N_l refers to the number of lender PM candidates, and k is the distance factor. The equation sums up M_d for all lender PM candidates, where M_{nd} refers to the M_d for the nth lender PM candidate. The distance factor is used to avoid producing the same result because a quite high M_{nd} and a quite low M_{nd} counteract each other in the sum result.

Algorithm 10 is used to produce the value of distance factor k. k is set at a higher value than default value 1 if the normalized size of used memory is higher than the lower threshold (line 1). Otherwise, k remains on default value 1 (line 4).

Algorithm 10: Distance Factor Calculation for Fitness Function

Input : M_{nd}
Output : k

if M_{nd} *is positive* **then**
  Set $k \leftarrow M_{nd} + 1$;
else
  Set $k \leftarrow 1$;
Return k;

5.1.3 Constrained Optimization

In the block processing of memory sharing, the lender PM planning is responsible for selecting a lender PM for each memory borrowing request in order to let the borrower PM know where to borrow memory resource. It aims to minimize the impact on lender PMs of sharing out memory resource. The impact is quantified by calculating the distance between the used memory and the threshold with a penalty of over-threshold. Therefore, the lender PM planning is formulated as the following constrained optimization problem with respect to the set of memory borrowing requests $R = \bigcup_{n=1}^{N_b} R_n$:

$$\begin{cases} \min\limits_{R} \mathbb{F} = \sum\limits_{n=1}^{N_l} (\dfrac{M_{nu}^{(mshr)}}{M_t} - T_{lower}) \times k \\ \text{s.t. } 0 \leq M_{nu}^{(mshr)} \leq M_t \\ \quad 0 \leq N_l \leq N_m \end{cases} \tag{5.5}$$

where $M_{nu}^{(mshr)}$ represents the amount of used memory for nth of lender PM candidates after planned memory sharing activation. A constraint is that a lender PM should not become memory overloaded, which is represented as an amount of the lender PM's used memory during memory sharing cannot be higher than its memory capacity. Another constraint is that lender PM candidates are part of or all PMs in the cloud data centre, which can be represented as that the number of lender PM candidates should be smaller than or equal to the number of all PMs.

5.2 Architecture of Block Processing Framework

The framework architecture of block processing of memory sharing has similar design as the instant processing of memory sharing framework described in Chapter 4. A global controller,

which performs as the controller, runs on a dedicated PM or the same PM with a cluster manager or hypervisor. Local controller is run on each node PM, which performs as the agent role of the controller-agent model.

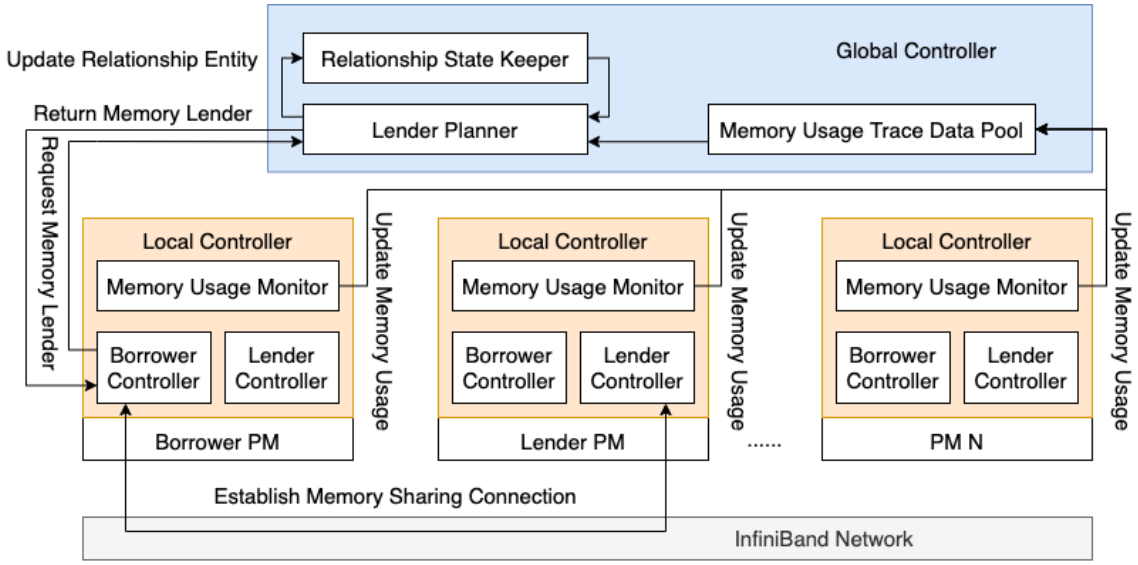


Figure 5.1: Architecture of Block Processing Framework of Memory Sharing

The detailed framework architecture of block processing of memory sharing is illustrated in Figure 5.1. The local controller is identical in both instant processing and block processing of memory sharing framework, while the global controller has several differences.

Memory Usage Data Pool stores only the most recent memory usage data of each PM because the lender planner does not require historic memory usage trace data for lender PM selection. It means that every time a new memory usage data is updated from the memory usage monitor of each local controller, the memory usage trace data pool overwrites the existing data.

Lender Planner allows the global controller to find the lender PM for the borrower PM with consideration of the optimization on resource provisioning of the whole cluster. The genetic algorithm, an evolutionary algorithm, is applied in the lender planner for making the optimized lender selection. In addition, the lender planner selects lender PMs for borrower PMs all at once, while the lender selector in instant processing of memory sharing (Figure 4.5) chooses one lender PM for one borrower PM at a time.

5.3 Global Controller Algorithm

This section describes algorithms for two types of global controller: one-to-many memory sharing controller and many-to-many memory sharing controller.

5.3.1 One-to-many Memory Sharing Controller

In a one-to-many controller of the block processing framework of memory sharing, memory borrowing requests are handled together by the lender PM planning algorithm. Depending on the number of borrowing requests to be handled, block processing of memory sharing may be efficient for handling large numbers of borrowing requests at once, while increased overhead may result from handling just a few borrowing requests at a time.

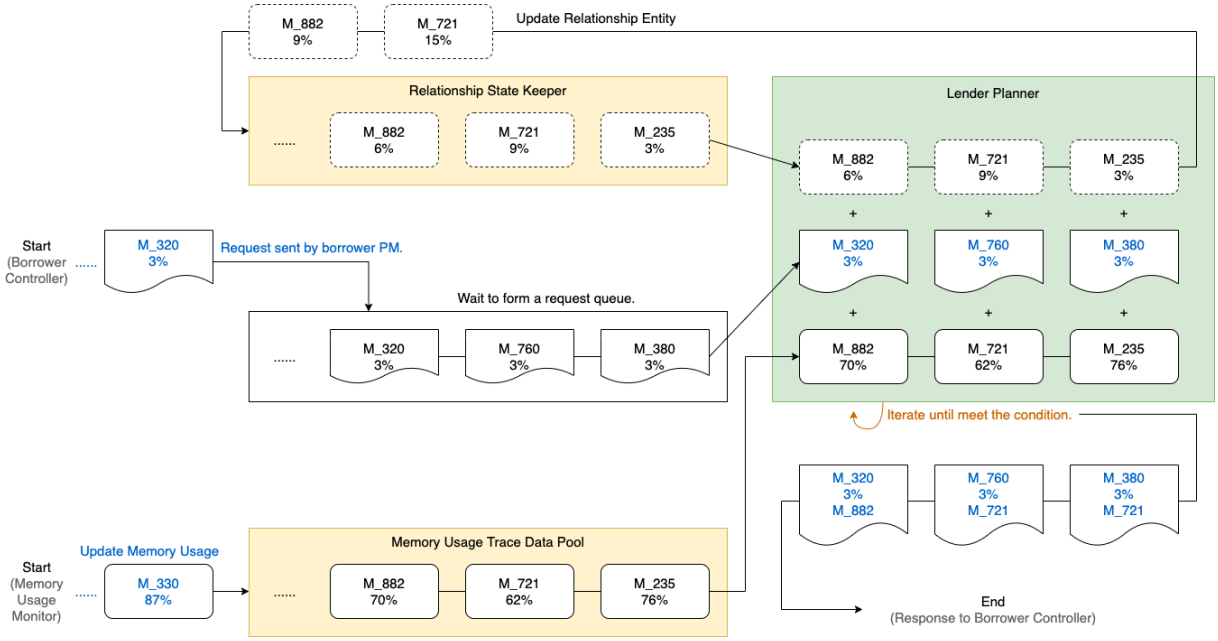


Figure 5.2: Process of One-to-Many Memory Sharing Controller in Block Processing

Figure 5.2 illustrates the process of the one-to-many memory sharing controller. Each memory borrowing request sent by the borrower PM is not handled as soon as it arrives the global controller. Instead, the global controller waits a tiny moment and queues the memory borrowing requests and handles them together. Hence, a request queue is maintained to temporarily hold borrowing requests. In Figure 5.2, borrowing requests sent from the borrower PM M_760 and M_380 are waiting until a request from borrower PM M_320 arrives. They are then passed to the lender planner.

During the lender PM planning process, the number of selected lender PM candidates may not be equal to the number of memory borrowing requests. The same lender PM may be selected for various memory borrowing requests when it is feasible. This benefits from block processing of the lender PM planning. In Figure 5.2's example, lender PM M_721 lends its spare memory resource for both borrower PM M_760 and M_380.

Algorithm 11: One-to-Many Control Algorithm for Lender Planning

Input : Memory usage trace data.
Output : Selected Lender PM Plan

```
1 Run the GA to get the calculated lender PM plan;
2 for each PM in lender PM plan of GA result do
3     Calculate M_s by Equation (4.1);
4     if M_s is positive then
5         Add this PM to selected lender PM plan;
6 Update borrower-lender relationship entry to Relationship State Keeper;
7 Return selected lender PM Plan;
```

The control algorithm for one-to-many lender planning is shown in Algorithm 11. It starts running the GA computation to obtain an optimized result (line 1). However, the result of the GA computation does not guarantee each lender PM has sufficient spare memory resource to share with the borrower PM, according to the computation goal of the GA. It is necessary to walk through all selected lender PMs to examine them against the threshold described in the previous chapter, and as shown in lines 2-5. The feasible lender PMs are added to the selected lender PM plan (line 5).

5.3.2 Many-to-many Memory Sharing Controller

Many-to-many memory sharing of block processing model has two steps more than the block processing of the one-to-many memory sharing controller: the borrowing request split and result grouping.

The borrowing request is split into multiple sub-requests as tasks, ahead of arriving at the wait queue. The memory size of each task could affect the performance of memory sharing. The tiny memory size of each task means there can be a lot of tasks for one borrowing request, which, in the worst case, can result in many lender PMs being selected for memory sharing. In the process demonstration in Figure 5.3, the borrower PM M_320 requests 3% normalized size

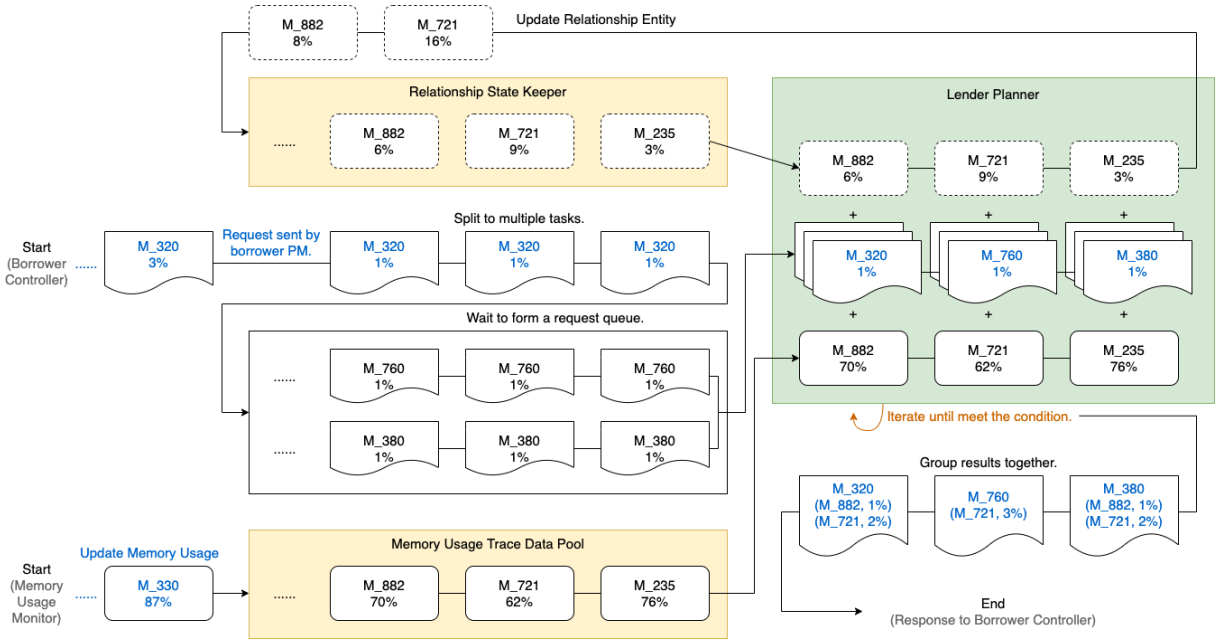


Figure 5.3: Process of Many-to-Many Memory Sharing Controller in Block Processing

of the remote memory resource. Such a request is split into three tasks in the split procedure, where each task involves 1% normalized size of the remote memory resource.

Result grouping concludes the result of the GA computation to form a single response for the borrower PM because the result of the GA computation only records selected lender PMs for each task instead of each borrower PM. If there is no result grouping, the global controller may respond to a borrower PM multiple times. There will be no problem if the lender PM is distinct in these responses. However, multiple responses may indicate the same lender PM as the worst case, which results in avoidable overhead. An example in Figure 5.3 is borrower PM M_760, which is indicated by the global controller borrowing 3% of memory resource from lender PM M_721, although the maximum number of lender PMs in case of borrower PM M_760 is three.

There is another reason for doing result grouping, which is to verify that the request of borrower PM is fulfilled. According to the fitness function of the GA computation, it is not guaranteed that every selected lender PM has satisfied the memory sharing requirement. The selected lender PM may become memory overloaded after sharing the memory resource to the borrower PM. Such a case can be found during result grouping by examining the memory usage of the post-sharing lender PM against the threshold. The borrower PM will be notified that the request cannot be satisfied if one or more its selected lender PMs has memory overloading risk.

Algorithm 12 shows the procedures of lender planning on the controller of many-to-many

Algorithm 12: Many-to-Many Control Algorithm for Lender Selection

```
   Input  : Memory usage trace data.
   Output : Selected Lender PM Plan
1  Split M_req to multiple lender selection tasks;
2  Run the GA to get the calculated lender PM plan;
3  for each PM in lender PM plan of GA result do
4      Calculate M_s by Equation (4.1);
5      if M_s is positive then
6          Add this PM to available lender PM plan;
7  for each request in original memory borrowing requests do
8      if this borrowing request is fulfilled in available lender PM plan then
9          Add lender PMs for this task to selected lender PM plan;
10 Update borrower-lender relationship entry to Relationship State Keeper;
11 Return selected lender PM Plan;
```

memory sharing. It starts from the borrowing request split (line 1) and GA computation (line 2). The result of the GA computation is stored as the calculated lender PM plan. Then, the feasible lender PM is picked from calculated lender PM plan to the available lender PM plan (lines 3-6). Finally, every memory borrowing request is verified with the available lender PM plan (lines 7-9). Lender PMs fulfilling memory borrowing requests are added to the lender PM plan as the response of the lender planner.

5.3.3 Capability and Feasibility of Global Controller

The prerequisite for the proposed global controller is the existence of one or more solutions. The solution always exists if the cluster runs normally where memory usage of the cluster is below its capacity. However, there will be no solution for the global controller if workloads demand more memory resources than the cluster's capacity. We assume this case would never happen because one or more additional PMs are booted up automatically for overcoming such issues, according to resource provisioning policy of the cluster supervisor system.

The global controller may find a wrong solution, although there is always a solution. The global controller may choose a lender PM for memory sharing. However, it is possible that memory sharing connections are rejected by the selected lender PMs, because of insufficient spare memory resources. The reason is the global controller initializes by randomly picking a candidate lender PM for each memory borrowing request. It then optimizes the initial memory

sharing plan with the object of minimizing memory usage of each PM in the cluster, without any verification of the feasibility of each selected lender PM. Thus, how to guarantee the proposed global controller can always find a correct solution becomes a future work.

5.4 GA-based Lender PM Planning Algorithm

This section describes a block processing method for the plan lender PM, where the planning algorithm uses the GA to gain optimized provisioning.

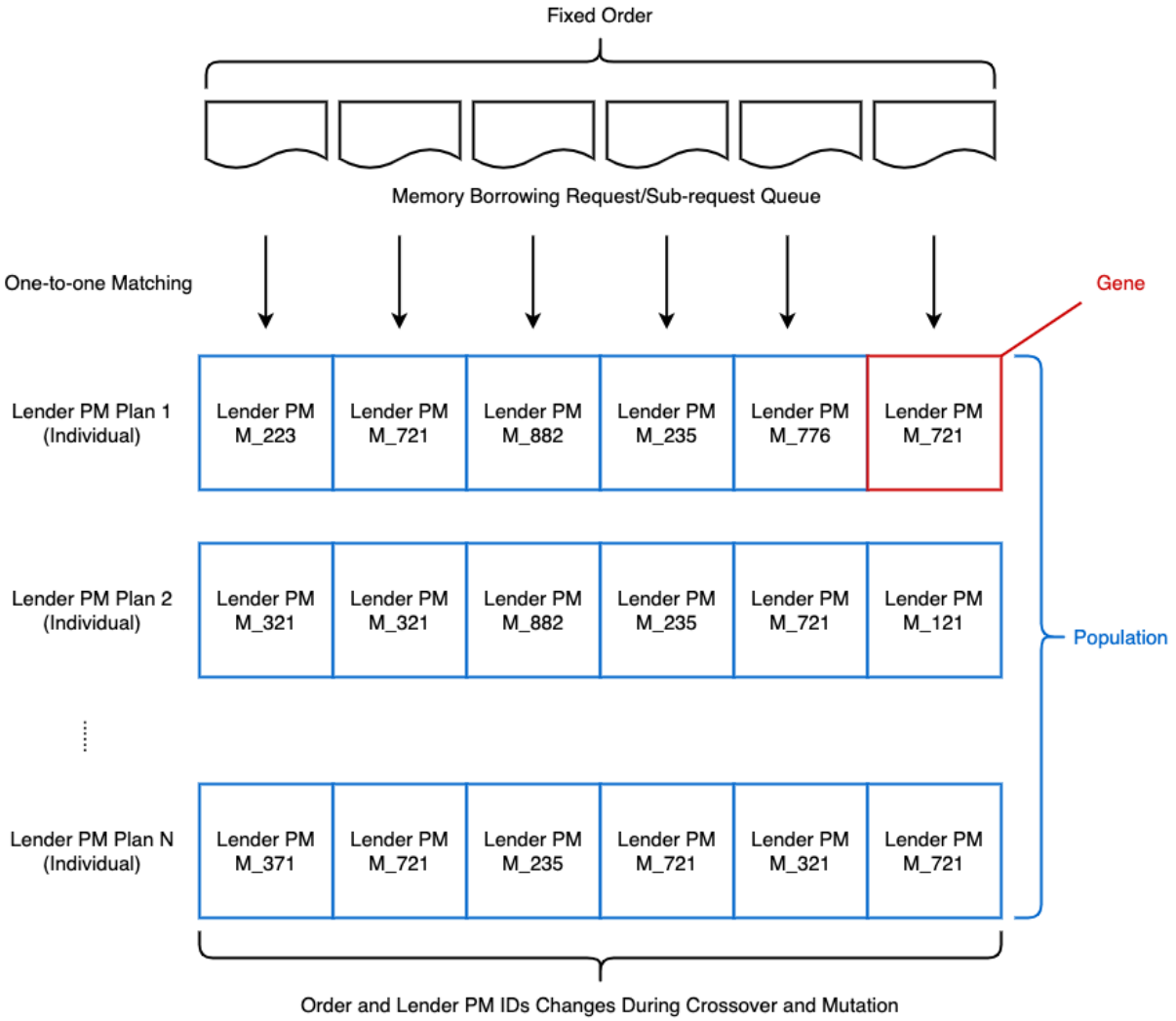


Figure 5.4: Process of Many-to-Many Memory Sharing Controller

The data structure of the GA is shown in Figure 5.4. The GA receives a list of memory borrowing request/sub-requests and list of lender PM candidates as parameters. The list of memory borrowing request/sub-requests comes in fixed order, which should not be changed during the computation. For each individual of the population, a gene representing a lender PM

ID comes from the given lender PM candidates, which is also one-to-one matched with an item in the list of memory borrowing request/sub-requests. During crossover and mutation of the GA, the order of genes and content of genes in individuals may change.

There are two variants of GA computation: brute-force based GA and clustering based GA. Brute-force based runs without knowledge of the historic trace data of memory usage. It considers all node PMs as lender PM candidates. Clustering based GA runs on top of the clustering method introduced in Section 4.5.2, which only considers clustering results as lender PM candidates.

Algorithm 13: GA Fitness Function Computation in Lender Selection of Memory Sharing

Input : A borrower-lender PM matching plan, memory usage of each PM, size of shared memory for Each Lender PM
Output : The plan's fitness value

```
1 for PM in the given plan do
2     Add the size of shared memory to the memory usage of this PM; and add the size of
        memory, which will be shared in the given plan, to the memory usage of this PM;
        – Equation (5.2)
3     Calculate the distance between threshold and memory usage of this PM; – Equation
        (5.3)
4     Calculate distance factor k by Algorithm 10;
5     Add the distance multiply by k to this plan's fitness value;
6 Return This plan's fitness value 𝔽;
```

The optimization goal of the GA is to minimize memory usage of all the lender PM candidates. The fitness function is shown in Algorithm 13. The fitness value represents the total distance between the selected lender PMs and the higher threshold. To calculate such a distance for each lender PM, the size of memory which has been shared to borrower PMs and the size of memory resource which is planned for sharing are added to the current memory usage of the lender PM (line 2). Then, the distance between the lender PM and the threshold is calculated and added to the fitness value (lines 3-5).

5.5 Experimental Studies and Evaluation

This section describes the experiments to investigate improvement of performance of memory sharing, by comparing the block processing framework of memory sharing and instant processing of memory sharing framework introduced in Chapter 4.

5.5.1 Experimental Setup

Both one-to-many and many-to-many block processing of memory sharing are conducted in the experiments. There are four types of experiments, which consist of four extra workloads: 2%, 3%, 4%, and 5%. Extra workload is added to the memory usage of each node PM of the cluster. The queue period is one second, which means memory borrowing requests are added into the queue and are handled every second by the block processing of memory sharing. In addition, similar to the previous chapter, Alibaba's trace data is used for the experiments. The simulation is implemented in Go programming language.

a Lender PM Candidate

Table 5.1: Number of lender PM candidates.

Additional Workload	Clustering	Mean-based FFD	GA-Clustering	GA
2%	469	4023	469	4023
3%	469	4023	469	4023
4%	469	4023	469	4023
5%	469	4023	469	4023

Table 5.1 shows the number of lender PM candidates set for proposed methods and benchmark methods. The clustering-based method (Clustering) has 469 lender PM candidates scattered in 7 groups. The proposed GA based on top of clustering (GA-Clustering) has 469 lender PM candidates, which is the same as clustering-based methods. However, these 469 lender PM candidates are not sorted because it is unnecessary for GA computation. Others consider all 4023 PMs as lender PM candidates.

b GA Setting

There are three settings for GA: crossover function, mutation function, and condition of termination. The crossover function used in experiments is Generalized N-point Crossover (GNX) proposed by Radcliffe and Surry [1995]. The mutation function permutes two genes at random 2 times. GA is terminated if one of the following conditions is met: the best fitness value is unchanged over 200 times; or computation takes more than 10 seconds.

5.5.2 Experimental Evaluation

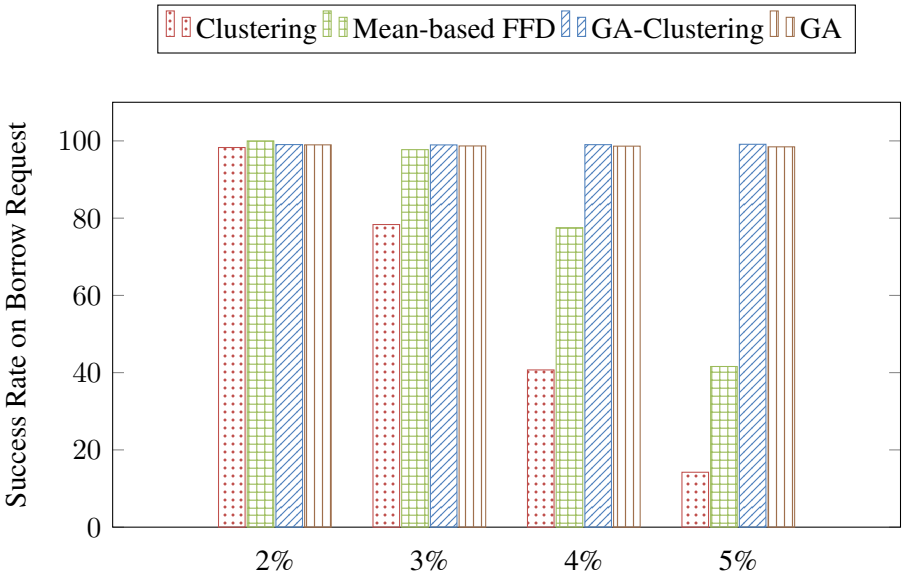
Both one-to-many memory sharing and many-to-many memory sharing are performed and evaluated in the experiments. Several numbers are recorded for analysis: numbers about memory sharing requested by the borrower PM, numbers about maintaining memory sharing connections between the borrower PM and the lender PM, and the number of distinct lender PMs used in the experiment. All experiments are repeated ten times, and recorded numbers are the average value of ten records.

a Success Rate on Pairing Lender PM to Borrower PM

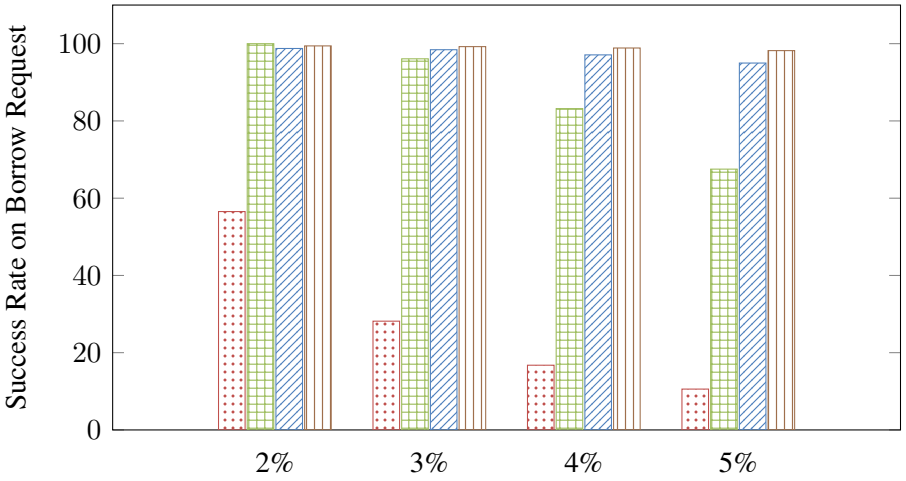
The success rate on pairing lender PM with borrower PM in many-to-many memory sharing is shown in Figure 5.5. It is calculated by dividing the count of successfully pairing by the total count of memory borrowing requested by borrower PM. Failure of pairing lender PM with borrower PM represents the global controller responds infeasible lender PMs to memory borrowing requests. Figure 5.5a shows the success rate for one-to-many memory sharing, while many-to-many is shown in Figure 5.5b. Both sub-figures are composed by the results on four levels of additional workloads, 2%, 3%, 4%, and 5%, with four different control algorithms to select the lender PM. Two of the control algorithms are from the instant processing of memory sharing framework introduced in the previous chapter, while the other two are GA based.

Both GA-based approaches have a similar performance. They also have a similar performance as mean-based FFD in 2% and 3% additional workloads for both one-to-many and many-to-many memory sharing. However, their success rates are maintained in 4% and 5% additional workloads, which is different from the performance of mean-based FFD and clustering. The reason is GA is able to find an optimized solution while FFD cannot.

Comparison between two GA based approaches shows that, GA-clustering has a slightly higher success rate than GA in one-to-many memory sharing, while GA has a better performance than GA-clustering in many-to-many memory sharing. This indicates two conclusions: 1) the small number of lender PM candidate, 469 out of 4023, does affect the success rate of pairing lender PMs with borrower PMs, no matter whether the clustering approach or GA-clustering approach. 2) GA-clustering proves the quality of clustering results. FFD may be inappropriate for the clustering approach, which has a low success rate.



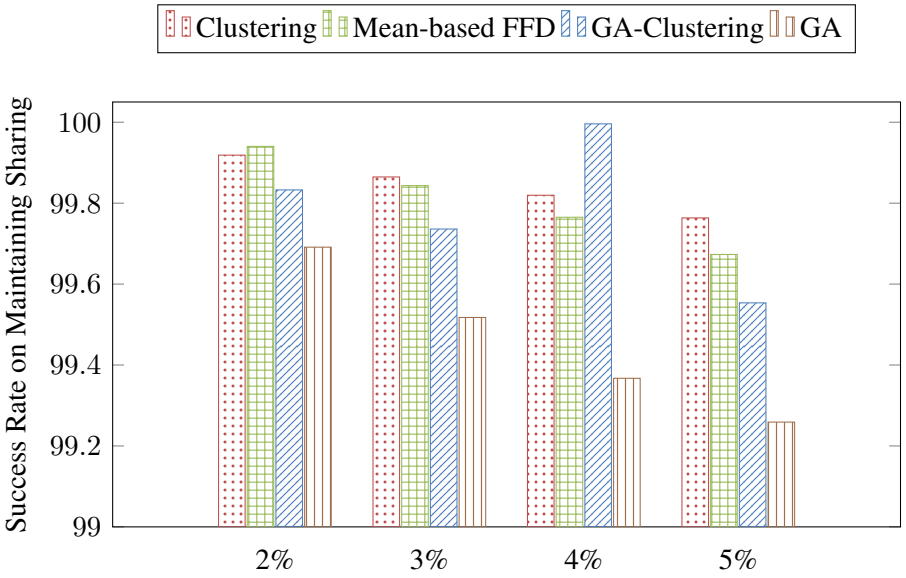
(a) Percentage of Successful Pairing Lender PM With Borrower PM in One-to-Many Memory Sharing



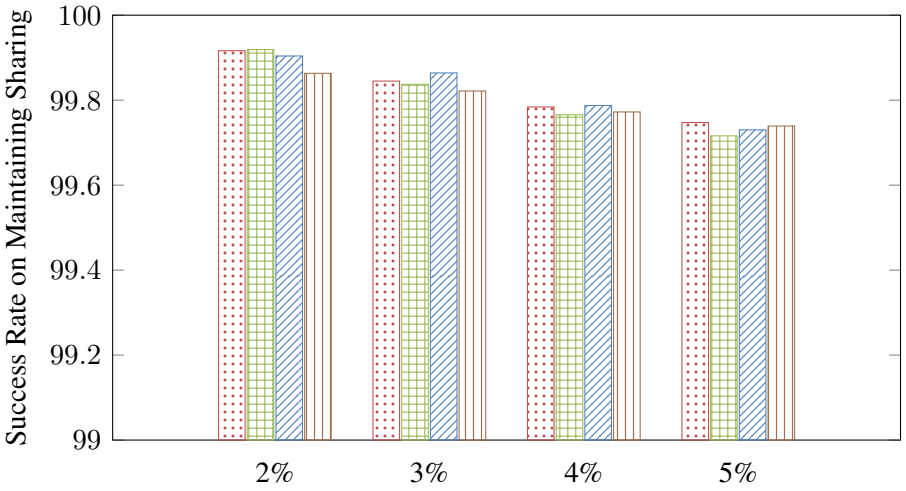
(b) Percentage of Successful Pairing Lender PM With Borrower PM in Many-to-Many Memory Sharing

Figure 5.5: Percentage of Successful Pairing Lender PM With Borrower PM in Block Processing of Memory Sharing

b Success Rate for Maintaining Memory Sharing Connection



(a) Percentage of Keeping Memory Sharing Connection in One-to-Many Memory Sharing



(b) Percentage of Keeping Memory Sharing Connection in Many-to-Many Memory Sharing

Figure 5.6: Percentage of Keeping Memory Sharing Connection in Block Processing Framework of Memory Sharing

Figure 5.6 demonstrates the success rate of keeping the memory sharing connection in one-to-many and many-to-many memory sharing. The rate is calculated on percentage of duration with which the lender PM can keep fulfilling the memory sharing requirement. Such duration is interrupted if the lender PM becomes memory overloaded during the connection of memory sharing with the borrower PM. In this experiment, all approaches have results over 99.2%, which may be considered as acceptable performance.

GA-clustering has a higher success rate than GA except for the 5% additional workload in many-to-many memory sharing. It has a lower success rate than mean-based FFD most of the time in one-to-many memory sharing. However, in the 4% additional workload of one-to-many memory sharing, its success rate reaches 99.90%. On the other hand, GA-clustering has a higher success rate than mean-based FFD in many-to-many memory sharing, except for the 2% additional workload.

Since the computation of the GA approach does not use historic knowledge nor prediction, it is possible that GA selects an inappropriate PM as the lender PM. An inappropriate PM can be the PM which has high standard deviation in time-series memory usage data. Memory usage of such a PM changes rapidly, which could cause numerous memory overloads for the lender PM.

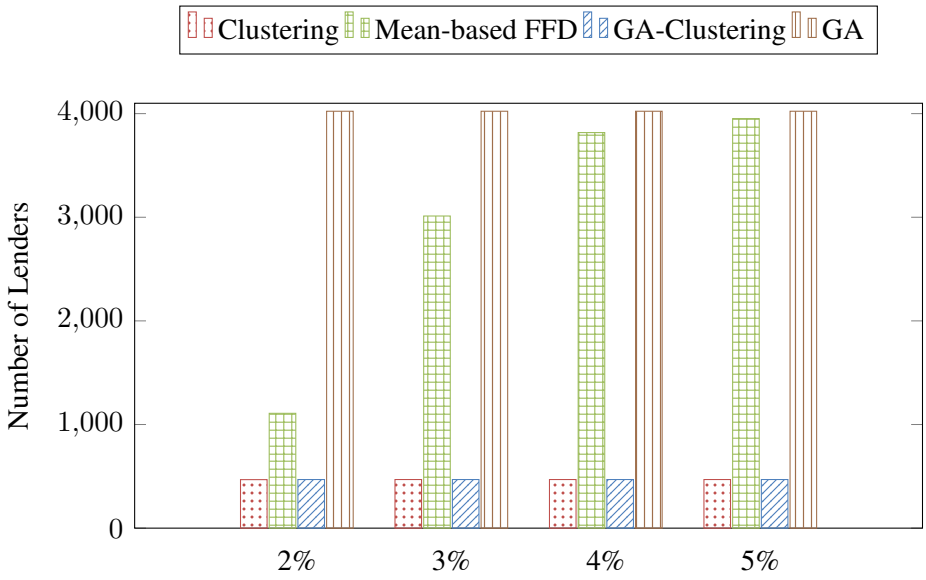
c Distinct Lender Count

Figure 5.7 illustrates the number of lender PMs used in the experiment. If a PM behaves as a lender PM multiple times, it is counted only once. The experimental result of clustering shows that both GA-clustering and GA has used all lender PM candidates during the experiment.

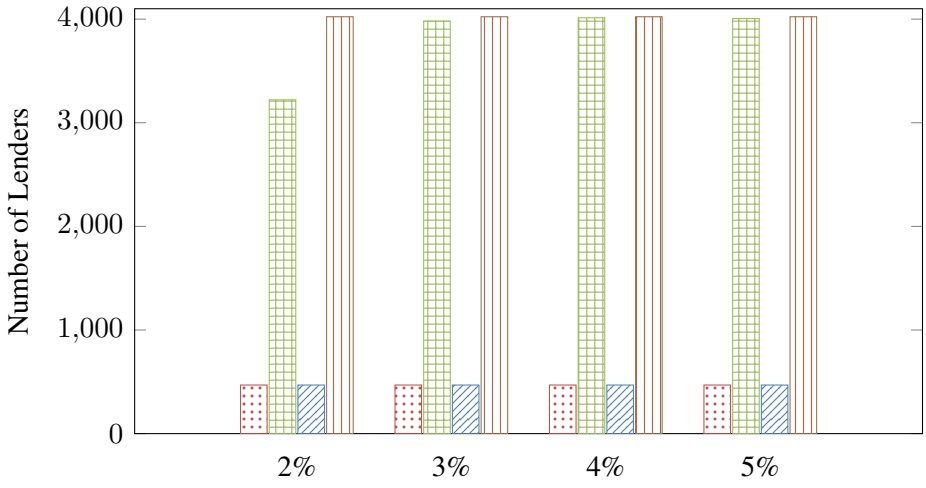
By considering the experiment results discussed above, it is believed that GA-clustering has a better performance than GA because it uses around one tenth of GA's lender PM candidates; 469 out of 4023. Moreover, it is assumed that a smaller number of lender PMs indicates a lower overhead of managing memory sharing, including establishing and disconnecting memory sharing connections between lender PMs and borrower PMs. The previous discussion also confirms that more distinct lender PMs can increase the possibility of interrupt on maintaining memory sharing connection.

5.6 Summary of Chapter

In this chapter, a block processing framework of memory sharing is presented for handling memory overload occurring on multiple PMs at the same time. The framework allows a lender to share its spare memory resource for multiple borrowers (one-to-many memory sharing), and allows multiple lenders to share their spare memory resource to multiple borrowers (many-to-many memory sharing). A genetic-algorithm based control algorithm is presented for planning



(a) Distinct Lender Count in One-to-Many Memory Sharing



(b) Distinct Lender Count in Many-to-Many Memory Sharing

Figure 5.7: Distinct Lender Count in Block Processing Framework of Memory Sharing

multiple lender PMs and multiple borrower PMs together, for both one-to-many and many-to-many memory sharing. In addition, the genetic-algorithm based control algorithm works for both full lender PM candidates 4023 PMs, and 469 PMs selected by the clustering method described in the previous chapter.

Experimental studies are conducted in simulation using Alibaba's public trace data for evaluating how much improvement our clustering-based memory sharing framework can bring to real world data centres. With a 5% increment of memory usage on each PM, our system can handle 99.1% of memory overload situations by utilizing spare memory resource of 11.7% PMs selected by clustering. The difference between the proposed GA approach and the GA-clustering approach is that only one tenth of PMs are considered as lender PM candidates for the GA-clustering approach, while the GA approach considers all PMs of the cluster as lender PM candidates in the experiment. Experimental results show that for a typical 3% additional workload, the GA-clustering approach can handle 98.7% memory overload situations in one-to-many memory sharing, and 98.3% memory overload situations in many-to-many memory sharing. The GA approach in typical 3% additional workload can handle 98.2% memory overload situations in one-to-many memory sharing, and 99.1% memory overload situations in many-to-many memory sharing. In addition, comparing to other levels of additional workloads, both GA-clustering and GA approaches show the similar performance. Usually, GA-clustering has a higher non-interrupt rate of maintaining memory sharing connection than GA, but it is still lower than the clustering-based approach presented in the previous chapter.

The next chapter concludes this thesis by summarizing the proposed memory sharing system and frameworks for one-to-many and many-to-many memory sharing. Suggestions for future research direction are also presented in the next chapter.

Chapter 6

Conclusions and Recommendations

This chapter summarizes the research development of a memory sharing framework for handling memory overload of PMs in cloud data centres. It outlines the main contributions and findings, discusses research limitations and highlights future research directions.

6.1 Summary of the Research

While the early emphasis of cloud data centre is on elastically providing computing resources to end users and computation tasks, cloud vendors are now more and more interested in over-committing their computing resources to maximize the resource utilization and minimize operating costs. In principle, the risks of over-committing computing resources can be hedged if each of the VMs on PMs consumes only a small portion of requested computing resources instead of continually taking full allocated computing resources. PMs can run out of computing resource capacity if such a hedge fails, and results in CPU or memory overload. To handle this problem on PMs, cloud vendors must live migrate VMs from CPU or memory overloaded PMs to underutilized PMs. However, in the worst case, live VM migration can cause cascading overloads in over-committed cloud data centres.

This research has proposed a systematic memory sharing framework for handling memory overload of PMs in over-committed cloud data centres. Memory sharing becomes possible since RDMA can provide superfast data transmission with low latency. Feasibility studies and three progressive levels of memory sharing have been presented in this thesis.

Firstly, feasibility on three levels of memory sharing has been examined. The three levels

are: one-to-one memory sharing which has one lender PM and one borrower PM, one-to-many memory sharing which has one lender PM, and many borrower PMs, and many-to-many memory sharing which has many lender PMs and many borrower PMs. Feasibility studies indicate memory sharing can be implemented by utilizing RDMA to swap unused memory pages to memory space in a remote PM, which forms a swap-based and remote block device-based memory sharing approach. Based on that, three levels of memory sharing have been also confirmed as feasible.

A one-to-one memory sharing system has been designed and physically implemented, according to the feasibility studies. It integrates a mechanism of threshold-based memory overload detection, and a unified control algorithm for sharing memory automatically. In this memory sharing system, a PM relies on the threshold-based memory overload detection to switch its role between memory lender PM and memory borrower PM on the fly. When a PM becomes a memory borrower PM, it can automatically negotiate and establish a memory sharing connection from a lender PM in order to borrow memory resource. When memory overload disappears and memory underutilization occurs, a PM can switch to a memory lender PM. A lender PM accepts a memory sharing request from a borrower PM only if it is feasible, when it has enough spare memory resource for lending.

One-to-many and many-to-many memory sharing have been jointly solved because they have very many similarities. An instant processing and a block processing for memory sharing have been proposed for both one-to-many and many-to-many memory sharing. The instant processing of memory sharing refers to immediately select the lender PM for the borrower PM. The block processing of memory sharing refers to a block of memory borrowing requests are being processed at a time.

The instant processing of memory sharing has been designed for high non-interrupt rate on maintaining the memory sharing connection, while minimizing the number of distinct lender PMs used. A high non-interrupt rate means high QoS of memory sharing and low risk of cascading overloads on the memory resource. Minimizing the number of distinct lender PMs used can reduce the overhead of managing memory sharing connection, where the overhead is produced by the memory sharing system detailed in Chapter 4. Two lender PM selection algorithms have been proposed, where one is clustering-based, and the other is mean-based FFD algorithm. A profile-guided clustering algorithm with tolerance of missing values has

been designed for filtering unwanted lender PM candidates while increasing the non-interrupt rate. Mean-based FFD is designed as the standard solution for instant lender selection since FFD is quick and efficient enough for such scenario.

The block processing of memory sharing, on the other hand, is not required to select the lender PM for borrower PM immediately. Instead, it waits to form a memory borrowing request queue which contains memory sharing requirements from multiple borrower PMs. It then processes these requests together as a whole in one algorithm. This design enables optimization for maximizing the rate of successful pairing of the lender PM and the borrower PM. The block processing of memory sharing has been designed with GA and proposed fitness function for the memory sharing issue. It has been designed with two different approaches: one is applying GA on all PMs of the cluster as lender PM candidates, while the other is applying GA on top of clustering results. Experimental studies have shown that GA on top of clustering achieves all goals. It not only maintains an acceptable high non-interrupt rate, but also has a high success rate of pairing lender PMs and borrower PMs for memory sharing, while it only requires one tenth of the cluster as lender PM candidates.

6.2 Limitations and Future Recommendations

The main limitation of this research is that the proposed approach requires more comprehensive practical testing. Firstly, the one-to-many and many-to-many memory sharing framework was not physically implemented and evaluated in real clusters nor data centres. Moreover, the latency to set up the memory sharing connection is instant in one-to-one memory sharing, while it is not clear on its impact to real running clusters and data centres. Secondly, only the Alibaba's cluster trace data was used in the simulation because other mainstream cloud vendors, such as AWS and GCP, had not yet disclosed memory usage information of PMs in their data centres. In future research, trace data from other clusters will be collected and used for a more comprehensive simulation-based evaluation and the proposed one-to-many and many-to-many memory sharing frameworks will be implemented and evaluated in real-world clusters.

Another limitation is that the memory size for sharing between the lender PM and borrower PM is fixed in one-to-many and many-to-many memory sharing. It means all borrower PM requests the same size of remote memory resource in the simulation. The size of remote

memory resource is equal to the additional workload. However, it is not necessary to borrow that size of remote memory because actual required memory resource can be less than the additional workload. In future research, one-to-many and many-to-many memory sharing will be improved to allow each borrower PM to request remote memory resources which is only sufficient for handling memory overload.

One of future works is about improved GA for memory sharing. In the proposed solution, GA may respond infeasible lender PMs for memory sharing because of its optimization strategy and the nature of GA. Thus, an improvement on GA is required in future work to allow the proposed method can be guaranteed to find a correct solution. In addition, it is needed to investigate the maximum latency can be accepted by block processing of memory sharing. Such latency includes queuing time, GA computation time, and time cost on memory sharing connection.

A future recommendation for memory sharing is to distribute inactive memory pages to main memory space of other PMs which have spare memory resource, instead of preparing an in-memory virtual block device and remotely sharing it as swap space to the memory overloaded PM. This may further increase utilization of the memory resource of the cluster, although the overhead of managing memory sharing can be higher than with the block device-based approach. Another future recommendation is to investigate how memory sharing and live VM migration work together to comprehensively handle the resource overloading problem in cloud.

Regarding network security, this research has not yet investigated any safety issue may be involved in memory sharing. For example, data has the chance to be leaked since memory pages are swapped and stored on remote PMs during memory sharing. This deserves to be researched in the future.

Literature Cited

Abdel-Basset, M., Manogaran, G., Abdel-Fatah, L., and Mirjalili, S. (2018). An improved nature inspired meta-heuristic algorithm for 1-D bin packing problems. *Personal and Ubiquitous Computing*, 22(5):1117–1132.

Abdul-Minaam, D. S., Al-Mutairi, W. M. E. S., Awad, M. A., and El-Ashmawi, W. H. (2020). An adaptive fitness-dependent optimizer for the one-dimensional bin packing problem. *IEEE Access*, 8:97959–97974.

Ahn, S., Kim, J., Lim, E., and Kang, S. (2018). Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high performance computing. *IEEE Access*, 6:26493–26504.

Alibaba Open Source (2018). Alibaba cluster trace program. Retrieved from `https://github.com/alibaba/clusterdata`.

Amit, N., Tsafrir, D., and Schuster, A. (2014). VSwapper: A memory swapper for virtualized environments. *ACM SIGPLAN Notices*, 49(4):349–366.

Ashouraei, M., Khezr, S. N., Benlamri, R., and Navimipour, N. J. (2018). A new SLA-aware load balancing method in the cloud using an improved parallel task scheduling algorithm. In *2018 IEEE 6th international conference on future internet of things and cloud*, pages 71–76, Barcelona, Spain. IEEE.

Baset, S. A., Wang, L., and Tang, C. (2012). Towards an understanding of oversubscription in cloud. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, pages 1–6, San Jose, CA, USA. USENIX Association.

Beloglazov, A. and Buyya, R. (2013). Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1366–1379.

Berndt, S., Jansen, K., and Klein, K.-M. (2020). Fully dynamic bin packing revisited. *Mathematical Programming*, 179(1):109–155.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Notices*, 41(10):169–190.

Brugger, B., Doerner, K. F., Hartl, R. F., and Reimann, M. (2004). AntPacking – an ant colony optimization approach for the one-dimensional bin packing problem. In *Evolutionary Computation in Combinatorial Optimization*, pages 41–50, Coimbra, Portugal. Springer.

Cao, R., Tang, Z., Li, K., and Li, K. (2021). HMGOWM: A hybrid decision mechanism for automating migration of virtual machines. *IEEE Transactions on Services Computing*, 14(5):1397–1410.

Cao, W. and Liu, L. (2018). Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud. In *IEEE International Conference on Big Data*, pages 191–200, Seattle, WA, USA. IEEE.

Cao, W. and Liu, L. (2020). Hierarchical orchestration of disaggregated memory. *IEEE Transactions on Computers*, 69(6):844–855.

Cardona, O. (2019). Towards hyperscale high performance computing with RDMA. Retrieved from `https://pc.nanog.org/static/published/meetings/NANOG76/1999/20190612_Cardona_Towards_Hyperscale_High_v1.pdf`.

Che, J., He, Q., Gao, Q., and Huang, D. (2008). Performance measuring and comparing of virtual machine monitors. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 2, pages 381–386, Shanghai, China. IEEE.

Choi, H.-H., Kim, K., and Kang, D.-J. (2017). Performance evaluation of a remote block device with high-speed cluster interconnects. In *Proceedings of the 8th International Conference on Computer Modeling and Simulation*, pages 84–88, Canberra, ACT, Australia. ACM.

Chou, L.-D., Yang, Y.-T., Hong, Y.-M., Hu, J.-K., and Jean, B. (2014). A genetic-based load balancing algorithm in OpenFlow network. In *Advanced Technologies, Embedded and Multimedia for Human-centric Computing*, pages 411–417, Taipei, Taiwan. Springer.

Choudhary, A., Govil, M. C., Singh, G., Awasthi, L. K., Pilli, E. S., and Kapil, D. (2017). A critical survey of live virtual machine migration techniques. *Journal of Cloud Computing*, 6(23):1–41.

Cusumano, M. (2010). Cloud computing and SaaS as new computing platforms. *Communications of the ACM*, 53(4):27–29.

Dasgupta, K., Mandal, B., Dutta, P., Mandal, J. K., and Dam, S. (2013). A genetic algorithm (GA) based load balancing strategy for cloud computing. *Procedia Technology*, 10:340–347.

Deshpande, U., Wang, B., Haque, S., Hines, M., and Gopalan, K. (2010). MemX: Virtualization of cluster-wide memory. In *39th International Conference on Parallel Processing*, pages 663–672, San Diego, CA, USA. IEEE.

Ding, Z. (2018). vdsm: Distributed shared memory in virtualized environments. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 1112–1115, Beijing, China. IEEE.

Dokeroglu, T. and Cosar, A. (2014). Optimization of one-dimensional bin packing problem with island parallel grouping genetic algorithms. *Computers & Industrial Engineering*, 75:176–186.

Dorigo, M., Birattari, M., and Stutzle, T. (2006). Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39.

Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. (2014). FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, pages 401–414, Seattle, WA, USA. USENIX Association.

Effatparvar, M. and Garshasbi, M. (2014). A genetic algorithm for static load balancing in parallel heterogeneous systems. *Procedia - Social and Behavioral Sciences*, 129:358–364.

Elmeleegy, K., Olston, C., and Reed, B. (2014). SpongeFiles: Mitigating data skew in Mapreduce using distributed memory. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 551–562, Snowbird, UT, USA. ACM.

Everman, B., Rajendran, N., Li, X., and Zong, Z. (2021). Improving the cost efficiency of large-scale cloud systems running hybrid workloads - a case study of Alibaba cluster traces. *Sustainable Computing: Informatics and Systems*, 30:100528.

Feldkord, B., Feldotto, M., Gupta, A., Guruganesh, G., Kumar, A., Riechers, S., and Wajc, D. (2018). Fully-dynamic bin packing with little repacking. In *45th International Colloquium on Automata, Languages, and Programming*, volume 107, pages 51:1–51:24, Prague, Czech Republic. Schloss Dagstuhl.

Feng, H., Ni, H., Zhao, R., and Zhu, X. (2020). An enhanced grasshopper optimization algorithm to the bin packing problem. *Journal of Control Science and Engineering*, 2020:1–19.

Gao, R. and Wu, J. (2015). Dynamic load balancing strategy for cloud computing with ant colony optimization. *Future Internet*, 7(4):465–483.

Golchi, M. M., Saraeian, S., and Heydari, M. (2019). A hybrid of firefly and improved particle swarm optimization algorithms for load balancing in cloud environments: Performance evaluation. *Computer Networks*, 162:106860.

Goldberg, D. E. and Holland, J. H. (1988). Genetic algorithms and machine learning. *Machine Learning*, 3(2):95–99.

Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., and Shin, K. G. (2017). Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pages 649–667, Boston, MA, USA. USENIX Association.

Guo, C., Wu, H., Deng, Z., Soni, G., Ye, J., Padhye, J., and Lipshteyn, M. (2016). RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215, Florianopolis, Brazil. ACM.

Guo, J., Chang, Z., Wang, S., Ding, H., Feng, Y., Mao, L., and Bao, Y. (2019). Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, Phoenix, AZ, USA. ACM.

Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. (2010). Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93.

Gupta, J. N. D. and Ho, J. C. (1999). A new heuristic algorithm for the one-dimensional bin-packing problem. *Production Planning & Control*, 10(6):598–603.

Guz, Z., Li, H. H., Shayesteh, A., and Balakrishnan, V. (2017). Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR'17, pages 1–9, Haifa, Israel. ACM.

Haouari, M. and Serairi, M. (2009). Heuristics for the variable sized bin-packing problem. *Computers & Operations Research*, 36(10):2877–2884.

Hines, M. R. and Gopalan, K. (2007). MemX: Supporting large memory workloads in Xen virtual machines. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, pages 1–8, Reno, NV, USA. ACM.

Hines, M. R., Gordon, A., Silva, M., Da Silva, D., Ryu, K., and Ben-Yehuda, M. (2011). Applications know best: Performance-driven memory overcommit with ginkgo. In *IEEE Third International Conference on Cloud Computing Technology and Science*, pages 130–137, Athens, Greece. IEEE.

Hoeflin, D. and Reeser, P. (2012). Quantifying the performance impact of overbooking virtualized resources. In *2012 IEEE International Conference on Communications (ICC)*, pages 5523–5527, Ottawa, ON, Canada. IEEE.

Huang, W., Gao, Q., Liu, J., and Panda, D. K. (2007). High performance virtual machine migration with rdma over modern interconnects. In *2007 IEEE International Conference on Cluster Computing*, pages 11–20.

IBM (2019). IaaS, PaaS and SaaS - IBM cloud service models. Retrieved from `https://www.ibm.com/au-en/cloud/learn/iaas-paas-saas`.

Ivkovic, Z. and Lloyd, E. (1998). Fully dynamic algorithms for bin packing: Being (mostly) myopic helps. *SIAM Journal on Computing*, 28(2):574–611.

Ji, S., Li, M. D., Ji, N., and Li, B. (2018). An online virtual machine placement algorithm in an over-committed cloud. In *IEEE International Conference on Cloud Engineering*, pages 106–112, Orlando, FL, USA. IEEE.

Jiang, L., Wang, K., and Zhao, D. (2018). Davram: Distributed virtual memory in user space. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 344–347, Washington, DC, USA. IEEE.

Johnson, D. S. (1974). Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314.

Kaur, A., Kaur, B., Singh, P., Devgan, M. S., and Toor, H. K. (2020). Load balancing optimization based on deep learning approach in cloud environment. *International Journal of Information Technology and Computer Science*, 12(3):8–18.

Kavis, M. J. (2014). *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. Wiley, 1st edition.

Kennedy, J. and Eberhart, R. (1995). Particle swarm optimization. In *Proceedings of International Conference on Neural Networks*, volume 4, pages 1942–1948, Perth, WA, Australia. IEEE.

Kissel, E. and Swany, M. (2016). Photon: Remote memory access middleware for high-performance runtime systems. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 1736–1743, Chicago, IL, USA. IEEE.

Kocharyan, A., Ekane, B., Teabe, B., Tran, G. S., Astsatryan, H., and Hagimont, D. (2020). A remote memory sharing system for virtualized computing infrastructures. *IEEE Transactions on Cloud Computing*. DOI:10.1109/TCC.2020.3018089.

Koh, K., Kim, K., Jeon, S., and Huh, J. (2019). Disaggregated cloud memory with elastic block management. *IEEE Transactions on Computers*, 68(1):39–52.

Kröger, B. (1995). Guillotineable bin packing: A genetic approach. *European Journal of Operational Research*, 84(3):645–661. Cutting and Packing.

Levine, J. and Ducatelle, F. (2004). Ant colony optimization and local search for bin packing and cutting stock problems. *Journal of the Operational Research Society*, 55(7):705–716.

Li, G., Chen, W., and Xiang, Y. (2021). Zweilous: A decoupled and flexible memory management framework. *IEEE Transactions on Computers*, 70(9):1350–1362.

Li, G. and Wu, Z. (2019). Ant colony optimization task scheduling algorithm for SWIM based on load balancing. *Future Internet*, 11(4).

Li, K. (1988). IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 94–101, University Park, PA, USA. Pennsylvania State University Press.

Li, K., Xu, G., Zhao, G., Dong, Y., and Wang, D. (2011). Cloud task scheduling based on load balancing ant colony optimization. In *6th Annual Chinagrid Conference*, pages 3–9, Liaoning, China. IEEE.

Li, Y., Tang, X., and Cai, W. (2016). Dynamic bin packing for on-demand cloud resource allocation. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):157–170.

Liang, S., Noronha, R., and Panda, D. K. (2005). Swapping to remote memory over InfiniBand: An approach using a high performance network block device. In *IEEE International Conference on Cluster Computing*, pages 1–10, Burlington, MA, USA. IEEE.

Lim, K., Turner, Y., Santos, J. R., AuYoung, A., Chang, J., Ranganathan, P., and Wenisch, T. F. (2012). System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, New Orleans, LA, USA. IEEE.

Liu, D., Tan, K., Huang, S., Goh, C., and Ho, W. (2008). On solving multiobjective bin packing problems using evolutionary particle swarm optimization. *European Journal of Operational Research*, 190(2):357–382.

Liu, H., Liu, R., Liao, X., Jin, H., He, B., and Zhang, Y. (2020). Object-Level memory allocation and migration in hybrid memory systems. *IEEE Transactions on Computers*, 69(9):1401–1413.

Liu, J., Wu, J., and Panda, D. K. (2004). High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198.

Liu, L., Qiu, Z., and Dong, J. (2017). A load balancing algorithm for virtual machines scheduling in cloud computing. In *2017 9th International Conference on Modelling, Identification and Control*, pages 471–475, Kunming, China. IEEE.

Makasarwala, H. A. and Hazari, P. (2016). Using genetic algorithm for load balancing in cloud computing. In *2016 8th International Conference on Electronics, Computers and Artificial Intelligence*, pages 1–6, Ploiesti, Romania. IEEE.

Martello, S., Pisinger, D., and Vigo, D. (2000). The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267.

Mauch, V., Kunze, M., and Hillenbrand, M. (2013). High performance cloud computing. *Future Generation Computer Systems*, 29(6):1408–1416.

Mellanox Technologies (2015). RDMA aware networks programming user manual. Retrieved from `https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`.

Mellanox Technologies (2019). What is RDMA? Retrieved from `https://community.mellanox.com/s/article/what-is-rdma-x`.

Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E., and Pendarakis, D. (2010). Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing*, pages 11–20, Washington, DC, USA. ACM.

Microsoft Azure (2019a). What is IaaS? infrastructure as a service. Retrieved from `https://azure.microsoft.com/en-au/overview/what-is-iaas/`.

Microsoft Azure (2019b). What is PaaS? platform as a service. Retrieved from `https://azure.microsoft.com/en-us/overview/what-is-paas/`.

Mishra, M., Das, A., Kulkarni, P., and Sahoo, A. (2012). Dynamic resource management using virtual machine migrations. *IEEE Communications Magazine*, 50(9):34–40.

Moltó, G., Caballer, M., and de Alfonso, C. (2016). Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems*, 56:1–10.

Mulahuwaish, A., Korbel, S., and Qolomany, B. (2022). Improving datacenter utilization through containerized service-based architecture. *Journal of Cloud Computing*, 11(1):1–29.

Newhall, T., Lehman-Borer, E. R., and Marks, B. (2016). Nswap2L: Transparently managing heterogeneous cluster storage resources for fast swapping. In *Proceedings of the Second International Symposium on Memory Systems*, pages 50–61, Alexandria, VA, USA. ACM.

Nishant, K., Sharma, P., Krishna, V., Gupta, C., Singh, K. P., Nitin, and Rastogi, R. (2012). Load balancing of nodes in cloud using ant colony optimization. In *UKSim 14th International Conference on Computer Modelling and Simulation*, pages 3–8, Cambridge, UK. IEEE.

Oura, H., Midorikawa, H., Kitagawa, K., and Kai, M. (2017). Design and evaluation of page-swap protocols for a remote memory paging system. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 1–8, Victoria, BC, Canada. IEEE.

Pan, K. and Chen, J. (2015). Load balancing in cloud computing environment based on an improved particle swarm optimization. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 595–598, Beijing, China. IEEE.

Patel, N. and Patel, H. (2018). An approach for detection of overloaded host to consolidate workload in cloud datacenter. *International Journal of Grid and High Performance Computing*, 10(2):59–69.

Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. (2013). Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–184, Davis, CA, USA. ACM.

Radcliffe, N. J. and Surry, P. D. (1995). Fitness variance of formae and performance prediction. *Foundations of Genetic Algorithms*, 3:51–72.

Ramezani, F., Lu, J., and Hussain, F. K. (2014). Task-based system load balancing in cloud computing using particle swarm optimization. *International journal of parallel programming*, 42(5):739–754.

Rao, R. and Iyengar, S. (1994). Bin-packing by simulated annealing. *Computers & Mathematics with Applications*, 27(5):71–82.

Roy, S., Kumar, R., and Prvulovic, M. (2001). Improving system performance with compressed memory. In *Proceedings 15th International Parallel and Distributed Processing Symposium*, pages 1–7, San Francisco, CA, USA. IEEE.

Ruan, Z., Schwarzkopf, M., Aguilera, M. K., and Belay, A. (2020). AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 315–332, Banff, Alberta, Canada. USENIX Association.

Seiden, S. S. (2002). On the online bin packing problem. *Journal of the ACM*, 49(5):640–671.

Seshadri, V., Pekhimenko, G., Ruwase, O., Mutlu, O., Gibbons, P. B., Kozuch, M. A., Mowry, T. C., and Chilimbi, T. (2015). Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. *ACM SIGARCH Computer Architecture News*, 43(3S):79–91.

Sridhar, R., Chandrasekaran, M., Sriramya, C., and Page, T. (2017). Optimization of heterogeneous bin packing using adaptive genetic algorithm. *IOP Conference Series: Materials Science and Engineering*, 183:012026.

Srinuan, P., Yuan, X., and Tzeng, N. (2020). Cooperative memory expansion via OS kernel support for networked computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2650–2667.

Svärd, P., Hudzia, B., Walsh, S., Tordsson, J., and Elmroth, E. (2015). Principles and performance characteristics of algorithms for live vm migration. *ACM SIGOPS Operating Systems Review*, 49(1):142–155.

Talaat, F. M., Saraya, M. S., Saleh, A. I., Ali, H. A., and Ali, S. H. (2020). A load balancing and optimization strategy (LBOS) using reinforcement learning in fog computing environment. *Journal of Ambient Intelligence and Humanized Computing*, 11(11):4951–4966.

Tan, Y., Wang, B., Yan, Z., Srisa-an, W., Chen, X., and Liu, D. (2020). APMigration: Improving performance of hybrid memory performance via an adaptive page migration method. *IEEE Transactions on Parallel and Distributed Systems*, 31(2):266–278.

Tomás, L. and Tordsson, J. (2013). Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, pages 1–10, Miami, Florida, USA. ACM.

Tsitsipas, A., Hauser, C. B., Domaschka, J., and Wesner, S. (2017). Towards usage-based dynamic overbooking in IaaS clouds. In *Economics of Grids, Clouds, Systems, and Services*, pages 263–274, Athens, Greece. Springer.

Visalakshi, P. and Sivanandam, S. (2009). Dynamic task scheduling with load balancing using hybrid particle swarm optimization. *International Journal of Open Problems in Computer Science and Mathematics*, 2(3):475–488.

VMware (2019). Migrating virtual machines. Retrieved from `https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vsphere.vcenterhost.doc/GUID-FE2B516E-7366-4978-B75C-64BF0AC676EB.html`.

Voorsluys, W., Broberg, J., Venugopal, S., and Buyya, R. (2009). Cost of virtual machine live migration in clouds: A performance evaluation. In *Cloud Computing*, pages 254–265, Beijing, China. Springer.

Waldspurger, C. A. (2003). Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194.

Wang, B. and Li, J. (2016). Load balancing task scheduling based on multi-population genetic algorithm in cloud computing. In *2016 35th Chinese Control Conference*, pages 5261–5266, Chengdu, China. IEEE.

Wang, Z., Wang, X., Hou, F., Luo, Y., and Wang, Z. (2016). Dynamic memory balancing for virtualization. *ACM Transactions on Architecture and Code Optimization*, 13(1).

Williams, D., Jamjoom, H., Liu, Y.-H., and Weatherspoon, H. (2011). Overdriver: Handling memory overload in an oversubscribed cloud. *ACM SIGPLAN Notices*, 46(7):205–216.

Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. (2007). Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, pages 229–242, Cambridge, MA, USA. USENIX Association.

Wu, R., Huang, L., and Zhou, H. (2019). RHKV: An RDMA and HTM friendly keyvalue store for dataintensive computing. *Future Generation Computer Systems*, 92:162–177.

Xiao, Z., Song, W., and Chen, Q. (2013). Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117.

Xu, P., He, G., Li, Z., and Zhang, Z. (2018). An efficient load balancing algorithm for virtual machine allocation based on ant colony optimization. *International Journal of Distributed Sensor Networks*, 14(12):1550147718793799.

Xue, H., Kim, K. T., and Youn, H. Y. (2019). Dynamic load balancing of software-defined networking based on genetic-ant colony optimization. *Sensors*, 19(2).

Xue, Y. and Zhu, Z. (2021). Hybrid flow table installation: Optimizing remote placements of flow tables on servers to enhance PDP switches for in-network computing. *IEEE Transactions on Network and Service Management*, 18(1):429–440.

Yadav, M. P., Akarte, H. A., and Yadav, D. K. (2021). *An Approach for Load Balancing Through Genetic Algorithm*, chapter 36, pages 511–524. Wiley, 1st edition.

Yadav, Y. and Rama Krishna, C. (2019). Real-time resource monitoring approach for detection of hotspot for virtual machine migration. In *Applications of Artificial Intelligence Techniques in Engineering*, pages 555–563, New Delhi, India. Springer.

Ye, D., Pavuluri, A., Waldspurger, C. A., Tsang, B., Rychlik, B., and Woo, S. (2008). Prototyping a hybrid main memory using a virtual machine monitor. In *IEEE International Conference on Computer Design*, pages 272–279, Lake Tahoe, CA, USA. IEEE.

Zhang, F., Liu, G., Fu, X., and Yahyapour, R. (2018). A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Communications Surveys & Tutorials*, 20(2):1206–1243.

Zhang, Q., Liu, L., Su, G., and Iyengar, A. (2017a). MemFlex: A shared memory swapper for high performance VM execution. *IEEE Transactions on Computers*, 66(9):1645–1652.

Zhang, W., Xie, H., and Hsu, C. (2017b). Automatic memory control of multiple virtual machines on a consolidated server. *IEEE Transactions on Cloud Computing*, 5(1):2–14.

Zhang, X., Shae, Z., Zheng, S., and Jamjoom, H. (2012). Virtual machine migration in an over-committed cloud. In *IEEE Network Operations and Management Symposium*, pages 196–203, Maui, HI, USA. IEEE.

Zhao, W., Wang, Z., and Luo, Y. (2009). Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review*, 43(3):37–47.

Zhou, K., Liu, W., Tang, K., Huang, P., and He, X. (2018). Alleviating memory refresh overhead via data compression for high performance and energy efficiency. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1469–1483.

Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. (2004). Dynamic tracking of page miss ratio curve for memory management. *ACM SIGPLAN Notices*, 39(11):177–188.