VIRTUAL MACHINE MIGRATION: GREEDY HEURISTICS AND MATHEMATICAL MODELS

Charles Boudreau

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
Concordia University
Montréal, Québec, Canada

December 2020

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Charles Boudreau**

Entitled: **Virtual Machine Migration:**
Greedy Heuristics and Mathematical Models

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

Master of Computer Science

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

Signed by the final examining committee:

_____ Chair
Dr. Dhrubajyoti Goswami

_____ Examiner
Dr. Dhrubajyoti Goswami

_____ Examiner
Dr. Olga Ormandjieva

_____ Supervisor
Dr. Brigitte Jaumard

Approved _____
Dr. Lata Narayanan, Chair of Department

_____ 20 _____ _____
Mourad Debbabi, Acting Dean
Gina Cody School of Engineering and Computer Science

Abstract

Virtual Machine Migration:
Greedy Heuristics and Mathematical Models

Charles Boudreau

Live Migration of virtual machines between different physical hosts is an important process for datacenter management, essential for safeguarding hardware integrity and controlling power consumption, among other functions, with no perceptible interruptions for the user of the virtual machine. As the migration operation has a cost in terms of power and service quality degradation, it is of interest to examine how to best conduct multiple live migrations such that the total time needed to complete all planned migrations is minimized. Scheduling of multiple VM migrations may also take into account the risk of bringing the system to a state where all planned migrations cannot be resolved due to deadlocks caused by resource dependencies.

In this work, we propose a set of solutions based on greedy heuristics for the VM migration problem. We have selected four possible criteria to base scheduling decisions on, and we evaluate the total migration time degree of completion of the planned migrations next to a baseline algorithm. Additionally, we propose two decomposed linear programming models intended for column generation solution: a time-based formulation, followed by a precedence-based formulation. We suppose that these decomposed formulations will lead to faster solution times over conventional, "compact" formulations due to their structure permitting the elimination of a large number of variables from explicit consideration when the continuous relaxation is solved with column generation techniques.

Acknowledgments

I would like to express my sincerest gratitude to my supervisor, Professor Brigitte Jaumard, for her continued support during the course of my graduate studies. I would also like to thank my parents for their support and encouragement.

Contents

List of Figures

List of Tables

List of Acronyms

ILP integer linear programming. 4, 10

MDG migration dependency graph. 25

MILP mixed integer linear programming. 13

QoS quality of service. 17

SLA service level agreement. 1

VM virtual machine. 1

Chapter 1

Introduction

1.1 General Background

Virtualization technology enables the partition of computing resources into separate environments in order to allow multiple users to make use of the same shared physical computing resources [32]. These separate environments, or Virtual Machines (VM), grant each user an operating system and applications housed on the same machine without mutually affecting each other. A development of the time-sharing approach, virtualization would not come into wide usage until improvements in network bandwidth would allow for the advent of Cloud Computing, in which vendors lease their spare computing resources for customers to access through an Internet connection, thus allowing users to make use of high-end hardware at a fraction of the price it would take for them to purchase said physical computing resources.

A large concentration of virtual machines on one or over several datacenters necessitates methods for their management. Virtual Machine migration allows the redeployment of Virtual Machines from one physical machine to another, in most cases without users being aware. Possible use cases include rebalancing VM load over the servers in order to preserve hardware integrity and Service Level Agreements (SLA), clearing a server on the verge of failing or scheduled to go down for maintenance, or optimizing power and network utilization.

VM migration can be live or non-live. In non-live migration, the VM is either suspended or shut down, and then migrated to its destination in its entirety. The migrating VM is thus unavailable to the user during the process. Live migration,

by contrast, allows the virtual machine to stay active and available during the almost totality of the migration process, being shut down for a small period of time, imperceptible to users in most cases, thus allowing users to continue to make use of it. Several different methods of achieving live migration exists, which have their advantages and faults concerning migration time and service interruption.

The performance of a migration operation is evaluated along several criteria, chief of which is the total migration time, or makespan, of the migration [32]. Beyond this, migrations incur other side-effects that a good migration should seek to minimize. Among these are *Downtime*, or the amount of time during which the virtual machine is out of service, *Total Network Traffic*, or the bandwidth usage over the network, as bandwidth used for the purpose of migration cannot be used for services which may require it, and *Service Degradation*, or how migrations affect other running services.

1.2 Project

This study concerns the elaboration of migration scheduling schemes in order to fulfill certain objectives. Said objectives may vary from obtaining a desired final placement of virtual machines from a given initial placement in the shortest time possible, or to dynamically find a new placement for the VMs in the given network that would fulfill certain conditions such as minimizing power consumption or improving service quality. This process assumes that the migrations are online and providing services to end users, and as such the migrations must be performed with minimal interruption to ongoing services. This requirement for live migration thus complicates the task by adding requirements to maintain a certain level of service quality for the duration of the move.

One issue to look out for during the migration process is the possibility of deadlocks. A "deadlock" occurs when a group of VMs are unable to move to their assigned destination due to a lack of available resources causing a cyclical dependence. Such cases, if present, essentially render the problem of devising a migration schedule impossible without organizing a temporary migration, where one of the deadlocked machines is temporarily moved to a server with spare resources in order to undo the deadlock and allow the planned migrations to complete.

Figure 1 show a very simple case of a deadlock. Here, both VM1 and VM2 are

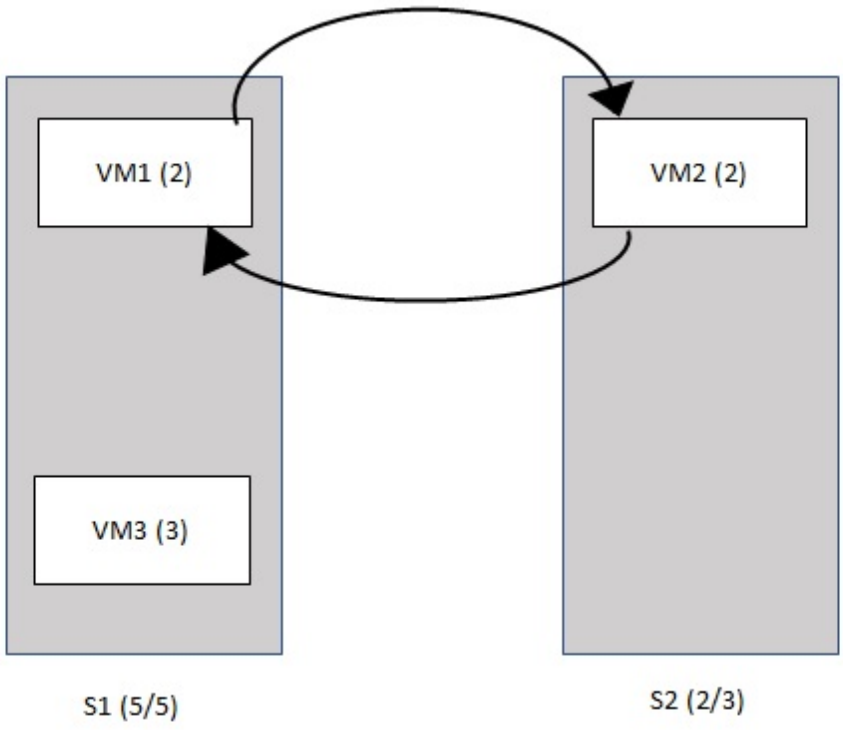

Figure 1: Simple Example of a Deadlock

unable to reach their destination due to a lack of available resources for both servers. The only way to remedy this state would be to either move VM3, or move VM1 or VM2 to another server in the network, long enough for the other to complete its migration and free up the necessary space.

Deadlocks can be classified into two types, "direct" or "indirect", based on how they arise [28]. A *direct* deadlock is present in the migration problem from the start, whereas an *indirect* deadlock comes to be as a result of poor choices made during the migration process. While there is no way to prevent the former, which can only be resolved through a temporary migration, the latter can be avoided provided a proper migration order is selected. The ability to avoid *indirect* deadlocks must therefore be considered when designing heuristics in order to ensure all possible migrations can be scheduled.

Where heuristics are concerned, there is also the possibility that a deadlock may occur as a result of a poor choice regarding migration order. While determining the

migration order, heuristics would need to find some way to avoid these "indirect" deadlocks, so as to avoid having to fix them down the line.

Another issue to address is the scalability of the solutions presented. As covered in the literature review, the majority of the solutions presented for the VM migration problem utilise either mathematical models or metaheuristics. We are interested in solutions that can be implemented in order to solve the problem in real time, so we need a solution whose execution time is short and scales well as the size of the problem increases. Mathematical Programming models are too slow to provide solution on-line, and while metaheuristics are faster than the latter, they are still not quite fast enough for this purpose. For this reason, we look into the use of greedy heuristics to solve the VM migration problem as their simplicity allows them to provide solutions relatively quickly even as the problem size increases.

Beyond the elaboration of greedy heuristic solutions, this study also concerns the creation of new mathematical programming models for the VM migration problem. While Integer Linear Programming (ILP) models cannot be used for on-line problem resolution, as previously stated, they nevertheless have value in that the solutions they provide can be used as a benchmark to evaluate the quality of other solutions. As the ILP models surveyed in this work are all conventional, "compact" models, we look into "decomposed" ILP models, who are structured in a manner that allows a technique knows as "column generation" to obtain a solution from the model faster than could be obtained by solving a more traditional formulation.

1.3 Contributions

We propose a set of greedy heuristics that aim to complete all scheduled migrations and avoid creating deadlock situations in the process, while keeping total migration time as low as possible. We use the result of the ILP model of Jaumard *et al.* [17] on a set of test problems to evaluate the performance of our heuristics. Experimental results show inconsistent performance compared to benchmarks and failure to avoid indirect deadlocks in some problems.

We also present two new large scale optimization models for the VM migration problem. The first one is a discretized, time-based model, and the second one, a model based on precedence relations between planned migrations. These decomposed

models are created for use with column generation, which can exploit the structure of a decomposed model in order to solve it faster than a conventional, "compact" model. As the time to solve these models can be quite long, improvements in the solution time are still valuable, even if these models will still not be fast enough for on-line solution generation.

1.4 Plan of the Thesis

The thesis is organized as follows. Chapter 2 covers the technical background for the problem of Virtual Machine migration. Chapter 3 collects the literature covering the research problem of the thesis. Chapter 4 details the heuristics used to attempt to solve the migration problems. Chapter 5 shows the results of the heuristics on a set of test problems. Chapter 6 presents a set of alternate modelings of the VM migration problem. Chapter 7 presents the conclusions and future research directions.

Chapter 2

Background

2.1 Datacenters

2.1.1 Migration Motivation

Virtual machine migration can be used to reduce power consumption in a datacenter by more efficiently distributing VMs in the system. One of the most significant sources of waste in such datacenters is the continued operation of lightly-loaded servers [12]. Redistributing VMs within the network would reduce power consumption by allowing more servers to be unloaded and powered off. Reduction of power consumption would also consequently lead to lowered carbon emissions. Such consolidation may also increase service quality, in the case of multiple VMs needing to communicate with each other, by reducing the distance between them.

Similarly, virtual machine migration allows for the continued use of a VM located on a server on the verge of failing or scheduled to go down for maintenance. By moving VMs to other servers through live migration, it is possible to perform maintenance on a physical machine without adversely affecting user experience. Thus, over time, maintenance operations can be performed on an entire datacenter in such a manner that is imperceptible to users.

VM migration is also used for load balancing within a system. Load balancing seeks to redistribute VMs within a network in order to promote access to resources as well as performance [6]. The aim is to minimize the number of both underloaded and overloaded servers, as the former are an inefficient allocation of power while the latter may potentially compromise service quality and hardware lifespan [32].

In cases where VMs can travel across multiple datacenters, moving VMs from one site to another can further improve performance [32]. For example, moving a VM to a datacenter closer to its user can improve performance by minimizing latency. The VM may also need to move again if the user is in motion, shifting to the closest machine as the user moves. Alternately, it may be advantageous to move VMs to locations where it is night time or the weather is cooler, in order to take advantage of the colder weather of these locations to regulate equipment temperature, thus saving power by relying less on powered means of heat control such as fans or air conditioning.

2.1.2 Migration: pre-copy/postcopy

The migration of a virtual machine from one host to another with no or very little interruption perceptible by the user can be handled in several ways, chief of which are the following.

The *pre-copy* approach migrates the VM's contents while the VM is still in operation. In the event that a transferred item has been modified on the original VM (the "page" has been "dirtied"), the modified content is also sent during the course of the migration operation. This process continues until either it reaches a previously determined iteration limit or the remaining content to be transferred is determined to be small enough. Then, the VM is halted and its processor state as well as the remaining unsent data is copied over to the migration's destination before being restarted [16].

The *post-copy* approach essentially reverses the steps in the *pre-copy* scheme. Here, the machine's CPU state is halted, transferred and restarted first, and memory transfer follows. This way, each memory page would be transferred at most once, contrary to *pre-copy*, where a page may be moved multiple times if it is faulted during the migration process. However, *post-copy* carries the risk that an as-of-yet untransferred page could be faulted, thus needing to go through the network to update it, see, e.g., [16].

Finally, *Hybrid Copy* acts as a compromise between the two previously-mentioned approaches. *Hybrid Copy* first begins by copying memory pages over to the destination server in much the same way *pre-copy* does, but only copies the most frequently-used pages. Once this is done, the VM state is transferred and the rest of the migration process resembles the *post-copy* method. *Hybrid copy* thus attempts to minimize the drawbacks of both previous methods by transferring a smaller amount of memory in

the "pre-copy" phase, and minimizing potential page faults in the "post-copy" phase by having transferred the most often-consulted pages beforehand.

2.2 Different Problem Statements

2.2.1 Problem Statement: Generalities

In this thesis, we investigate the VM migration problem stated as follows. Consider a could infrastructure, i.e., a data center with a given set of servers (S), indexed by SV. Each server hosts a given number of VMs. Let V denote the set of VMs, indexed by VM.

Our objective is to find an order of migrations that allows us to take a given "current" placement (PL_c) and bring it to a desired "target" placement (PL_n), where "placement" refers to an assignment of VMs to servers in the network (i.e., datacenter). At the end of the migration process, all VMs must be in their designated server as detailed by the final placement, and this state must be reached in the shortest time possible. Each VM requires the use of a certain number of processors ($\text{REQ}_{\text{VM}}^{\text{CPU}}$) and units of RAM ($\text{REQ}_{\text{VM}}^{\text{RAM}}$) in order to fulfill its duties. Servers are limited in how many VMs they can accommodate following their own resources, for both thier number of processors ($C_{\text{SV}}^{\text{CPU}}$) and their RAM ($C_{\text{SV}}^{\text{RAM}}$). Here, VM resource requirements are considered static and do not vary with workload.

This work concerns the intra-DC case specifically, which means all servers are part of the same datacenter. In addition, no special constraints apply to the links connecting the servers together, so the migration time of an individual VM is primarily a function of it's RAM requirements, and migrations are assumed to have their own dedicated bandwidth, so no other function of the datacenter can interfere with migration or otherwise alter migration time.

2.2.2 Initial & final states

In this case, both an initial state and final state are given, and the model or heuristic is tasked with moving VMs until the VMs are hosted in their assigned servers as designated by the Final state. The model or heuristic moves towards this final state while trying to minimize makespan as well as any other constraints imposed upon it

(Network Usage, Power Consumption, etc.).

2.2.3 Initial state but no final state

In this case, there is a given initial state but no final state, and the algorithm is tasked with performing migrations until the system state fulfills a given goal. VMs could be moved around to fulfill goals such as reducing power consumption through consolidation onto as small a set of active Servers as possible. In such a case, the total migration time could be controlled by adding a cost or penalty to moving VMs around, thus discouraging frivolous moves. Other goals such as load balancing could be fulfilled by moving VMs until the number of Over and Under-utilized Servers has been minimized.

Chapter 3

Literature Review

We present a review of studies covering solutions to the virtual machine migration problem. Section 3.1 covers solutions that employ mathematical models such as ILP models to derive solutions. Section 3.2 covers a variety of heuristic solutions, both classical and meta-heuristic ones, as well as other techniques such as reinforcement learning. In the last section of this chapter, we explain why representing precedence relations faithfully using a graph is difficult, and why many methods that make use of such a graph choose to simplify the precedence relations that bind the VMs.

3.1 Exact Methods

Methods using different types of mathematical models, which we refer to as "exact methods", are of interest first and foremost as a means of evaluating the performance of other types of solution methods such as heuristics. They are not viable for solving problems in a practical context due to poor scaling with problem size. We next distinguish the studies focusing on intra-site vs. cross-site, although models are not very different.

3.1.1 Intra-site Migration

Ghribi *et al.* [13] propose a mathematical model that combines new VM allocation and VM consolidation, seeking to allocate VMs while also minimizing the number of migrations and power consumption. Power consumption is indirectly minimized by attempting to maximize the number of "idle" servers, or servers that are not currently

hosting a VM. The authors compare the performance of their model with an energy consumption-conscious variant of the Best Fit algorithm. Results show that the exact algorithm uses anywhere from 10-50% less servers than the Best-Fit algorithm, depending on dataset characteristics. When comparing the combination of allocation and migration algorithm with allocation alone, the former allows the switching off of 10-20% more servers. Moreover, both exact algorithms show consistent improvement in energy savings over best fit. Unfortunately, the convergence time of the exact methods grows exponentially with the number of servers, growing past acceptable standards for practical applicability when trying to consolidate in a network of 20 nodes hosting 80 VMs, which limits their practicality

Onoue *et al.* [23] propose an ILP model for migration scheduling in addition to a heuristic algorithm. The model seeks to minimize both overall migration time (i.e., makespan) and number of migrations, while respecting constraints pertaining to the completion of all underway migrations, respecting resource capacities on the servers and the network itself, and ensuring each VM is only deployed on 1 server at any given time. Results show that, on a two eight-core 2.90 GHz processors and 64 GB RAM, the ILP was often unable to determine a solution in a reasonable amount of time. In some cases, the ILP could not resolve at all before reaching its time limit, which was set at 1 hour.

Nasim *et al.* [21] present a mathematical model for VM migration that also takes into account uncertainties in regards to machine resource requirements and power consumption. The authors note that most other models assume input is known exactly, when this may not be the case in practice. Power consumption models, VM resource demand and migration overhead among others may vary and render an "optimal" solution that assumes input to be known and fixed as unfeasible. The model, based on Robust Optimization, assumes inputs are not exactly known, but fall within a given set of bounds. Results compare the performance of the model for various data instances where uncertainty may lie in one of the aforementioned inputs and determine the tradeoff between power consumption and probability of violating the constraints. By adjusting the model's level of protection against uncertainty, it is possible to reduce the probability of an SLA violation from 50% at 0 protection to less than 1 %. This reduction of SLA chance comes at a power cost that is variable according to the degree of uncertainty in the inputs .

Xu *et al.* [31] propose an optimization model for VM migration that pays special attention to the costs incurred by deciding to migrate. The authors note that many other power-aware VM migration schemes seek to save energy by minimizing the number of migrations overall, without taking into account migration costs which can vary based on individual machines and circumstances. The authors also take into account VM requirements that can change from one instance of time to another, and thus need to address sudden violations of server capacity. To this end, they propose a model whose objective is to minimize the number of active servers operating at any instance of time, subject to constraints pertaining to server capacity, SLA violations and controlling whether or not to migrate a VM based on its remaining execution time.

Saber *et al.* [26] examine the use of optimization solvers such as CPLEX applied to VM Migration problems, with special attention given to multi-objective optimization problems. The authors note that the so-called "best" placement for the VMs in a datacenter may depend on what objective is being pursued, as choosing to optimize either power consumption or reliability of services may lead to different outcomes. Their model, in addition to having the usual constraints in place to make sure individual VMs can only exist in one place at any instance of time, can only change location under specific conditions and must respect the resource capacities of the server they are currently hosted on, also feature constraints relating to the services provided by a VM. VMs of the same service should be spread out for replication purposes, and some VMs of different services may need to be within vicinity of each other due to some dependence. The model has three different objectives to consider: power, migration and reliability cost. Results show that even for single objective problems, the solver struggles to deliver solutions in an acceptable timeframe once larger instances are reached. In order to solve single and multi-objective problems in a timely manner, the problem needs to be relaxed by increasing the optimality gap or limiting the directions in the search space to explore.

Jaumard *et al.* [17] propose a sequence-based model for VM migration and compare its performance to a time-indexed formulation. The advantage of this sequence-based formulation over its time-based peer lies in the overall lower number of variables required in order to express the problem, due mainly to not having to explicitly model the state of the system at each possible time index, where the number of relevant

time indices can be quite large, depending on the problem. Results confirm that the sequence-based model achieves superior performance to the time-index model, finding the optimal makespan for all problems in the test dataset while maintaining a lower computation time in all cases. The model also allows for intermediate migrations to be conducted, but results on the data show that it rarely improves the optimal solution, although the impact of allowing these migrations on total computation time is minimal.

3.1.2 Cross-site Migration

Liu *et al.* [20] present an ILP model for the VM Migration problem, focusing on cross-site VM migration rather than the usual intra-site migration. The authors pay special attention to the possibility that a substandard migration sequence has the chance to cause congestion along the links between sites. The model seeks to maximize the number of planned migrations to be completed while taking into account the available capacity of the inter-network links, as well as considering any inter-VM communication that could further reduce the resources available for migration. The model does not allow for more than one migration per step, however, and the authors acknowledge that the execution time for the Mixed Integer Linear Programming (MILP) problem is too slow for on-line problem resolution (taking 94.92 seconds to solve a problem with 4 sites and 7 migration requests). Consequently, the authors also propose a heuristic based on the MILP that performs almost as well while keeping complexity and computational times down.

Gupta *et. al.* [15] investigate the effects of VM migration in a cross-site situation, where each site is subject to different rates concerning power consumption, varying on an hourly basis. The authors create a model for VM migration that attempts to exploit these spatial and temporal differences in energy prices through plotting migrations over a period of several hours, while also taking into account the costs incurred by performing these migration operations. The model for the total power consumption of the network takes into account the power consumption of the servers, racks and migrations. The power consumption of the servers scales with their VM load, that of the racks scales with the number of non-idle servers, and for migrations, power consumption scales with the total amount of data transferred. Results show that the migration scheme offers savings from 10 to 25 % over a 4-hour exercise. But

the model unfortunately does not scale up to determine performance over a 24-hour period.

3.1.3 Concluding Remarks

Among the models surveyed, we observe that all models have constraints relating to assignment of VMs (a VM may only be assigned to one server at any time) and respecting server or datacenter resource capacity constraints. While the end goal is generally stated to reduce energy consumption, some objective functions simply try to minimize active servers while others attach a power cost to states and actions in the model. Among models that implement the latter approach, the majority agree on the power cost of a server scaling in some way with the number of VMs housed on it. Models generally differ in constraints in order to fulfill specific subgoals, such as considering power limits for individual servers, inter-VM communication or service spread over a network.

The proposed ILP models reviewed are all "compact" , where the model was created to represent the problem with as few variables as necessary. Typically, this is desirable, as the lower number of variables leads to a less complex model that is easier to optimize. Conversely, a decomposed model has more variables than an equivalent compact formulation, but, paradoxically, it's structure may allow us to ignore many of these extra variables and lead to faster solution times than the compact formulation. None of the surveyed references consider decomposition models for use in, e.g., a column generation solution scheme. Decomposition models are interesting in that they may result in more scalable models that their "compact" counterparts. In Chapter 6, we propose two types of decomposition models.

3.1.4 Possible Solution: Decomposition Modelling

We look into decomposition modeling in hopes that, by elaborating such a model for the VM migration problem, we can generate solutions faster than what would be possible with the previously established compact formulations. A decomposed model is an ILP model that takes a form similar to Dantzig-Wolfe decomposition, whether it was originally conceived this way or was created through a transformation of a compact ILP. The decomposed model is split into a Master Problem and one or

several subproblems, also known as pricing problems. The appeal of this formulation over the compact model is that it allows us to cut down the number of considered variables to a smaller, more meaningful subset [10]. Columns are added to the basis in a manner reminiscent of the Simplex method, but these columns are obtained through the optimization of the pricing problems. Once none of the subproblems can be further optimized, the algorithm halts.

3.2 Heuristics

Onoue *et al.* [23] introduce, in addition to the optimisation model stated above, a heuristic algorithm for virtual machine migration based on a graph seeking to model dependencies between the VMs in the system. The modeling of the full dependency graph begins with the creation of small dependency graphs for each VM to be migrated, an edge from that VM to the set of VMs currently residing on its destination is added if it is determined that the VM considered cannot migrate before at least part of the set. All these smaller dependency graphs are then assembled into a larger one representing the problem in whole. The authors use this dependency graph to calculate each VM's *migration weight*, or priority based on number and size of migrations that depend on it. The proposed algorithm greedily selects eligible VMs based on their migration weights in descending order. The dependency graph is updated following every finished migration. Results show that, when compared to a simple greedy heuristic that selects migrations based on time to complete and the Optimization model mentioned in the previous section, the dependency graph-based heuristic solves every problem in the given set, almost always outperforming the simple heuristic, and is able to find a solution for problems on which the optimization model had timed out, taking a few seconds at most. The dependency graph used in this solution does not exactly model the dependency relations between the VMs, only modeling AND-type relations and treating OR-type dependencies and AND-types. This is further discussed in section 3.3.

Jeiroodi [18] proposes several improvements to the algorithm proposed by Onoue *et al.* [23]. One of these improvements involves the selection of servers to act as temporary hosts in the event of an intermediate migration. The proposed algorithm uses a formula based on the available resources of the server in both the initial and

final configuration in order to select servers for intermediate migrations while minimizing the chance that the migration will be delayed or blocked before being engaged. Another proposed improvement is in enabling the adjustment of the timing of intermediate migrations according to network load, in order to allow users to choose to optimize either makespan or service quality by adjusting the maximum threshold for network load under which intermediate migrations may go forward. Results show that the proposed improved algorithm can significantly reduce the number of intermediate migrations conducted in order to solve the migration problems, and thus make the migration more power efficient, while remaining competitive with Onoue *et al.* [23]'s algorithm in terms of makespan.

Gilesh *et al.* [14] propose a heuristic algorithm for the purpose of consolidating VMs in such a way as to admit all incoming VMs into the system, based on a Simulated Annealing (SA) meta-heuristic. As such, the exact final assignments are not given beforehand but determined at runtime, so long as they fulfill a given goal (in this case, allow all entering VMs to be hosted somewhere in the system. The algorithm works by performing perturbations on the current assignments, or moving VMs around if said move is deemed desire able enough, and either accepting the resulting assignments if they improve the objective value, or possibly accepting the assignments based on probability if they do not. The current "temperature" is then updated based on a given *cooling rate* value, and this process repeats until a given temperature threshold is reached. The authors compare the Simulated Annealing-based parallel migration scheme to a sequential migration model and a parallel migration model elaborated by Song *et al.* [27], and find that their SA-based algorithm outperforms the sequential algorithm by a factor of up to 7 and the parallel one by a factor of 3 when considering a high number of migrations.

Qi *et al.* [24] present a model for energy consumption of VMs as well as a heuristic method to apply the principles of the model in a reasonable time. The model breaks down performance of the datacenter into three key areas: Energy consumption (including VMs, physical machines and switches), downtime, or access time while migration is in progress, and resource utilization, then tries to optimize these three areas. The heuristic based on the model is a genetic algorithm based on the Non-dominating Sorting Genetic Algorithm III (NSGA-III) proposed by Deb *et al.* [8] The algorithm sets the "scheduling strategies", or final VM placements, as "genes", and

fitness is evaluated through a weighted sum of the three components of the optimization model's objective function. The algorithm then performs crossover and mutation operations for a given number of iterations, and finally a best solution is selected, according to a weighted value based on the three previously-mentioned criteria. Results show that, compared to a scheduling method based on shortest paths and another Energy-aware Scheduling Method (ESM), the author's heuristic performs better in all areas, significantly so when compared to the benchmark.

Nazir *et al.* [22] propose a heuristic for VM migration heuristic with emphasis on maintaining Quality of Service (QoS) as to fulfill previously-agreed to SLAs . The authors present a scheme for both VM placement and migration, handled through a broker which receives user requests and assigns VMs to Servers accordingly. For placement, VMs are greedily assigned to hosts based on estimated power consumption, provided they do not put the host into a critical state. For migration, VMs in any host considered overloaded are offloaded into preferably underutilized hosts, then the algorithm searches for pairs of underutilized hosts to move VMs from one host to the other in an attempt to shut down as many PMs as possible. VMs are not moved if they are expected to finish their tasks within a certain delay. Performance evaluation is done along five main criteria: power consumption, number of migrations, Millions of Instruction per Second, or MIPS, which determines length of execution for a task, SLA violations, measured in terms of provided vs. requested MIPS, and Task Execution time. When measured against the NPA and DVFS policies, the proposed QoS-MMP algorithm offers less energy consumption, migrations and SLA violations.

Kanniga Devi *et al.* [11] present an algorithm for load balancing with particular attention to network constraints, in addition to a load monitoring algorithm that minimizes the query time through leveraging dominating sets. The authors note that many VM migration algorithms do not focus on network level resources and aim to resolve this by proposing an algorithm for load balancing that models the network of physical machines using a graph. The ST-LVM-LB algorithm handles load balancing by categorizing PMs into *overloaded* and *underloaded* sets, finding neighboring pairs that come from different sets and splitting the load difference by moving VMs from a server in the overloaded set to one in the underloaded set. In the event that there remains some servers in different categories that are not adjacent, paths are calculated between these remaining PMs and migrations executed. The proposed

algorithm, called ST-LMV-LB, outperformed the other two benchmark algorithms (DMA and Sandpiper) [19] [30] in total migration time, migration cost and network overhead.

Rahmani *et al.* [25] have devised an algorithm for VM migration that takes into account sudden fluctuations in workload that may affect the performance of the system. As these sudden bursts of activity can be short-lived, it becomes important to distinguish whether a change of state warrants a migration or not. The authors propose an algorithm to determine both the opportune time to migrate and the manner in which it is conducted. They do this by looking beyond the average load of a Server in the present time, but also looking at a weighted sum of both a current Server's current load and it's average load in the past. This value is used to determine if a Server is over or under-loaded, and these states are then used to determine how migrations are carried out, with the assumption that the algorithm would be less reactionary and thus make moves only when necessary. Results show that the Burst-conscious algorithm, when compared to algorithms that are sensitive to workload explosions, performs less migrations and suffer less Service violations, while maintaining comparable power consumption, showing that it is more efficient.

Basu *et al.* [4] propose an algorithm for VM migration based on reinforcement learning. The authors argue that the myopic nature of greedy heuristics makes them ill-suited to dealing with dynamic workloads, and that a reinforcement learning-based solution would be more adaptable. The solution proposed, contrary to other proposed solutions using reinforcement learning, is both scalable in real time and does not need extensive training. The algorithm proposed, known as MEGH, chooses how to conduct migrations by attempting to minimize a cost function predicting the cumulative cost of the move into the future. Estimations of future cost are rough at the start, but are further revised with each new iteration. Results show that MEGH outperforms competing Reinforcement Learning solutions in terms of cost and number of migrations on different sets of planned workloads.

Tian *et al.* [28] propose a heuristic algorithm that combines a method to avoid unnecessary deadlocks by looking ahead with a nature-inspired metaheuristic called Chicken Swarm Optimization, for the purpose of determining placements that would provide savings in power consumption. In CSO, the solutions, in this case, placements, have their fitness evaluated and are classified as roosters, hens and chicks,

which determines where they can search for new solutions. New generations of solutions are thus generated and evaluated until the algorithm terminates. The authors also attempt to prevent potential deadlocks through the addition of an algorithm that checks whether a placement would result in a deadlock if they tried to move to it from the initial placement, and alter target Servers for VMs that cannot reached their original target. Results show that the Chicken Swarm Algorithm obtains faster convergence on both synthetic and real-world datasets, over similar deadlock-avoidance algorithms.

3.2.1 Concluding Remarks

In the works involving heuristics reviewed above, we note that the problem statement for the VM migration problem varies greatly. One example of this variation lies whether or not the resource demands of the VM are static or dynamic. Some of the works [25] [4] consider the resource requirements of the VMs to vary along with their workload over the course of the exercise while other formulations have resource requirements fixed. Performance evaluation also varies greatly, as the works all have different valuations for execution time, power consumption and SLA violation management. Some works evaluate solutions based on how quickly a desired state is reached, whereas others have a fixed period of time during which the algorithm must make moves to minimize the power consumption, as determined by the authors' power model, over the period of the experiment. These differences in problem statement definition make it difficult to compare the different algorithms amongst each other.

Many of the works use a generated dataset, with the dataset that was used in the experiments not available for consultation, while the algorithms used for the generation are more often provided, though not always. There also seems to not be a set standard for the parameters of the VMs and servers of the dataset, although Amazon EC2 is referenced somewhat frequently when referring to VM parameters. Conversely, many of the works that do provide links to their data have a problem statement on our own, usually concerning the performance evaluation over a set period of time given a trace of workloads, rather than trying to bring the system a desired state in a minimal amount of time, as is our case. In these cases, Cloudsim is often mentioned as the simulation environment, with the traces provided by Planetlab.

3.3 Exact Modelling with a Precedence Graph

At the start of this project, we originally attempted to present a dependency graph-based algorithm similar to the one in Onoue *et al.* [23], with the idea that we would modify the graph to more accurately reflect the dependency relations between the VMs by using an AND-OR graph similar to [1], but we had to abandon this approach due to difficulties with designing and implementing such a graph. We detail below why creating an exact modeling of precedence relations is not a trivial task.

The two main reasons it is difficult to obtain an exact modelling of the dependencies of the VMs using an AND/OR graph is the potentially large size of the generated graph and the potential appearance of unforeseen dependencies in the future, after the initial generation of the dependency graph.

First, We had thought to model VM dependencies using a graph similar to [1], where OR nodes would represent VMs and AND nodes act as "aggregator" nodes used to represent dependency on a set of VMs. The issue with this modelling is that the total number of nodes in such a graph is potentially exponential. While the number of OR-nodes is the same as the number of VMs in the system, the number of AND-nodes depends on the number of potential choices to be made. For example, say a VM4 needs two of VM1,VM2 or VM3 to have moved in order to finish it's migration, as shown in Figure 2. On the dependency graph, this would need to be represented by adding three AND nodes: one for possibility {VM1, VM2}, one for {VM2, VM3} and one for {VM2, VM3}

Second, our problem differs from the one portrayed in [1] in that there are dependencies that may arise in the future that need to be accounted for. For example, see Figure 3. Server 1 containing an outgoing VM1 and has two incoming VMs: VM2 and VM3. There is enough space for one of VM2 and VM3 but not both, so long as VM1 has not moved.

If we were to construct the dependency graph at the beginning of the problem, before any move has been made, we would not indicate that VMs 2 and 3 have any dependencies, as they are both able to be scheduled at this point in time. If we were to choose one to execute, such as VM2, then the dependency graph would then have to be modified, as VM3 now depends on the exit of VM1 to be able to be scheduled. As such, the utility of a given graph modelling would cease to be accurate after each "round" of migrations. In order to remedy this, we could generate a new dependency

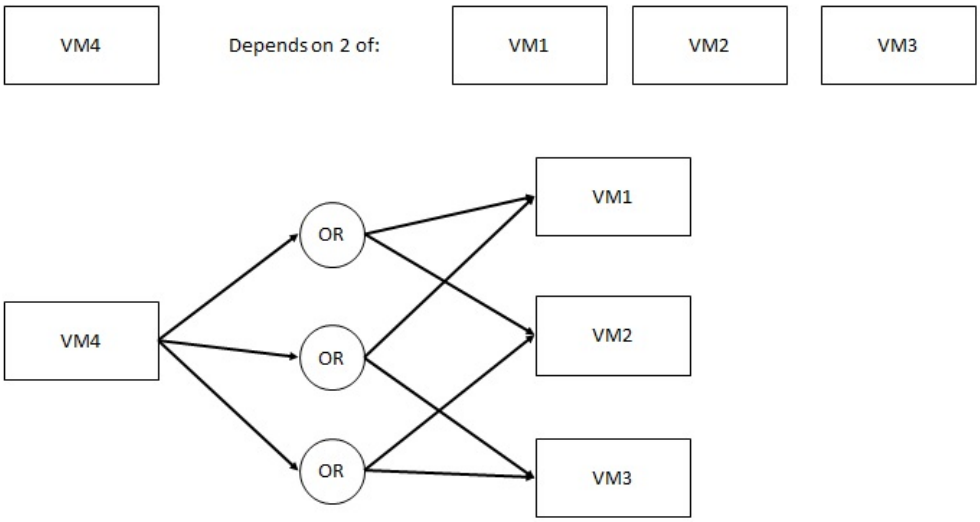

Figure 2: Attempt to Model OR Dependencies Using an AND-OR Graph

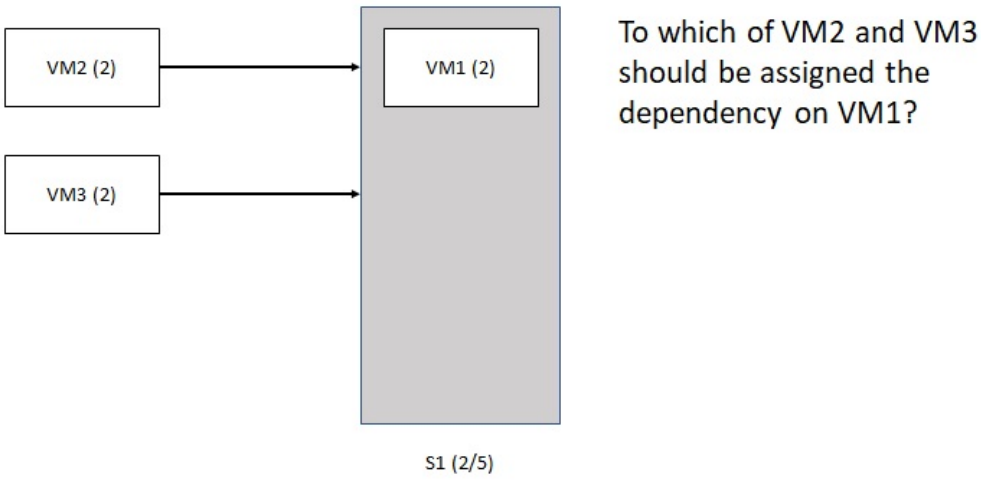

Figure 3: Example of Issue with d=Dependency Assignment

graph for each new move, adding to the algorithm's complexity, or adopt a more conservative approach by indicating a dependency on VM1 for both VMs 2 and 3. This approach is used by Onoue *et al.* [23], and compromises on the accuracy of the represented precedence relations in favor of producing a relatively simple model that is much easier to implement. We could also address the case brought up by Figure 3 by choosing either at random or through some other selection process which VMs have dependencies and which do not. This simplifies graph building, but carries the risk of creating a graph that could lead to a deadlock, where an alternate choice would lead to a deadlock-free graph.

3.4 Deadlocks in Generated Instances

The migration problems used for testing the performance of the heuristics presented in this work, as well as the methods presented in [18] and [17], are synthetic problems created via the generator we elaborated in [5]. The generated dataset is detailed further in Chapter 5. We designed the generator such that, for most of the available scenarios used to generate the problems' final state, this final state is generated with the initial state as a base by moving VMs around one-by-one while ensuring that the capacity constraints of the servers are respected at all times in order to ensure that the final state generated is reachable under the constraints of the problem. As such, "direct" deadlocks will not be present in most of the generated problems, and so intermediate migrations should not be necessary to solve these. One of the scenarios used in the dataset was designed to move the VMs without respecting capacity constraints in between the initial and final state in order to generate problems with "direct" deadlocks, however results from Jaumard *et al.* [17] show that doing so is difficult, short of creating problems in which the servers are so heavily loaded there is no spare space to conduct any migratios.

Chapter 4

Greedy Heuristics

4.1 Motivation

As discussed in the literature, solutions based on meta-heuristics are often chosen to solve the VM migration problem, often preferred over classical, greedy heuristics. Despite this, we have elected to elaborate solutions based around greedy heuristics due to concerns about how computation time scales with problem size, having in mind the need of real-time scalable algorithms. As we are interested in solutions that can be applied *on-line*, it is crucial for us that the solution method scales well as the problem size increases. While meta-heuristics scale better than ILP models, their computation time is still too long for them to be useful in a practical, *on-line* scenario. We have therefore chosen to prioritize simpler solutions as we need them to be as fast as possible.

The two dependency graph-based solutions presented in 4.2 make use of a sub-algorithm to break up any deadlocks encountered during the migration process by organizing intermediate migrations to break resource dependency cycles. In practice, concerning the generated migration problems also used in this work, Jaumard *et al.* [17] shows that, for an ILP solution, allowing the use of intermediate migrations has little effect on the solve time for the vast majority of problems, indicating that deadlocks encountered by heuristics are overwhelmingly of the self-inflicted, "indirect" variety, arising from poor choices in migration ordering. In light of this, we have decided to forgo the use of a deadlock circumvention component for out algorithms, as it should not be necessary to solve the problems in our dataset, provided our

heuristic solutions are sufficiently proficient at avoiding indirect deadlocks.

4.2 Benchmarking

In this section, we present two heuristic algorithms addressing the VM migration problem using a dependency graph to determine precedence relations between the migrating VMs, which are then used to decide migration order. These two algorithms will serve as benchmarking tools to evaluate the performance of our own heuristics, in addition to the ILP results. We find it relevant to compare our results with these in order to see how our results compare to a competing heuristic method, rather than to only consult the ideal results generated by the ILP. Subsection 4.2.1 details the algorithm presented in Onoue *et al.* [23], and subsection 4.2.2 presents a refined version of the Onoue algorithm, correcting some of its faults.

4.2.1 Onoue *et al.* (2017)

Onoue *et al* [23]'s proposed VM Migration algorithm uses a dependency graph-based approach to prioritize which VMs are moved first in accordance with their precedence relation with other VMs in the network. The algorithm necessitates the generation of a migration dependency graph for all of the VMs in the network in order to establish these precedence relations. As shown by figure 4, in order to create the complete graph, they begin by taking each VM from the network, denoted as v and creating an individual dependency graph for it by first checking if v can be scheduled for migration and, if not, adding an arc indicating a dependency between v and any VM on v's destination server that is not part of the final placement for that server, indicating that v cannot move until one or all of these VMs move. Once an individual graph has been generated for all VMs, all graphs are then combined into a single, possibly disjoint graph.

The creation of the dependency graph is needed to determine the *migration weight* of VMs, the value used to determine which VMs migrate first. The *migration weight* of a VM is defines as its migration plus the migration weight of all other VMs that depend on it to move. Within the dependency graph, there may be cases of circular dependencies where some subgraphs form loops resulting in no VM being free of dependencies. The authors address this through the scheduling of a temporary

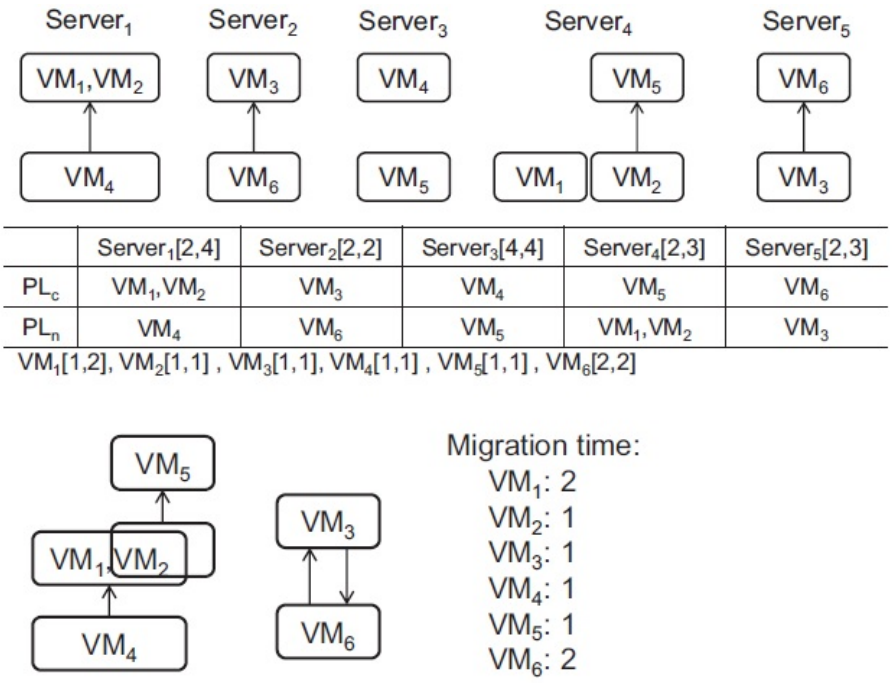

Figure 4: Onoue dependency graph creation, taken from [23]

migration of one of the VMs in order to break the cycle.

The Migration Dependency Graph (MDG)-based algorithm proper works by first generating the dependency graph for the network. VMs to be migrated are then sorted by their migration weight as determined by the graph. VMs with no dependencies that are able to be migrated are thus identified and scheduled. In the event that no such VM can be found, temporary migrations are engaged in order to break any dependency cycles. Once scheduled migrations have finished, the migration graph is updated and the process repeats until all VMs reach their destination.

4.2.2 Khodayar's Algorithm (2019)

Khodayar *et al.* [18] propose an improved version of the algorithm proposed by Onoue, addressing some issues found in the latter. This new algorithm improves on Onoue's by attempting to reduce the number of temporary migrations that would occur during the solution process. It does so through two means: waiting until there are less concurrent migrations before attempting to break deadlocks, and by selecting temporary servers in order to minimize any potential conflicts with current or future migrations.

First, Onoue's algorithm plans temporary migrations almost as soon as a cycle is found, but does not schedule them until they are chosen to be scheduled, in the same way regular migrations are prioritized. This means that a planned, temporary migration could be made invalid due to other migrations moving a VM onto the designated temporary server and thus preventing the migration. Khodayar *et al.* address this by waiting until the number of pending migrations drops under a pre-determined threshold. While the safest way to resolve this issue is to not plan temporary migrations until there are no more pending migrations, this does lead to an increase in total migration time. The use of a threshold thus allows management of the risk of having a temporary migration blocked vs the increase in total migration time.

Second, Onoue *et al.* do not detail how the selection of a temporary server is conducted. As the choice of temporary server can affect likelihood that a temporary migration may be interfered with, Khodayar *et al.* propose a score function to evaluate the servers in the network and determine which one should be used as a temporary server. Servers are scored according to a formula considering their current and final capacities in order to minimize the chance of temporary migrations being interfered with.

4.3 Greedy Heuristics

In this section, we first explain our approach to heuristic designing in subsection 4.3.1. In subsection 4.3.2, we propose a set of decision criteria for greedy heuristics based on our reflections.

4.3.1 General Greedy Framework

The proposed greedy heuristics fall under two different types of approaches.

In the first approach, the idea for the heuristic would be to make move increase the number of options at the next time step, or at least to limit options as little as possible. The logic behind this being that, the more options for moves we have, the less likely we are to run into a situation where no more moves are possible yet the final state was not reached, likely due to a deadlock.

We attempt to estimate how a given migration influences the number of options in the future by looking at the servers it involves (the migration's origin and destination

servers). We choose migrations where the origin server of said migration is heavily requested as a destination by other migrations in the system and where the server designated as the migration's destination is not heavily requested as a destination. By prioritizing migrations as such, we free up space on some servers that would be heavily requested and possibly rendering more pending migrations possible, while having the migrated VM consume resources on a Server that would impede as few other migrations as possible.

There are several different ways we can implement the above approach. The first and most simple one is to look at each migration, then look at the number of incoming migrations on both the origin and destination servers of said migration, and prioritize migrations with the highest difference between incoming migrations on origin and destination. This approach can be further refined by looking ahead by one move to see the difference in possible moves before and after the migration has been made. Of course all migrations are not homogeneous, and some migrations may be more demanding in terms of resources than others. In order to take this into account, we can also make decisions based on the amount of resources requested rather than raw number of incoming migrations.

The second Approach revolves around looking at how many migrations a server is involved in, be it as origin or destination, and attempt to prioritize migrations where one of the servers involved has the least possible number of transactions pending. This means that, ideally, we select migrations that happen to be the last one the source or destination (or both) server is involved in, thus removing it from the set of servers involved in future migrations.

4.3.2 Criteria

We now refine the criteria defined in the previous section. It leads to the following heuristics.

INC-DIFF: Organize migrations by the difference in number of incoming migrations on the origin server ($IN(\mathrm{SRC}_{\mathrm{VM}})$) vs. the number of incoming migrations on the destination server ($IN(\mathrm{DST}_{\mathrm{VM}})$).

$$\max_{\mathrm{VM} \in V} \{IN(\mathrm{SRC}_{\mathrm{VM}}) - IN(\mathrm{DST}_{\mathrm{VM}})\}$$

We sort migrations in descending order according to this value, which should result in prioritizing migrations where there is heavy demand (in terms of the number of migrations) on the origin server and little demand on the destination. Executing these moves first should likely render other migrations, that could not be scheduled previously due to lack of available resources on their target server (in this case, the origin server of migrations just scheduled), available while also seeking to impede as few other planned migrations as possible by prioritizing migrations whose target server is not heavily requested as a destination.

PER-FULL: A more detailed take on the previous heuristic. It takes a given migration, looks at the involved servers, (source and destination) and counts the number of migrations that can be scheduled at this point in time having these two servers as a destination. It then looks at the assumed state of these two servers if the migration currently being evaluated were to be completed (that is, moved from its source to its destination, with resource usage updated to match) and counts the number of possible migrations on the two servers of interest. We then take the difference between the number possible after and before the migration and sort migrations to prioritize the highest of this value.

$$\max_{\text{VM}\in V} \{A_{\text{VM}} - B_{\text{VM}}\}$$

$$A_{\text{VM}} = A_{\text{VM}}^{\text{SRC}_{\text{VM}}} + A_{\text{VM}}^{\text{DST}_{\text{VM}}} \quad ; \quad B_{\text{VM}} = B_{\text{VM}}^{\text{SRC}_{\text{VM}}} + B_{\text{VM}}^{\text{DST}_{\text{VM}}}$$

where A_{VM} represents the number of migrations possible after executing the migration in question, and B_{VM} represents the number of migrations possible before then. The difference of the two shows the increase or decrease in potential moves if we were to schedule this migration.

RES-DM : Functions similarly to the above-detailed heuristics, but instead of looking at the number of incoming migrations, it takes into account the number of resources requested.It can be written as follows:

$$\min_{\text{VM}\in V} \left\{R^{\text{SRC}_{\text{VM}}} - R^{\text{DST}_{\text{VM}}}\right\}$$

$$\text{R}^{\text{SV}} = \min_{\square\in\{\text{CPU,RAM}\}} \frac{RA_{SV}^{\square} - RI_{SV}^{\square}}{C_{\text{SV}}^{\square}},$$

where:
$RA^{\square}_{\text{SV}}$ = amount of resource $\square$ available in SV
$RI^{\square}_{\text{SV}}$ = amount of resource $\square$ requested by incoming VMs on SV
$C^{\square}_{\text{SV}}$ = maximum capacity for server SV for resource $\square$.

We choose the minimum value in order to determine which of the different resources is the bottleneck. Difference between available and requested resources is normalized over the server's total capacity in order to account for different range of values between the different resources.

We propose this alternative to the above heuristics relying on the number of migrations in order to account for the heterogeneous resource demands of VMs in the network. These demands result in the amount of space free on one server and consumed on another as a result of migration varying depending of the VM being moved, and thus in some cases it might be advantageous to engage a migration that would allow the future scheduling for migration of one large VM, rather than one that would allow multiple smaller VMs to move as a result.

For example, assume we have the choice between conducting a migration that would, upon completion, allow 3 more migrations that require 2 resources apiece vs. another migration that would allow upon completion 1 other migration requiring 9 resources. It may be advantageous to choose the latter, as the one migration would free 9 units all on it's origin server, leading to the high likelihood that an incoming migration on that server that was previously blocked would now be possible, whereas allowing the first migration would permit the three smaller migrations, which would free a total of 6 units that may be spread among up to 3 servers. We suppose that freeing a large amount of space on one server may lead to unblocking more migrations than freeing a small amount of space on multiple servers.

REM-MV: For a given migration, we look at its source and destination server and check the number of other migrations they are involved in, as either source or destination, and assign to the migration the minimum value of the two. We then sort migrations by this value and select them in ascending order, updating the values at each time step. Through this method, we seek to reduce as quickly as possible the number of servers that are involved in the migration process, either as source or destination node. Criterion can be mathematically written as follows:

$$\min_{\text{VM}\in V} \left\{ \min \left\{ RM^{\text{SRC}_{\text{VM}}}, RM^{\text{DST}_{\text{VM}}} \right\} \right\} \qquad \text{where} \qquad RM^{\text{SV}} = IN(\text{SV}) + OUT(\text{SV}).$$

4.4 Impact of Preprocessing

In order to reduce the search space for the greedy heuristics, we add a preprocessing procedure to run ahead of the heuristic at each decision step. This preprocessing algorithm functions by scheduling migrations which are determined to be "trivial", meaning that scheduling them would have no impact on any other migration in the system, and whose scheduling would not impact any other migration. As such, they can be scheduled for migration immediately without fear of causing an indirect deadlock.

We identify these "trivial" migrations by finding, for each server, the list of migrating VMs that have said server as their intended destination. We then verify if the server has the spare resources to accommodate all incoming migrations and, if so, we schedule all incoming migrations. Doing so would eliminate these migrations from consideration by the heuristic without changing the final migration order, as it is given that these migrations would be scheduled anyway by the heuristic.

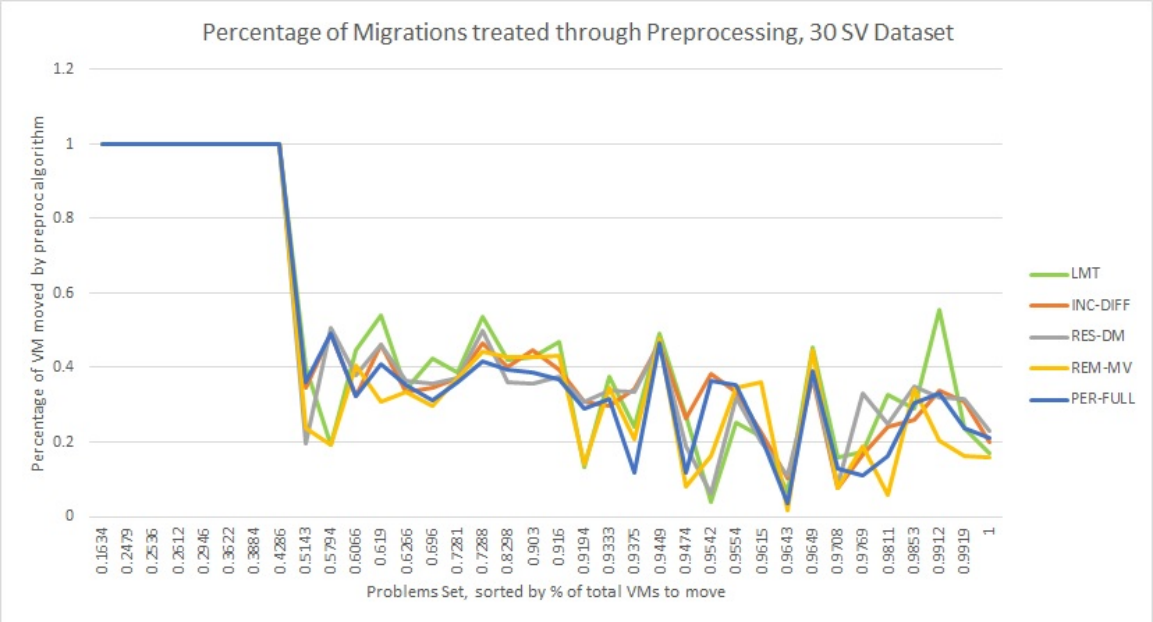

Figure 5: Percentage of total migrations treated through pre-processing

Figure 5 shows, for the 30 Server dataset, the percentage of migrations that were scheduled through preprocessing for each problem of one of the datasets. Results show that the number of migrations treated through preprocessing scales with the apparent difficulty of the problem, as estimated through the percentage of total migrations

that need to be moved. Trivial problems can be resolved completely through the preprocessing method, and as the difficulty of the problem increases, the number of migrations treated through preprocessing decreases. The majority of problems fall within a 20 to 40 % range for number of migrations treated through preprocessing. The amount of migrations treated this way for a given problem also varies slightly based on the heuristic used, as different heuristics lead to different problem states which can have more or fewer "trivial" migrations.

4.4.1 General Algorithm

Algorithm 1 General Heuristic Algorithm

```
for all M do
    Add M to origin(M)'s list of outgoing migrations
    Add M to destination(M)'s list of incoming migrations
end for
while There are scheduled migrations left to complete do
    while There are planned migrations that can be scheduled do
        sort planned migrations by chosen criteria
        schedule first available migration
        update lists on concerned servers
    end while
    find scheduled migration with earliest end time
    advance time and mark all migrations completing at this time as complete
    update lists on concerned servers
end while
```

Algorithm 4.4.1 shows the general algorithm used for all the criteria previously listed, with the sorting of planned migrations changing depending on the criteria used. At the beginning, we first loop through all the migrations in order to set up, for all servers, lists for migrations incoming and outgoing on that server. These lists are used to quickly refer to values (incoming migrations, resources) used in the calculation of priority regarding scheduling, through the selected criteria.

Once this is done, the main loop consists of scheduling as many migrations as possible at a given time step, and then advancing time by finishing a scheduled migration. The algorithm terminates when there are no more scheduled migrations to complete.

Chapter 5

Numerical Results

In this chapter, we present the results and analysis of the performance of our heuristics. Section 5.1 describes how the generator used to create our problem datasets functions, and gives details on the generated dataset used in our experimentation.Section 5.2 describes our method for evaluating the performance of our solutions. Section 5.3 contains the results and analysis of the performance of our heuristics, compared both amongst themselves and to the benchmarking algorithms listed at the start of the section.

5.1 Data Sets

The dataset for this work was made using an in-house migration problem generator [5]. The generator outputs an initial and final state for a network based on input values given by the user. The user can determine problem parameters such as the total number of servers and VMs, servers' possible resource capacities and VMs' resource demands, and the manner in which the final state is decided.

The number of servers in the network is given as an integer, while the total number of VMs in the network can be determined either with a given number or through assigning "classes" to the servers. A server's "class" determines how many VMs it can hold by filling it with newly-generated VMs until it reaches the class' accepted range for server load. For example, a newly-generated, empty server is assigned a class "40-60" will have newly-generated VMs until its load rises to between 40 and 60%.

The CPU and memory resource capacities for servers and requirements for VMs are given by the user. The user supplies a list of possible values for server CPU cap, server memory cap, VM CPU requirement, VM memory requirement. Newly-generated Servers and VMs have their caps and requirements determined by randomly pulling from these user-given pools of values.

For this particular work, we have supplied resource requirement values for the VMs based on the Amazon EC2 instances [2], and the supplied values for the server capacities are based on the parameters of the Dell servers found at [9]. Table 1 displays the possible values for resource requirements or capacity, whichever applies. These values are displayed as (CPU, RAM).

Servers	VMs
(28,128)	(2,4)
(56,256)	(2,8)
(78,512)	(4,8)
(112,768)	(4,16)
-	(4,122)
-	(8,16)
-	(8,32)
-	(8,244)
-	(16,64)
-	(16,122)
-	(16,488)
-	(36,72)
-	(48,192)
-	(64,256)

Table 1: Possible Resource Values for Servers and VMs

The final placement in a generated problem is determined by migrating VMs from the initial placement in accordance with a user-determined "scenario". Such scenarios usually aim to produce a final placement that would reflect the purpose of migration in a practical situation.

Scenarios include:

- **FAIL:** Fail scenario. In this scenario, a number of servers has been designated to shut down and all VMs must be migrated off to other servers. The percentage of failing servers is a user-determined parameter.

- **CONS:** Consolidation scenario. In this scenario, VMs are moved around until

all servers are either empty, or their load falls within a pre-determined interval. The interval of the accepted range for the load of the servers is a user-determined parameter. (ex: CONS-90-100 means that all servers where both CPU and RAM load does not fall within the 90 to 100% range must be emptied or have VMs added or removed until their load % complies.

- **LBAL:** Load balancing scenario. Attempt to minimize the variance in Load for all Servers. No user parameters.

- **DSHF:** Deadlock shuffle scenario. Random redistribution of VMs with no underlying justification. An attempt at generating difficult to solve problems, even if they do not really reflect a real-world scenario.

All scenarios except DSHF move VMs one-by-one while taking resource constraints into account in order to ensure the final placement has a solution. DSHF removes all VMs, places them in a global container, and redistributes them randomly to Servers, which may then result in the problem having no solution.

Additionally, the user may also determine the percentage of VM that will be shuffled around randomly prior to the start of the scenario proper. This aims to make the problem generation less deterministic.

For this work, we have generated several sets of migration problems, each set, denoted by the number of servers in the problem (30,50,100,200,300 and 500 servers). For each set, we have 36 generated problems covering every combination of the following parameters: All available problem scenarios (FAIL,CONS,LBAL,DSHF) with the amount of failing servers in FAIL set to either 1/3 or 2/3 of the total servers, and the acceptable load range for CONS is set to either 90-100 % or 100-100%. The percentage of VMs shuffled prior to that start of the algorithm is set to 0, 50 or 100% and the load for the Servers in the initial state is set to 60-80 % or 80-100 %. As stated previously, the number of VMs in a given problem is determined dynamically when generating using class loads, although it still scales with the number of servers. Table 2 provides a range for the number of VMs in the problems for each of the generated datasets.

Dataset (by number of servers)	range for number of VMs
30 SV	100-150
50 SV	150-250
100 SV	350-500
200 SV	750-950
300 SV	1200-1400
500 SV	2000-2200

Table 2: Range for Number of VMs in Each Dataset

5.2 Performance Criteria & Testing Environment

The performance of a given heuristic is determined by two main criteria : the total migration time, or makespan, and the percentage of planned migrations that were successfully completed. We are looking to minimize both of these values, although we assign higher value to solutions that complete all required migrations over those that do not, regardless of achieved makespan. This means that a solution with a relatively high makespan but that completes all assigned migrations will be considered more valuable that one which reports a low makespan but has some scheduled migrations left incomplete. Additionally, as we are concerned with application in an on-line scenario, the claculation time for the algorithms is also considered as part of performance evaluation.

The hardware used for the testing of the heuristics and ILP solution consists of a cluster composed of 40 Intel Xeon CPU E5-2660 v3 @ 2.60 GHz with 2 threads per processor, and 125 GBs of system memory.

5.3 Comparison of Heuristics

We evaluate the performance of the heuristics detailed in Chapter 4 by using the following other methods as baselines:

LMT: Longest Migration Time. This greedy heuristic orders migrations in descending order of the time needed to complete, and serves as a baseline for comparing the other criteria.

ONOUE: We outline it in Chapter 4, and details can be foudn in [23]. It contains a deadlock resolution component using temporary migrations.

KHODAYAR: Improved version of the above ONOUE, as detailed in [18]. Improvements include optimized selection of temporary servers for deadlock resolution.

ILP: Linear Programming model for the Virtual Machine Migration problem, as detailed in [17]. Maximum time allowed to search for a solution is limited to 2000 seconds or 33 and 1/3 minutes.

5.3.1 Intra-Greedy Heuristic Comparison

In this subsection, we compare the performance of the 4 greedy heuristics detailed in the previous chapter in order to determine which of the four would be most promising going forward. We add the results of the ILP to provide an ideal to attempt to reach, and the results of LMT to show a minimum standard for our heuristics to surpass. We've ran the heuristics over all datasets and provide detailed results for the 30 and 200 server datasets in figures 6 and 7. Each point on the horizontal axis indicates a problem, and the numbers show the percentage of total VMs that are intended to move. We treat this value as an indicator of problem difficulty. The lines represent the makespan of the solution and the bars show, if any, the percentage of planned migrations left incomplete by the solutions.

Heuristic Name	Completion %
LMT	0.6231 %
INC-DIFF	0.6811 %
RES-DM	0.7198 %
REM-MV	0.5266 %
PER-FULL	0.6715 %

Table 3: Migration Average Completion Rate for Each Heuristic

Table 3 displays the percentage of solvable problems in the dataset where the heuristics were able to migrate all planned migrations. Here, "solvable problems" is defined as all problems where the ILP is able to find a solution, across all datasets (30,50,100,200,300,500 servers), which amounts to a total of 207 problems. The table shows that 3 of the 4 proposed heuristics show improvement over LMT, with RES-DM reaching a near 10 % improvement, but we are still a long way off from finding a heuristic that completely eliminates the possibility of indirect deadlocks. REM-MV is shown to perform worse than LMT in this metric.

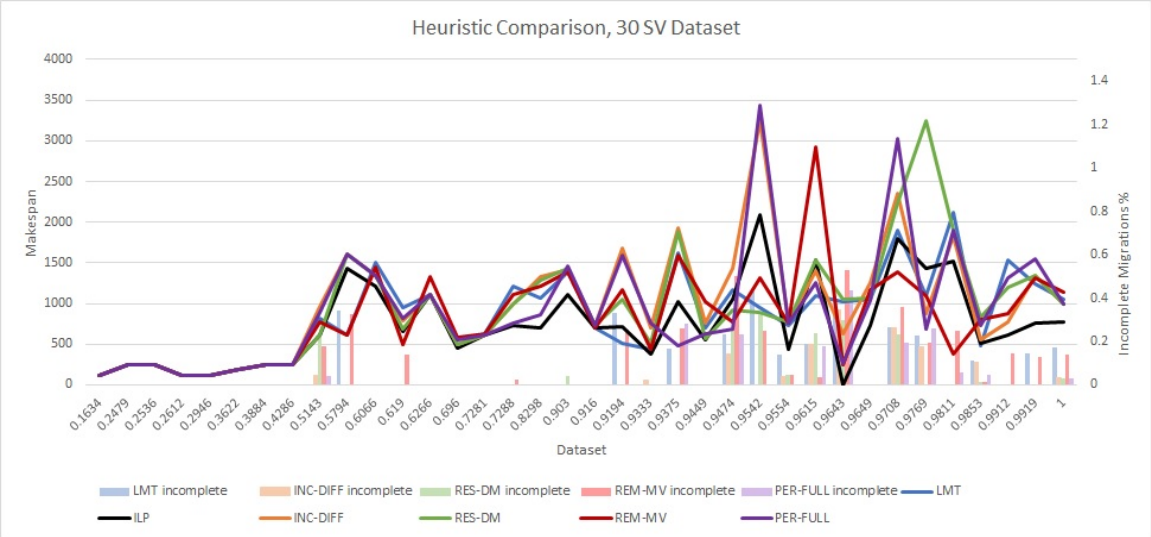

Figure 6: Comparison of Heuristics on 30 Server Dataset

On both Figures 6 and 7, we can observe that the two most efficient heuristics are RES-DM and PER-FULL, as they have the highest rate of problem completion while remaining competitive in terms of makespan. However, no heuristic appears to have a clear advantage and performance varies on a case-to-case basis. By contrast, REM-MV shows the poorest results, being outperformed by the basic LMT.

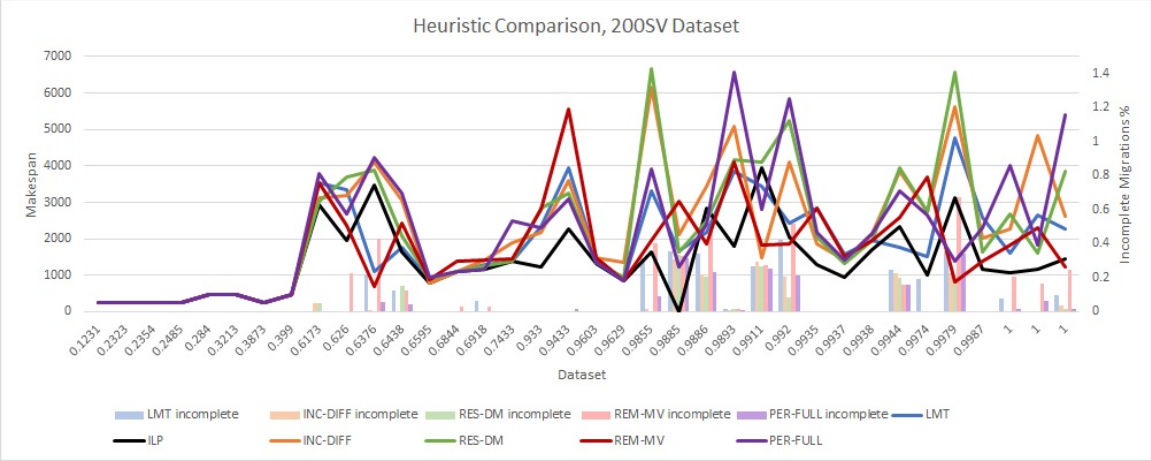

Figure 7: Heuristic Comparison for 200 Server Dataset

5.3.2 Best Greedy vs. Dependency-graph Heuristics

In this subsection, we select one of the heuristics from subsection 5.3.1 we consider to have the best performance, and test it against the precedence-based solutions outlines in Chapter 4. Results in subsection 5.3.1 lead us to select RES-DM for this exercise. We compare its results to that of ONOUE and KHODAYAR on all the datasets and present detailed results for the 30 and 200 server sets in Figures 8 and 9.

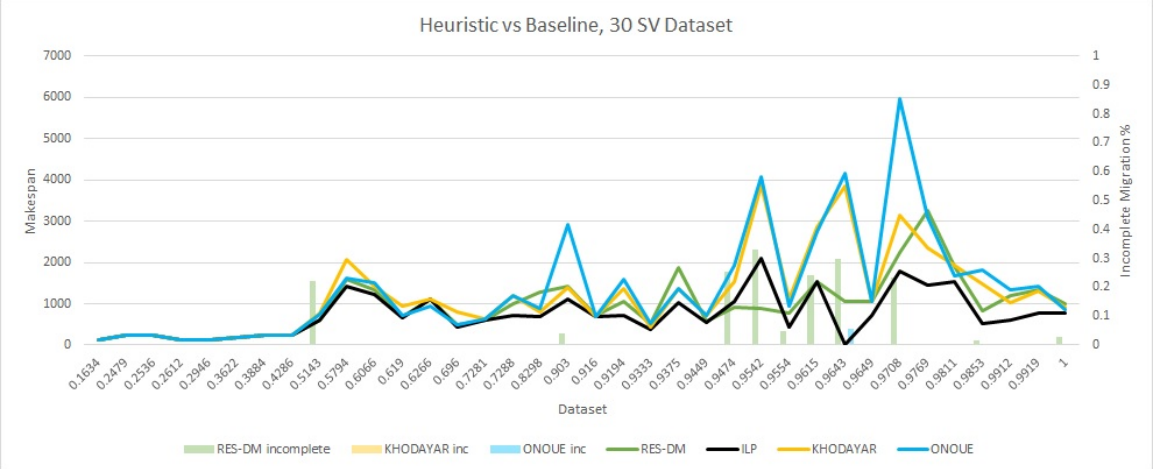

Figure 8: Comparing Best Heuristic with Baseline for 30 Server Dataset

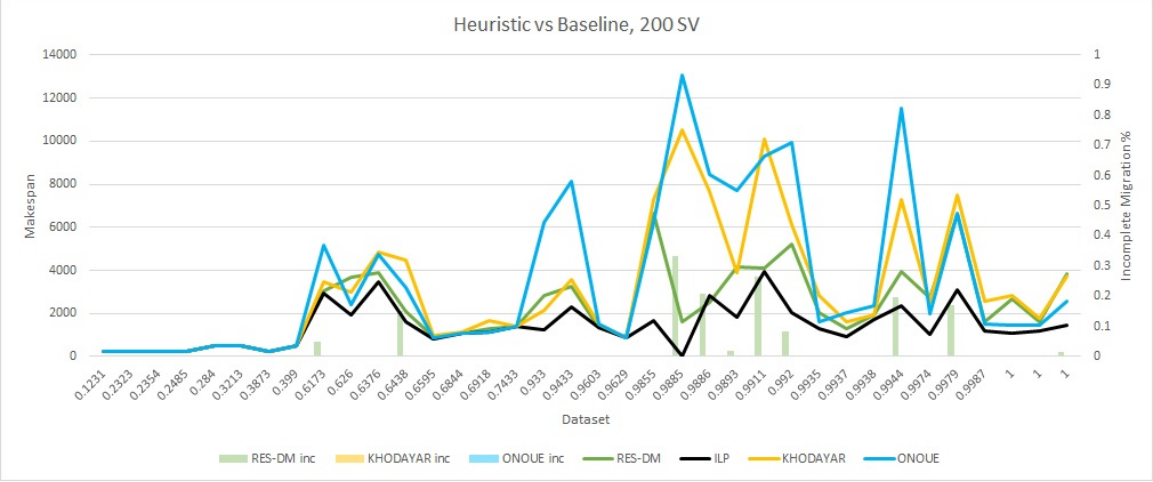

Figure 9: Comparison of Best Heuristic with Baseline for 200 Server Dataset

In figures 8 and 9, we observe that, on problems where our heuristic is able to complete all planned migrations, the makespan of our greedy solution shows no clear

pattern performance-wise, sometimes being outclassed by both ONOUE and KHODAYAR, while at other time beating one or both. The greedy heuristic is unfortunately unable of completely solving all problems that are solvable without the use of temporary migrations, as indicated by ILP results. This means we were unable to find a heuristic solution that can completely avoid indirect deadlocks and thus is reliant on some sort of deadlock resolution sub-algorithm, as are the aforementioned precedence-based algorithms.

5.3.3 Computational Times

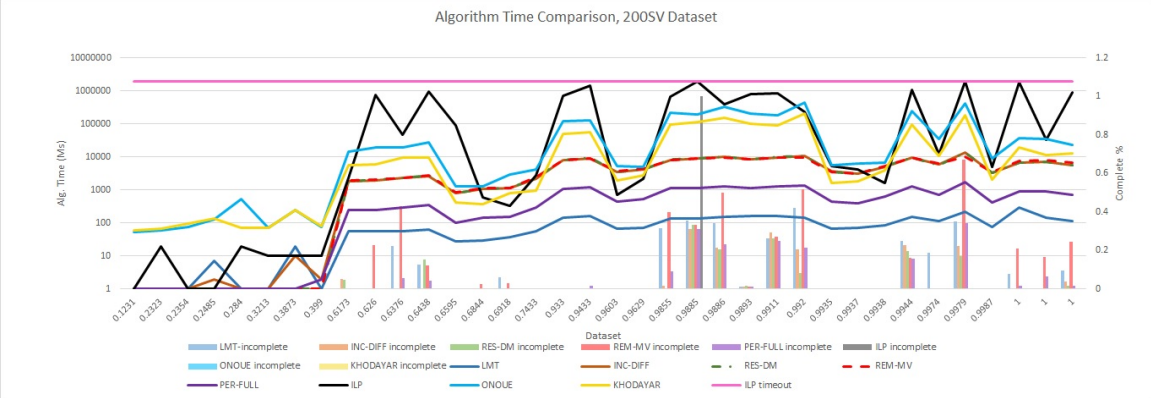

Figure 10: Computational Times Comparison for Heuristics on 200 Server Dataset

Figure 10 shows the algorithm computational time in milliseconds for the 200 server dataset for the proposed heuristics, as well as LMT and the ILP for comparison. We observe that three of the heuristics have computational time significantly higher than LMT, to the point where they are outrun by even the ILP on "easier" problems, if we were to take the number of incomplete migrations by the heuristics on a given problem as an indicator of problem difficulty. PER-FULL seems more efficient, outpacing the ILP on every problem in terms of time, however it is questionable whether the marginally improved performance of LMT, as discussed in 5.3.1, is worth the notable increase in computational time.

Figure 11 shows the average algorithm time by datacenter size for the heuristics and the ILP, counting solved problems, incomplete solutions and timeouts. Results confirm that PER-FULL's faster execution speed over the other proposed heuristics holds for all sizes of datacenter tested. We also note that the added layer of complexity

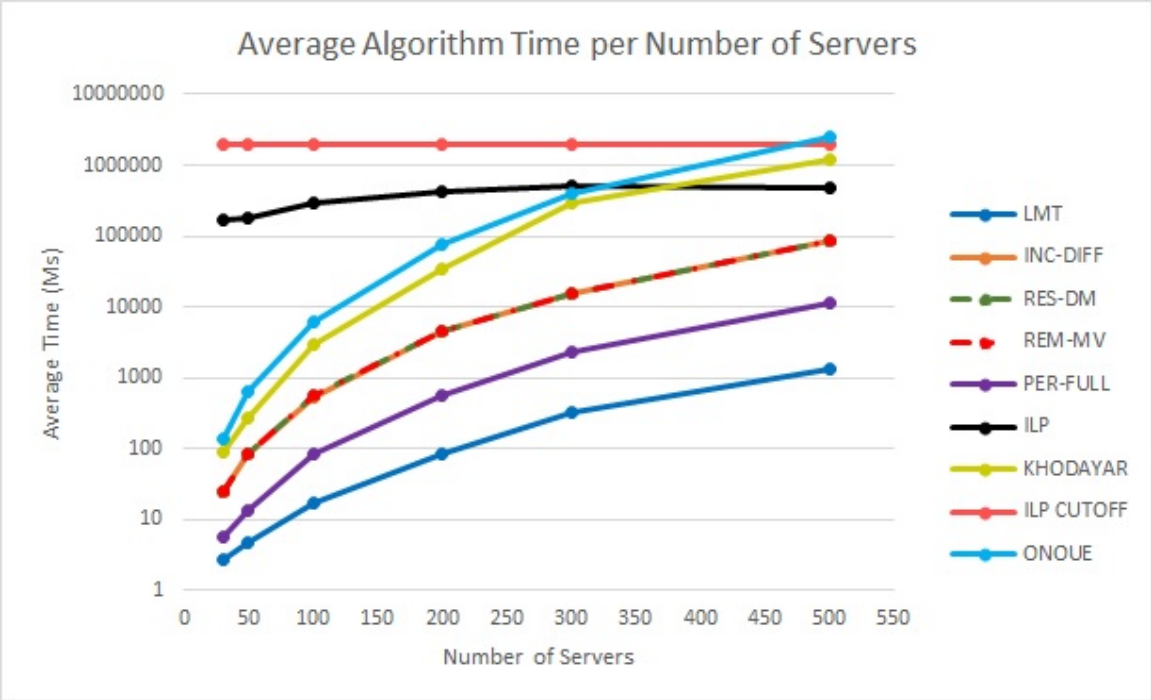

Figure 11: Average Algorithm Time over All Datasets

in heuristic design over LMT leads to a noticeable penalty in computational time and scaling with problem size, casting doubts on whether the meager performance increase previously mentioned is worth this trade-off. We note here that the ILP curve seems to level off, which is due to the time limit of 200 seconds put in place to solve a given problem. The migration dependency graph-based solutions, ONOUE and KHODAYAR, have no such limit set in place which explains why they register a greater average computational time than the ILP for the larger datasets.

5.3.4 Conclusions

To summarize, while our greedy heuristics are able to somewhat compete with the precedence-based heuristics on certain cases, they still fail entirely to solve some instances which should be theoretically solvable (i.e., could migrate all VMs), as indicated by the ILP results. In many cases, our heuristics run into deadlocks and terminates without solving while the ILP finds a migration plan, implying the migration of all VMs necessary for solving the problem is possible without needing to use temporary migrations.

We were unable to propose a greedy heuristic that significantly improves avoidance of indirect deadlocks compared to the longest migration time-based solution used as one of our benchmarks, and the results on problems where it does find a complete solution do not show the achieved makespan to be consistently superior to the chosen benchmark algorithms. In addition, when taking computational time into account, the slight improvements with respect to both makespan and number of completed migrations that our heuristics may achieve over a simpler solution such as LMT are difficult to justify when contrasted with the noticeable increase in computational time that our heuristics require compared to these simpler heuristics.

We think that it would have been interesting to design selection criteria in the greedy heuristic that would give more consideration to the characteristics of the given problem, such as the scenario type or the size and load of the datacenter. As explained in Section 5.1, data sets were generated using different parameters, and it would have been of interest to analyze the results in light of their characteristics, such that we could potentialy find which problems are more likely to produce deadlocks based on these, so that we may reserve deadlock avoidance strategies such as an n-step look-ahead to deal only with the most difficult problems.

Chapter 6

ILP Models

In this chapter, we discuss large scale ILP formulations and propose two decomposition models of our own for the VM migration problem. Section 6.1 reviews the classical, "compact" formulations and why we find them limited. In section 6.2, we propose two decomposition models: the first model is a time-indexed formulation that we anticipate will not lead to a significant improvement of the scalability of the model compared to existing "compact" models, while the second model we propose is an event-based decomposition which would seem more promising regarding scalability.

6.1 Classical ILPs and their limitations

As discussed in Chapter 2, different compact ILP models (i.e., usually with a polynomial number of variables and constraints) have been proposed, but they lack scalability, i.e., cannot solve large or difficult data instances with more than a few hundred VMs or tens of servers in a reasonable amount of time, where "reasonable" implies a viable time frame for on-line problem resolution. ILP models may not be a viable solution for solving problems as they appear, but the solutions they produce are still valuable within the context of heuristic solution evaluation. While the time constraints for finding a solution in such a context are not nearly as strict as for on-line problem resolution, it remains relevant to find ways to reduce the time needed to produce a solution, especially as the problem size increases. We thus propose two decomposition models formulated to be solved with column generation techniques in mind.

Column generation formulations offer some advantages over traditional, compact problem formulations. According to [3], "compact" formulations may have a "weak" LP relaxation, meaning that the distance between the relaxation's and the integer's optimal value is significant. Reformulating the compact problem into a column generation one can strengthen the relaxation. Second, the compact formulation may be structured in a way that negatively impacts branch-and-bound methods, i.e., the derivation of an integer solution. Branch-and-bound performs poorly on problems whose structure is too "symmetric" (i.e., such that a permutation of the variables leads to the same solution) because branching makes little change in the problem. This can be addressed by reformulation of the problem. Surprisingly, even if it usually means greatly increasing the number of variables, it leads to more efficient solution schemes, as the latter one only implicitly enumerate all the variables.

6.2 Proposed Decomposition Models

We propose two decomposition models for the VM migration problem: a discretized, time-based model and an event-based model, based on the models introduced in [17].

6.2.1 List of Common Parameters and Variables of the two Models

The following list details parameters and variables that are common to both models. Variables specific to a given model are detailed in their respective sections.

Parameters

$\text{VM} \in V$: A Virtual Machine in the network

$\text{SV} \in S$: A server in the network

$C_{\text{SV}}^{\text{CPU}}$: Maximum number of processors for server SV

$C_{\text{SV}}^{\text{RAM}}$: Maximum amount of memory for server SV

$\text{REQ}_{\text{VM}}^{\text{CPU}}$: Number of processors required by VM to be housed on a server SV

$\text{REQ}_{\text{VM}}^{\text{RAM}}$: Amount of memory required by VM to be housed on server SV

$\text{MIGR}_{\text{VM}}^{\text{SV},\text{SV}'}$: Amount of time needed to migrate VM from SV to SV′, where $\text{SV}, \text{SV}' \in S$

$\square$: Set of Resources, $\square = \{\text{CPU}, \text{RAM}\}$.

Variables

C_{max} : Total Makespan

6.2.2 First Decomposition Model: A time-index model

This first decomposition model relies on the concept of *path configuration*, which we define in this section. We then describe the specific set of parameters and variables and then the master and pricing problems.

Path Configuration

The key set of variables of the first decomposition model relies on the concept of path configuration. Before we define it, we first need to introduce a spatio-temporal graph to express the options each VM has for scheduling the time of its migration. Such a graph details the possible options a VM has for moving from it's source Server to it's intended destination. The graph $G^{\text{VM}} = (V^{\text{VM}}, L^{\text{VM}})$ is defined by *(i)* its set of nodes V, where each node of the graph has two coordinates (x, y): the x coordinate is an instance of time (t index) and the y coordinate is a server (in this case either the VMs source SRC or destination DST), and *(ii)* its set of links, $\ell \in L$ where ℓ represents the possible actions a VM can take at some point in time t, be it to either remain on it's current Server until the next time step or move from a SRC Server to DST.

A path configuration $p \in P_{\text{VM}}$ defines a possible decision concerning the time of the migration. Indeed, in G, p represents a possible path for VM. Figure 12 shows a simplified example of such a graph, with $p1$ and $p2$ representing two possible path configurations, or migration scheduling decisions for the VM this graph concerns. The horizontal axis represents time while the vertical axis represents the possible locations the VM can be in, in this case the source and destination nodes. A VM choosing to move at $t = 1$ would be selecting $p1$, whereas a VM moving at $t = 4$ would select $p2$. Such a graph will typically have much more than two paths.

A path configuration is characterized by the following parameters:

$a_{\text{VM},\text{SV}}^{t,p} = 1$ if VM is using SV at time t while path p is chosen, 0 otherwise.

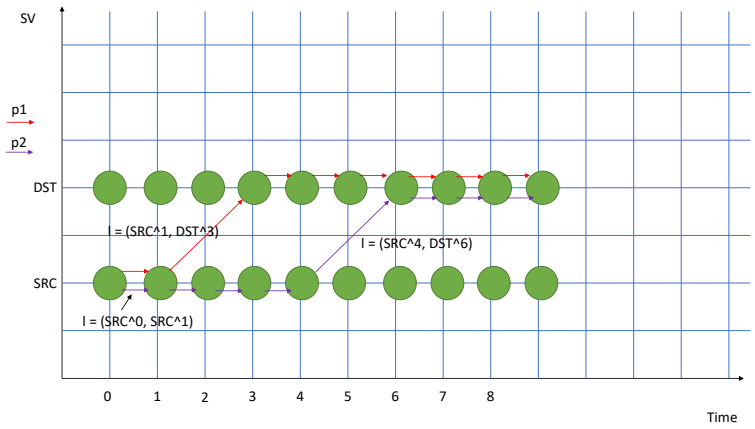

Figure 12: Spatio-Temporal Graph for VM

$\text{SV}^t = \{\text{SRC}^t_{\text{VM}}, \text{DST}^t_{\text{VM}}\}$: Set of nodes in the graph, which represent the VM in question's source or destination server at some point in time t.

Denote by $\omega^{\text{IN}}_{\text{VM}}(\text{SV}^t)$ the set of incoming arcs for node SV^t, and by $\omega^{\text{OUT}}_{\text{VM}}(\text{SV}^t)$ the set of outgoing arcs for node SV^t. They are further detailed as:

$$\omega^{\text{IN}}_{\text{VM}}(\text{SRC}^t_{\text{VM}}) = \{(\text{SRC}^{t-1}_{\text{VM}}, \text{SRC}^t_{\text{VM}})\} \tag{1}$$

$$\omega^{\text{OUT}}_{\text{VM}}(\text{SRC}^t_{\text{VM}}) = \{(\text{SRC}^t_{\text{VM}}, \text{SRC}^{t+1}_{\text{VM}}), (\text{SRC}^t_{\text{VM}}, \text{DST}^{t+\text{MIGR}^{\text{SRC,DST}}_{\text{VM}}}_{\text{VM}})\} \tag{2}$$

$$\omega^{\text{IN}}_{\text{VM}}(\text{DST}^t(\text{VM})) = \left\{(\text{DST}^{t-1}_{\text{VM}}, \text{DST}^t_{\text{VM}}), (\text{SRC}^{t-\text{MIGR}^{\text{SRC,DST}}_{\text{VM}}}_{\text{VM}}, \text{DST}^t_{\text{VM}})\right\} \tag{3}$$

$$\omega^{\text{OUT}}_{\text{VM}}(\text{DST}^t_{\text{VM}}) = \left\{(\text{SRC}^t_{\text{VM}}, \text{SRC}^{t+1}_{\text{VM}})\right\}. \tag{4}$$

Parameters of the Master Problem

the other parameters used by the decomposition model are as follows:

$p \in P_{\text{VM}}$: Path for VM

$a^{t,p}_{\text{VM,SV}}$: $= 1$ if VM is using SV at time t while path p is chosen, 0 otherwise.

Variables of the Master Problem

x_{VM}	$= 1$	if VM has been moved, 0 otherwise
z_p	$= 1$	if path p selected for VM, 0 otherwise
T_{VM}	$=$	end time for migration of VM
$\text{C}_{\max}$	≥ 0	defines the makespan.

Master Problem

Minimize:

$$\mathrm{C}_{\max} - \textsc{penal} \sum_{\textsc{vm} \in V} x_{\textsc{vm}} \tag{5}$$

subject to:

$$\sum_{p \in P_{\textsc{vm}}} z_p = x_{\textsc{vm}} \qquad \textsc{vm} \in V \tag{6}$$

$$\sum_{p \in P_{\textsc{vm}}} \textsc{end}_p z_p = T_{\textsc{vm}} \qquad \textsc{vm} \in V \tag{7}$$

$$T_{\textsc{vm}} \leq \mathrm{C}_{\max} \qquad \textsc{vm} \in V \tag{8}$$

$$\sum_{\textsc{vm} \in V} \sum_{p \in P_{\textsc{vm}}} \textsc{req}^{\square}_{\textsc{vm}} a^{t,p}_{\textsc{vm},\textsc{sv}} z_p \leq C^{\square}_{\textsc{sv}} \qquad t \in T, \textsc{sv} \in S, \square \in \{\textsc{cpu}, \textsc{ram}\}. \tag{9}$$

$$x_{\textsc{vm}} \in \{0,1\} \qquad \textsc{vm} \in V \tag{10}$$

$$z_p \in \{0,1\} \qquad p \in P^{\textsc{vm}}, \textsc{vm} \in V \tag{11}$$

$$T_{\textsc{vm}} \geq 0 \qquad \textsc{vm} \in V, \tag{12}$$

$$\mathrm{C}_{\max} \geq 0. \tag{13}$$

The objective function (5) minimizes the makespan of the solution while also penalizing planned migrations that were not completed. Constraints (6) indicate that if a VM migrates, then it must make use of a path. Constraints (7) and (8) indicate that the makespan is bounded by the end time of the latest migration to finish. Constraints (9) define the resource capacity constraints for each server.

Variables of the pricing problem

φ_ℓ	$= 1$	if path uses link ℓ, 0 otherwise.
$a^t_{\textsc{vm},\textsc{sv}}$	$= 1$	if VM is using SV at time t while path p is chosen, 0 otherwise.

Pricing Problem

We have one pricing problem for each VM, i.e., we need to consider two particular servers, the source and the destination server of VM. As we do not consider any intermediate server in the migration, we have that: Output of the pricing problem ≡ input of master (coefficients of the z_p variables)

$a^{t,p}_{\textsc{vm},\textsc{sv}} = 0$ if $\textsc{sv} \notin \{\textsc{dst}_{\textsc{vm}}, \textsc{src}_{\textsc{vm}}\}$.

(pricing variable) $\quad \varphi^t_\ell \quad \rightsquigarrow \quad a^{t,p}_{\textsc{vm},\textsc{sv}}$ (coefficient of z_p)

Minimize:

$$0 - u_{\text{VM}}^{(6)} - \sum_{\square \in \{\text{CPU,RAM}\}} \sum_{t \in T} \sum_{\text{SV} \in S} u_{\text{SV},t,\square}^{(9)} \text{REQ}_{\text{VM}}^{\square} a_{\text{VM,SV}}^{t}. \tag{14}$$

subject to:

$$\sum_{\ell \in \omega_{\text{VM}}^{\text{OUT}}(\text{SRC}_{\text{VM}}^{0})} \varphi_\ell = \sum_{\ell \in \omega_{\text{IN}}(\text{DST}_{\text{VM}}^{|T|})} \varphi_\ell = 1 \tag{15}$$

$$\sum_{\ell \in \omega^{\text{OUT}}(\text{SV}_{\text{VM}}^{t})} \varphi_\ell = \sum_{\ell \in \omega_{\text{IN}}(\text{SV}_{\text{VM}}^{t})} \varphi_\ell \qquad \text{SV}_{\text{VM}} \in \{\text{SRC}_{\text{VM}}^{t}, \text{DST}_{\text{VM}}^{t}\} \in S, t \in T \tag{16}$$

$$a_{\text{VM,SV}_{\text{SRC}}}^{t} \geq \varphi_\ell \qquad \ell = (\text{SV}_{\text{SRC}}^{t-1}, \text{SV}_{\text{SRC}}^{t}) \in L, t \in T \tag{17}$$

$$a_{\text{VM,SV}_{\text{SRC}}}^{\tau} \geq \varphi_\ell \qquad \ell = (\text{SV}_{\text{SRC}}^{t-\text{MIGR}_{\text{VM}}^{\text{SV,SV}'}}, \text{SV}_{\text{DST}}^{t}) \in L, \; 0 \leq t - \text{MIGR}_{\text{VM}}^{\text{SV,SV}'} < \tau \leq t, t \in T \tag{18}$$

$$a_{\text{VM,SV}_{\text{DST}}}^{\tau} \geq \varphi_\ell \qquad \ell = (\text{SV}_{\text{SRC}}^{t-\text{MIGR}_{\text{VM}}^{\text{SV,SV}'}}, \text{SV}_{\text{DST}}^{t}) \in L, \; 0 \leq t - \text{MIGR}_{\text{VM}}^{\text{SV,SV}'} < \tau \leq t, t \in T \tag{19}$$

$$a_{\text{VM,SV}_{\text{DST}}}^{t} \geq \varphi_\ell \qquad \ell = (\text{SV}_{\text{DST}}^{t-1}, \text{SV}_{\text{DST}}^{t}) \in L, \; t \in T : t \geq \text{MIGR}_{\text{VM}}^{\text{SV,SV}'}. \tag{20}$$

$$\varphi_\ell \in \{0, 1\} \qquad \ell \in L \tag{21}$$

$$a_{\text{VM,SV}}^{t} \in \{0, 1\} \qquad \text{VM} \in V, \text{SV} \in \{\text{SRC}_{\text{VM}}, \text{DST}_{\text{VM}}\}. \tag{22}$$

Equation (14) shows the objective function of the pricing problem, calculating the reduced cost for each VM and fixing the resource utilization variables. Constraint (15) sets the initial and final placement of the VM at the start and end of the migration. Constraint (16) ensures the continuity of the path between the start and end time. Constraints (17), (18), (19) and (20) establish the resource utilization rules for VMs based on their current location and whether they are in transit. Constraints (17) and (20) cover resource usage for all consecutive instances of time where the VM stays on a server, whether it is the origin or destination, and Constraints (18) and (19) cover the span of time during which the VM moves from its origin server to its destination. During this span, resources are consumed on both servers.

6.2.3 An event-based model

We now present a decomposition model for an event-based formulation proposed in [17]. Unlike the previous model, this formulation does not track system state at each time index and focuses how many VMs are consuming resources in a given server in the worst case. For a given server, VMs that have some relation to that server are tracked through a or b: a tracks whether two VMs with the same destination server have both started moving towards it, and thus both consume resources on that server, and b controls whether two VMs, one with the server in question as it's destination and the other with the server as its source, are both consuming resources at the same time, due to the leaving VM having not finished its outgoing migration before the incoming VM starts its own. More concisely, the model relies on the concept of a server configuration:
$\gamma \in \Gamma_{\text{SV}}$ denotes a generic configuration for server SV. It corresponds to a set of precedence relations between the VMs associated with S (either moving to or from it). These determine whether one VM arrives before another, or whether one VM has left the Server before another one begins moving towards it. Each configuration γ is characterized by the following two sets of parameters:

$a^{\gamma}_{ji}, i \neq j$: $\text{SV} \in \text{DST}_{\text{VM}_i} \cap \text{DST}_{\text{VM}_j}$,

- = 1 if VM_j starts its migration towards SV before or at the same time as VM_i starts its migration towards SV,
- = 0 otherwise.

$b^{\gamma}_{ji}, i \neq j$: $\text{SV} \in \text{DST}_{\text{VM}_i} \cap \text{SRC}_{\text{VM}_j}$

- = 1 if VM_j completes its migration out of SV after VM_i starts its migration towards SV,
- = 0 otherwise.

Figure 13 illustrates why we have chosen to track resource constraints in this manner. Here, VM_i and VM_j are two VMs with the same intended destination , which is the source server for VM_k. The value of a_{ij} is set to 1, as VM_i has moved to the destination server before VM_j, and the value of b_{kj} is set to 1, as VM_k has yet to complete its migration out of the server before VM_j started its own migration towards

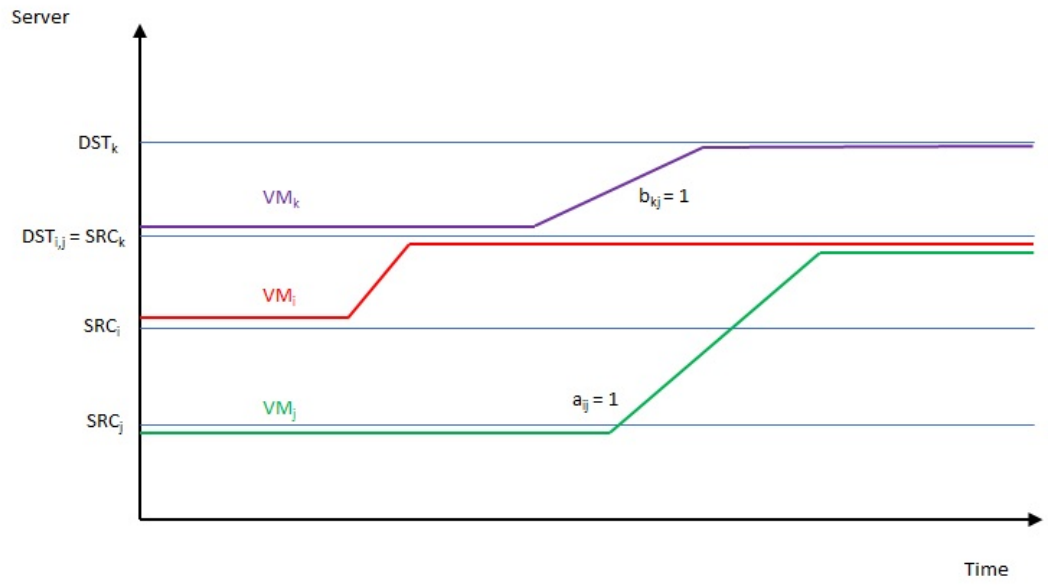

Figure 13: Visualisation of Event-based Resource Constraint Tracking, Adapted from [17]

it. Tracking events in this manner allows us to validate that the capacity constraints of the server have not been violated at any point during the migration process without having to get a snapshot of the server at each instance of time. These variables allow us to determine the existence of the point in time where the server has a maximum of resources consumed, that is to say when VM_j is migrating to it while VM_i is already there and VM_k has still not left. If, at this point, resource capacity constraints are respected, then we need not concern ourselves with the state of the server at any other time during the migration process

Variables of the Master Problem

$$z_\gamma = \begin{cases} 1 & \text{if configuration } z_\gamma \text{ was chosen for server SV} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{\text{VM}}^{\text{MIGR}} = \begin{cases} 1 & \text{if VM has been migrated from its source to its destination server} \\ 0 & \text{otherwise} \end{cases}$$

$x_{\text{VM}}^{\text{SV}}$ = time at which VM starts its migration towards SV

$$\alpha_{\text{SV}}^{\text{SV}'} = \begin{cases} 1 & \text{if SV and SV}' \text{ represent the same server} \\ 0 & \text{otherwise} \end{cases}$$

Master Problem

$$\min \text{C}_{\max} + \text{PENAL} \sum_{\text{VM} \in V} x_{\text{VM}}^{\text{MIGR}} \tag{23}$$

subject to:

$$x_{\text{VM}}^{\text{DST}} + \text{MIGR}_{\text{VM}}^{\text{SRC,DST}} \leq \text{C}_{\max} \qquad \text{VM} \in V \tag{24}$$

$$\sum_{\gamma \in \Gamma_{\text{SV}}} z_\gamma = 1 \qquad \text{SV} \in S. \tag{25}$$

$$\text{REQ}_{\text{VM}_i}^{\square} + \sum_{\gamma \in \Gamma_{\text{SV}}} \left(\sum_{\text{VM}_j \in V : \text{DST}_i = \text{DST}_j} \text{REQ}_{\text{VM}_j}^{\square} a_{ji}^{\gamma} + \sum_{\text{VM}_j \in V : \text{DST}_i = \text{SRC}_j} \text{REQ}_{\text{VM}_j}^{\square} b_{ji}^{\gamma} \right) z_\gamma \leq C_{\text{DST}_i}^{\square} \qquad \square \in \{\text{RAM}, \text{CPU}\}, \text{VM}_i \in V. \tag{26}$$

$$M \sum_{\gamma \in \Gamma_{\text{SV}}} a_{ji}^{\gamma} z_\gamma \geq (x_{\text{VM}i}^{\text{SV}} - x_{\text{VM}j}^{\text{SV}}) \qquad \text{VM}_i, \text{VM}_j \in V, \text{SV} \in \text{DST}_{\text{VM}_j} \cap \text{DST}_{\text{VM}_i} \tag{27}$$

$$M(\sum_{\gamma \in \Gamma_{\text{SV}}} a_{ji}^{\gamma} z_\gamma - 1) \leq (x_{\text{VM}i}^{\text{SV}} - x_{\text{VM}j}^{\text{SV}}) \qquad \text{VM}_i, \text{VM}_j \in V, \text{SV} \in \text{DST}_{\text{VM}_j} \cap \text{DST}_{\text{VM}_i} \tag{28}$$

$$-M \sum_{\gamma \in \Gamma_{\text{SV}}} b_{ji}^{\gamma} z_\gamma \leq x_{\text{VM}i}^{\text{SV}} - x_{\text{VM}j}^{\text{SV}} - \text{MIGR}_{\text{VM}j}^{\text{SV,SV}'} \qquad \text{VM}_i, \text{VM}_j \in V, \text{SV} \in \text{SRC}_{\text{VM}_j} \cap \text{DST}_{\text{VM}_i} \tag{29}$$

$$-M \left(\sum_{\gamma \in \Gamma_{\text{SV}}} b_{ji}^{\gamma} z_\gamma - 1 \right) \geq x_{\text{VM}i}^{\text{SV}} - x_{\text{VM}j}^{\text{SV}} - \text{MIGR}_{\text{VM}j}^{\text{SV,SV}'} \qquad \text{VM}_i, \text{VM}_j \in V, \text{SV} \in \text{SRC}_{\text{VM}_j} \cap \text{DST}_{\text{VM}_i} \tag{30}$$

$$z_\gamma \in \{0, 1\} \qquad \gamma \in \Gamma_{\text{SV}}, \text{SV} \in S \tag{31}$$

$$x_{\text{VM}}^{\text{MIGR}} \in \{0, 1\} \qquad \text{VM} \in V \tag{32}$$

$$x_{\text{VM}}^{\text{SV}} \geq 0 \qquad \text{SV} \in S, \text{VM} \in V \tag{33}$$

$$\alpha_{\text{SV}}^{\text{SV}'} \in \{0, 1\} \qquad \text{SV}, \text{SV}' \in S. \tag{34}$$

The objective function (23) of the master problem seeks to minimize the total makespan of the solution while also keeping the total number of migrations down via a penalty term. Constraint (24) established the lower bound on the makespan as the latest

migration. Constraint (25) limits the number of precedence configurations selected per server to 1. Constraint (26) ensures the resource capacities of the server are respected at all times by checking that, for each selected configuration and for each VM, the requirements of that VM, any other VM also entering the server, and any VM that has not yet left the server respect the server's max capacity for all resource types. Constraints (27) and (28) represent precedence relation constraints, ensuring that $a_{ji}^{\text{SV}} = 1$ only holds if the migration of VMj towards the server does not succeed that of VMi. Constraints (29) and (30) cover another aspect of precedence relation constraints, ensuring that $b_{ji}^{\text{SV}} = 1$ only holds if VMj finishes migrating out of the server after VMi starts it's own migration towards the server.

Pricing Problem

Each pricing problem is associated with a server. Therefore, we only consider a subset of the VM set V containing VMs which are either the "incoming" or the "outgoing" on the Server in question. Let us denote by V_{SV} the set of such VMs for server SV.

Variables of the pricing problem

$a_{ji}, i \neq j$: $\text{SV} \in \text{DST}_{\text{VM}_i} \cap \text{DST}_{\text{VM}_j}$,

- $= 1$ if VM_j starts its migration towards SV before or at the same time as VM_i starts its migration towards SV,
- $= 0$ otherwise.

$b_{ji}, i \neq j$: $\text{SV} \in \text{DST}_{\text{VM}_i} \cap \text{SRC}_{\text{VM}_j}$

- $= 1$ if VM_j completes its migration out of SV after VM_i starts its migration towards SV,
- $= 0$ otherwise.

We write below the pricing problem associated with server SV.

Objective: Reduced cost of variable z_γ with $\gamma \in \Gamma_{\text{SV}}$

$$\min -u^{(25)}$$
$$- \sum_{\mathrm{VM}_i \in V_{\mathrm{SV}}} \left(\sum_{\mathrm{VM}_j \in V : \mathrm{DST}_i = \mathrm{DST}_j} \left(\sum_{\square \in \{\mathrm{RAM}, \mathrm{CPU}\}} \mathrm{REQ}^{\square}_{\mathrm{VM}_j} u^{(26)}_{\square,i} + u^{(27)}_i + u^{(28)}_i \right) a_{ji} \right.$$
$$\left. + \sum_{\mathrm{VM}_j \in V : \mathrm{DST}_i = \mathrm{SRC}_j} \left(\sum_{\square \in \{\mathrm{RAM}, \mathrm{CPU}\}} \mathrm{REQ}^{\square}_{\mathrm{VM}_j} u^{(26)}_{\square,i} + u^{(29)}_i + u^{(30)}_i \right) b_{ji} \right) \quad (35)$$

subject to:

$$a_{ij} + a_{ji} \leq 1 \qquad \mathrm{VM}_i, \mathrm{VM}_j \in V_{\mathrm{SV}}, \mathrm{DST}_i = \mathrm{DST}_j. \quad (36)$$
$$a_{ji} \in \{0,1\} \qquad i \neq j, \mathrm{SV} \in \mathrm{DST}_{\mathrm{VM}_i} \cap \mathrm{DST}_{\mathrm{VM}_j}, \mathrm{VM}_i, \mathrm{VM}_j \in V \quad (37)$$
$$b_{ji} \in \{0,1\} \qquad i \neq j, \mathrm{SV} \in \mathrm{DST}_{\mathrm{VM}_i} \cap \mathrm{SRC}_{\mathrm{VM}_j}, \mathrm{VM}_i, \mathrm{VM}_j \in V \quad (38)$$

The objective function (35) sets the reduced cost of variable z_γ with $\gamma \in \Gamma_{\mathrm{SV}}$. Constraint (36) serves as a tiebreaker.

6.2.4 Solution process

Solving the linear relaxation of the master problem: column generation.

The decomposed models presented above are solved through Column Generation, also called delayed linear programming, following the process in [7]. Figure 14 depicts a flowchart representing the solution process for Column Generation as well as Branch and Bound. For Column Generation, we first convert the Master Problem into a Restricted Master Problem (RMP), which is a formulation of the Master Problem that only keeps a subset of variables that are deemed "sufficiently meaningful" [10], as the Master Problem may contain too many variables to solve in a timely manner. After the Primal Bound and Dual Bound are set, the Restricted Master Problem, relaxed into a linear problem, is solved and the Primal Bound is updated [29]. Next, the pricing problems are solved in order to generate columns with negative reduced cost, i.e., columns such that their addition in the Restricted Master Problem will lead to an improvement of the objective of its linear relaxation. A subset of these columns is then chosen to be added to the basis of the RMP and the Dual Bound is updated. In the event that there are multiple pricing problems, as in our project, the new dual

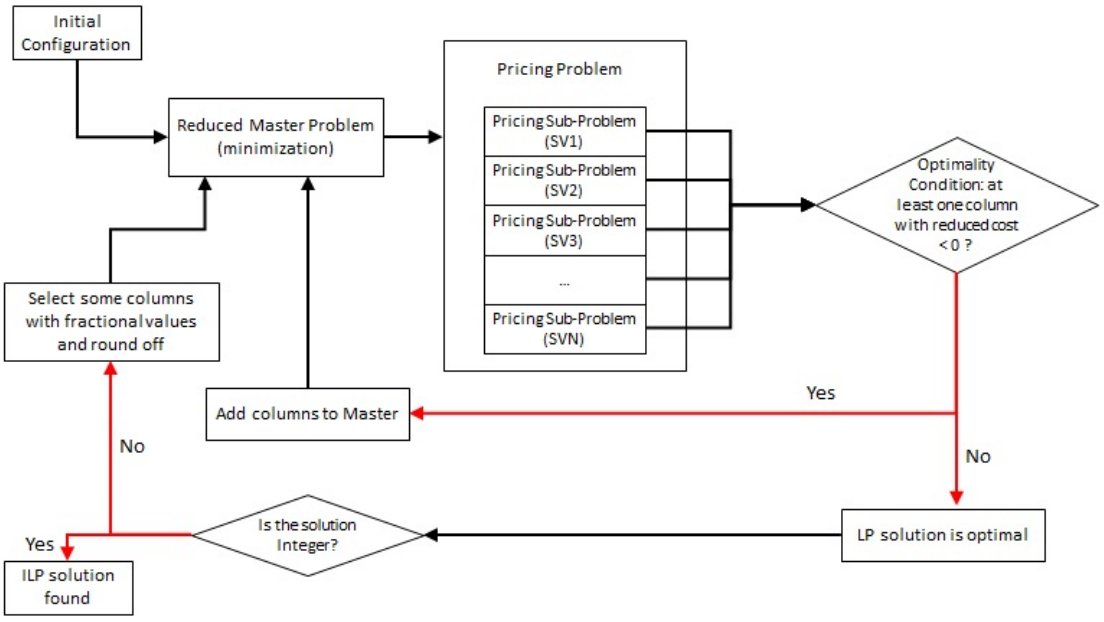

Figure 14: Solution Process for Column Generation and Branch and Bound

bound is calculated through a formula involving all pricing problems. The RMP is then solved again and the process repeats until the pricing problems can no longer generate columns with negative reduced cost or the Primal Bound is equal to the Dual Bound.

Deriving an ILP solution

There are two options:

Option 1: Deriving a heuristic ILP solution. In such a case, we get the optimal ILP solution of the last restricted master problem: this is not necessarily the optimal ILP of the master problem, but usually provides a good ILP solution whose accuracy can be evaluated as follows:

$$\varepsilon = \frac{\tilde{z}_{\mathrm{ILP}} - z^{\star}_{\mathrm{LP}}}{z^{\star}_{\mathrm{LP}}},$$

where

$z^{\star}_{\mathrm{LP}}$ denotes the optimal solution of the linear relaxation of the master problem

$\tilde{z}_{\mathrm{ILP}}$ denotes the "heuristic" ILP solution

$z^{\star}_{\mathrm{ILP}}$ denotes the optimal ILP solution.

We then have:

$$z^{\star}_{\text{ILP}} \leq \tilde{z}_{\text{ILP}}.$$

Option 2: Deriving an optimal ILP solution. It requires a Branch-and-Price algorithm, which essentially implements column generation within a branch-and-bound framework. The branching is typically done on the variables of the original, compact formulation (the x variables in this case) rather than on the variables introduced by the decomposition (z_p here) [29].

Branch-and-Price typically goes as follows: first, if the optimal solution returned by the relaxed RMP is integer, we stop, as this solution is the optimal solution. If not, we need to select one of the variables x^* that do not have integer values and split them into two subproblems with feasible ranges $X \cap \{x : x \leq \lfloor x^* \rfloor\}$ and $X \cap \{x : x \geq \lceil x^* \rceil\}$ [29]. From there, there are two ways we can proceed: the branching constraint is enforced either in the master problem, or in the pricing problems. Which of the two options is the better choice is debatable: typically, the second option may lead to better bounds at the cost of higher complexity [29].

An alternate branching scheme involves branching on the decomposition variables (often denoted by λ in a typical Dantzig-Wolfe decomposition, z in our case). The branching constraints are:

$$\sum_{k:y_{r,k}=1,y_{s,k}=1} \lambda_k = 1 \qquad \text{and} \qquad \sum_{k:y_{r,k}=1,y_{s,k}=1} \lambda_k = 0$$

where r, s are rows, k indicates a column in a 0-1 matrix Y, The solution to the master problem must be integer in the event that we cannot come up with a branching pair [3]. As row pairs are finite, so are the number of branches to explore, and each branch cuts a large number of variables from consideration. The idea behind this particular branching scheme is that branching eventually leads the coefficient matrix to become totally balanced, meaning the LP relaxation only has integral extreme points.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We proposed a set of greedy heuristics using different selection criteria that would seek to complete the generated migration problems in reasonable time without the use of temporary migrations to resolve indirect deadlocks. Results show minimal improvement over a basic greedy heuristic sorting by longest migration time, concerning both makespan and ensuring all migrations in a given problem complete. Designing greedy heuristics capable of avoiding "indirect" deadlocks is particularly difficult. We have, as of yet, been unable to devise some criteria on which to base scheduling decisions that would completely eliminate the chance to inadvertently bring the network into an unsolvable state. The case may be that indirect deadlock avoidance cannot be realized using only information available in the present time, and may need some sort of means of looking several steps ahead in order to ensure deadlock avoidance, and not have to rely on a means of repairing deadlocks in order to complete all scheduled migrations.

We also propose two decomposition ILP models assuming the use of column generation algorithms for solving the VM migration problem. The first proposed model is a *time-based* model that tracks the position of the VMs in the network at each discrete instance of time. While this type of modelling is relatively simple to elaborate, we suspect that it does not scale up well due to the large number of variables necessary for modelling the entire network state at every instance of time. The second model proposed, conversely, is an *event-based* model where we establish precedence

relations between the migrations, leading to a lower number of total variables compared to the previous, *time-based* model, which hopefully leads to better scaling with problem size, especially when compared to the compact formulations. We note that in practice, even if the number of variables is large, the column generation approach obtains solutions without having to explicitly consider all of them.

7.2 Future Work

Future work may involve further improvement of greedy heuristics. One possible direction could be to take advantage of parallel processing in order to have several greedy heuristics with different selection criteria running simultaneously. If the best criteria varies on a case-by-case basis, then we can have access to multiple solutions and select the best one for the current problem, without increasing computation time by running multiple heuristics sequentially. Next, we can revise the modelling of the problem to be more true to life. For example, during migration, the current scheme has a migrating VM reserve the totality of its resource requirements on both the source and destination servers. It would be more realistic to model the problem such that a transiting VM's requirements on the source change gradually as the migration progresses (as the data is transferred from source to destination, the VM could gradually free space on the source to match, and possibly permitting another migration to begin earlier than it would otherwise.) Finally, the proposed decomposed LP models can be implemented and tested in order to verify assertions that these models can achieve superior computation times compared to compact formulations.

Bibliography

[1] G.M. Adelson-Velsky and E. Levner. Project scheduling in and–or graphs: A generalization of Dijkstra's algorithm. *Mathematics of Operations Research*, 27(3):504–517, 2002.

[2] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/ Online; Accessed on : Dec. 23, 2020.

[3] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, and P.H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[4] D. Basu, X. Wang, Y. Hong, H. Chen, and S. Bressan. Learn-as-you-go with Megh: Efficient live migration of virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1786–1801, 2019.

[5] C. Boudreau. Algorithms for optimized virtual machine migration scheduling. Technical report, Concordia university, 2018. Undergraduate research report.

[6] Z.C. Chaczko, V. Mahadevan, S. Aslanzadeh, and C. Mcdermid. Availability and load balancing in cloud computing. In *International Conference on Computer and Software Modeling*. IACSIT Press, Singapore, 2011.

[7] V. Chvatal. *Linear Programming*. Freeman, 1983.

[8] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. *IEEE transactions on evolutionary computation*, 18(4):577–601, 2013.

[9] Dell servers. https://www.dell.com/fr-fr/work/shop/serveurs-dell-poweredge/sc/servers/poweredge-rack-servers Online; Accessed on : Dec. 23, 2020.

[10] J. Desrosiers and M. Lübbecke. A primer in column generation. In *Column generation*, pages 1–32. Springer, 2005.

[11] R. Kanniga Devi, G. Murugaboopathi, and M. Muthukannan. Load monitoring and system-traffic-aware live vm migration-based load balancing in cloud data center using graph theoretic solutions. *Cluster Computing*, 21(3):1623–1638, 2018.

[12] F. Farahnakian, P. Liljeberg, and J. Plosila. Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 500–507, 2014.

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

[14] M.P. Gilesh, S. Satheesh, A. Chandran, S.D.M. Kumar, and L. Jacob. Parallel schedule of live migrations for virtual machine placements. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 64–70. IEEE, 2018.

[15] A. Gupta, U. Mandal, P. Chowdhury, M. Tornatore, and B. Mukherjee. Cost-efficient live vm migration based on varying electricity cost in optical cloud networks. *Photonic Network Communications*, 30(3):376–386, 2015.

[16] M.R. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[17] B. Jaumard, O. Gluck, and D. Le. Effectiveness of heuristics for VM migration. In *submitted for publication*, 2020.

[18] K. Jeirroodi. Efficient heuristics for virtual machine migration in data centers. Master's thesis, Concordia University, Montreal, Canada, 2019.

[19] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*, pages 373–381. IEEE, 2006.

[20] J. Liu, Y. Li, D. Jin, L. Su, and L. Zeng. Traffic aware cross-site virtual machine migration in future mobile cloud computing. *Mobile Networks and Applications*, 20(1):62–71, 2015.

[21] R. Nasim, E. Zola, and A.J. Kassler. Robust optimization for energy-efficient virtual machine consolidation in modern datacenters. *Cluster Computing*, 21(3):1681–1709, 2018.

[22] B. Nazir et al. Qos-aware vm placement and migration for hybrid cloud infrastructure. *The Journal of Supercomputing*, 74(9):4623–4646, 2018.

[23] 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, 2017.

[24] L. Qi, Y. Chen, Y. Yuan, S. Fu, X. Zhang, and X. Xu. A qos-aware virtual machine scheduling method for energy conservation in cloud-based cyber-physical systems. *World Wide Web*, pages 1–23, 2019.

[25] S. Rahmani and V. Khajehvand. Burst-aware virtual machine migration for improving performance in the cloud. *International Journal of Communication Systems*, 33(7):e4319, 2020.

[26] T. Saber, A. Ventresque, J. Marques-Silva, J. Thorburn, and L. Murphy. Milp for the multi-objective vm reassignment problem. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 41–48. IEEE, 2015.

[27] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. Parallelizing live migration of virtual machines. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 85–96, 2013.

[28] 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 & Fuzzy Systems*, 32(2):1389–1400, 2017.

[29] F. Vanderbeck and L.A. Wolsey. Reformulation and decomposition of integer programs. In *50 Years of Integer Programming 1958-2008*, pages 431–502. Springer, 2010.

[30] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

[31] H. Xu, Y. Liu, W. Wei, and Y. Xue. Migration cost and energy-aware virtual machine consolidation under cloud environments considering remaining runtime. *International Journal of Parallel Programming*, 47(3):481–501, 2019.

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