Efficient Heuristics for Virtual Machine Migration in Data Centers

Khodayar Jeirroodi

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

July 2019

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Khodayar Jeirroodi

Entitled: Efficient Heuristics for Virtual Machine Migration in Data Centers

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

Master of Computer Science

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

Signed by the final examining committee:

_____ Chair
Dr. Weiyi Shang

_____ Examiner
Dr. Dhrubajyoti Goswami

_____ Examiner
Dr. Tse-Hsun Chen

_____ Thesis Supervisor(s)
Dr. Brigitte Jaumard

_____

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean,

Date _____

ABSTRACT

Efficient Heuristics for Virtual Machine Migration in Data Centers

Khodayar Jeirroodi

Live migration of virtual machines is one of the essential virtualization technologies which enables the consolidation and load balancing in cloud data centers without interrupting the services. Main goals for optimizing a single virtual machine live migration is to minimize migration time, transferred data and downtime. Planning multiple live migrations in a data center has an essential impact on feasibility of consolidation and quality of services during migrations, however, optimizing parallel VM migrations has been studies less. Minimizing makespan (total migration time) while reducing energy and service quality degradation caused by using datacenter resources for migrations, are the main objectives of the problem. One of the issues in planning multiple live migrations is to detect and consider migrations order dependency constraints and possible deadlocks caused by lack of enough free resources in servers during the process. In the literature, exact mathematical models are not scalable and heuristics are not optimal and they don't consider the quality of service and energy efficiency of migration process when resources are restricted.

In this work we propose a heuristic algorithm for scheduling the migration of virtual machines in a data center in order to minimize makespan (total migration time) and solve the conflicts (deadlocks) caused by limitation of resources with minimum cost and quality degradation.

Acknowledgements

I would like to express my deepest appreciation and gratitude to my supervisor Professor Jaumard for all the things I learned from her during my study. Her guidance, support and patience greatly helped and inspired me for this study.
Also I would like to thank my wife Anahita and my parents for their continuous support, encouragement and their kindness.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cloud computing and virtualization technologies are growing very fast and big companies such as Google and Amazon are developing and providing cloud-based services for their customers. Virtual machines act like images of real machines in data centers and provide resources for applications. With vast grow in demand and complexity of the applications, cloud service providers need to assure quality of services for their customers, dedicate required physical resources and network bandwidth to follow Service Level Agreements (SLA). The number of virtual machines and average workloads change during the time. For example, some virtual machine might be shut down and new virtual machines may be required, also services run in each virtual machine and their load (memory, processor or bandwidth) will change. Therefore it is required to occasionally relocate virtual machines for efficient resources usage and ensuring the quality of service. The virtual technologies management system in a data center, moves virtual machines between the servers (physical machines) on the basis of a virtual machine placement algorithm, for several purposes such as balancing the load between physical machines to ensure more optimized utilization of resources and quality of service [1].

Furthermore for optimizing utilization of resources such as data center power consumption and site cooling systems, one way is to reduce the data center energy waste by the server consolidation technique, where virtual machines are packed into fewer number of servers (physical machines). Likewise for other reasons (e.g., customer preferences, failure or planned maintenance), we might need to change virtual machines placements in physical sources across the cloud. The techniques for finding the optimized placement of virtual machines are called server consolidation.

Live migration of virtual machines, allows datacenters to relocate the virtual machines while services they provide do not stop but for a tiny time between stopping the virtual machine on legacy location and starting its copy on the destination server. There are various studies to address optimization of virtual machine live migrations in different directions, such as memory migration optimization, power consumption of a single live virtual machine migration, migrations over WAN and optimizing live migration of multiple virtual machines [2].

Most studies try to optimize the live migration of a single virtual machine, but in real applications such as server consolidation in datacenters we need to migrate several virtual machines in a short period of time. Studies such as Onoue *et al.* [3] , Sun *et al.* [4] and Deshpande and Keahey [5] try to reduce the makespan which is the total time between start of the first migration and the time when the last virtual machine finishes its migration. Although reducing the makespan, helps quality of service for the users via reducing the time window during which users expect degradation of service, but considering the quality of service degradation caused by intermediate steps and order of migration can strongly affect the service quality and hence the objectives of the planned migrations (e.g., server consolidation), which is the main focus of this study.

1.2 Virtual machine migration

Server consolidation, in general tries to pack a number of virtual machines on a fewer number of servers to optimize resource utilization and reduce power consumption regarding server constraints and possible SLA requirement. The important feature that makes the server consolidation technique even more attractive is virtual machine live migration. With using virtual machine live migration, one can transfer a running virtual machine from a server to another one without considerable service downtime [1].

In live migration of a virtual machine, if we use the pre-copy technique, while the services are still running on the virtual machine on source server, all of the memory pages of virtual machine are transferred to the destination server. Based on the size of the memory and page dirtying rate, the process of copying memory pages might be repeated for dirtied memory pages until it reaches a threshold for the number of iterations or size of the dirtied page. Then the virtual machine on source server stops and the remaining dirties pages with CPU state will be transferred to the destination server and the virtual machine starts on the destination server [6]. The time between stopping virtual machine and restarting it on target server, depends on memory size, dirty page rate and bandwidth and resources of both servers and network, which can be as small as few tens of milliseconds.

There are several studies that solve the server consolidation and virtual machine optimized placement problems. Many of them use different approaches of bin packing problem. Some other use different methods for forecasting usage of physical resources (RAM, CPU, I/O) and according to thresholds of high and low usages find new placements for virtual machines assigned to those locations. One step of the problem which has been considered less, is moving virtual machines to new placements via a seamless virtual machines migration. Keeping services running during the migration (without considerable downtime) and finish the migrations in desired time with minimum service degradation is an important part of the server

consolidation process. The migration step can be considered along with the placement problem or as a separate one (where we assume new and legacy placements are given). In both cases finding order (scheduling) and migration plan for migrating virtual machines is not straightforward. Migration process needs resources (CPU, memory, etc.) in both source and destination. Also the amount of network bandwidth used for migrations has restrictions. If we prefer faster total transition time, we might confront SLA violations. Also long migration time leads to longer time for the current unfavorable state of placements which voids the optimization and extends the period of possible SLA violations and service degradation because of the migration process.

During the migration process, we might need to move virtual machine to a temporary server to solve a blocked situation, we call them temporary or transient migrations since later we must migrate them again to the original destination. Temporary migrations are costly, because they consume resources and degrade quality of service(s) running on those machines during migration.

1.3 Contribution

In this study we propose heuristics for virtual machine migration scheduling and planning algorithm which tries to find the optimal order of virtual machine migrations to reduce the makespan considering service level degradation cause by intermediate steps and temporary migrations. The final heuristic is evaluated by solving the migration scheduling problem for generated datasets and is compared with a heuristic algorithm and the results of two exact models in the literature. Also we compare results of the intermediate heuristics with the final heuristic as we gradually constructing our final heuristic.

1.4 Thesis plan

The thesis is organized as follows. Chapter 2 gives the background and detailed statement of the problem, the heuristic approach for scheduling virtual machine migrations. In Chapter 3 we review the previous studies. Chapter 4 gives the details of the heuristics algorithms proposed by this study. In Chapter 5 we have the numerical results and comparison between the approaches, and Chapter 6 concludes the study and introduces ideas for future work.

Chapter 2

Problem Statement

In this chapter we review the background of virtual machine consolidation and migration problems. In Section 2.1 we review the problem of optimized placement of virtual machines. The placement/consolidation problem can be studied alongside the migration problem, which is reviewed in Section 2.3. Background of optimizing virtual machine migrations (with given placements) which is the main focus of this study is introduced in Section 2.2

2.1 Virtual machine placement

The number of virtual machines and their average workloads change during the time. For example, some virtual machine might be shut down and new virtual machines may be required, also services run in each virtual machine and their load (memory, CPU, Bandwidth, etc.) might change and therefore it is profitable to regularly relocate virtual machines for efficient resources usage and ensuring the quality of the services running on virtual machines. The virtual technologies management system moves virtual machines between the data centers (physical machines) for several purposes such as load balancing between physical machines. Virtual machine placement is studied from different aspects and by using different methods.

Virtual machine placement problem

In this section, we review the concepts and methods used for virtual machine placement problem. In a datacenter with several servers (physical machines), each server contains multiple running virtual machines, for several reasons such as optimal placement, maintenance, etc we might need to have a new placement for the virtual machines. Finding new or optimized placements for all or part of the virtual machines is a problem every cloud service provider must manage regularly. This problem has been addressed in many studies with different approaches and domains. Frequent additions and removals of virtual machines beside changes in the load and resource requirement of each virtual machine, make the legacy placement of virtual machine in data centers non-optimized. There are many studies trying to find an optimized virtual machine placement with different objectives such as load balancing or reducing number of active data centers [1]. Some of the purposes of changing virtual machines placements are:

- Load balancing
- Server consolidation
- Service level agreement
- Hardware maintenance
- Maximizing reliability
- Optimizing network traffic

Constraints of virtual machine placement problem can be physical machine resources (CPU, memory, disk), network capacity, SLA, etc.
The output of this problem is a new placement for a subset of virtual machines. Some of the studies consider the migration of virtual machine to new placement combined with the placement problem. These studies are reviewed in Section 2.3

Virtual machine placement problem objectives

Regularly the strategy used for solving a virtual machine placement problem has objective and constraints similar to a bin packing problem (multi capacity bin packing). In a classic bin packing problem the aim is to place objects as dense as constraints allow in fewer number of containers, while in virtual machine placement the purposes of the new placement can be resource shortage or reducing energy consumption (active server energy consumption, cooling system, ...) maintenance and overhead costs of running physical machines. Bloglazov *et al.* [7] set upper and lower thresholds for total CPU utilization by virtual machines allocated on a server. If the CPU utilization of a server falls below the lower threshold, all of the virtual machines of the server are planned to migrate to new locations and if CPU utilization exceed the upper threshold some virtual machines must be migrated in order to prevent SLA violations occur due to the lack of available resources.
Another example of optimization objective that can be considered is network traffic. For example Meng *et al.* in [8] formulate the virtual machine placement based on the network traffic. In their study, virtual machines with large mutual bandwidth usage are assigned to the servers in close proximity to achieve better network scalability.
In Deng *et al.* work, [9], problem of minimizing SLA violations with finding the optimize placement of virtual machines, based on several parameters such as reliability of physical machine (server) is studied.

Virtual machine placement key problem constraints

The key constraints for assigning virtual machines to servers in a datacenter (physical machines) are the capacity of servers and network bandwidth. In several studies, this capacity is divided into Memory, CPU, Disk resource and server network bandwidth.
The above mentioned constraints are the key ones, other constraints such as hardware reliability, inter-VM performance degradation, cooling systems can also be

considered [1].

Onoue *et al.* [3] puts the constraints for the processors and memory size in a way that summation of memory and processor demands for virtual machines assigned to a server must not exceed its capacity for any of the resources.

Li *et al.* [10] add the constraint of network capacity with the same logic, the total network usage of all virtual machines running on any single server must not exceed the network capacity upper bound of that server.

Beside considering the resource optimization obtained from consolidating virtual machines on fewer servers, the possible performance degradation must be regarded. Although CPU, memory and I/O are split between virtual machines, shared caches and memory bandwidths can degrade performance significantly, it is also possible that if we put certain combination of virtual machines on a server, the contention decreases the performance and violates quality of service requirement, or longer task execution times in virtual machines result in more energy consumption than non-consolidated state. Roytman *et al.* [11] determine how much each virtual machine service quality is degraded when placed with different sets of virtual machines to be consolidated and identify which and how many virtual machines can be placed on a server such that required performance is maintained. For this purpose, they assume degradation of a virtual machine 'A' when it runs with virtual machine 'B' on the same server, and consider maximum tolerable degradation of each virtual machine as a constraint.

Reliability of servers has direct effect on virtual machine service SLA, several short term turning on and off a physical machine will reduce the lifetime and reliability of data center devices. Also packing more virtual machine into one server, increases its temperature which leads to higher risk of hardware failure and less reliability. Sansottera *et al.* [12] consider power, performance and reliability aspects simultaneously on their consolidation model with focus on performance and reliability requirements to guarantee average response time constraints for virtual machine services, in addition using an active/active sparing model for servers in which the data center uses one of the active servers in a round-robin fashion so as

to make the services more stable and reliable.

Legal mandates governing user data can result in constraints for the virtual machine processing those data, restricted physical location of data or data isolation, and co-location limitations can also affect virtual machine placement constraints [13].

Static or dynamic virtual machine replacement

Virtual machines replacement can be done statically or live (dynamically). In static approach mapping of virtual machines to servers is not changed over specific period of time. The load balance of each server is estimated based on the peak resource demand of virtual machines. At time intervals, the load balance and unusual changes will be detected and new virtual machine placements will be calculated. In dynamic virtual machine, the cloud computing management platforms monitors the workloads of the servers and network, and if resource utilization and demand of any of the virtual machines exceeds a threshold, placements of virtual machines will be optimized. Wolk *et al.* [14] compare static against dynamic consolidation method in a real data center experiment. Their study concludes static method leads to higher energy efficiency compared to dynamic allocation with only a modest level of overbooking. This is partly due to migration overheads and response time peaks caused by live migration (details are beyond the scope of this study).

Virtual machine placement SLA

Availability of services, service response time, resources and maximum downtime of a service indirectly affect the strategy by which we assign and manage the resources of virtual machines. Also constraints such as data location and shared infrastructure resources are the examples of constraints based of the defined Service Level Agreement (other than technical constraints) [13].

Virtual machine placement problem solving methods

Virtual machine consolidation problem can be mapped into a version of bin packing problem, (Lee *et al.* [15], cited in Varasteh and Goudarzi [1]), where virtual machines as objects must be packed into servers which are similar to bins. Resources of servers here are placement constraints. Methods for the exact solution, such as linear programming, dynamic programming and stochastic programming are NP-hard. For overcoming complexity of this problem, several heuristic methods are used. Among the heuristic methods greedy algorithm of First Fit Decreasing (FFD) is used more frequently, although FFD method has major limitation that forces the problem to be one dimensional and servers have the same resource capacity (Lee *et al.* [15], cited in Varasteh and Goudarzi [1]), but it has advantage of being simple and fast. Dosa [16] approved that number of bins found by first fit decreasing algorithm is not more than $11/9\ OPT + 6/9$ where OPT is the optimal number of bins found by an exact algorithm. Other than known bin packing algorithms, there are several studies that implement specific heuristics for solving virtual machine consolidation problem.

2.2 Virtual machine live migration

Live migration of a virtual machine is the process of moving a virtual machine with running service between locations (servers) without stopping the services during the process. This is done by moving the memory state of the running virtual machine from the source server to destination server while the VM is running on either source or destination. There are two main approaches for migrating virtual machines pre-copy and post-copy.

Pre-copy technique

This is the most general technique of moving a live virtual machine. In this technique the memory and processor state of a virtual machine is transferred in iterations from the source host to the destination host, whereas the virtual machine is still running on the source host. In the first iteration, the entire state of the virtual machine is transferred to the destination host. Changed pages of memory is resent to the destination in the next iteration if it has been changed (dirtied) until the remaining dirtied page memory can be transferred in a tiny stop and copy phase (which is negligible in service) or number of iterations reaches a threshold. In pre-copy approach the virtual machine keeps running on the source server until the migration to the destination server is done, so if the process is interrupted the state of the virtual machine will not be lost. Most of the hypervisors such as Virtual MachineWare, KVirtual Machine and Xen use this approach.[2]

Post-copy technique

In this approach first the processor state is transferred to the destination server and the services start working on destination server, then the memory of the stopped virtual machine in source will be transferred. If there is demand for a part of the memory during running of the service in destination server before all the memory pages are moved, it will be fetched from the source which is called page-fault. Although this approach results in shorter migration time but has drawbacks of service degradation due to accessing required memory via network also has risk of loosing the latest service state if the virtual machine in the destination fails.

Hybrid technique

In this approach, hypervisor first transfers a part of the memory from source to destination with pre-copy approach then starts the virtual machine in destination and continue transferring memory pages via post-copy approach.

Inter-datacenter virtual machine migration

Live migration of a virtual machine is relocating the virtual machine between two servers, these servers can be located in the same datacenter or different datacenters. The server consolidation problem principally is inside one datacenter [1], therefore the migration problem is also inside the same datacenter, which is likewise the domain of this study. Optimizing virtual machine live migration between datacenters mostly consists of routing and bandwidth assignment for migrations, for instance Ayoub *et al.* [17] propose an algorithm for efficient routing and bandwidth assignment for single and parallel virtual machine migrations between datacenters in an optical network.

Virtual machine live migration problem

In this section, we review the problem of migrating virtual machines from a current placement to a new placement. New placement is calculated based on the optimized assignment of virtual machines to servers (regarding the constraints), the migration phase must be planned independent of implementation of the first phase for finding feasible and optimal order (schedule) to relocate virtual machines to new places. Impulsive migration plan might be impractical due to deadlocks or SLA violation caused by overloading network resources, also if migration process takes too long the overhead can violate optimally of the re-placement, similarly state of the virtual machines might change during a long period of migration. Scheduling virtual machine migrations can be categorized in two approaches, first one is migrating one virtual machine at a time and the other one is scheduling parallel migrations. Also we can migrate virtual machines inside a datacenter or between datacenter. In the following sections we review shortly the single virtual machine migration and also inter-datacenter migration, then in more details we review multiple virtual machine migrations inside a datacenter which is the focus of this study.

Migrating a single virtual machine

Problem of optimizing a single virtual machine migration, instead of optimizing the order of several virtual machine migrations, focuses on optimizing the process of migration itself. This problem aims to optimize the migration duration, minimize the data transferred in the network, reduce energy consumption and minimize the downtime of the services when switching from the virtual machine stopped in the source to the one in the destination. Example of such optimization is using a techniques to predict the dirty pages of memory in advance to optimize the memory migration by reducing migration time and/or downtime of the service [2]. Another domain of the study is power consumption for live migrating virtual machines, major application of virtual machine migration is physical server consolidation which aims to reduce the energy consumption is datacenters, this additional energy consumption is due to having duplication of virtual machines simultaneously and also network resources used for conducting the migrations. [2]. Although this study's center of attention is optimizing migration of several virtual machine, but among its objective is reducing number of temporary migrations which results in reducing energy consumption for the total migration process.

Multiple virtual machine migrations

With current technologies, parallel migrations are possible. However, all of these live migrations cannot be done concurrently because of capacity constraints of the network and also possible blockages caused by servers limited capacities to host the virtual machines of the current and new placements simultaneously.
Finding the optimized order of migrations will reduce the total migration time (we will use the term 'makespan') to gain more stable service and avoid service quality degradation due to the resource usage by migrations. Main inputs of the this problem are, virtual machines characteristics (CPU, memory, bandwidth, dirty page rate), servers (physical machines) capacities, network bandwidth and legacy and new placements for each virtual machine. Other constraints such as

maximum total time for migrations also can be part of the inputs. In the following subsections, we review the virtual machine migration problem.

Virtual machine migration problem objective

Main objective for migrating virtual machines is to minimize makespan or total migration time, by total here we mean the time until the last re-located virtual machine migrates to its new server and all the virtual machines are moved to their assigned servers based on the new placements. Minimizing total migration time results a smaller impact on performance degradation due to virtual machine migrations, also reduces the waiting time for adding new virtual machines after new placements are achieved. Any VM addition should be delayed until all migrations are complete because shortages of destination servers can occur while virtual machines are migrating. In terms of management tasks, it is also beneficial for cloud providers to know how long it takes for all virtual machines to finish migrating [3]. Also, the migration process must not take so long in a way that there are considerable changes in virtual machines service load which might make the current placement non-optimized again.

Virtual machine migration problem constraints

The order (scheduling) of virtual machine migration cannot be impulsive, there are constraints must take into account otherwise the migration plan might not be practical. Reaching optimal time for migrating virtual machines not only optimizes the process (i.e., energy consumption), sometimes taking too much time, puts down the main aim of migration (i.e., SLA violation raised by migration). For optimizing total migration time, several constraints at source and target servers of migrations and datacenter network must be considered. Also, the migration timing (individual virtual machine migration and total migration time) and path must not violate any SLA related to network links bandwidth dedicated to services [18].

Dow and Matthews [13] in addition to constraints for resources such as memory, CPU, disk or networking I/O, consider the constraints of network management resources such as hypervisor low memory conditions, virtual machine:virtual machine anti-colocation, and virtual machine:Host anti-co-location constraints. In this study authors use $A*$ search algorithm for which they define cost of each movement for virtual machines in network based on the type of the movement (random move, moves that violates any constraints etc.) some movements such as move has a high cost (e.g., invalid) that it is unlikely they are being chosen in algorithm. We can categorize these constraints based on the constraint emerge from physical machines or the ones come from network links (paths). In addition, there is another type of problem (deadlock) that can be caused by server or links constraints.

Number and order of migrations

If we assume there is a need to change the current placement to the new one and multiple individual virtual machine migrations must be done in one server, if all of these migrations are done simultaneously the excessive utilization of the resources in the server (CPU, memory etc.) might violate the quality of service for running virtual machines, also it is possible that simultaneous migrations take too long which affects the optimally of the new placement. Beside the limitation of resources, traffic and bandwidth of the paths might affect the number and order of migration. it is possible that the shortest path (physical distance, less number of routers/switches) between the source and destination for a virtual machine A does not have the required capacity for performing the migration without effecting the quality of services using that path simultaneously, or the free bandwidth is not enough to perform multiple simultaneous migrations in the desired time window, we have to delay some of migrations or change their migration paths. Oneue *et al.* [3], for avoiding traffic congestion assume parallel degree for network links, which is based on maximum number of parallel migrations in each output links.

Deadlocks

If at the moment of migration there is not enough resources (processor, memory or service bandwidth) on the destination server, the migration can not be done and we have a deadlock. There are two main types of deadlocks [19], one is indirect deadlock which occurs when a virtual machine needs resources in a server that are currently used by another virtual machine that is planned to be relocated. In this case we must be sure to plan the migration of the second virtual machine in prior to the first one and can be solved by ordering the migrations considering resource shortages. Other type of deadlock is when there the dependency of migrations for releasing the resources makes a cyclic graph which can be solved by more complex methods such as planning some temporary migrations. One reason that makes the resource shortages which lead to deadlock more possible is the fact that in a datacenter during migrating a virtual machine, twice of the required resources for running that VM is needed.

One way to solve this infeasibility of the migrations is to change the allocation of virtual machines in the new placement in a way to avoid having such set of migrations [19]. Another approach is to migrate a set of virtual machines to a temporary physical machine in order to make the migrations possible.

For example we assume virtual machine A currently in location m is planned to migrate to location n and virtual machine B which is currently at virtual machine n is planned to be placed in m. If both machines can not host A and B simultaneously (physical resource or network bandwidth limitations), we have to move one of them to a temporary location or shut down one of them in order to free one of the physical machines for the first step of placement. As an example in work of Onoue *et al.* [3], they use migration dependency graph (described in detail in Section 4.1.1) to gain the optimal order of migrations, in the algorithm they are solving the deadlocks by migrating a low cost virtual machine(s) in cycles to temporary positions which must be retrieved to the original destinations later.

Individual virtual machine migration duration

One of inputs for the problem of optimizing multiple virtual machine migrations is the migration time for an individual virtual machine. In virtual machine live migration, full state of the machine running in the source server must be transferred to the destination, but the predominant part is the memory. Many of the studies in this domain use the memory size value as input without loss of generality. In our study, although we can use this approximation, there are more precise approaches to calculate the time needed for migrating one virtual machine with given memory size, dirty page rate and assigned migration bandwidth. In [20] authors use a simple mathematical formula to calculate the migration time. With using pre-copy method for migrating a virtual machine, with memory size of V_m and average dirty page rate of D Mbps, it needs n iteration to complete transferring memory and the dirtied pages generated during the process. If V_i is the amount of data sent during iteration i we have:

$$V_i = \begin{cases} \frac{V_m}{R} + \tau & \text{if } i = 1 \\ D \times T_{i-1} & \text{otherwise.} \end{cases}$$

If we consider R as network bandwidth for the migration, τ as the delay between each iteration and $\lambda = D/R$, we can calculate each iteration time :

$$T_i = \frac{V_i}{R} + \tau = \begin{cases} V_m/R + \tau & \text{if } i = 1 \\ T_{i-1} \times \lambda + \tau & \text{otherwise.} \end{cases}$$

We can conclude T_i as :

$$T_i = \frac{V_m}{R}\lambda^{i-1} + \tau\frac{1-\lambda^i}{1-\lambda} \quad i = 1, 2, ...n$$

$$T_v = \sum_{i=1}^{n} T_i + T_{down} = \frac{V_m(1-\lambda^{n+1})}{R(1-\lambda)} + \tau\frac{n(1-\lambda) - \lambda(1-\lambda^{n+1})}{(1-\lambda)^2} + t_{down},$$

Where T_v is the total migration time for an individual virtual machine v, which can be used in problems of optimizing multiple virtual machine migrations.

2.3 Mixed problem

The virtual machine consolidation problem can also consider a more optimized migration possibility along with the problem of placement. As discussed, its possible that migration constraints make the placement solution impractical, also optimal solution of new placement might become ineffective by a excessive time and energy consuming migration plan. For tackling these issues some of the studies consider limitations for migrations or change the objective of the problem to include the migration criteria.

Problem objective

In some of the mixed problem studies, the authors use the same objectives of a general virtual machine placement problem and they consider the migrations process by adding related constraints to the problem. Some of the studies change the problem objective in a way to add the effects of migrations to the problem. [21] Verma *et al.* add a migration cost estimated by quantifying the decrease in throughput because of live migration and estimating the revenue loss because of the decreased performance (as given by SLA) to the problem formulation.
Li *et al.* [10] add an objective for minimizing the migration cost beside the objectives of placement (for example minimizing number of active PMs), to minimize the number of migrations in addition to minimizing unbalancing between resources. Migration cost consists of a constant overhead cost for each migration and also a cost related to pre-copy and stop-and-copy phases duration.

Problem constraints

In general, a mixed problem contains the constraints of virtual machine placement problem and for migration constraints most of the studies, try to avoid migration complexities by limiting the number of migrations in virtual machine consolidation. Takeda and Takemura [22] use a first fit decreasing algorithm to assign virtual machines to PMs and for limiting number of migrations they only consider virtual machines which are currently in high-load or low-load PMs. A high load PM is the one which has at least one resource (for example CPU) utilization above the threshold. Also in their algorithm sending or receiving PMs are not candidate for new migration. The algorithm tries to balance the virtual machine placement by moving virtual machine from low load/high load PMs to the most loaded PM with enough resources.

In the study of Takahashi *et al.* [23], to prevent servers performance loss during migrations, authors add a limitation in a way that each server can send or receive at most one virtual machine per time to prevent performance degradation due to migration collision.

Some studies try to add a migration cost penalty related to number of migrations to objective function, Beloglazov *et al.* [7] for optimized virtual machine allocation, set objective of reducing energy consumption in cloud data center management, in their heuristic algorithm the number of required migrations is minimized through a policy to select the minimum number of virtual machines to be migrated with the goal of minimizing energy consumption and overcome servers CPU utilization violations (out of upper lower thresholds).

2.4 Statement of scheduling virtual machine live migrations problem

The objective of the problem is to find the order of migrations and temporary location assignments (if any) to migrate a set of virtual machines from a legacy

(current) server placement to a new one inside a datacenter in a way to minimize makespan (total migration time) and number of temporary migrations.
Temporary migrations affects the resource (energy) utilization and quality of services during migrations, by minimizing number of temporary migration we can reduce the energy consumption and possible SLA degradation.

Assumptions:

1. Virtual machine migrations are intra-DC (physical servers are hosted within the same data-center).
2. We consider the pre-copy live migration strategy.
3. Migration time for each virtual machine is calculated based on memory size and dirty page rate of each virtual machine [20].
4. Dedicated bandwidth is the same for all migrations.
5. Migration bandwidth assignment is greater than dirty page rate of any virtual machine.
6. Both current and new placements are feasible.
7. Datacenter network structure follows FAT tree topology.

In most recent cloud implementations, cloud backbone network structure has fat tree structure, by which we can abstract the data center network as a bandwidth guaranty network and the capacity between each two servers is based on whole network pipeline free capacity (total parallel degree) and direct links from source and destination to first switches, no need for tracing the exact capacities in paths between the servers [4],[3]. In this study we assume the datacenter network has a fat tree topology.

Inputs:

1. Virtual machines information
   (a) Processor (CPU) size of each virtual machine
   (b) Memory size of each virtual machine
   (c) Network bandwidth (for service) size of each virtual machine
2. Servers (for convenient and based on the context we might use the terms 'Physical Machine' , 'PM' or 'location' for servers)
   (a) Processor (CPU) capacity of each location
   (b) Memory capacity of each location
   (c) Network bandwidth of each location
3. Current assignments of virtual machines to locations (PL^{CURRENT}), for each VM i, S_{ij} if virtual machine i is assigned to server j in current placement.
4. New replacement of virtual machines to locations (PL^{NEXT}), for each VM i, R_{ij} if virtual machine i is assigned to server j in new placement.
5. Datacenter capacity for parallel migrations

Constraints:

1. There is always a working copy of each virtual machine (except a tiny down-time between shutting down a VM in the source and starting it in the destination server)
2. After the migration, each virtual machine is assigned to one and only one location (destination location which is provided).
3. None of the servers has any overloaded resources at any time.

Output:

1. Order of virtual machine migrations.
2. Each virtual machine migration start and finish timestamp.
3. Location of each virtual machine during migrations.
4. Each server load (resource usage) before, after and during the migrations.
5. Datacenter network load during migrations.
6. Makespan (total migration time).

Chapter 3

Literature Review

In this chapter, we review the studies in the domain of virtual machine migration optimization problem with different objectives and methods. First we explore the related studies in Section 3.1 and then we summarize our review in Section 3.2.

3.1 Related works

Virtual machine migration is a technology that expands the area of improving manageability of datacenters for optimizing the resource usage and increase the quality of services. Live migration allows a virtual machine continue to run during migration without disrupting service or services running on it, which brings a vast area of interest for studies. Optimization techniques for virtual machine live migration can target makespan (total migration time), downtime and total transferred data during migration [2]. The focus of this study is optimization of multiple parallel migrations with the objective of reducing makespan with minimizing number of possible temporary migrations. Temporary migrations are for overcoming possible cyclic dependencies during the migrations and affects energy consumption, total transferred data and service quality which can be a part of the objectives or constraints of the problem (more details in Section 4.1.3). The literature of minimizing makespan of multiple migrations problem, can be classified based on

the techniques to achieve this objective, for example finding the order and paths of migrations, or minimizing makespan by optimizing memory transferring techniques such as pre-copy or post-copy. Also another principle for categorizing is the mathematical method used for solving the problem, linear programming, heuristic, greedy etc. Also we can categorize the studies based on the problem boundaries, virtual machine placement optimization, virtual machine migration(s) optimization or mixed problem (detail definition can be found in Chapter 2).

Onoue *et al.* [3] define a Mixed Integer Linear Program (MILP) to solve the multiple migrations problem. Their exact method has current and new placements as input and objective of minimizing makespan (total migration time) and the reducing unnecessary (temporary or transient) migrations. Their objective function has a total number of migration steps (multiples of the greatest common divisor of migration times) and a second term which is a coefficient of number of total migrations as a penalty to minimize the number of temporary migrations. Nine constraints assure the completion of migrations and resource limitation during and at the end of migrations. Their MILP formulation is not scalable and based on their evaluation section, it cannot finish the calculation for certain number of test datasets in less than one hour, which was the limit of timeout for their MILP solver. They also propose a heuristic with the use of migration dependency graph which also has been used in this study. The drawbacks and missed parts of their heuristic are addressed in Chapter 4.

In [24], Ghribi *et al.* propose an exact method (ILP) for optimizing placements of virtual machines on servers (consolidation) along with optimizing number of migrations to minimize the energy consumption which is considered as a mixed problem (defined in Section 2.3). In their evaluation section, their exact method does not seem scalable, and the execution time when the number of active nodes (servers) is more than 20 is in the scale of tens of hundreds of minutes [24].

Kherbache *et al.* in [25] describe a migration scheduler module which uses the resource constrained project scheduling algorithm (RCPSP). For overcoming the NP-Hard performance of the algorithm they propose to simplify the problem and

use a heuristic. The outputs of their migration scheduler are best moment to start migration(s) also the amount of bandwidth to allocate to migrations. Inputs are virtual machine workload, the network topology and user specified constraints.

In [26] Bari *et al.* propose a simple heuristic to optimize the sequence of virtual machine migrations. Their heuristic groups the virtual machines based on the resources needed for migrations and prioritize the migrations of VMs in a group that has the maximum migration time and more dependant groups. For evaluation, they compared it with a very simple heuristic that find and returns a feasible and not necessarily optimized solution. In the output of their algorithm they assume virtual machine and network downtime which is against the main advantage of live migration and quality of service expected from this technology.

Sun *et al.* in [4], try to minimize the total migration time (makespan) of parallel migrating virtual machines with use of a combination of pre-copy and post-copy migration strategies. Their study though tries to minimize the migration time of parallel multiple migrating virtual machines but does not consider optimization of sequence of migrations and dependency conflicts.

Gilesh *et al.* [27], define an MILP for minimizing or bounding the cost of migrations in cloud data centers. They define their exact problem formulation and then for offering computational feasibility they propose a greedy and a meta-heuristic methods. Their greedy algorithm sorts virtual machines based on the calculated migration time and start to migrate feasible ones. One drawback of their meta-heuristic approach is to limit the number of successful migration after a certain number of iteration in the algorithm and also do not have a mechanism to consider the number of temporary migrations. Based on their simulated results, the meta-heuristic approach makespan is on average 25% less than their greedy algorithm.

Deshpande and Keahey [28] study the effect of contention between service bandwidth request and virtual machine migrations in a datacenter. The pre-copy and post-copy migration strategies affect traffic of the source or destination servers differently during migrations, these conditions affect the quality of service in datacenter and also migration makespan. They propose a traffic-sensitive live virtual

machine migration technique to optimize the combination of pre-copy and post-copy techniques for migrations of the co-located VMs (those located on the same source host), instead of relying on a single pre-determined technique for all migrations.

Jaumard *et al.* [29], propose an exact sequence-based model of Mixed Integer Linear Program (MILP) and compare it with other exact models such as Onoue *et al.* [3]. Their sequence-based MILP model is highly efficient and can solve sufficiently big virtual machine migration problem instances in a few seconds except for infrequent instances which need temporary migrations. In other part of their study, they investigate the necessity of intermediate migrations for a minimum makespan solution, their model can solve most of the problem instances without need of any temporary migrations which is a big advantage compared to other exact models and heuristics in the literature. Their numerical results provide a proper source for evaluating the results of the heuristic of this study.

3.2 Summary

Number of studies addressing virtual machine migration optimization is not considerable compared to studies in other areas of cloud computing. Some of the studies try to optimize makespan and quality of service of virtual machine migrations by selecting an optimized combination of post-copy and pre-copy techniques during migrations which is not the area of this study. The studies attempt to solve the virtual machine live migration scheduling with an exact method (e.g., MILP) try to solve a problem that by nature is NP-HARD, so they are likely not scalable. The heuristics algorithms in the literature often don't consider the resource limitation dependencies which beside the memory size of migrating virtual machine, is the main factor for prioritization some migrations over others in scheduling. Bari *et al.* [26] try to consider this dependency with putting migrating virtual machines in groups based on the resource needed. Onoue *et al.* [3] in a more advanced approach introduce the dependency graph which creates a chain

of dependencies based on the resource limitations in the current placements of the datacenter.

The heuristic algorithms do not consider the migration dependency factor or if so, they do not control the number of unwanted temporary (transient) migrations. Jaumard *et al.* in [29] show that in an exact solution there can be no or very few transient migrations.

In this study, we propose a heuristic which with considering migration dependency concept, minimizes the number of transient migrations and also optimizes the makespan. We compare our results with heuristic and exact model of [3] and also results from exact model in [29].

Chapter 4

Virtual Machine Live Migration Scheduling

In this chapter, we introduce new heuristics for multiple virtual machine migrations scheduling. We explore different heuristics which gradually improve the algorithm to reach the final heuristic with aim of optimizing makespan and number of temporary migrations. We also review possible trade-off between makespan and temporary migrations in the algorithm.

4.1 Migration priorities and dependency weights

A very basic heuristic for migrating virtual machines from the current placement to the new one, sorts the migrating virtual machines in descending order of their migration times and starts migrating the feasible ones in the same descending order. This approach faces two main drawbacks, first it is not optimized since the priority of the migrations are only based on their migration times and the algorithm does not consider the dependencies. A simple example is when a virtual machine with large memory size cannot migrate due to the lack of resources in the destination which is being held by one or more virtual machines smaller in memory size. Other problem, deadlock happens when there is one or more cyclic chain

of dependencies, a basic heuristic can not detect or solve this type of complexity. For addressing these issues we can use migration dependency graph, which has been used in Onoue *et al.* [3], the authors propose a heuristic algorithm which makes use of a migration dependency graph (detailed definition in Section 4.1.1) and migration dependency weights (details in Section 4.1.2), to find the priority (dependency sequence) of the migrations and also to detect possible cyclic dependencies which results in deadlocks (Section 2.2). First we define the dependency graph and the algorithm to weight the migrations.

4.1.1 Dependency graph

To find the feasible sequence of migrations and the most prior ones, Onoue *et al.* [3] define a migration dependency graph $d = (V^{\mathrm{D}}, L^{\mathrm{D}})$ [3] which is generated based on this observation that a migration is possible when there is no shortage of resources in the destined server to host a new virtual machine. In Onoue *et al.* [3] migration time for each virtual machine assumed to be equal to its memory size (without loss of generality), more accurate formula for calculating the migration time is described in Section 2.2.

Algorithm 1: Generating dependency graph

Result: Dependency graph
Data: VMs, Servers, PL^{CURRENT} and PL^{NEXT}, VM migration times

$C_s \leftarrow$ free resources in server s
$R_v \leftarrow$ resource demand for VM v
$In_s \leftarrow$ list of VMs migrating to server s
$Out_s \leftarrow$ list of VMs migrating from server s
$d \leftarrow \{\}$ // empty graph
foreach $s \in$ *Servers* **do**
    $In_s \rightarrow$ sort descending migration time
    foreach $v \in In_s$ **do**
        if *s has enough resources for v* **then**
            $C_s = C_s - R_{\mathrm{v}}$
            $In_s = In_s - \{v\}$ // remove v from In_s
    end
    if $In_s \neq \emptyset$ **then**
        add an edge from In_s to Out_s
        add In_s and Out_s to d
end

In Algorithm 1, we assume each *dependency edge* is from a set of migrating virtual machines to a server, for which there are not enough resources at the current time in the destination server, to the set of virtual machines which are currently in the server (PL^{CURRENT}) and will migrate from the server based on PL^{NEXT}. This dependency edge means the coming virtual machines can migrate after one or more of the virtual machines in the server at PL^{CURRENT} move from the server (finish migrating) and release resources. To achieve this in Algorithm 1, for each server, we sort the coming VMs in descending order of their migration times (coming VMs are the virtual machines which supposed to migrate to the current server in PL^{NEXT}) and by iterating over them, we check if there are enough resources on the server for each of them based on the PL^{CURRENT}, if so we reserve the required resources (subtract the required resources from the available resources) and remove the VM from list of coming VMs until there are not enough free resources for any of the VMs. If one or more VMs left in the set of coming VMs, we draw an edge from this set to the set of outgoing VMs from this server.

Consider the example of Figure 4.1, current and next placements are as described in Table 4.1. In this example we assume migration times are equal to memory size of each virtual machine without loss of generality.

	Server #1	Server #2	Server #3	Server #4	Server #5
Server resources	memory: 20 processor: 15	memory: 20 processor: 15	memory: 25 processor: 30	memory: 30 processor: 30	memory: 40 processor: 35
$PL_{CURRENT}$	VM_1, VM_2	VM_8	VM_5, VM_9	VM_4, VM_7, VM_3	VM_6
PL_{NEXT}	VM_7	VM_9	VM_8 , VM_3, VM_1	VM_6, VM_2	VM_5,VM_4
Dependencies		VM_9 –> VM_8	{VM_8, VM_1} –> {VM_5, VM_9}	{VM_6, VM_2} –> {VM_4, VM_7, VM_3}	-
VM resources (memory , processor)					
	VM_1:(5,5) VM_6:(16,16)	VM_2:(10,8) VM_7:(2,2)	VM_3:(5,5) VM_8:(10,8)	VM_4:(15,10) VM_9:(10,8)	VM_5:(8,8)
VM migration times					
	VM_1:5 VM_6:16	VM_2:10 VM_7:2	VM_3:5 VM_8:10	VM_4:15 VM_9:10	VM_5:8

TABLE 4.1: $PL_{CURRENT}$ and PL_{NEXT} of the virtual machine migration

For example in Figure 4.1, server 3 currently has 7 units of memory and 14 units of processor as free resources. After sorting the virtual machines migrating to server 3 in descending order of migration times, for example $\{VM_8, VM_3, VM_1\}$, we cannot migrate VM_8 due to the lack of enough free resources in the server. If

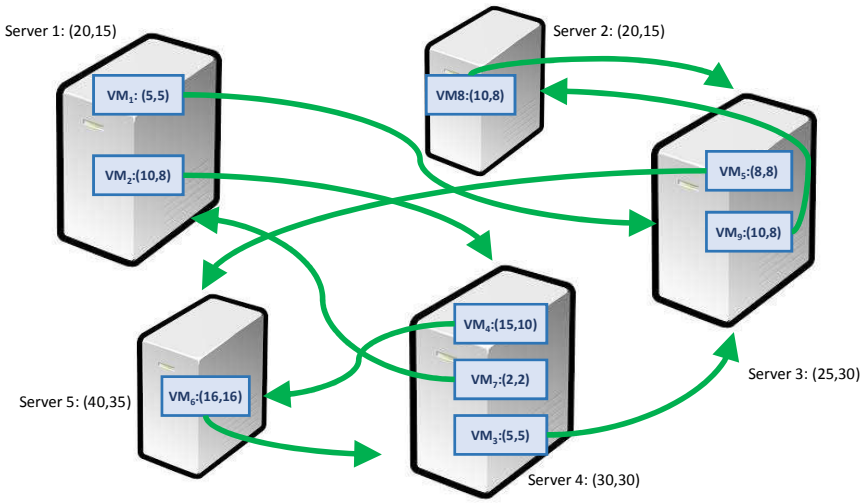

FIGURE 4.1: Virtual machine migration example

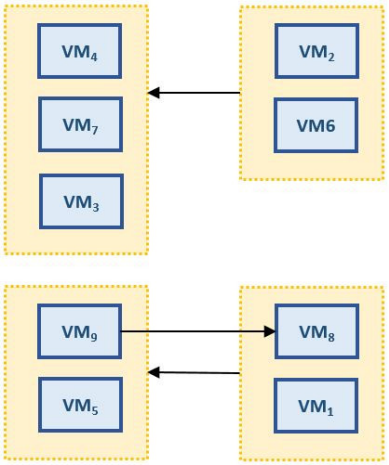

FIGURE 4.2: Example of migration dependency graph

we check the next candidate, we can migrate VM_3 so we remove it from the list of migrating virtual machines to the server and update the free resources to {2,9} (memory and processor respectively). This is the last possible migration, so there is an edge from $\{VM_8, VM_1\}$ to $\{VM_5, VM_9\}$. We do the same for all servers to obtain the whole dependency graph. The entire migration dependency graph generated based on dependency graph of each server, is depicted in Fig. 4.2.

As we describe in Algorithm 1, also shown in the example, one edge from a set of virtual machines to the other one, does not state an exact dependency relation

between two sets. This is because we put an edge from VMs of blocked incoming migrations to all outgoing migrations of a server. It is possible that by migrating a subset of VMs in the destination node, there are enough resources to free enough spaces for coming VMs. This concludes that the dependency graph is not an exact modeling of the dependency relations.

4.1.2 Migration weights

Migrating the virtual machines in descending order of their migration weights instead of migration times, probably decreases the makespan in a heuristic algorithm by considering the dependencies. Migration weights are calculated based on virtual machines migration times and migration dependencies. The migration weight for each virtual machine is the sum of its migration time and migration weights of its dependent virtual machines. After generating migration dependency graph *MDG*, migration weights are calculated as described in Algorithm 2. In this algorithm first we calculate the dependency weights for the VMs in the sets with no incoming edge in a dependency graph (the VMs which no other VM is dependant to them) which is equal to their migration time. And then we add this weight to the migration times of the immediate dependant VMs. When the migration weights of all dependant VMs calculated, we remove incoming edge of the dependant VMs and repeat the process. If there is no VM set with no incoming edge in the dependency graph either we have finished weighting all the VMs in the dependency graph or we need to migrate temporarily a VM set with minimum summation of migration times to generate a node with no incoming edge (line 21 in Algorithm 2). This step is just for calculating the migration weights and later whether temporary migrations are really required or not, is decided during migration process in the main algorithm.

As an example in Figure 4.3, the migration times for VM_1, VM_2, VM_3 and VM_4 are 2, 3, 1, 2 respectively. For calculating the migrations weight we start from

Algorithm 2: Calculating Migration Weights

Result: Migrations Weights w

Data: VMs, Dependency Graph d , migration times (M_v : migration time of VM v)

```
1  Nodes ← List of VMs in g;
2  w ← {}        // array of migration weights
3  foreach v ∈ Nodes do
4  |   w[v] ← 0
5  end
6  repeat
7  |   o ← get a VM set with no incoming edge in d;
8  |   if (o is not empty) then
9  |   |   foreach v ∈ o do
10 |   |   |   w[v] ← w[v] + M_v;
11 |   |   |   if v has outgoing edges (d) then
12 |   |   |   |   Dep ← Dependant VMs of v;
13 |   |   |   |   foreach D ∈ Dep do
14 |   |   |   |   |   w[D] ← w[D] + w[v] ;
15 |   |   |   |   end
16 |   |   |   |   remove outgoing edges of V (d)
17 |   |   |   add w[v] to w  ;
18 |   |   |   remove V from Nodes
19 |   |   end
20 |   else
21 |   |       // when o is empty ;
22 |   |   V_t ← VM set with min sum of migration times;
23 |   |   remove outgoing edges from V_t (d)
24 |   end
25 until Nodes is empty;
26 Return w;
```

VM_1 which has no incoming edge. Migration weight for VM_1 is equal to its migration time which is 2. Then for the VMs in the set $\{VM_1\}$ which is dependant to the set $\{VM_2, VM_3\}$, we calculate migration weight of 5 (3 + 2) for VM_2, and for VM_3, 3 (1 + 2). In the next step for VM_4 migration weight equals to 10 (2 + 5 + 3). Based on these weights the migration order will be [VM_4 , VM_2 , VM_3 , VM_1], which for this simple example can be directly derived from the dependency graph.

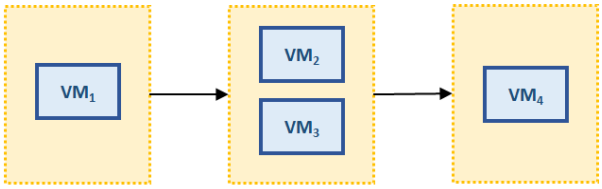

FIGURE 4.3: Example of dependency graph

4.1.3 Onoue *et al.* heuristic

In this heuristic virtual machines are migrated in an order that meets the feasibility and also optimally for reducing the makespan (total migration time). First for feasibility, the algorithm needs to find migration candidate VMs which are not forced to wait for availability of resources in the destination they reside in PL^{NEXT}, in other words virtual machines which are in nodes (sets) where there is no outgoing edge in the dependency graph. The availability of bandwidth for migrating also must be verified.

For finding the deadlocks or cyclic dependencies, the algorithm checks each connected component of the dependency graph, they look for feasible migration by checking whether there is any VM without outgoing edges in the connected component, if no it means there is no virtual machinethat can start its migration and we have a deadlock, in this case we have to solve such cyclic dependencies by moving a VM set to a temporary location to make some migrations feasible. These transient migrations have additional costs (increased makespan, quality degradation) for the migration planning, for this reason Onoue *et al.* [3] select a source node (VM set) in the cyclic connected component of the dependency graph, with one with minimum sum of migration weights. We can use migration weights for getting the order of migrations. When there are multiple VMs ready to be migrated it is more optimized if we prioritize the ones with more dependent migrations, in other words VMs with bigger migration weights.

The complete heuristic is shown in Algorithm 3. First it starts to find the possible deadlocks by checking connected components of the dependency graph (Algorithm 3 line 9). If any of the connected components does not have at least one feasible

migration, in other words a VM set without outgoing edges in dependency graph (independent node) (Algorithm 3 line 11), it will be added to the list of cycles. Afterward it solves the deadlocks by transforming migration dependency graph (Algorithm 3 line 20), which is described in detail in Algorithm 4. Then it checks the independent migrations and starts the feasible ones in descending order of migration weights by checking if there are enough resources in the destination server and enough network bandwidth (Algorithm 3 line 21). After finding and starting a set of migrations, it looks for the next set of VMs among the ongoing migrations that are going to finish their migrations first, and finishes their migration by releasing the resources on the source servers and subtracting the passed time from the remaining time of all ongoing migrations (Algorithm 3 line 32). After starting new migrations or finishing a set of migrations the dependency graph and migration weights are updated.

4.1.4 Corrections and completions

In this section we describe some added parts to Onoue *et al.* [3] heuristic which we found by some means necessary for implementation of the algorithm and we proposed a solutions for them.
One of the modifications, *shuffling the temporary migrations*, in fact improves the Algorithm 3, but it was necessary to construct an algorithm to finish all migrations for data sets with average and high complexity to provide us a source for evaluating our own heuristic.

Retrieving temporary migrations

For solving the cyclic dependencies, transform MDG method (Algorithm 4) inside Algorithm 3 creates temporary (transient) migrations. After adding these temporary migrations, because they have the highest weight in the related connected components, they are going to migrate as soon as other constraints are solved (for example network bandwidth). We need to keep the track of original migrations

Algorithm 3: Onoue *et al.* [3] algorithm

Result: Total migration time

Data: M: migration times, S: server capacities, V : virtual machine capacities, PL^{CURRENT}, PL^{NEXT}, n: topology graph

```
d ← generateDependencyGraph (PL^CURRENT , PL^NEXT , S) (d = (V^D, L^D))
c ← {}      // MDG cycles;
x ← {}      // migrateable VMs;
r ← {}      // migration time steps;
l ← {}      // next migration candidate VMs;
t ← {}      // independent VMs after solving cycles;
i ← {}      // list for keeping independent VMs;
w      // migration weights;
foreach connected component g in d do
    w ← CalculateMigrationWeights(g , M);
    i ← getVMsWithoutOutgoingEdge(g, w) ;
    if i = ∅ then
        add g to c
    else
        add i to l
    end
end
repeat
    if c is not empty then
        t ← TRANSFORMMDG(c, PL^CURRENT, PL^NEXT, S, V)
    foreach vm in l do
        if destination has enough resources and path has bandwidth then
            reserve resources;
            add vm to x;
            remove vm from l;
    end
    add t to l;
    if x is empty AND l is empty then
        unsolvable deadlocks
    else
        y ← min migration time calculated for x (w);
        foreach vm in x do
            migrate vm and release resources ;
            update d = G_vm
            i ← get VMs without outgoing edges ∈ g;
            if i is empty and g is not empty then
                add g to c
            else
                add i to l
            end
        end
        add y to r
    end
until d is empty;
return r
```

Algorithm 4: TRANSFORM MDG : method in Algorithm 3

Result: create independent VMs

Data: cMDG, PL^{CURRENT}, PL^{NEXT}, S, V, n

```
1  VMs_t ← find VM set with min sum of migration times in target nodes of cMDG
2  s_c ← current server of vs_t in PL^CURRENT
3  s_n ←  servers for each VM ∈ VMs_t in PL^NEXT
4  s[v]        // new location for transient migration for VM v
5  isSolvable ← true        // boolean if cycles is solvable
6  foreach v in VMs_t do
7  |   foreach s in n do
8  |   |   if s ≠ S_n[v] AND s ≠ S_c AND s has enough resources for v in PL^CURRENT then
9  |   |   |   s[v] ← s ;
10 |   |   |   break
11 |   end
12 |   if s[v] ≠ null then
13 |   |   add new transient migration (v , s[v])
14 |   else
15 |   |   isSolvable ← false ;
16 |   |   break
17 |   end
18 end
19      // all VMs can be migrated to temporary locations?
20 if isSolvable then
21 |   remove incoming edge to VMs_t , update cMDG Return (VMset without outgoing
   |    edges in cMDG)
22 else
23 |   cMDG has a unsolvable deadlock
24 end
```

(PL^{NEXT}) and as soon as all the migrations in the related connected component finish their migration add a new migration from temporary server to the original destination in PL^{NEXT} for temporarily moved VMs (Algorithm 5), so in next iteration the dependency graph includes migrations which retrieve the temporary ones, otherwise we will reach a different or even infeasible PL^{NEXT} during the migrations or when all migrations are done. The authors might consider this step in their implementation of the heuristic, but it is not mentioned clearly in their paper [3].

Algorithm 5: Finishing next wave of migrations

```
1  OM ← List of ongoing migrations
2  t_m ← remaining time of an ongoing migration m
3  start
4  t ← min t_vm for virtual machines ∈ OM
5  foreach vm ∈ OM do
6      if t_vm = t then
7          ...      // release resources in source server and start vm in destination
8
9          PL_vm^CURRENT = PL_vm^NEXT
10         if vm was a temporary migration then
11             PL_vm^NEXT = Original(PL_vm^NEXT)
12     else
13         t_vm ← t_vm − t
14     end
15 end
```

Shuffling temporary migrations

In the heuristic methods we cannot find the possible optimal temporary servers for transient migrations. In case of blockage, we can change the temporary migration and add a new or final location for some of the temporary migrations (shuffling temporary, for solving blockages). It happens when the locations selected for temporary migrations become unavailable before they start migrating. For example if there is another migration with higher weight that starts migrating (reserving resources) on temporary server and reserve it resources. In this situations we need to find a new temporary server, otherwise we will encounter blocked migration(s). With having a better method for choosing the temporary servers for transient migrations, we can decrease these type of blockages with a better greedy selection of temporary servers (as described in Section 4.2.2). As an example in Figure 4.4 , we can see adding *shuffling temporary migration* step can increase the number of successful migration specially when number of total migrations increases. The chart in Figure 4.4, compares Onoue *et al.* [3] algorithm for datasets (33 datasets) of 100 servers with and without adding the shuffling temporary migration steps to the algorithm. As we go right in horizontal axis number of VMs to be migrated in datasets increases.

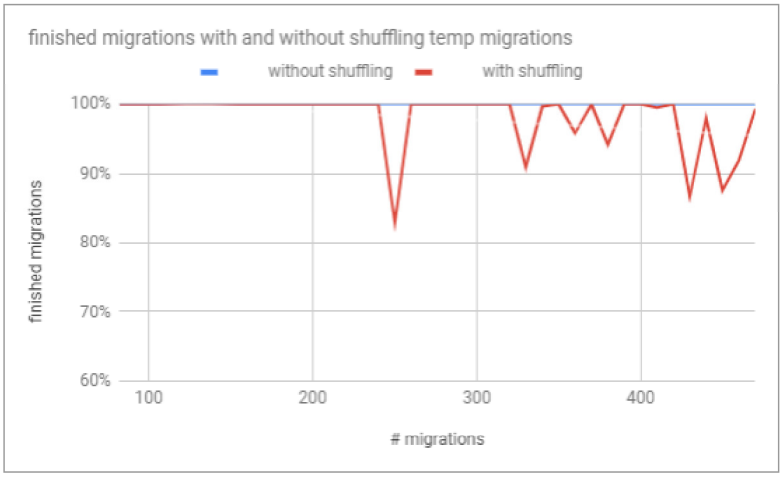

FIGURE 4.4: Shuffling temporary migrations effect on total finished migrations

4.2 Scheduling virtual machine migrations

In this section we propose our heuristic for minimizing the makespan (total migration time) of multiple simultaneous virtual machine migrations. We use the concept of migration dependency graph introduced in Onoue *et al.* [3] (described in Section 4.1.3).

4.2.1 Introducing contributions of this study

In our heuristic we use the strategy of the first fit decreasing (FFD) algorithm. The weights or sizes of migrations are their dependency weights calculated by migrations dependency graph (described in Section 4.1.2) to assure priority of the migrations based on the availability of resources on source and destination servers of the migrations. We also use the dependency graph to find the possible deadlocks. For solving the deadlocks we must move one or more of migrating

VMs in the cycle to a temporary server other than source and destination of their migration, similar to what described in Algorithm 4.
Based on our observations on Onoue's algorithm we found and solved the below mentioned issues and improved the makespan, energy consumption and quality degradation of the solution in our heuristic:

1. Retrieving temporary migrations as early as possible.
2. Adjusting temporary migration locations
3. Reducing the number of temporary migrations.
4. Reducing makespan.
5. Option of trade-off between makepsan and quality degradation during migrations

Temporary migrations are not desirable, since we have to conduct an extra migration also we need to put a VM in an unplanned server for a period of time, for addressing item 1, we need to wait for the availability of resource in the original destination of the migrations and move the temporary migration VM set to the original destination, more description can be found in Section 4.1.4. This step is added to our heuristic in line 13 of Algorithm 6.
Item 2 is needed to make sure the temporary migrations can start as soon as possible, otherwise the number of temporary migrations and a chance of migration blockages increase. The server loads are changing during the migrations and optimally of selected temporary locations is not guaranteed, so if they cannot start migration right away due to unavailability of resources on the temporary location, we need to check and adjust the location of temporary migrations. This is described in Section 4.1.4.
Temporary migrations increase energy consumption, quality degradation and makespan, to reducing number of temporary migrations, item 3, we introduce a new policy for selecting the time to generate temporary migrations and also a method to select temporary migration locations more optimally. These two help decreasing the

number of temporary migrations effectively, the details are in Section 4.2.2.
Waiting for a more optimal moment to conduct the temporary migrations, as a solution for item 3, has a drawback of increasing makespan due to reducing the usage of the available bandwidth for migrations. For overcoming this issue and minimizing the makespan, (item 4), we add a step which checks if usage of the network drops below a threshold (Section 4.2.4) and solves possible cycles and start some temporary migrations. Also better detection of cycles can help to manage temporary migrations earlier (Section 4.2.3).
Beside reducing the makespan, minimizing the migration costs (energy consumption and/or quality degradation) is part of the objective of the problem (item 5), reducing number of temporary migration with cost of increasing the makespan, can reduce the energy consumption and quality degradation which is described in Section 4.2.2, Subsection *'Minimum number of transient migrations strategy'*.

Algorithm 6 shows the initial version of the proposed heuristic. The inputs of the algorithm are virtual machines and servers capacities, migration times (described in Section 2.2), legacy and new placements. With these inputs we can extract the list of migrations and generate migration dependency graph. For finding and starting migrations we iterate over all migrations sorted on descending order of their dependency weights (line 6) and check if we can start migrating them. The method *resourcesAvailable* checks if the destination server of the migration has enough resources (at the current moment) to host the VM, also if there is enough network bandwidth to convey the migration. The method *startMigrating*, reserves the resources on the destination server and updates the network capacity for migrations (network parallel degree).
Also we check if there are temporary migrations for which the original destination (not the temporary location) has enough resources at the moment, because during the possible period for pending temporary migrations, the servers capacities might change so there is a possibility of moving the VM of a temporary migration to its original destination. The started migration are kept in a list (x). If this list, x, is

empty and we still have undone migrations it means there are locked migrations that can be found by checking the connected components with no independent VM (line 22), then we solve the cycles (same as *Transform MDG* described in Algorithm 4) and update the dependency graph to have new migrateable VMs.
At the end of each iteration, we look into ongoing migrations and finish the next set of them with lowest remaining migration time in method *finishNextMigrations* (as shown in Algorithm 5).

4.2.2 Reducing number of temporary migrations

Algorithm 3 is a greedy one and order of migrations and temporary locations chosen for transient migrations are not optimal, hence in a complex dataset for virtual machine migrations planning, it results in considerable number of temporary migrations. In our experiences with generated datasets (described in Section 5.1) there are lots of temporary migrations needed for finishing migrating all of virtual machines to their final destination. For dataset of 50 servers, the average ratio of number of transient migrations to number of total planned migrations (relocated virtual machines) is about 40 percent (Figure 4.5). The more complex the problem (more migrations, highly loaded servers, etc.) the more number of temporary migrations are needed.
These temporary migrations are highly costly, they expand the resource usage during these unwanted migrations and decrease the quality for the services running on them and also on other VMs residing on the same servers by using sever and network resources. Also temporary migrations increase the chance of new blockages, by occupying twice the amount of resources during the migration. With average of 40 percent increase in number of migrations for achieving a new placement (see Figure 4.5), any service degradation or energy consumption that is a proportion of the number of migrations will also increase with that amount. In switch based networks such as fat tree topology a part of the energy consumption by network is a proportion of number of bits conveyed by the network [30], so these excess

migrations increase the energy consumption for migrations with the same rate of increase in number of migrations, subsequently.

We propose two approaches for reducing the number of temporary migrations in our method, first *'waiting for ongoing migrations to be finished before adding transient migrations'* and also *'optimizing selection of temporary servers'* for those migrations.

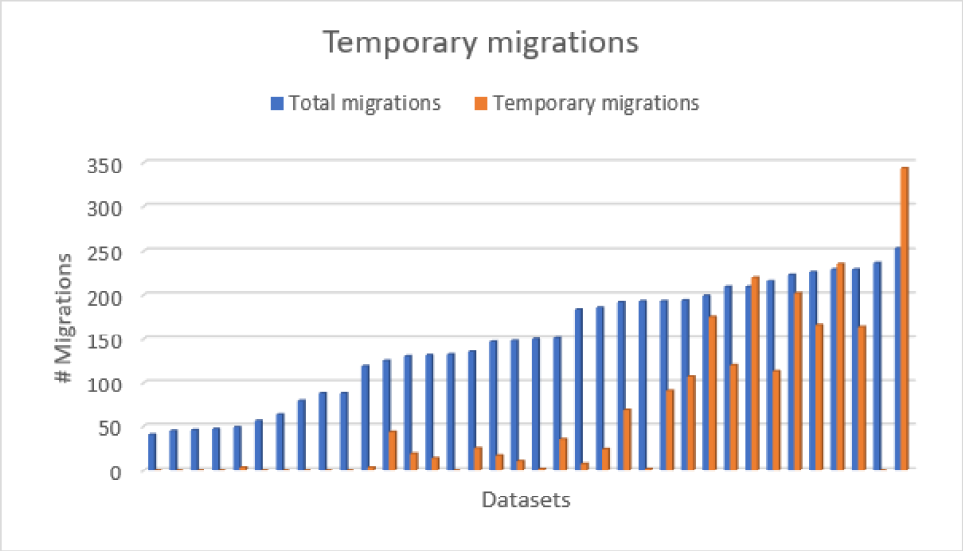

FIGURE 4.5: Temporary migrations generated for Algorithm 3 (datasets of 50 servers)

Waiting for completion of ongoing migration

Based on the datacenter network's parallel degree for concurrent migrations, the number of simultaneous migrations can affect free resources of the servers and create new and strict dependencies in the network. Independent of the method used for virtual machine migration (pre-copy , post-copy, or mixed), during the migration each virtual machine occupies twice the amount of physical resources in servers, since after starting the migration and before starting the copied virtual machine in the destination at (PL^{NEXT}) there is two copies of the virtual machine in the datacenter. If we want to generate the dependency graph during migrations there will be less resources at destinations which results more possible

Algorithm 6: Heuristic: Initial version

Result: Total migration time

Data: M: migration times, S: server capacities, V : virtual machine capacities, PL^{CURRENT}, PL^{NEXT}, n : datacenter network

```
1  d ← generateDependencyGraph (PL^CURRENT , PL^NEXT)
2  m ← {}        // all migrations;
3  c ← {}        // MDG cycles;
4  x ← {}        // ongoing migration VMs;
5  l ← {}        // feasible migration candidate VMs;
6  while m ≠ empty do
7      if l ≠ empty then
8          sort (l)        // descending dependency weight;
9          foreach vm ∈ l do
10             if resourcesAvailable(vm, n) then
11                 startMigrating(vm);
12                 add vm to x;
13             else if vm ≡ tempMigration & resourcesAvailableForOriginal(vm,n) then
14                 startMigratingToOriginal(vm);
15                 add vm to x;
16             end
17         end
18         remove x from l;
19     end
20     if x = empty then
21         foreach connected component g in d do
22             i ← getVMsWithoutOutgoingEdge(g) ;
23             if i = ∅ then
24                 add g to c
25             else
26                 add i to l
27             end
28         end
29         if c ≠ empty then
30             SolveCycles(c);
31             d ← generateDependencyGraph (PL^CURRENT , PL^NEXT , n);
32             l add getVMsWithoutOutgoingEdge(d);
33         else
34             shuffleTempMigrations();
35             d ← generateDependencyGraph (PL^CURRENT , PL^NEXT ,n);
36             i ← getVMsWithoutOutgoingEdge(d);
37             if i ≠ ∅ then
38                 l add i;
39             else
40                 infeasible migrations
41             end
42         end
43     if x ≠ empty then
44         x remove finishNextMigrations(x) ;
45         GD ← generateDependencyGraph (PL^CURRENT , PL^NEXT , n);
46         l add getVMsWithoutOutgoingEdge(GD);
47     end
48 end
```

dependencies. Also for solving possible deadlocks there are less available temporary locations in the network. For this purpose in our heuristic when we start migrating a group of virtual machine, we wait until this wave of migrations finishes (with some exceptions described in Section 4.2.4), then we regenerate and look for possible cycles in the dependency graph. As we can see in Algorithm 6 line 20, if there is no ongoing migrations then we check if a connected component does not have any VMs without incoming edges and solve the deadlock, it means unless there is no feasible migration in the whole network we are not trying to solve the cyclic connected components in the dependency graph by adding new temporary migrations.
Since we are not solving the cycles as they appear in the dependency graph, we don't need to iterate over the connected components in dependency graphs (to detect cycles) as done in algorithm of Onoue *et al.* [3] (Algorithm 3 line 9), and we can loop over the migrations as done in Algorithm 6, line 6.

We must mention here, although the approach of the heuristic in Algorithm 6 decreases the number of temporary migrations significantly, it has a drawback of increasing the makespan (total migration time), which is caused by not using the full capacity of network when there are cyclic connected components waiting for the ongoing migrations to be finished. This issue can be solved by checking the number of ongoing migrations against a threshold and adding more transient migrations to the network (Section 4.2.4).

Optimize temporary location selection

For temporary migrations we can chose any server other than the current location and the migration destination of the virtual machine(the candidate VMs for temporary migration are inside a cyclic dependency graph component, which indicates they have an outgoing edge and they cannot migrate to their original destination at the current moment). The simple way of selecting a temporary location for a transient migration is to search for a server that currently has the free capacity

to host the virtual machine, which is used in Onoue *et al.* [3] heuristic in method *TransformMDG* (Algorithm 4 line 7).

The parameters affect the selection of a temporary server (in a heuristic manner) are the conditions of the server at current and new placements. For this purpose we define a new formula to calculate a *score* for each server.

$$S_t(s) = f_c \times c_c(s) + f_n \times c_n(s) \tag{4.1}$$

total score for server s $\leftarrow S_t(s)$

current capacity (PL^{CURRENT}) factor $\leftarrow f_c$

new capacity (PL^{NEXT}) factor $\leftarrow f_n$

free capacity of server s in (PL^{CURRENT}) $\leftarrow c_c(s)$

free capacity of server s in (PL^{NEXT}) $\leftarrow c_n(s)$

In Equation 4.1, free capacity for current assignments (c_c) is calculated based on the free capacity of each server at the current situation of the datacenter (not the starting legacy assignments before the migrations started). Free capacity based on the PL^{NEXT} (c_n) is the free capacity of the server at new placements, which is the free capacity of the server when all migrations are done. Diagram in Figure 4.6 depicts the average number of temporary migrations with different values for f_c and f_n after executing Algorithm 6 for 36 datasets of 50 servers.

Selecting the temporary migration based on their free resources after all migrations are done, reduces the number of temporary migrations. With this observation we can say it happens because we have to put transient VMs in locations which are less likely destination for the coming migrations in next steps to avoid creating new infeasible migrations (dependencies) which generates new temporary migrations consequently. So selection of temporary servers (line 7 Algorithm 4) from the feasible servers (those have enough resources for the transient VM in PL^{CURRENT})

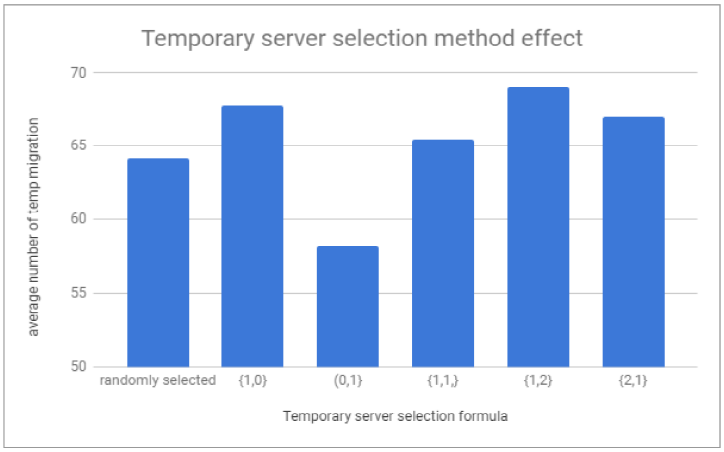

FIGURE 4.6: Temporary server selection with different f_c and f_n

we select the one with the maximum S_t.

Minimum number of transient migrations strategy

In Algorithm 4, if a connected component in the migration dependency graph, ($cMDG$) does not have any virtual machine set with no outgoing edges is con cyclic dependencies and we need transient (temporary) to solve the deadlock. For transient migrations we select a set of virtual machines with minimum summation of migrations times in the $cMDG$, the purpose of this selection is to minimize time required for moving the transient migration to their temporary locations, but it does not minimize the number of transient migrations. In cases where we want to minimize the *number* of temporary migrations (for example with the objective of affecting less number of services with quality degradation) in the step of solving the deadlocks, we can select the minimum size virtual machine set instead of choosing the set with minimum summation of migration times. This selection has a drawback of increasing the makespan and also amount of data flow in the datacenter which results in more energy consumption, but decreases the number of transient migrations.

4.2.3 Large connected components

For big connected components in a busy datacenter, waiting for dependencies to be solved, increases the overall time. In Algorithm 3, for finding the feasible migrations which can be detect by selecting virtual machines belong to sets without outgoing edges in dependency graph, we iterate over connected components in the dependency graph. In complex datacenter server placements (with higher number of migrations relative to number of virtual machines or servers), we might have large connected components in dependency graphs. So if we have cycles in a connected component which need transient migrations to proceed, the process of solving cycles waits until all other feasible migrations in connected components are done. This excess waiting increases the makespan.
The above observation is considered in Algorithm 7. If there is no ongoing migrations (there was no feasible migration to start) we check for every cycles inside the dependency graph without considering the connected components (line 20). This means if we have complex connected components we start solving possible cyclic dependencies earlier and concurrently, which results to conducting more required transient migrations simultaneously.

4.2.4 Speeding up by solving largest cycle periodically

Minimizing number of temporary migrations by waiting for ongoing migration to be finished before solving the deadlocks has a drawback of declining usage of free network capacity for conducting the temporary migrations. It means even when there is free capacity for the network and feasible candidate servers for transient migrations, we are not starting the transient migration immediately and the goal of reducing temporary migrations we are increasing the makespan (total migration time). One solution for tackling this issue is to avoid waiting unconditionally for the ongoing migrations to be finished. We can have a threshold for number of ongoing migrations after which we can start moving transient migrations. The affect of this change (beside decreasing makespan) on number of temporary migration

Algorithm 7: Heuristic

Result: Total migration time

Data: M: migration times, S: server capacities, V : virtual machine capacities, PL^{CURRENT}, PL^{NEXT}, n : datacenter network

```
d ← generateDependencyGraph (PL^CURRENT , PL^NEXT)
m ← {}        // all migrations;
c ← {}        // MDG cycles;
x ← {}        // ongoing migration VMs;
l ← getVMsWithoutOutgoingEdge(d)        // feasible migration candidate VMs;
while m ≠ empty do
    if l ≠ empty then
        sort (l)        // descending dependency weight;
        foreach vm ∈ l do
            if resourcesAvailable(vm, n) then
                startMigrating(vm);
                add vm to x;
            else if vm ≡ tempMigration & resourcesAvailableForOriginal(vm,n) then
                startMigratingToOriginal(vm);
                add vm to x;
            end
        end
        remove x from l;
    end
    if x = ∅ & m ≠ ∅ then
        c ← detectCycles(d) ;
        if c ≠ empty then
            SolveCycles(c);
            d ← generateDependencyGraph (PL^CURRENT , PL^NEXT , n);
            l add getVMsWithoutOutgoingEdge(d);
            if l = ∅ then
                infeasible migrations
        else
            shuffleTempMigrations();
            d ← generateDependencyGraph (PL^CURRENT , PL^NEXT ,n);
            i ← getVMsWithoutOutgoingEdge(d);
            if i ≠ ∅ then
                l add i;
            else
                infeasible migrations
            end
        end
    end
    if x ≠ empty then
        x remove finishNextMigrations(x) ;
        GD ← generateDependencyGraph (PL^CURRENT , PL^NEXT , n);
        l add getVMsWithoutOutgoingEdge(GD);
    end
end
```

depends on the complexity of the network. Based on our datasets, for simple and average complexities it decreases the number of temporary migrations too, though for some of the highly loaded networks in sample data it increases the number of temporary migration slightly which is neglectable considering lowering makespan more effectively.

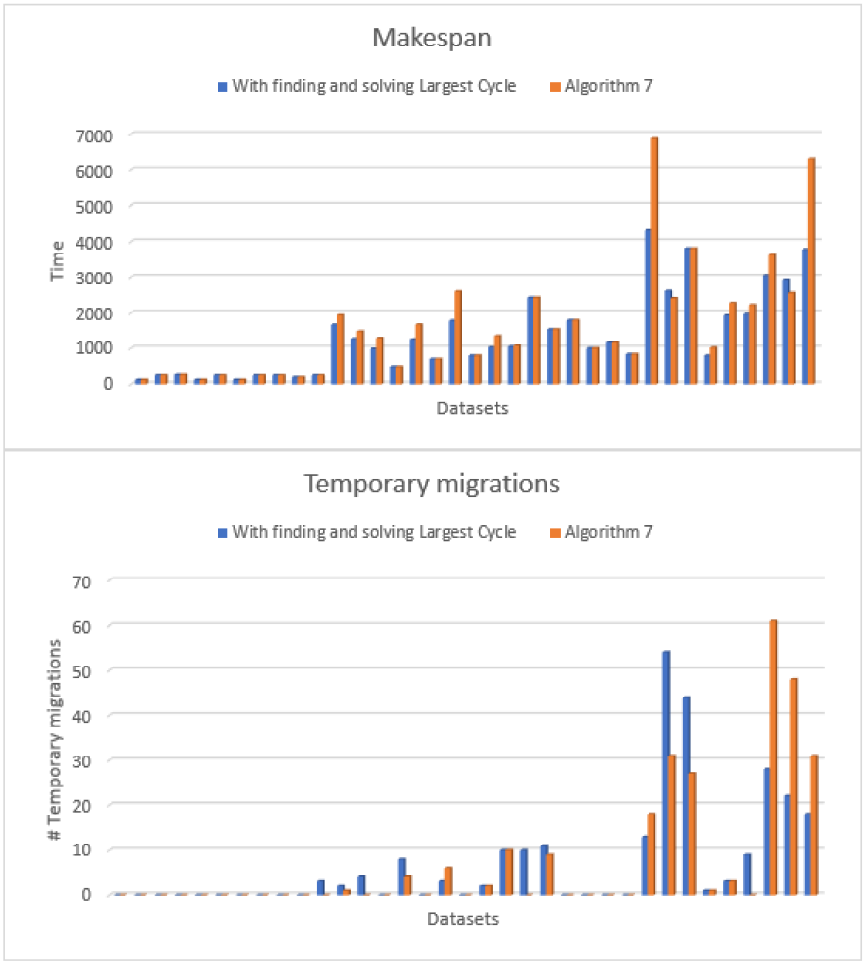

FIGURE 4.7: Effects of adding the method of solving largest cycle to Algorithm 7

Diagram in Figure 4.7 shows the results of running Algorithm 7 with and without method of solving the largest cycle in each iteration. As depicted in the diagram, the average of total migration time is reduced by 15% while the number of total temporary migration has a 2.7% rise. Sample dataset consists of 50 servers, the horizontal axis is the number of migration for each dataset and the threshold in line 20 Algorithm 8 is the number of migrations divided by number of servers.

Final Heuristic

With applying all the optimizations described in this chapter to our heuristic, the final Algorithm 8 is concluded. The goal of this algorithm as mentioned in Chapter 2, is to reduce the makespan of simultaneous migrations while trying to avoid unnecessary transient migration as much as possible to avoid service quality degradation and excess energy consumption. The evaluation of this heuristic has been done against exact methods and other heuristics in the literature in the next chapter.

In this algorithm we iterate over the migrations as we mentioned in beginning of the Section 4.2. Checking for migrateable VMs and start their migrations, lines 7 to 19, is the same as Algorithm 7. While we still have ongoing migrations, we check if the number of proceeding migrations is less than a threshold, we find the largest cycle and by solving it we make more feasible migrations ready to be started (line 24). If all the migrations are finished, we try to solve all the cycles to make best use of the free network bandwidth for temporary and new feasible migrations, we also check and if there is no new cycles (to generate new temporary migrations), we try to find new locations for possible blocked temporary migrations (line 35) by shuffling pending temporary migrations.

In the last part of the algorithm, we finish next wave of ongoing migrations, we find the smallest remaining migration time among the ongoing migrations and subtract it from remaining migration times of all ongoing migrations (inner method *finishNextMigrations()*). For the ongoing migrations with zero remaining migration times (finished one), we mark the migration finished and release the resources at the source server. Finishing a migration affects the migration dependency graph and migration weights, which must be updated in this step. After updating dependency graph, we might have new VM set without outgoing edges, in other words new feasible migrations which must be added to the corresponding list (lines 45 to 49). We repeat this procedure until all the migrations are finished.

Algorithm 8: Final Heuristic

Result: Total migration time

Data: M: migration times, S: server capacities, V : virtual machine capacities, PL^{CURRENT}, PL^{NEXT}, n : datacenter network

```
1  d ← generateDependencyGraph (PL^CURRENT , PL^NEXT)
2  m ← {}        // all migrations;
3  c ← {}        // MDG cycles;
4  x ← {}        // ongoing migration VMs;
5  l ← getVMsWithoutOutgoingEdge(d)        // feasible migration candidate VMs;
6  while m ≠ empty do
7      if l ≠ empty then
           ... start migrating VMs in l
19     end
20     if 0 < x.size ≤ threshold then
21         C_l ← getLargestCycle(d) ;
22         SolveCycle(C_l);
23         d ← generateDependencyGraph (PL^CURRENT , PL^NEXT , n);
24         l add getVMsWithoutOutgoingEdge(d);
25     end
26     if x = ∅ & m ≠ ∅ then
27         c ← detectCycles(d) ;
28         if c ≠ empty then
29             SolveCycles(c);
30             d ← generateDependencyGraph (PL^CURRENT , PL^NEXT , n);
31             l add getVMsWithoutOutgoingEdge(d);
32             if l = ∅ then
33                 infeasible migrations
34         else
35             shuffleTempMigrations();
36             d ← generateDependencyGraph (PL^CURRENT , PL^NEXT ,n);
37             i ← getVMsWithoutOutgoingEdge(d);
38             if i ≠ ∅ then
39                 l add i;
40             else
41                 infeasible migrations
42             end
43         end
44     end
45     if x ≠ empty then
46         x remove finishNextMigrations(x) ;
47         GD ← generateDependencyGraph (PL^CURRENT , PL^NEXT , n);
48         l add getVMsWithoutOutgoingEdge(GD);
49     end
50 end
```

Chapter 5

Numerical Results

The heuristics described in Chapter 4 are implemented and evaluated by running them for generated datasets. Although the main contribution of this study is represented in final heuristic (Algorithm 8), in Chapter 4 we introduced some intermediate heuristics to test and compare improvement steps. The numerical results for comparing those heuristics can be found in previous chapter (for example Figures 4.4 and 4.7). Also, we introduced a modified version of final heuristic with the strategy of minimizing number of temporary migration in Section 4.2.2, this heuristic is compared with final heuristic in this chapter (Section 5.2).
In this chapter, we introduce our data generator in Section 5.1. We compare the results of the final heuristic with other heuristics and exact models in Section 5.2.

5.1 Data generator

During the implementation and for evaluating the heuristics of this study we used the data instances generated for a parallel study conducted by Jaumard *et al.* in [29]. The data instance generator, generates datasets based on different input parameters, each dataset consists of servers and their characteristics (available memory, processor and network bandwidth), virtual machines and their capacities, current and new placements from which we get the migrations. The values for

virtual machines resources are based on the Amazon EC2 instances [31]. For capacities of the servers, we used the characteristics of Dell servers found on [32]. The possible pairs of values (c, m), as c stands for number of CPU units (virtual CPU for VMs) and m for memory (GiB) for Servers are (28,128), (56,256), (78,512) and (112,768) and for virtual machines are (2,4), (2,8), (4,8), (4,16), (4,122), (8,16), (8,32), (8,244), (16,64), (16,122), (16,488), (36,72), (48,192) and (64,256). For generating dissimilar datasets, beside the number of servers there are other inputs to determine the characteristic of each instance :

1. Initial load of server resources, the range which indicates servers load of resources (CPU and memory) for current placements.

2. Placement scenarios: for generating the new placement of VMs in datacenter we can choose among the scenarios which are based on real-world problems of virtual machine placement problems. Each scenario is briefly described as follow:

   (a) Load Balancing, this scenario aims to redistribute the VMs in datacenter in way that servers have approximately the same load/occupancy.

   (b) Consolidating, servers consolidation in datacenter aims to optimized the usage of the servers based on turning off excess servers by packing VMs more optimized in fewer servers. In this scenario VMs are migrated until each servers has a load inside an acceptable interval or being empty (turned off).

   (c) Server failure/maintenance, is a case where number of servers in the datacenter are going to be stopped and all VMs resided in those servers are migrated to other servers. After number of failing servers is selected randomly, each related VM must be moved to a random feasible server.

   (d) Deadlock shuffle, this scenario does not simulate a real world datacenter situation, by redistributing all the VMs in a datacenter this scenario creates a complex migration plan for testing extreme situation such as having deadlock (cyclic) migration sequences.

3. Random factor/shuffling, another optional input is a factor for randomly migrate a portion of VMs in the datacenter, to add more complexity to the scenarios for simulating the unknown situations in real-world.

For each different number of servers (data center size) of 30, 50, 100, 200, 300 and 500 we generated 32 data set with different above mentioned parameters.

5.2 Comparing the results

In this section, we compare the result of running the final heuristic with other algorithms for different datasets. The comparisons aim to evaluate the results based on the makespan (total migration time) and number of intermediate (temporary) migrations. We also compare the makespan and number of temporary migrations and their bandwidth usage of minimum number of temporary migrations strategy with final heuristic.

Comparing with Onoue *et al.* heuristic [3]

Sample results of running the final heuristic against the Onoue *et al.* heuristic, Figure 5.1 depicts the results for running final heuristic for a dataset of 50 servers, the horizontal axis is proportion of number of migrations to number of virtual machines in each dataset.
In Figure 5.2 we can see the same results for datasets of 100 servers.
As we can see in both Charts, we can migrate the virtual machines with less temporary migrations, the average of temporary migration for datasets of 50 servers are $\sim 88\%$ less than heuristic of Onoue *et al.* , for datasets of 100 servers number of temporary migrations reduced by $\sim 92\%$ on average. This reduction, are more noticeable when the number of migrations in a datacenter increases. The makespan is also slightly better in the optimized heuristic, while the main objective was to reduce the number of temporary migrations.

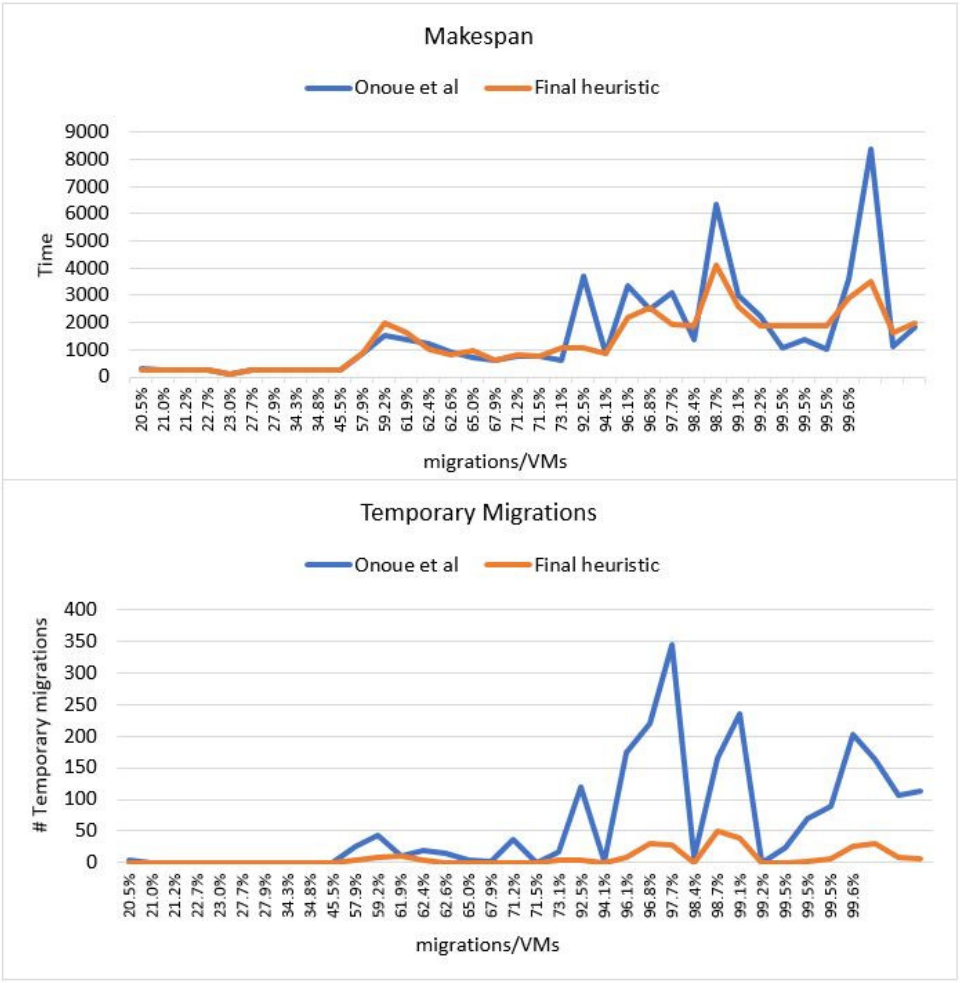

FIGURE 5.1: Comparing makespan and number of temporary migrations of final heuristic and Onoue *et al.* heuristic - datasets of 50 servers

Comparing with First Fit heuristic

In figure 5.3, we are comparing the result of heuristic 8 with the results of another heuristic, an FF heuristic (First Fit) method introduced in [29]. For datasets of 30 server, in table 5.1. The heuristic 8 has on average a smaller makespan of 22% and also finishes all the migrations where the FF algorithm can not finish $\sim$ 11% of the datasets (dependency deadlocks).

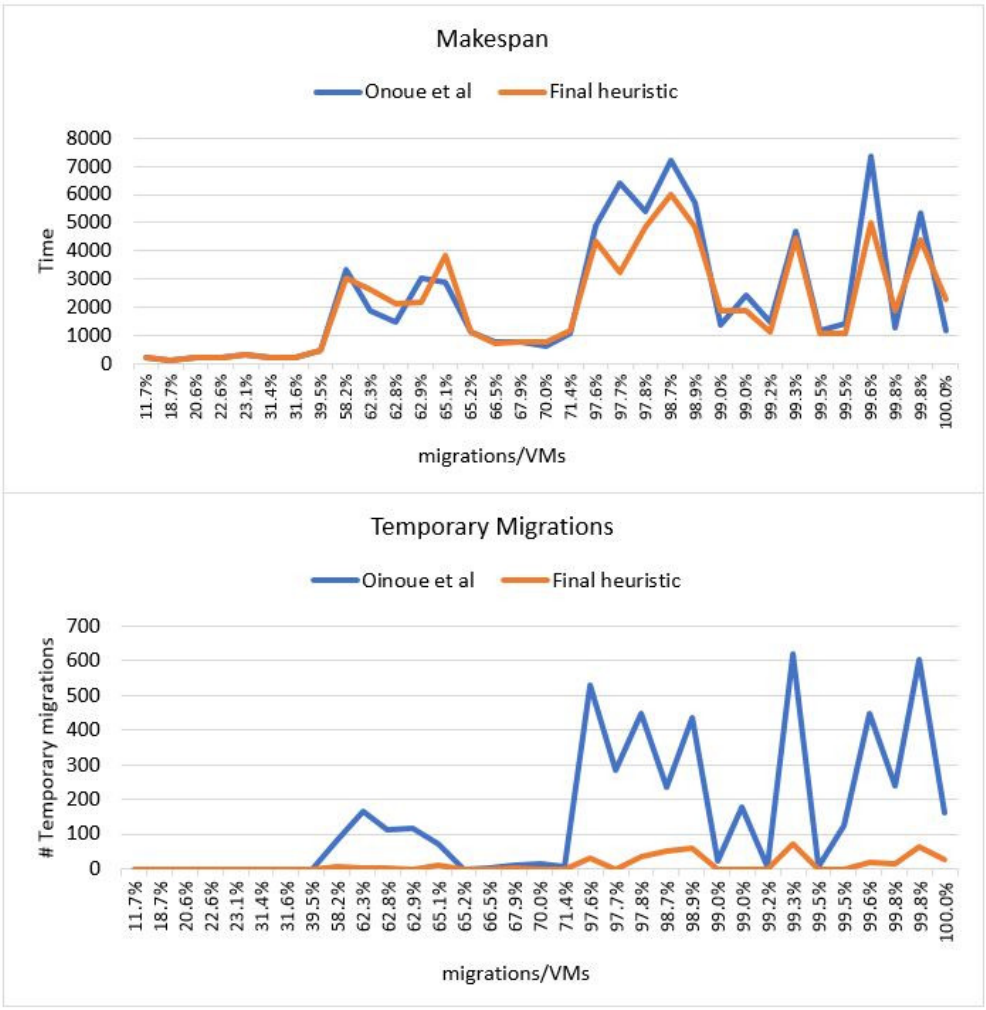

FIGURE 5.2: Comparing makespan and number of temporary migrations of final heuristic and Onoue *et al.* heuristic - datasets of 100 servers

Comparing the results of big dataset

The results for dataset of 200 servers in depicted in chart 5.4 and table 5.2. In the chart the results for two datasets with unfinished migrations for FF method (datasets DS 29 and 30) have been removed. On average the makespans for heuristic method of algorithm 8 is approximately 37% smaller. For comparing how long the process takes for each method, we can check the cpu time for FF and heuristic 8 (in ms).

	classification							First Fit heuristic				Heuristic			
Dataset Id	Placement Scenario	Type	Initila Load	Shuffle	Number of SV	Number of VMs	VM to migrate	Makespan	#VM moved	#temp migr	finished %	Makespan	#VM moved	#temp migr	finished %
DS 1	Cons	90-100	80-100	0	30	121	30	245	30	0	100%	245	30	0	100%
DS 2	Cons	100-100	80-100	0	30	134	35	123	35	0	100%	123	35	0	100%
DS 3	Cons	100-100	60-80	0	30	121	47	245	47	0	100%	245	47	0	100%
DS 4	Cons	90-100	60-80	0	30	126	54	245	54	0	100%	245	54	0	100%
DS 5	Cons	100-100	80-100	50	30	126	78	945	78	0	100%	945	78	9	100%
DS 6	Cons	100-100	60-80	50	30	118	86	733	86	0	100%	1213	86	0	100%
DS 7	Fail	1/3	80-100	50	30	140	72	759	72	0	100%	775	72	4	100%
DS 8	Cons	90-100	80-100	50	30	158	99	1117	99	0	100%	1117	99	6	100%
DS 9	DSHF		60-80	100	30	136	134	2069	134	2	100%	1495	134	4	100%
DS 10	Cons	100-100	60-80	100	30	100	100	1253	100	1	100%	917	100	10	100%
DS 11	Fail	1/3	60-80	0	30	127	46	193	46	0	100%	193	46	0	100%
DS 12	Fail	2/3	80-100	0	30	153	25	123	25	0	100%	123	25	0	100%
DS 13	Fail	2/3	60-80	0	30	129	38	123	38	0	100%	123	38	0	100%
DS 14	Fail	1/3	60-80	50	30	122	74	1511	74	0	100%	1415	74	7	100%
DS 15	Cons	90-100	60-80	50	30	114	83	611	83	0	100%	631	83	0	100%
DS 16	Cons	90-100	60-80	100	30	124	123	2273	123	2	100%	1309	123	20	100%
DS 17	Fail	1/3	80-100	100	30	124	114	2555	114	5	100%	1365	114	16	100%
DS 18	Fail	1/3	80-100	0	30	138	35	245	35	0	100%	245	35	0	100%
DS 19	Fail	2/3	60-80	50	30	125	87	501	87	0	100%	797	87	1	100%
DS 20	Fail	2/3	80-100	50	30	126	73	2427	73	1	100%	2087	73	8	100%
DS 21	LBAL		80-100	50	30	134	121	1509	121	0	100%	1387	121	0	100%
DS 22	DSHF		60-80	50	30	127	120	707	120	0	100%	707	120	0	100%
DS 23	LBAL		60-80	50	30	105	98	417	98	0	100%	445	98	0	100%
DS 24	LBAL		60-80	100	30	114	110	1051	110	0	100%	1051	110	0	100%
DS 25	DSHF		60-80	0	30	157	150	1275	150	1	100%	1111	150	0	100%
DS 26	Cons	90-100	80-100	100	30	133	126	1305	97	0	77%	1543	126	12	100%
DS 27	Fail	2/3	60-80	100	30	106	104	2083	104	0	100%	1927	104	2	100%
DS 28	DSHF		80-100	0	30	153	146	1241	93	0	64%	3871	146	106	100%
DS 29	DSHF		80-100	50	30	130	127	3657	127	0	100%	2369	127	11	100%
DS 30	LBAL		80-100	100	30	156	150	1263	127	2	85%	2863	150	24	100%
DS 31	Fail	1/3	60-80	100	30	113	112	973	112	0	100%	1041	112	2	100%
DS 32	Fail	2/3	80-100	100	30	128	120	1983	120	2	100%	1359	120	8	100%
DS 33	DSHF		80-100	100	30	112	108	2143	78	2	72%	3847	108	45	100%
DS 34	Cons	100-100	100-80	100	30	137	133	6969	133	5	100%	3147	133	23	100%
DS 35	LBAL		60-80	0	30	119	109	733	109	0	100%	697	109	0	100%
DS 36	LBAL		80-100	0	30	141	117	803	117	0	100%	803	117	1	100%

TABLE 5.1: Comparing results of final heuristic with FF heuristic - datasets of 30 servers

Comparing with exact algorithms

Onoue *et al.* MILP

Figure 5.5 and Table 5.3 depict the comparison of the result (Makespan - total migration time) for the datasets where the results have been calculated with the

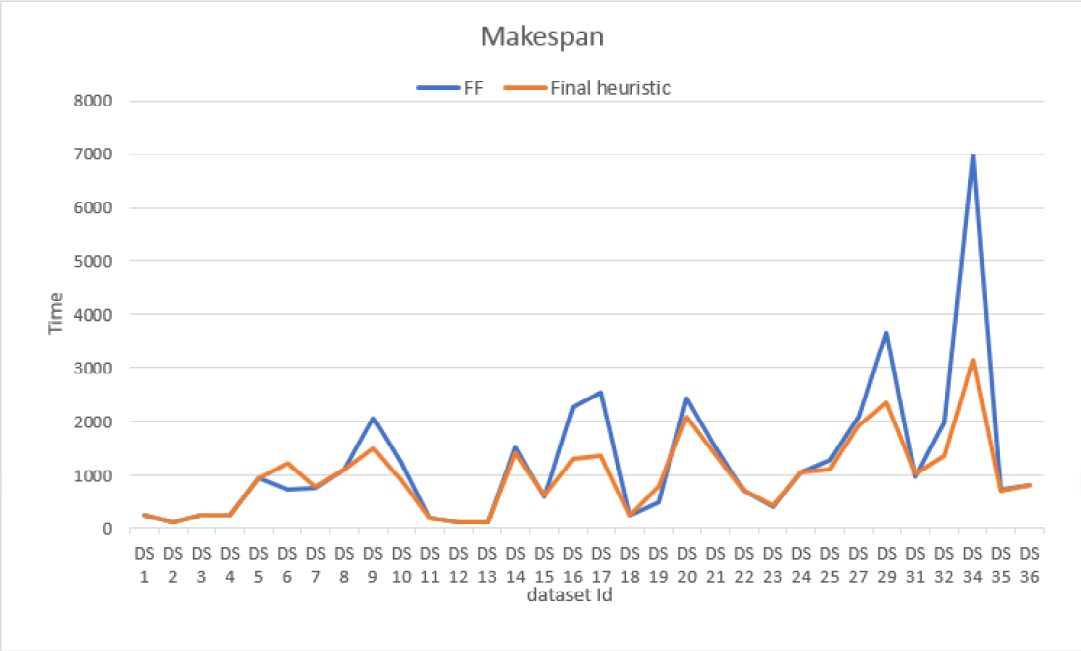

FIGURE 5.3: Comparing makespan of final heuristic with FF heuristic - datasets of 30 servers

exact method introduced in Onoue *et al.* [3]. The results are not complete on all the datasets due to setting a time limit for exact model to complete the calculations. The time limit was fairly long and the experience shows that the exact method is not salable for relatively big or complex problems. Nevertheless the results are compared with the heuristic for the finished runs.

Sequence based MILP

This mixed integer linear programming method which is introduced in the study of Jaumard *et al.* [29] is much more scalable than the MILP of Onoue *et al.* [3], which offers an excellent range of results for evaluating the result of the heuristic in this study. As we can see in Fig. 5.6, the makespan for datasets of 30 servers with the exact method introduced in [29] (MILP-JGL), is almost 30 percent less than the results of the final heuristic of this study.

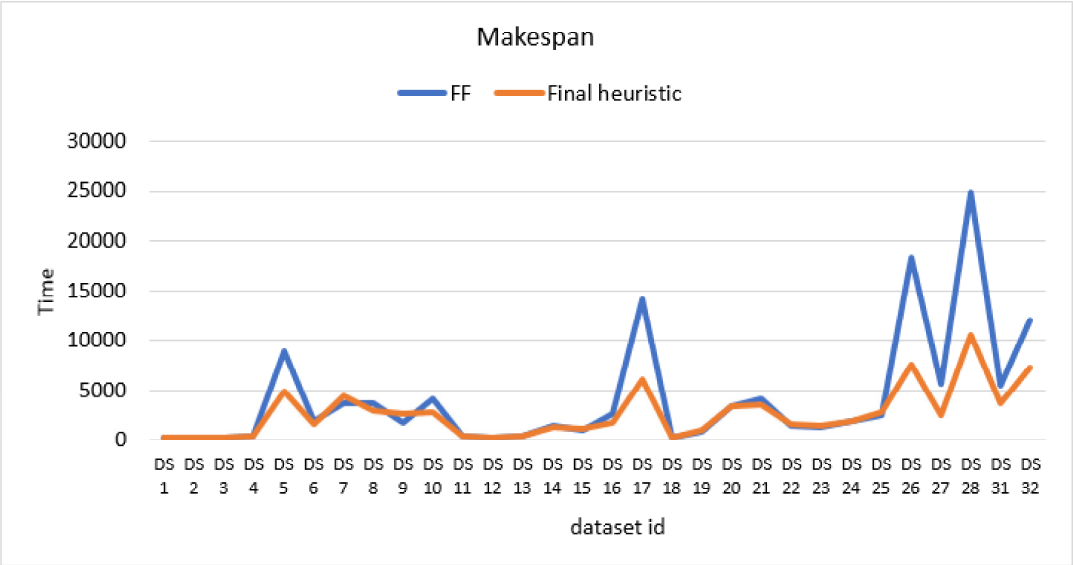

FIGURE 5.4: Comparing makespan of final heuristic with FF heuristic - datasets of 200 servers

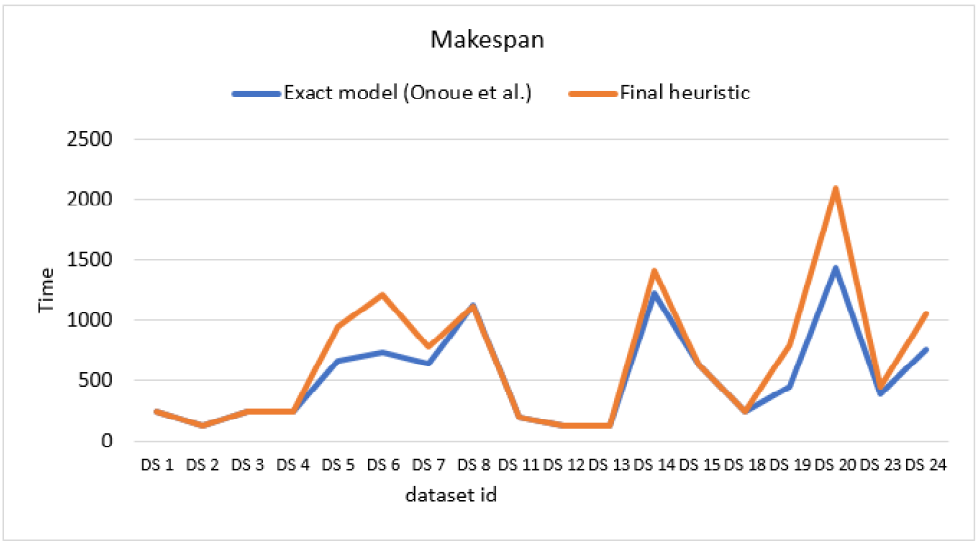

FIGURE 5.5: Comparing makespan of final heuristic with Onoue *et al.* exact model (MILP) - datasets of 30 servers

Numerical results for minimum temporary migrations heuristic

In section 4.2.2, we discussed a strategy to minimize number of temporary migrations with a cost of increasing the amount of memory transferred for temporary migrations. Here is the numerical results for comparing the number of temporary migrations for datasets with 50 servers. As we can see in Fig 5.7 the number of

	classification							First Fit heuristic			heuristic		
Dataset Id	Placement Scenario	Type	Initila Load	Shuffle	Number of SV	Number of VMs	VM to migrate	Makespan	CPU Time	Finsihed%	Makespan	cpu Time (on Linux Server) @edmonds.encs	Finsihed%
DS 1	Cons	90-100	80-100	0	200	871	205	245	9.26E+06	100%	245	77	100%
DS 2	Cons	100-100	80-100	0	200	853	212	245	5.24E+06	100%	245	300	100%
DS 3	Cons	100-100	60-80	0	200	772	299	257	1.20E+07	100%	257	697	100%
DS 4	Cons	90-100	60-80	0	200	792	316	489	1.05E+07	100%	489	73	100%
DS 5	Cons	100-100	80-100	50	200	894	570	8989	3.05E+07	100%	4853	9,235	100%
DS 6	Cons	100-100	60-80	50	200	756	523	1873	2.28E+07	100%	1679	1,209	100%
DS 7	Fail	1/3	80-100	50	200	890	573	3757	2.14E+07	100%	4457	9,301	100%
DS 8	Cons	90-100	80-100	50	200	885	554	3645	2.13E+07	100%	3013	6,657	100%
DS 9	DSHF		60-80	100	200	782	781	1831	1.97E+07	100%	2591	1,879	100%
DS 10	Cons	100-100	60-80	100	200	815	815	4225	2.93E+07	100%	2853	19,561	100%
DS 11	Fail	1/3	60-80	0	200	817	232	489	1.17E+07	100%	489	60	100%
DS 12	Fail	2/3	80-100	0	200	878	204	245	8.92E+06	100%	245	61	100%
DS 13	Fail	2/3	60-80	0	200	800	257	489	1.37E+07	100%	489	64	100%
DS 14	Fail	1/3	60-80	50	200	822	611	1437	1.91E+07	100%	1379	854	100%
DS 15	Cons	90-100	60-80	50	200	789	540	1099	1.75E+07	100%	1163	3,853	100%
DS 16	Cons	90-100	60-80	100	200	831	831	2617	2.99E+07	100%	1753	10,878	100%
DS 17	Fail	1/3	80-100	100	200	880	873	14143	4.96E+07	100%	6099	204,629	100%
DS 18	Fail	1/3	80-100	0	200	926	114	245	4.56E+06	100%	245	685	100%
DS 19	Fail	2/3	60-80	50	200	740	488	919	2.01E+07	100%	1005	367	100%
DS 20	Fail	2/3	80-100	50	200	878	542	3335	1.69E+07	100%	3455	3,286	100%
DS 21	LBAL		80-100	50	200	917	865	4089	2.79E+07	100%	3581	64,085	100%
DS 22	DSHF		60-80	50	200	788	783	1459	2.75E+07	100%	1595	15,288	100%
DS 23	LBAL		60-80	50	200	756	726	1343	1.99E+07	100%	1485	1,808	100%
DS 24	LBAL		60-80	100	200	809	804	1953	2.07E+07	100%	1953	8,653	100%
DS 25	DSHF		60-80	0	200	767	762	2529	2.09E+07	100%	2857	3,683	100%
DS 26	Cons	90-100	80-100	100	200	874	864	18329	6.13E+07	100%	7641	155,017	100%
DS 27	Fail	2/3	60-80	100	200	772	770	5637	2.73E+07	100%	2493	10,889	100%
DS 28	DSHF		80-100	0	200	870	860	24867	4.20E+07	100%	10543	111,249	100%
DS 29	DSHF		80-100	50	200	898	890	8193	3.26E+07	76%	10087	87,700	100%
DS 30	LBAL		80-100	100	200	840	831	4157	3.91E+07	98%	3909	98,746	100%
DS 31	Fail	1/3	60-80	100	200	773	773	5503	3.82E+07	100%	3747	13,029	100%
DS 32	Fail	2/3	80-100	100	200	826	814	12029	4.52E+07	100%	7301	94,568	100%

TABLE 5.2: Comparing results of final heuristic with FF heuristic - datasets of 200 servers

temporary migrations with the new strategy is fewer than the number of temporary migrations in heuristic 8 (on average 61% less) but the amount of bandwidth (memory transferred) for temporary migrations is on average 32% more, also the makespan is almost 10% more with the new strategy.

	classification							Exact Model (Onoue et al.)				Heuristic			
Dataset Id	Placement Scenario	Type	Initila Load	Shuffle	Number of SV	Number of VMs	VM to migrate	makespan	#VM moved	#temp migr	finished %	makespan	#VM moved	#temp migr	finished %
DS 1	Cons	90-100	80-100	0	30	121	30	245	30	0	100%	245	30	0	100%
DS 2	Cons	100-100	80-100	0	30	134	35	123	35	0	100%	123	35	0	100%
DS 3	Cons	100-100	60-80	0	30	121	47	245	47	0	100%	245	47	0	100%
DS 4	Cons	90-100	60-80	0	30	126	54	245	54	0	100%	245	54	0	100%
DS 5	Cons	100-100	80-100	50	30	126	78	661	78	0	100%	945	78	9	100%
DS 6	Cons	100-100	60-80	50	30	118	86	733	86	0	100%	1213	86	0	100%
DS 7	Fail	1/3	80-100	50	30	140	72	637	72	0	100%	775	72	4	100%
DS 8	Cons	90-100	80-100	50	30	158	99	1131	99	0	100%	1117	99	6	100%
DS 11	Fail	1/3	60-80	0	30	127	46	193	46	0	100%	193	46	0	100%
DS 12	Fail	2/3	80-100	0	30	153	25	123	25	0	100%	123	25	0	100%
DS 13	Fail	2/3	60-80	0	30	129	38	123	38	0	100%	123	38	0	100%
DS 14	Fail	1/3	60-80	50	30	122	74	1221	74	0	100%	1415	74	7	100%
DS 15	Cons	90-100	60-80	50	30	114	83	627	83	0	100%	631	83	0	100%
DS 18	Fail	1/3	80-100	0	30	138	35	245	35	0	100%	245	35	0	100%
DS 19	Fail	2/3	60-80	50	30	125	87	449	87	0	100%	797	87	1	100%
DS 20	Fail	2/3	80-100	50	30	126	73	1435	73	0	100%	2087	73	8	100%
DS 23	LBAL		60-80	50	30	105	98	383	98	0	100%	445	98	0	100%
DS 24	LBAL		60-80	100	30	114	110	761	110	0	100%	1051	110	0	100%

TABLE 5.3: Comparing results of final heuristic with MILP-Onoue - datasets of 30 servers

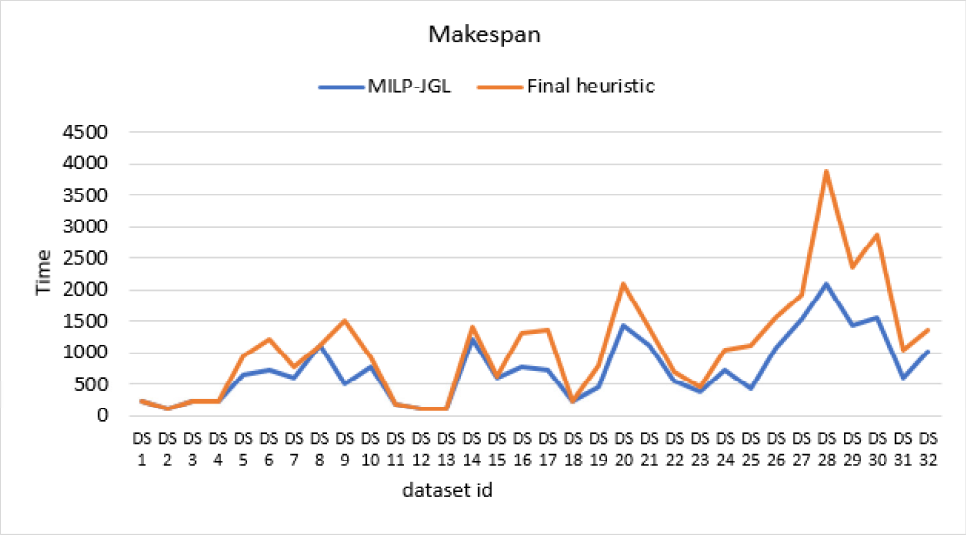

FIGURE 5.6: Comparing Makespan of final heuristic with MILP of Jaumard *et al.* (MILP-JGL) - datasets of 30 servers

Numerical results based on the virtual machine placement scenarios

In the previous section, we compared the Onoue *et al.* (Algorithm 3) and final heuristic (Algorithm 8) for datasets containing data generated with a combination of different scenarios. For comparing each placement scenario separately, we

	classification							MILP-JGL	Heuristic
Dataset Id	Placement Scenario	Type	Initila Load	Shuffle	Number of SV	Number of VMs	VM to migrate	Makespan	Makespan
DS 1	Cons	90-100	80-100	0	30	121	30	245	245
DS 2	Cons	100-100	80-100	0	30	134	35	123	123
DS 3	Cons	100-100	60-80	0	30	121	47	245	245
DS 4	Cons	90-100	60-80	0	30	126	54	245	245
DS 5	Cons	100-100	80-100	50	30	126	78	661	945
DS 6	Cons	100-100	60-80	50	30	118	86	733	1213
DS 7	Fail	1/3	80-100	50	30	140	72	611	775
DS 8	Cons	90-100	80-100	50	30	158	99	1117	1117
DS 9	DSHF		60-80	100	30	136	134	516	1495
DS 10	Cons	100-100	60-80	100	30	100	100	769	917
DS 11	Fail	1/3	60-80	0	30	127	46	193	193
DS 12	Fail	2/3	80-100	0	30	153	25	123	123
DS 13	Fail	2/3	60-80	0	30	129	38	123	123
DS 14	Fail	1/3	60-80	50	30	122	74	1221	1415
DS 15	Cons	90-100	60-80	50	30	114	83	611	631
DS 16	Cons	90-100	60-80	100	30	124	123	765	1309
DS 17	Fail	1/3	80-100	100	30	124	114	715	1365
DS 18	Fail	1/3	80-100	0	30	138	35	245	245
DS 19	Fail	2/3	60-80	50	30	125	87	449	797
DS 20	Fail	2/3	80-100	50	30	126	73	1435	2087
DS 21	LBAL		80-100	50	30	134	121	1119	1387
DS 22	DSHF		60-80	50	30	127	120	561	707
DS 23	LBAL		60-80	50	30	105	98	383	445
DS 24	LBAL		60-80	100	30	114	110	733	1051
DS 25	DSHF		60-80	0	30	157	150	439	1111
DS 26	Cons	90-100	80-100	100	30	133	126	1061	1543
DS 27	Fail	2/3	60-80	100	30	106	104	1529	1927
DS 28	DSHF		80-100	0	30	153	146	2091	3871
DS 29	DSHF		80-100	50	30	130	127	1441	2369
DS 30	LBAL		80-100	100	30	156	150	1543	2863
DS 31	Fail	1/3	60-80	100	30	113	112	611	1041
DS 32	Fail	2/3	80-100	100	30	128	120	1019	1359

TABLE 5.4: Comparing makespan of final heuristic with MILP of Jaumard *et al.* (MILP-JGL) - datasets of 30 servers

generated datasets of 50 servers based on each scenario (between 30 - 60 datasets for each scenario). The results (makespan and number of temporary migrations) are depicted in Figure 5.8, the makespans are moderately better for final heuristic compared to Onoue *et al.* heuristic (Algorithm 3), while number of temporary

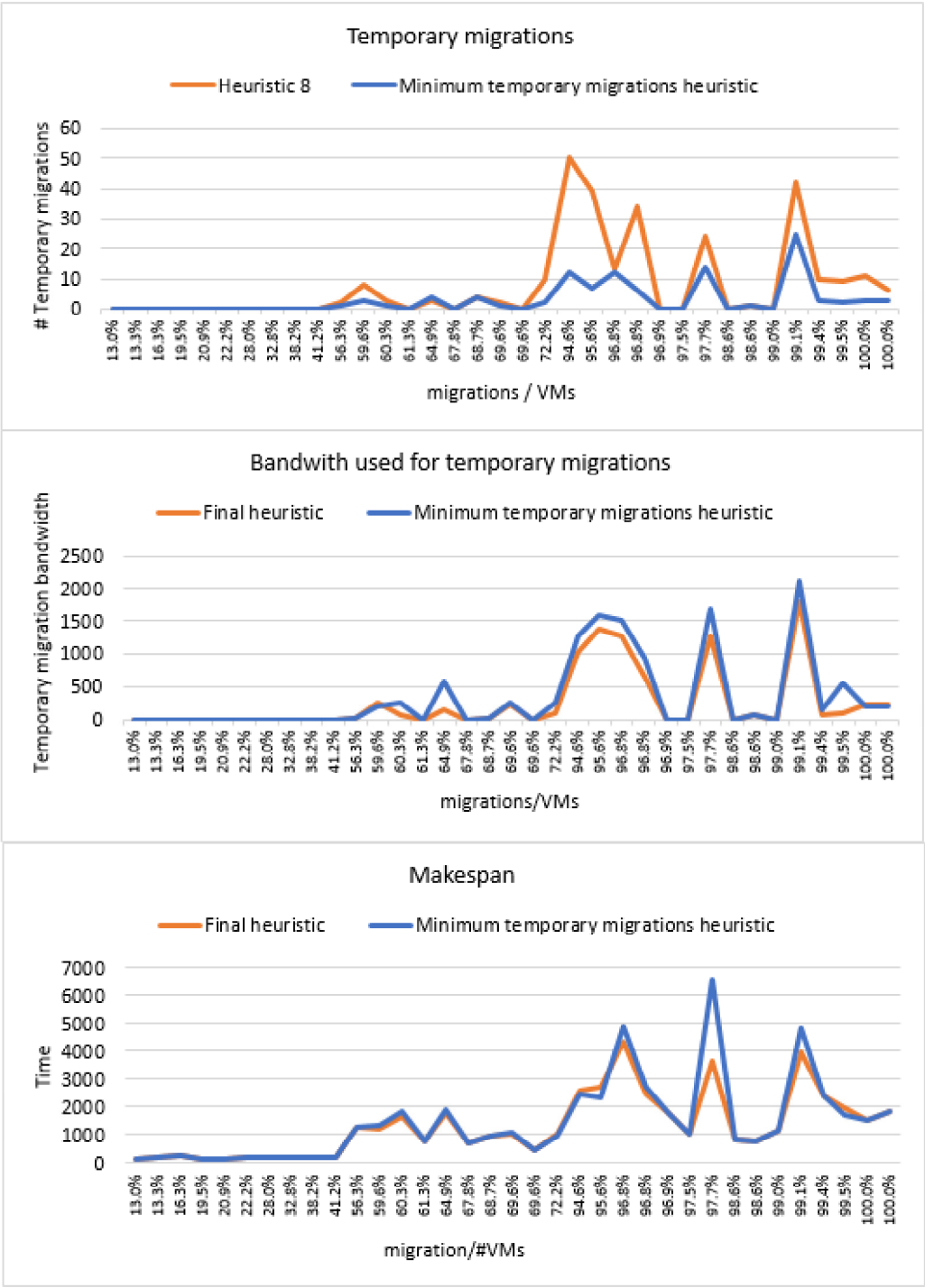

FIGURE 5.7: Compare results of final heuristic with minimum temporary migrations heuristic - datasets of 50 servers

migrations are on average 86% less than Onoue *et al.* heuristic. As we can see for the *deadlock shuffle* scenario which aims to create complex migration plans, makespan of final heuristic is clearly less than the other one.

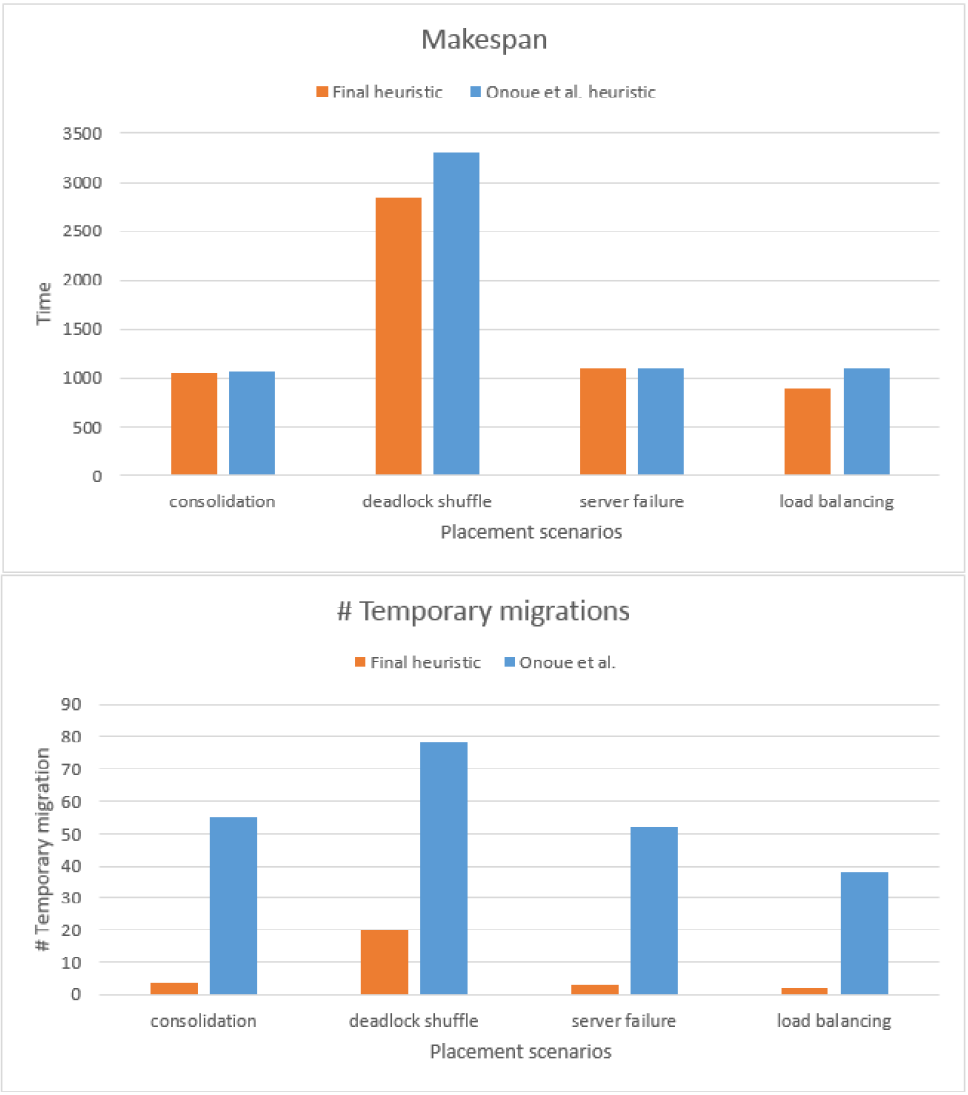

FIGURE 5.8: Comparing makespan and number of temporary migrations of final heuristic - grouped by different Scenarios - datasets of 50 servers

Chapter 6

Conclusion and Future Works

In this study, we designed heuristics for optimizing migration of multiple virtual machines with the main goal of minimizing the makespan. Our final heuristic has objective of minimizing makespan of multiple VM migrations while reducing number of temporary migrations or the data transferred for temporary migrations (different scenarios). By nature this problem has computational complexity, most of the exact mathematical models in the literature are not scalable except a recent one which is mentioned in this study and is used for evaluating our results. Heuristics in the literature are scalable, however beside trying to optimize makespan, they do not consider other factors such as number of temporary migrations they use to reach the final state of placements. Also, heuristics in the literature do not guaranty finishing all the migrations for complex problem where limitation of resources makes it hard to find a feasible order of migrations. In our heuristic, we achieve to reduce number of temporary migrations beside minimizing the makespan with observing the behavior of algorithm in various difficult cases. We evaluated our heuristic with comparing parameters such as makespan, number of temporary migrations and calculation (running) time of our algorithm with other heuristics and also exact models in the literature. The results show the quality of our solution.

Future work will include a mix problem where beside minimizing makespan and number of temporary migrations for multiple parallel VM migrations, optimization of each individual migration will be considered too. This can be achieved apparently by adding dynamic bandwidth assignments to migrations based on the network load and VM characteristics beside dynamic selection of migration techniques (pre-copy, post-copy or hybrid) for each migration.

Bibliography

[1] A. Varasteh and M. Goudarzi. Server consolidation techniques in virtualized data centers: A survey. *IEEE Systems Journal*, 11(2):772–783, 2015.

[2] M. Noshy, A. Ibrahim, and H. Arafat Ali. Optimization of live virtual machine migration in cloud computing: A survey and future directions. *Journal of Network and Computer Applications*, 110:1–10, 2018.

[3] K. Onoue, S. Imai, and N. Matsuoka. Scheduling of parallel migration for multiple virtual machines. In *IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 827–834, March 2017.

[4] G. Sun, D. Liao, V. Anand, D. Zhao, and Y. Hongfang. A new technique for efficient live migration of multiple virtual machines. *Future Generation Computer Systems*, 55 (Supplement C):74–86, 2016.

[5] U. Deshpande and K. Keahey. Traffic-sensitive live migration of virtual machines. *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 50–60, 2015.

[6] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpac, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286. USENIX Association, 2005.

[7] A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755–768, 2012.

[8] X Xiaoqiao, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. *IEEE Annual Joint Conference of the IEEE Computer and Communications Societies - INFOCOM*, pages 1–9, 2010.

[9] W. Deng, F. Liu, H. Jin, X. Liao, and H. Liu. Reliability-aware server consolidation for balancing energy-lifetime tradeoff in virtualized cloud datacenters. *International Journal of Communication Systems*, 27(4):623–642, 2013.

[10] R. Li, Q. Zheng, X. Li, and J. Wu. A novel multi-objective optimization scheme for rebalancing virtual machine placement. *IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 710–717, 2016.

[11] A. Roytman, A. Kansal, S. Govindan, J. Liu, and S. Nath. Pacman: Performance aware virtual machine consolidation. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 83–94. USENIX, 2013.

[12] A. Sansottera, D. Zoni, P. Cremonesi, and W. Fornaciari. Consolidation of multi-tier workloads with performance and reliability constraints. *International Conference on High Performance Computing and Simulation (HPCS)*, pages 74–83, 2012.

[13] E. Dow and J. Matthews. Wayfinder: parallel virtual machine reallocation through A* search. *Memetic Computing*, 8(4):255–267, 2016.

[14] A. Wolke, M. Bichler, and T. Setzer. Planning vs. dynamic control: Resource allocation in corporate clouds. *IEEE Transactions on Cloud Computing*, 4(3):322–335, Jan 2016.

[15] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation, 2011.

[16] G. Dosa. The tight bound of first fit decreasing bin-packing algorithm. In *Proceedings of the First International Conference on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, ESCAPE'07, pages 1–11. Springer-Verlag, 2007.

[17] O. Ayoub, F. Musumeci, M. Tornatore, and A. Pattavina. Efficient routing and bandwidth assignment for inter-data-center live virtual-machine migrations. *IEEE/OSA Journal of Optical Communications and Networking*, 9(3):B12–B21, 2017.

[18] K.s Tsakalozos, V. Verroios, Mema Roussopoulos, and A. Delis. Live vm migration under time-constraints in share-nothing iaas-clouds. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2285–2298, 2017.

[19] F. Tian, R. Zhang, J. Lewandowski, K.-M. Chao, L. Li, and B. Dong. Deadlock-free migration for virtual machine consolidation using chicken swarm optimization algorithm. *Journal of Intelligent and Fuzzy Systems*, 32(2):1389–1400, 2017.

[20] U. Mandal, P. Chowdhury, M. Tornatore, C. Martel, and B. Mukherjee. Bandwidth provisioning for virtual machine migration in cloud: Strategy and application. *IEEE Transactions on Cloud Computing*, 6(4):967–976, Jan 2018.

[21] A. Verma, P. Ahuja, and A. Neogi. *pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems*, pages 243–264. Springer Berlin Heidelberg, 2008.

[22] S. Takeda and T. Takemura. a rank-based vm consolidation method for power saving in data centers. *Information and Media Technologies*, 5(3):994–1002, 2010.

[23] S. Takahashi, H. Nakada, T. Atsuk, K. Tomohiro, S. Maiko, and Y. Akiko. Virtual machine packing algorithms for lower power consumption. *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 161–168, 2012.

[24] C. Ghribi, M. Hadji, and D. Zeghlache. Energy efficient vm scheduling for cloud data centers: Exact allocation and migration algorithms. *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 671–678, 2013.

[25] V. Kherbache, E. Madelaine, and F. Hermenier. Scheduling live migration of virtual machines. *IEEE Transactions on Cloud Computing*, pages 1–14, 2018.

[26] M. F. Bari, M. F. Zhani, Q. Zhang amd R. Ahmed, and R. Boutaba. Cqncr: Optimal vm migration planning in cloud data centers. In *2014 IFIP Networking Conference*, pages 1–9, June 2014.

[27] M. Gilesh, S. MadhuKumar, and J. Lillykutty. Bounding the cost of virtual machine migrations for resource allocation in cloud data centers. In *ACM SAC Cloud Computing Track*, pages 201–206, 2018.

[28] U. Deshpande and K. Keahey. Traffic-sensitive live migration of virtual machines. *Future Generation Computer Systems*, 72:118–128, 2017.

[29] B. Jaumard, O. Gluck, and D. Le. Effectiveness of heuristics for vm migration. *Paper in preparation*, 2019.

[30] O. Popoola and B. Pranggono. On energy consumption of switch-centric data center networks. *The Journal of Supercomputing*, 74(1):334–369, Jan 2018.

[31] Amazon ec2 instance types - amazon web services. https://aws.amazon.com/ec2/instance-types. Online; accessed 20-June-2019.

[32] Dell servers. https://www.dell.com/fr-fr/work/shop/serveurs-dell-poweredge/sc/servers/poweredge-rack-servers. Online; accessed 15-June-2019.