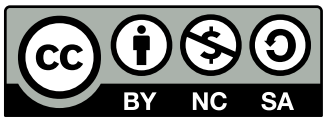
Aalto University
School of Science

Master's Programme in Mathematics and Operations Research

Optimizing application placement in private cloud environment under capacity constraints

Eero J. A. Virmavirta

Master's Thesis
2025



Aalto University
School of Science

Author Eero J. A. Virmavirta

Title Optimizing application placement in private cloud environment under capacity constraints

Degree programme Mathematics and Operations Research

Major Systems and Operations Research

Supervisor Prof. Ahti Salo

Advisor M.Sc. (Tech.) Onni Lampi

Collaborative partner Retail Logistics Excellence - RELEX Oy

Date 28 April 2025 **Number of pages** 64+4 **Language** English

Abstract

Modern cloud-based data centers depend on systematic capacity management methods to enable efficient utilization of computational resources. The continuing surge in demand for computational power is driving the data-intensive software industry toward building massive centralized data centers, which require sophisticated resource allocation algorithms to allow coherent consolidation of those resources. Improving the utilization of data center hardware allows the server fleet size to be decreased, reducing the costs and the environmental footprint of the data center operations.

In this thesis, we investigate a private cloud environment, in which customer applications are allocated to the self-managed server fleet of the organization. We model the allocation system as a vector bin packing problem with two separate resources: static RAM usage, and periodically varying CPU utilization. The objective of our model is to minimize the number of utilized servers, while providing the required resources for each application. Due to the computational complexity of the optimization problem, we develop five heuristic algorithms for solving the problem. The performance of these heuristics is rigorously evaluated in several scenarios simulating real-life occurrences. Further, we apply the most suitable heuristic to the data of a real data center of the organization, and investigate whether the current allocation process could be improved utilizing the algorithm.

The evaluations show that the most suitable heuristic for the system based on the combination of solution quality and computational efficiency is the first-fit-decreasing algorithm. Applying this algorithm to real data, the results show that the data center server fleet size can be reduced by almost a third compared to the current allocation, given our simplifying assumptions regarding the CPU utilization of the applications. Furthermore, we show that in case of disturbances disabling parts of the server fleet, we can utilize the heuristic to reallocate on average over 90% of the hostless applications back to the remaining fleet, if up to 40% of the fleet becomes unavailable. Generally, the thesis suggests that the resource utilization of the private cloud system can be improved with simple, systematic allocation procedures.

Keywords Resource allocation, capacity management, cloud computing, optimization

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Eero J. A. Virmavirta

Työn nimi Sovellusten sijoittamisen optimointi yksityisessä pilvilaskentaympäristössä kapasiteettirajoitteiden vallitessa

Koulutusohjelma Mathematics and Operations Research

Pääaine Systems and Operations Research

Työn valvoja Prof. Ahti Salo

Työn ohjaaja DI Onni Lampi

Yhteistyötaho Retail Logistics Excellence - RELEX Oy

Päivämäärä 28.4.2025 **Sivumäärä** 64+4 **Kieli** englanti

Tiivistelmä

Nykyaikaiset pilvilaskentaan nojaavat datakeskukset hyödyntävät järjestelmällistä kapasiteetinhallintaa laskentaresurssien tehokkaan käytön mahdollistamiseksi. Laskentatehon kasvava kysyntä on johtanut keskitettyjen datakeskusten rakentamiseen tietointensiivisellä ohjelmistoalalla. Näissä datakeskuksissa tarvitaan kehittyneitä algoritmejä resurssien allokointiin, jotta resurssitarpeet voidaan yhteensovittaa tehokkaasti. Datakeskusten laitteiston käyttöastetta parantamalla keskusten palvelinverkoston kokoa voidaan pienentää, mikä vastaavasti vähentää operaatioiden kustannuksia sekä ekologista jalanjälkeä.

Tässä diplomityössä käsitellään yksityistä pilvilaskentaympäristöä, jossa asiakassovellukset on allokoitu organisaation omalle palvelinverkostolle. Allokaatiosysteemi mallinnetaan työssä vektoripakkausongelmana (vector bin packing problem), jossa on kaksi resurssirajoitetta: staattinen hajasaantimuisti (RAM) sekä säännöllisesti vaihteleva suorittimen (CPU) käyttö. Mallin tavoitteena on tarvittavien palvelimien määrän minimointi. Koska ongelma on laskennallisesti hyvin raskas, sen ratkaisemiseksi esitellään viisi heuristiikkaa. Heuristiikkojen suorituskykyä arvioidaan tosielämää mukailevissa tilanteissa, joiden perusteella valitaan systeemiin sopivin algoritmi. Valittua algoritmia sovelletaan organisaation datakeskukseen, ja tutkitaan, voiko nykyistä allokaatioprosessia tehostaa heuristiikan avulla.

Arvioinnit osoittavat, että tulosten laadun sekä laskennallisen tehokkuuden kannalta sopivin heuristiikka on first-fit-decreasing -algoritmi. Kun algoritmia sovelletaan oikeaan datakeskukseen, palvelinverkoston kokoa voidaan pienentää lähes kolmanneksella, ottaen huomioon sovellusten suoritinkäytöstä tehdyt yksinkertaistavat oletukset. Tämän lisäksi työssä arvioidaan sovellusten takaisinsijoittamista palvelimille, kun osa verkostosta poistuu käytöstä häiriön vuoksi. Tulokset osoittavat, että heuristiikalla keskimäärin yli 90% kodittomista sovelluksista voidaan sijoittaa takaisin palvelimille, kun jopa 40% verkostosta poistuu käytöstä. Kokonaisuudessaan diplomityön pohjalta voidaan todeta, että kyseisen pilvi-infrastruktuurin resurssien käyttöä voidaan parantaa järjestelmällisten menetelmien avulla.

Avainsanat Resurssiallokaatio, kapasiteetinhallinta, pilvilaskenta, optimointi

Preface

Today marks the culmination of my academic journey. I have soared like a flight-impaired eagle – with a graceful yet tentative pace – toward the pinnacle of higher education: the title of Master of Science in Technology. This gömböc-esque voyage has seen its stable and unstable points, but through trials and tribulations, the forbidden fruit that is graduation, will soon touch my dehydrated lips. I shall carry with me the life lessons and distilled wisdom that my selective cognition has gathered in Otaniemi, for the entirety of my remaining life.

First and foremost, I wish to thank my advisor, Onni, for making both of my theses a reality. Your result-driven, no-nonsense approach helped this thesis to be completed not only on time, but with great success. Thank you Ahti, for your supervision, expertise, and time. Additionally, I want to thank Jäynäjäbät, and all my other colleagues from the Guild of Physics, for your cooperation, sparring, and support in our shared effort toward striving for excellence in academia.

Beyond my studies, the vast and historical student culture of Otaniemi has played a remarkably significant role in my early adult life. For all the magnificent and (un)forgettable moments, I wish to thank all my companions and stakeholders. Thank you BY, and all my roommates for the explosive communal years; they felt like an eternity. Thank you for the all-knowing N. Salttu and the workers of the salt mines, for the endlessly jubilant labor. Thank you, the honorable members of Mallutoimikunta, as well as the fine people of Civitoimikunta, Merikerho, and FTMK'20, for your lasting, fiery friendship. Thank you Teekkari village, Campus section, and the secret catacombs of Nektariveljeskunta, for the throbbing, organic endeavors we have experienced together. Thank you, my favourite Otaniemi structures - Merikabinetti, Takkakabinetti, Stockmann-kabinetti, and the Temple of Lemminkäinen - for your majesticity and lawfulness. Thank you my brothers-in-arms J. A. Pellonpää, Ana, lämä, Ozy et. al. for the warm nights of true liberty. Finally, I wish to thank my partner, Velma, for her love and support. All of you have molded me into the person I am today.

To honor the spirit of Wappu, I shall now lay down my laptop at the base of Wappucivi. I will do so for the glory of Teekkarius, for the bastion of our society that is academic freedom, and most of all, for the friends we made along the way.

Otaniemi, 28.4.2025

Eero J. A. Virmavirta

Contents

Abbreviations

BFD	Best-fit-decreasing
BPP	Bin packing problem
CPU	Central processing unit
CSP	Cutting stock problem
DC	Data center
FFD	First-fit-decreasing
KP	Knapsack problem
NF	Next-fit
PM	Physical machine (server)
PSO	Particle swarm optimization
RAM	Random access memory
VBPP	Vector bin packing problem
VM	Virtual machine (server)
WFD	Worst-fit-decreasing

1 Introduction

Allocating scarce resources is a topic of growing interest across industries. Efficient resource allocation is essential for performance optimization and waste minimization in various systems, ranging from supply chains and logistics to computational infrastructures. In cloud computing, allocating resources such as memory and computational power has become increasingly crucial in the past decades due to exponentially growing amounts of data and computational requirements [1]. Meeting this demand calls for more efficient resource management to ensure cost-efficiency while maintaining required system performance. A systematic resource allocation process allows cloud providers to reduce their hardware costs and environmental footprint while increasing the predictability and automation possibilities of the system. Better capacity management thus directly results in higher profitability for the cloud provider.

To tackle the challenge of capacity utilization, modern data centers have implemented advanced algorithms and models to ensure efficient usage of computational resources. The research on capacity management in cloud-based data centers is extensive. In section 3, we provide a comprehensive review of the common approaches for modeling resource allocation problems on a general level, and building on this, we narrow our focus to capacity management in cloud computing and data centers. However, most methods for capacity allocation rely on the free migration of applications from one server to another or simply allocating individual computational tasks to on-demand servers with sophisticated scheduling algorithms [2, 3]. These methods are not directly applicable to the problem of this thesis, as here, we investigate a data center where the allocation of applications to servers is considered static. Transferring applications from one server to another is possible and frequently executed, but it produces outages for application end users and requires manual labor from data center administrators. Thus, it cannot be practiced without consideration.

In this thesis, we model a resource allocation system in a private cloud environment as a multi-dimensional vector bin packing problem (VBPP). VBPP is a generalized version of the one-dimensional bin packing problem (BPP), a classic problem in combinatorial optimization, in which a set of heterogeneous items must be packed into a set of bins while minimizing the number of total bins utilized [4]. Here, web applications are modeled as said items, and they are allocated to servers (bins), considering the limited memory and computational resources of the servers. As such optimization problems quickly become computationally very challenging as the number of elements increases, we develop five heuristic algorithms for solving the problem, in addition to an exact algorithm. The performance of the proposed algorithms is evaluated through simulations and analyses using generated pseudo-realistic data.

In section 6, we apply the most suitable algorithm to a private cloud data center to investigate whether the application placement within the server fleet can be improved. As the computation hardware of the data center is self-managed and more importantly self-funded, more efficient resource utilization yields evident and measurable value for the organization. Additionally, we study the resilience of the data center by modeling application placement in situations in which portions of the server fleet become unusable.

The fundamental research objective of this thesis is to investigate whether there is room for improvement in the application placement process through optimization methods. If we can show that server utilization can be improved with systematic allocation of resources, the organization can implement the suggested methods, or focus further investigations on improving the allocation process according to the findings of this work. If this thesis allows the organization to reduce server fleet size now or in the future even slightly, the cost reductions on hardware acquisitions and power consumption will be considerable.

2 Definitions

In this section, we provide background for the thesis and define the necessary concepts for the modeling and discussion of resource allocation in the framework of cloud-based computing environments.

2.1 Cloud computing

Cloud computing is a framework that enables the use of central computing resources, such as memory, databases, processing power, etc. – remotely over the internet. With cloud computing, users can share the same physical resources simultaneously, allowing more constant and efficient utilization of those resources. This is a vast improvement from the traditional computing model, in which each computer utilizes its own local resources. Cloud computing architecture allows end users to utilize on-demand resources as they require, while the cloud provider handles the management and maintenance of the underlying cloud system and hardware. The users are generally charged only for the resources that they use, which makes cloud computing much more elastic and scalable compared to localized computing systems. The clear majority of modern corporations prefer cloud computing due to the scalability and efficiency compared to local computing, in addition to the outsourced resource management. [1]

Cloud computing systems can be roughly divided into two distinct groups: public clouds and private clouds. In a public cloud, the computing resources are available for the general public or a group of organizations, and the cloud system is owned by the company managing and selling the cloud resources. The largest and most commonly known cloud computing systems provided by software giants like Amazon (AWS), Microsoft (Azure), and Google (GCP) are great examples of public clouds. In contrast, private clouds are operated by a single organization, and the cloud system is managed either by that organization or an external party. A private cloud solution provides more security and control for the organization, but the disadvantages come from worse scalability and often higher costs. When computational demand increases, the physical resources of the private cloud system must be expanded, essentially by increasing the server fleet size, whereas with public clouds, increased computational requirements can often be handled simply by paying a larger subscription fee for the cloud provider. [1]

In this thesis, we mainly focus on private cloud environments, as the system we investigate in sections 4 through 6 is facilitated on a private cloud. However, we do consider public clouds as well in the literature review in section 3.2.

2.2 Server

On a general level, a server can be considered as any system that provides resources and services to its clients [5]. In this thesis, we consider computer server machines that are utilized for hosting web applications allocated to them. Servers contain a limited amount of resources, such as memory, storage, and computational power, which can be capitalized by the corresponding applications.

Traditionally, a server has been defined as a physical machine (PM) with local hardware. However, through the process of virtualization, the resources of the physical server can be divided into separate virtual machines (VM) that can themselves act as servers. In essence, virtualization is a process that allows the abstraction of computational resources of the physical machine into multiple virtual entities. Each VM can run its own operating system and host its own applications, essentially behaving as a stand-alone server, even though it is sharing physical resources with other VMs. This technology helps better consolidate server resources as applications with differing software needs can be hosted on the same PM. Thus, virtualization enhances the flexibility, utility, and scalability of the hardware. Virtualization is widely utilized in cloud computing systems due to these major upsides. [6]

2.3 Data center

Individual servers are the building blocks of computing infrastructure, but in the realm of computation-intensive software industry, they are rarely sufficient for serving any parts of the system on their own. Servers are often clustered together in vertical frames, referred to as racks, designed to house and organize servers and other necessary IT hardware. The function of racks is to provide centralized management and maintenance of critical components for the servers, and simply to allow them to be organized in a spatially efficient manner. A room, an entire building, or a large campus full of server racks that are clustered together is referred to as a data center (DC).

The increased demand for computation in many industries has driven the construction of massive DCs across the globe. Additionally, as more and more organizations are outsourcing their computing needs to cloud-based DCs, the trend of creating large centralized computing facilities is further accelerated. Large-scale DCs are especially popular indeed among public cloud providers, that serve a wide base of organizations and individuals. The scale of the DCs improves the flexibility of the systems and the consolidation of computing resources but bears the price of more difficult management and optimization. [2]

The vast majority of modern DCs rely on virtualization to provide maximal flexibility and centralized control, among other reasons. An overview of a cloud infrastructure hosted in a DC could be illustrated as depicted in figure 1. End users send computational requests, or tasks, to the VMs that are hosting their applications. Those VMs are allocated to physical servers located in the DC. The allocation of applications is often not static to specific VMs, but the tasks are elastically assigned to suitable machines according to the details of the request and the current loads of the servers. Due to this dynamic process, the resources can be efficiently utilized, as the tasks are allocated and scheduled to continuously maximize the resource utilization on servers. However, the complexity of the process makes it extremely difficult to optimize.

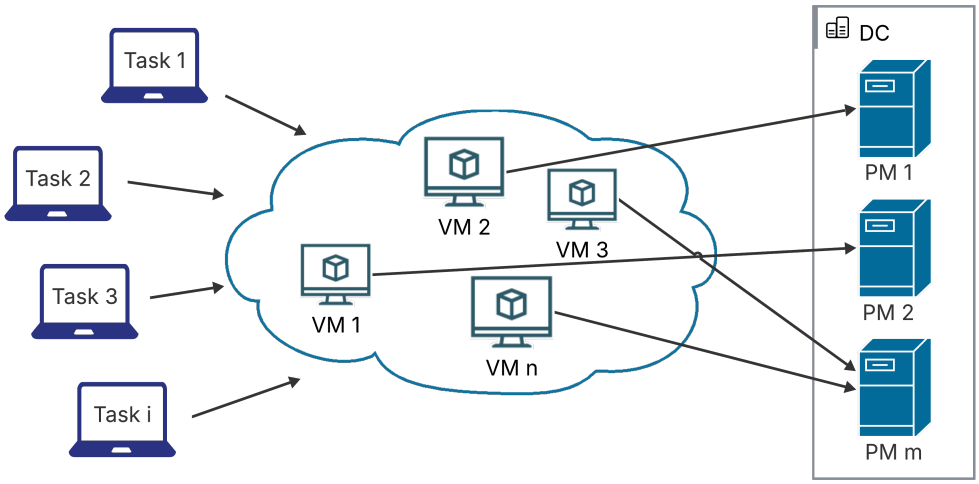


Figure 1: An exemplary cloud infrastructure with i tasks, n virtual machines, and m physical servers.

2.4 Random access memory

Random access memory (RAM) refers to the hardware of the server or other computer machine that is used for storing application programs and data once the machine is operational. RAM is a volatile memory since all data stored on it is removed when the server is powered off. Thus, data that needs to be permanently saved must be stored elsewhere, commonly on the hard drive of the server. Effectively, RAM can be considered the main memory component of the server, and its capacity plays a key role in the powerfulness of the server. The size of RAM is usually measured in GB or TB. [7]

In this thesis, RAM is one of the main components to be optimized. The applications allocated to servers require certain amounts of RAM, and the RAM capacity of the servers largely restricts how many applications each server can host. On a general level, we can assume that the more RAM capacity a server has, the more efficiently it is possible to allocate the set of applications. In addition to applications, other programs on the server, such as the operating system, also require some RAM capacity. Thus, we can never allocate the full RAM capacity to application programs, and there often exists a somewhat strict limit on which portion of RAM we should allocate to applications. In our case, the baseline limit is usually 80% of maximum capacity. From now on, when discussing the memory of a server or the memory requirement of an application, we refer to RAM.

2.5 Central processing unit

The central processing unit (CPU) is commonly described as the operational brain of the computer. CPU executes operations, performs calculations, and processes data based on the instructions of the memory unit. Once the CPU has executed its tasks, the results are delivered back to the memory unit. CPUs are utilized in many devices,

from smartphones to servers, to execute all required computational tasks. [7]

A key metric regarding the computational power of a CPU is the number of physical cores it contains. A core is an independent processing unit within the CPU, that can execute computational requests separately from other cores. Interestingly, a physical core can also perform two tasks simultaneously using a technology called hyper-threading (also called simultaneous multithreading). Hyper-threading makes the physical core appear as if there were two independent virtual cores acting side-by-side, even though they are sharing the physical components. This allows the CPU to divide tasks between these virtual cores further extending the parallel computational capabilities of the server. However, this does not inherently imply that hyper-threading increases the computational capacity of the server. Rather, it allows the computation to be logically parallelized, potentially providing significant performance increases for applications that support parallelization. Additionally, one server can have multiple nodes, which are essentially fully independent computational entities with their own CPUs. This means that a server with two 64-core nodes, with hyper-threading in use, has a total of $2 \cdot 64 \cdot 2 = 256$ threads and can therefore perform 256 computational tasks simultaneously, given that all tasks are suited for parallelization. [8]

The computational load of applications and tasks is often measured in CPU time, most commonly CPU seconds. One CPU second means that a task utilizes the computational efforts of one core for one second. If the task requires more computational effort, it either needs to utilize more cores for the computation, or the computation will last longer. A task that requires 512 CPU seconds in total, can thus be performed in two seconds with a 256-thread server if there are no other overlapping tasks on the server competing for the same resources, given that the application is capable of dividing the computation to multiple cores. Later in this thesis, we use CPU seconds to determine the workload required by applications at different time slots during the day.

2.6 Application

In this thesis, we view applications as computer programs that are used to access and manage customer software programs and data. Applications cannot function on their own, they need to be hosted on a server at all times. Applications require certain resources from the servers, mainly RAM and CPU capacity. The RAM requirement of an application is static and predetermined based on recent peak memory usage. The realized RAM use of an application varies periodically, as the memory load increases over time until a scheduled database cleanup is performed. The CPU requirements of applications are modeled in this thesis based on their historical CPU usage.

3 Review of resource allocation

This section reviews the most common approaches for modeling resource allocation problems used in the literature, and, in addition, discusses the most relevant resource allocation schemes within the realm of computer servers and data centers.

3.1 Common models for resource allocation

Resource allocation generally refers to the problem of assigning available resources to agents in a system. Shareable resources are often limited or their manufacturing and utilization incur costs. Thus, it is desired to allocate those resources as efficiently as possible, while fulfilling the needs and demands of the agents.

Decision-making problems on resource distribution have been extensively studied in operations research. In this section, we describe four archetypal models for resource allocation problems and discuss their respective variations, applications, and solution procedures.

3.1.1 Knapsack problem

Many resource allocation problems can be cast under the umbrella of the knapsack problem (KP). KP is one of the most studied problems in combinatorial optimization. KP was first mentioned in literature in 1896 [9], although it has been investigated in folklore throughout history. KP has applications in many fields, for example, cargo loading in logistics [10], agricultural land-use problems [11], and multimedia storage systems [12].

The classic version of the problem entails selecting items from a selection, each with an assigned value, to be placed inside a knapsack while respecting the weight limit of the knapsack. The goal is to select the set of items that maximizes the value of the knapsack contents. The standard KP can be formulated as

$$\begin{aligned} \max \quad & \sum_{i \in N} p_i x_i \\ \text{subject to} \quad & \sum_{i \in N} w_i x_i \leq c \\ & x_i \in \{0, 1\}, \ \forall i \in N, \end{aligned}$$

where N is the set of items each with a profit, p_i, and a weight, w_i, subjected to the capacity limit of the knapsack, c. Binary decision variable x_i indicates whether item i is selected or not. [13]

In addition to the standard KP, several extensions of the KP have been introduced to respond to more complex scenarios. In the fractional KP, items are divisible, meaning that only a fraction of an item can be placed in the knapsack. In some variations, each item can be selected multiple times. If there is a limit on how many times each item can be selected, the problem is called bounded, otherwise, it is an unbounded KP [14]. Further variations of the KP are multidimensional KP (MdKP) and multiple

KP (MKP). Introducing additional constraints besides weight transforms the problem into an MdKP. Having more than one knapsack, which rather often is the case, yields MKP. [15]

Although the formulation and the idea of KP are very simple, the complexity of the problem appears when developing solution algorithms for the problem. Although fairly attractive exact solution algorithms utilizing methods such as dynamic programming and branch-and-bound have been introduced [16], it is widely believed that no exact algorithms that solve the KP in polynomial time exist. Consequently, the calculation time of algorithms increases exponentially as the number of items, knapsacks, dimensions, and other possible factors increases. This characteristic means that the problem is at minimum NP-hard [17]. Nevertheless, extensive research around KPs has led to the development of various approximate algorithms, many of which yield close to optimal results in reasonable calculation time. [18, 19]

3.1.2 Bin packing problem

Changing the optimization objective from maximizing profit to minimizing the number of knapsacks effectively transforms the problem into the bin packing problem (BPP). In BPP, the goal is to pack a set of items into the minimum number of bins while respecting the capacity limits of each bin. Unlike KP, BPP does not usually assign specific profits to the items. [4]

The formulation of the standard BPP closely resembles that of the KP. Notable differences are the objective function and an additional constraint requiring that each item must be packed into exactly one bin

$$\begin{aligned} \min \quad & \sum_{j \in M} y_j \\ \text{subject to} \quad & \sum_{i \in N} w_i x_{ij} \leq c y_j, \quad \forall j \in M \\ & \sum_{j \in M} x_{ij} = 1, \quad \forall i \in N \\ & x_{ij}, y_j \in \{0, 1\}, \quad \forall i \in N, \ \forall j \in M, \end{aligned}$$

where, N is the set of items with weight, w_i, and M is the set of available bins with capacity c. The binary decision variables are y_J indicating whether bin j is used, and x_{ij} depicting if item i is packed in bin j.

Similarly to KP, BPP has several extensions to better model various real-life scenarios. These include variable size bin packing, in which the bins are not of uniform size, and multidimensional versions, in which items and bins are subject to additional capacity requirements and constraints [20]. These multidimensional packing problems are often divided into two categories based on how the separate constraints interact. Consider the problem of packing two-dimensional objects into a rectangular bin. Since the two constraints are length and width, one can naturally pack multiple thin stripes into a single bin, even if the length of one item spans the entire length dimension of the bin. Now, consider a bin in which the constraints are length and weight. If one

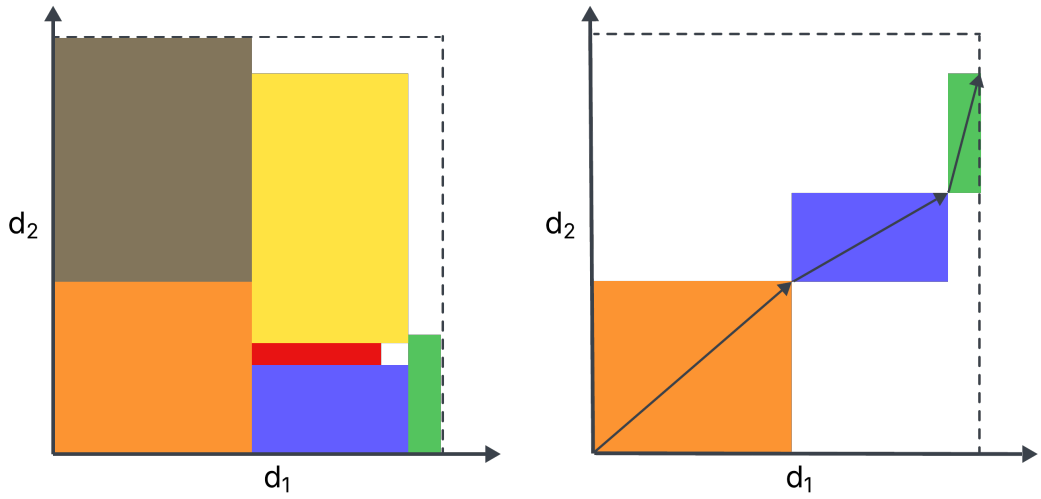


Figure 2: The difference between geometric (left) and vector (right) bin packing.

packs a short and heavy item that occupies a fraction of the length capacity, but the entire weight capacity, no additional items can be packed in the bin, even though the length dimension is poorly utilized. The first scenario is an example of a geometric BPP, and the second is often referred to as a vector BPP. The distinction between these types is illustrated in figure 2. [21, 22]

BPP is widely used for modeling common resource allocation problems such as vehicle cargo loading [23, 24], storage optimization [25], allocating computational resources [26], and countless other real-life scenarios. While the applications of BPP are often very similar compared to KP, their key difference lies in the optimization objectives: BPP focuses on minimizing the space required for packing, while KP aims to maximize profit with a fixed packing capacity.

Similarly to the KP, BPP has been proven to be strongly NP-hard [17]. As a consequence, extensive research has been dedicated towards developing exact, approximate, and heuristic algorithms for solving BPP. The vast catalogue of literature underscores the significance of the problem and the interest that the scientific community has in this issue. [27, 28, 29, 30]

3.1.3 Cutting stock problem

The problem of cutting specific-length pieces from standard-sized stock material while minimizing the waste generated in the cutting process is known as the cutting stock problem (CSP) [31]. As one can guess, CSP originates from the timber industry, which is still one of the main application domains of the problem. However, it has also been used to model various other systems in which the goal is to reduce wasted resources. An example of this is memory allocation in servers, with the objective of minimizing unutilized memory capacity [32].

CSP is closely related to BPP and can be formulated in a fairly similar manner. Both models aim to minimize the number of utilized base elements. The key distinction

between the models is that while BPP focuses usually on packing unique items, in CSP there is usually demand for cutting multiple identical pieces.

There are many ways to state the standard CSP, but maintaining similar formulations as for the earlier models, we can express

$$\begin{aligned}
\min \quad & \sum_{j \in M} y_j \\
\text{subject to} \quad & \sum_{i \in N} l_i x_{ij} \leq L y_j, \quad \forall j \in M \\
& \sum_{j \in M} x_{ij} = d_i, \quad \forall i \in N \\
& y_j \in \{0, 1\}, \quad \forall j \in M, \\
& x_{ij} \in \{0, 1, 2, \dots\}, \quad \forall i \in N, \ \forall j \in M.
\end{aligned}$$

For purposes of illustration, we have replaced the weight assigned to items with length, l_i, and the capacity is indicated by the total length of the stock material, L. The restriction that items can only be included once is removed, and replaced with the notation that pieces in each length must be cut according to the respective demand, d_i.

CSP has been commonly extended to higher-dimensional versions, which in two-dimensional settings essentially means cutting pieces from rectangular stock material. In many applications, the cutting process has limitations so that only orthogonal cuts can be performed, and each cut has to span the entire width of the piece. These are so-called guillotine cuts, which have also been extensively studied. The difference between guillotine and non-guillotine cuts is illustrated in figure 3. The benefit of guillotine cuts is that they are usually faster to compute, and more efficient to perform in real-life systems. While not restricting oneself to guillotine cuts usually yields better optimal solutions due to added flexibility in selecting cutting patterns, the optimal cuts might prove infeasible or slow to perform with machinery limitations. Therefore, guillotine cuts are often preferred in industry applications. [4]

Like KP and BPP, CSP also belongs to the set of NP-hard problems. A famous exact algorithm leveraging constraint relaxations and delayed column generation was

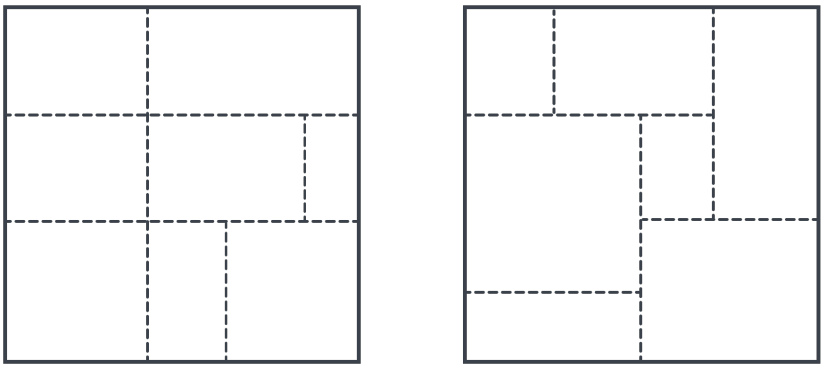


Figure 3: Examples of guillotine (left) and non-guillotine (right) cuts.

developed by Gilmore and Gomory in 1961, which has been widely utilized and further refined since [33]. However, with large problem instances, exact algorithms become computationally infeasible. This has led to the development of various heuristic methods for approximating CSP solutions. [34]

3.1.4 Scheduling problem

Consider a set of tasks that must be performed by a processor with limited resources. Each task is assigned with corresponding due dates, profits, relative importance, and other factors. Dynamically allocating the resources of the processor to perform each task can be formulated as a scheduling problem.

In its simplest form, a scheduling problem includes one processor and a set of tasks. However, it can be extended in many ways, such as incorporating multiple processors or introducing tasks that consist of multiple subtasks that need to be executed in a specific order on designated processors. An example of such scheduling pattern is illustrated in figure 4. The scheduling problem shares many similarities with the resource allocation problems presented earlier, as we again strive for efficient resource utilization. However, scheduling introduces a temporal dimension to the system, requiring decisions on the timing of resource utilization. [35]

Many kinds of different systems can be modeled through the scheduling problem. Some common applications include scheduling processes in traditional manufacturing lines [36], minimizing inventory and transportation costs in logistics [37], and deciding the timing and scope of potential projects or investments [38].

Even with simple formulations, the scheduling problem has been proven NP-complete and approximating solutions is even more difficult than in the static resource

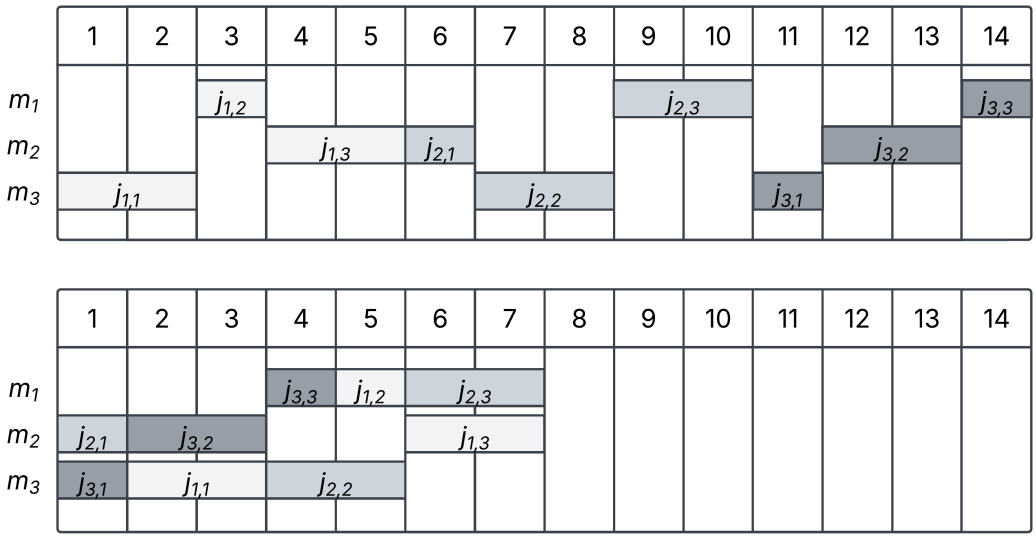


Figure 4: Illustration of two scheduling patterns of the same tasks. Each of the three tasks is divided into three subtasks, which must be performed on specific machines in a specific order.

allocation problems discussed earlier. Fortunately, efficient heuristic algorithms for solving the problem have been extensively studied, providing attractive alternative solution methods for many different scheduling scenarios. [39]

3.2 Resource allocation in servers and data centers

The energy consumption of DCs has surged in recent years due to increasing computational demand, which has driven the creation of massive DC facilities worldwide. In 2023, DCs accounted for 4,4% of total energy consumption in the US, up from 1,9% in 2018, and projections estimate that this share could rise to 6,7% - 12% by 2028, largely due to the extensive calculation power requirements of AI applications. [40]

Among the many energy-consuming components in a DC, the largest consumer is the server fleet, accounting for almost 50% of the total energy usage of the facility [41]. The energy consumption of active servers is not static, as servers running on higher capacity require more energy. The increase in consumption is not linear, as idle servers (operating at minimal capacity) utilize more than 50% of energy compared to the peak capacity energy consumption [42]. Thus, reducing the number of active servers is key to limiting DC energy consumption. Additionally, downsizing the server fleet lowers cooling demand, reduces the required floor space, and of course brings down the cost of server acquisitions, which account for 45% of total financial costs of running a DC [43].

Achieving the goal of reducing server fleet size calls for comprehensive capacity management throughout the processes of the DC. This entails allocating appropriate computational resources to applications, task scheduling policies, demand forecasting, resource scaling, constant monitoring and tuning of the system, and much more [3]. A variety of commercial solutions have been developed to tackle the whole capacity management process or specific parts of it. Examples of these are Muse [44], Océano [45], PACMan [46], and Kubernetes [47], which is considered the industry standard today. This section does not focus on these solutions. Rather, we survey the core principles of DC capacity management that directly relate to resource allocation problems. These entail the task allocation on servers and the virtual machine placement on physical machines, as described in section 2.3.

3.2.1 Task allocation

The main idea of task allocation is to distribute requested jobs from clients among the available resources in a fast and cost-efficient manner. The tasks to be assigned have specific resource requirements and deadlines, and once they have been executed, the tasks cease to exist. Task allocation is thus typically modeled as a dynamic process in which the goal is to allocate the requested resources for the tasks while balancing between underutilizing available resources (wasted resources and money), and overloading the servers (degraded performance). Dynamic task allocation is illustrated in figure 5. [48]

The most common approach to modeling task allocation is the scheduling problem. As the scheduling problem is known to be strongly NP-hard, tasks are usually allocated

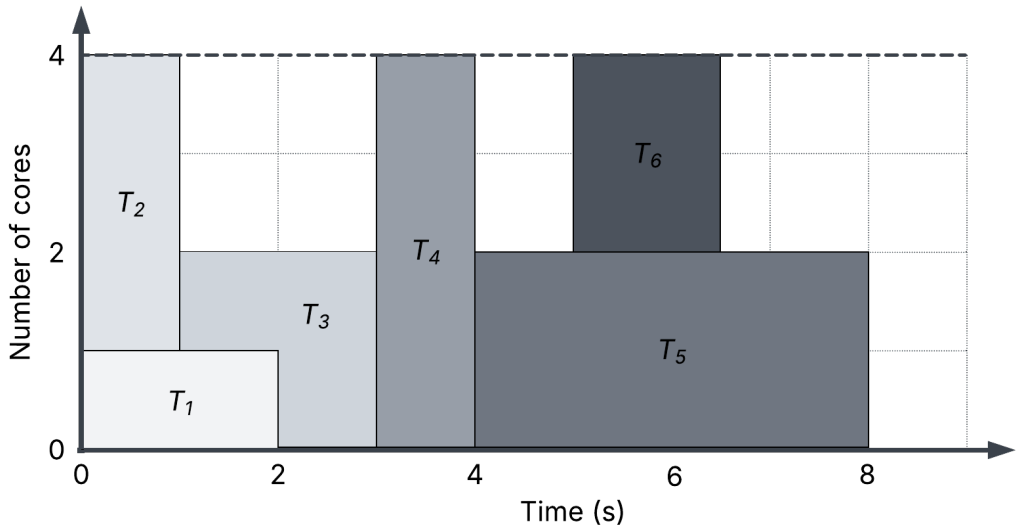


Figure 5: Illustration of dynamic task allocation on one four-core server. The loads of the tasks range from two CPU seconds (T_1) to eight CPU seconds (T_5).

to servers using some heuristic algorithm or general policy. Task scheduling algorithms in computing systems have been extensively studied for at least 50 years, and there is still plenty of research ongoing [49, 50, 51, 52], reflecting the importance and complexity of the task scheduling problem in computing.

Beyond the inherent difficulty of the scheduling problem, many cloud resource-specific aspects add to this difficulty. The increasing number and complexity of tasks demand efficient and scalable solutions frameworks for scheduling, which lead to larger and harder-to-manage DCs. Additionally, the computational requirements of the tasks are oftentimes unknown in advance, which complicates the allocation. The heterogeneous resource types of virtualized DCs and their interdependencies also increase the degree of difficulty in predicting the required resources. [2]

Besides the scheduling problem, different methods for modeling task allocation systems have been proposed. The system can be modeled through queueing theory [53], in which tasks are clustered into queues, and their mean response times are predicted and used for optimization. A control theoretic approach [54] using feedback from the current allocation status has also been explored. Chen et. al. [55] combine queueing and control theories in a task allocation system with promising results.

The majority of researchers approach task allocation with the objective of minimizing the number of utilized servers or maximizing the total resource utilization on servers, though these objectives are fundamentally similar. Besides this, alternative objectives have also been investigated. The problem can be modeled with the objective of minimizing energy consumption [56], which is related to resource usage but not linearly, thus possibly yielding different results. Other objective approaches include balancing the load on servers as equally as possible [57], and minimizing the execution time of each task [58]. Eventually, the goal in essentially all approaches is utilizing

resources as efficiently as possible.

Recently, researchers have investigated more detailed and descriptive ways of modeling task allocation systems, including multi-objective optimization and additional constraints. Introducing more complex problem formulations allows the system to be more accurately modeled, with the price of added computational complexity. To tackle this, more advanced algorithms, such as particle swarm optimization [59], ant-colony optimization [60], and several evolutionary algorithms [61] have been fine-tuned and applied to allocation systems.

Another approach for allocating resources for tasks or applications is using fairness metrics to ensure each task is allocated at least some resources. One fairness-driven approach is max-min fairness, which maximizes the smallest share allocated to any application. A more sophisticated and general approach is dominant resource fairness (DRF), in which each application first receives a share of the resource it most desperately needs [62]. DRF is especially useful when the expectation is that not all requests can be fully satisfied. The goal is often searching for Pareto efficient allocations, meaning that no application can improve its allocation without hurting some other application. DRF model has also further extensions, involving more complex infrastructures and objectives [63, 64].

3.2.2 Virtual machine placement

The hardware resources of a physical server can be divided into separate computation entities through virtualization (as explained in section 2.2). These so-called virtual machines can execute tasks and host applications rather independently, provided they are also hosted on a physical server that allows the VM to harvest their resources such as memory, CPU, and storage. The resource management problem at this stage is determining how to select suitable servers to host these VMs, to allow efficient utilization of the resources of the physical servers. The problem is illustrated in figure 6. In most systems, VMs can be migrated from one server to another, and their capacities can be adjusted to an extent. However, the flexibility of VM scaling and migration varies by system. In many cases, VM placement is modeled as a static operation, though a dynamic aspect can be created by solving the model consecutively. Research on both static and dynamic VM placement methods has been extensively conducted for the past 20 years. [65]

Static VM placement on servers strongly relates to the BPP. Items with certain sizes need to be packed into bins (servers) with capacity restrictions, with the goal of minimizing the number of bins utilized. Usually, the VM models have multiple resource requirements that are (at least on the macroscopic level) independent. This allows the system to be more specifically modeled with vector bin packing, rather than geometric bin packing. This modeling has been used in many papers on VM placement, though naming conventions have not always been consistent across studies [66, 67, 68, 69, 70, 71].

While modeling VM placement as a static process is computationally simpler, it often fails to adapt to the variation of VM resource requirements, leading to lackluster resource utilization. Speitkamp et. al. [72] consider a system in which the resource

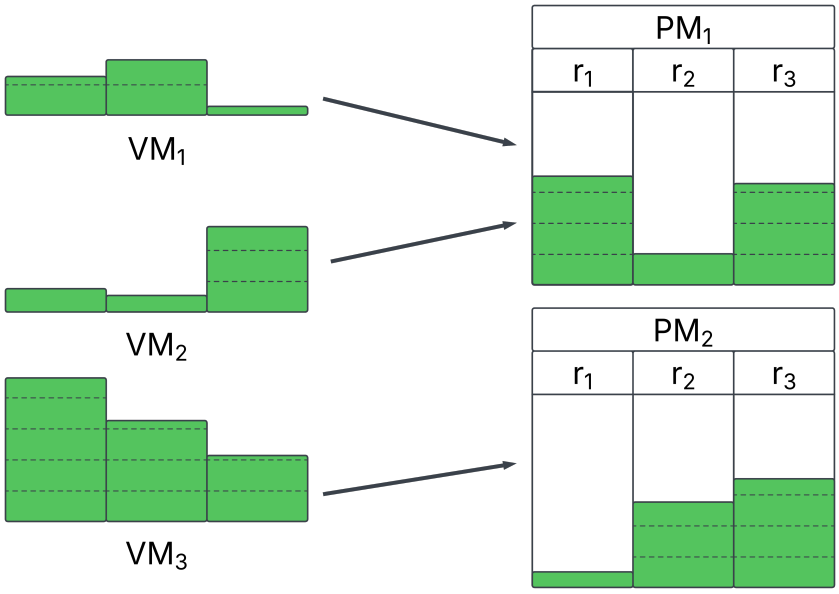


Figure 6: VM placement problem with three resource constraints.

requirements of VMs vary over time. They investigate allocating VMs to servers both with and without considering the fluctuation in requirements, and the dynamic version results in 31% of hardware savings. This efficiency was enabled by the seamless migration of VMs from one server to another, allowing servers to be shut down during quiet hours.

Similarly to the task allocation problem in DCs, the objectives in VM placement models usually focus on minimizing the number of physical servers or maximizing their utilization rates. However, the statically modeled placements often need to be re-evaluated due to some occurrence, e.g. variation of VM resource requirements, server maintenance, or shifts in total computational demand. Since in many systems, it is preferred to avoid migrating VMs from one server to another, minimizing the number of migrations can become a new key objective in VM placement problems [73, 74]. Essentially this means that the system needs to compromise between minimizing server fleet size and mitigating the number of migrations, with a trade-off ratio suitable for the specific needs of the system.

Solution approaches for the VM placement problem often revolve around heuristic strategies, because globally optimal solutions or even their approximations cannot be attained except for rather limited sizes of problem instances. Most commonly used heuristics include variations of classic greedy algorithms, such as first-fit and best-fit, and their extensions first-fit-decreasing (FFD) and best-fit-decreasing (BFD), which have yielded promising results in many systems. More complex methods for VM placement include several metaheuristic algorithms, such as genetic algorithm and simulated annealing, which reduce the solution space and utilize high-level heuristics to potentially locate optimal solutions. These methods have often produced very efficient VM placements, but their performance cannot be guaranteed in all systems. They are computationally more expensive compared to simpler heuristics. [70, 75, 76]

The modeling of static VM placement problem also relates to allocating applications to servers, as long as those allocations are also considered static. The physical procedure of assigning a VM to a server differs from the assignment of applications, but from a resource utilization standpoint, the processes can be considered similar. Therefore, the same models and solution frameworks have been consistently used for statically allocating applications to servers.

4 Modeling a capacity allocation system

In this chapter, we present a private cloud-based computational environment, in which a set of applications are to be hosted on a set of physical servers. The eventual goal is to minimize the number of utilized servers without compromising the service provided to customers. We develop the mathematical formulation of the system and discuss its solutions and present five heuristic algorithms for generating approximate solutions for the resource allocation problem.

4.1 Description of the system

We consider a private cloud environment, with one level of resource allocation. This means that there is no separate task allocation and VM placement to be executed. The system consists of a set of applications that needs to be hosted on a fleet of physical servers. The allocation is modeled static, meaning that the allocation is carried out only once. A dynamic aspect can be added by solving the model multiple times while altering the set of servers and applications based on changing conditions in between iterations.

The applications in our system have two separate resource requirements, namely RAM and CPU usage. The amount of RAM required by each application is static and it is manually predetermined based on historical maximum RAM usage.

CPU usage, however, varies significantly for all applications, mainly depending on the time of day. As the model is static, the CPU demand at separate time frames needs to be defined, instead of having a constant CPU capacity reservation. The variation is modeled by dividing the day into multiple intervals and defining the computational requirement separately for each time slot based on the maximum CPU usage within that slot. An example CPU reservation based on the usage pattern is illustrated in figure 7. We use the maximum CPU usage instead of averaging over the total CPU seconds within the time slot (essentially meaning the integral of CPU usage) because the average would in some cases significantly limit the CPU usage, lengthening runs and possibly causing applications to miss important deadlines.

The length of the CPU usage time slots is initially set to one hour. Thus, each slot is equal. There is however no reason that the time slots must be equally spaced. If deemed advantageous, we can shorten the time slots during busy hours, and lengthen them when there is less competition for the CPU resources. Additionally, if the one-hour time slots prove excessively lengthy, we can condense the slots to 30 minutes, 15 minutes, or even one-minute length. This may be necessary if the applications have short but high spikes in their CPU usage patterns, resulting in excessive capacity reservations around those spikes. Nevertheless, this means that initially, our model has 25 separate resource dimensions: one for RAM, and 24 for CPU utilization.

In this model, each application requires the same amount of CPU regardless of the day. Therefore, we can perform the allocations based on a 24-hour time frame, and the allocations will not violate constraints either in the future. This is a simplification, as there can be variation between days, and the total CPU and memory requirements can change over time. This can be taken into account by re-estimating the application

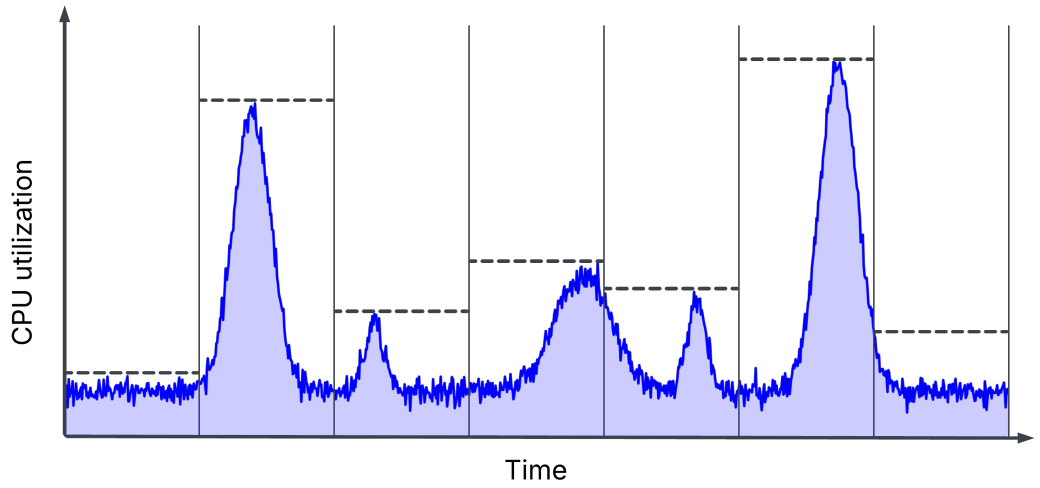


Figure 7: The reserved CPU capacity for each time period is determined based on the peak CPU utilization of the application within that period.

requirements regularly and performing reallocations when necessary. Additionally, a longer time frame could be used for defining the CPU usage patterns to better detect the actual requirement and the variations between days.

The full server fleet of the organization consists of five DCs, which are geographically and infrastructurally independent. The separation of DCs is not taken into account in the allocation process in our model, as we model the migration of an application similarly within a DC and between DCs, even though there are real-life blockers in migrating an application between DCs. The separation of DCs can, however, play a role when considering a disturbance (e.g. fire, power outage) that would disable an entire DC. We can then expect that such occurrence would disable one fifth of the total fleet, instead of the entire fleet.

All servers have strict RAM and CPU capacities that cannot be exceeded. The CPU capacity for servers is defined based on the number of physical cores the server contains. The capacity is static at all times, but in the model, we split it into different time slots, similarly as with the applications. This is to track the CPU usage on servers during these slots. Initially, we model each server similarly, meaning that all capacity restrictions are homogeneous. However, we keep the option of including heterogeneous server capacities in the model later. This adds another layer of complexity to the model but could prove mandatory when applying the model to real-life data. The number of servers that can be utilized is not restricted in the model.

The private cloud system that is utilized specifically by one organization provides some unique features that differentiate the system from other cloud infrastructures. Compared to massive public cloud systems of e.g. Google and Amazon that many companies utilize for their computations, a small private cloud system is far less scalable. Whereas with public cloud providers, computational resources can be extended by simply updating the subscriptions, in a private system, increasing computational resources essentially always means purchasing new servers. This also underscores

the importance of efficient resource utilization within the organization, as it allows maintaining a smaller fleet. A clear benefit of an independent infrastructure is that it makes workload forecasting possible. A public cloud provider can essentially never know in advance which type of jobs it will need to host, while a private system with static allocations can to a large extent predict the requirements based on historical data. The static allocation and predictable computational requirements are the pieces that enable the efficient utilization of server resources and allow us to model the system as described in this section.

4.2 Mathematical formulation

We model the resource allocation system as a multidimensional vector bin packing problem. This is a common method in the literature for modeling VM and application placement problems, as described in section 3.2.2. VBPP is discussed in detail in section 3.1.2.

The objective of our model is to minimize the number of utilized servers while allocating a set of applications to the server fleet and respecting the capacity constraints of each server. To avoid infeasibly large applications, we restrict the resource requirements, r, to $r_{ad} \leq c_d$, for all applications a in all dimensions d, meaning that an empty server needs to have enough capacity to host any application in the system. In reality, there are applications with even greater resource requests, but these are not considered in this model. In the actual infrastructure, the largest applications are hosted on dedicated servers with sufficient resources. The number of utilized servers is not limited, but we set an upper bound for formulation purposes. This upper bound, m, is large enough so that it does not restrict the number of utilized servers.

We state the problem as the following integer linear program

$$\text{Min.} \quad \sum_{s=1}^{m} y_s \tag{1}$$

$$\text{st.} \quad \sum_{s=1}^{m} x_{as} = 1, \quad \forall a \in \{1, \ldots, n\} \tag{2}$$

$$\sum_{a=1}^{n} r_{ad} x_{as} \leq c_d y_s, \quad \forall s \in \{1, \ldots, m\}, \forall d \in \{1, \ldots, D\} \tag{3}$$

$$x_{as} \in \{0, 1\}, \quad \forall a \in \{1, \ldots, n\}, \forall s \in \{1, \ldots, m\} \tag{4}$$

$$y_s \in \{0, 1\}, \quad \forall s \in \{1, \ldots, m\}. \tag{5}$$

The decision variables and parameters used in the formulation are explained in table 1. The decision variable x contains each application-server combination, meaning that x has $n \cdot m$ elements. The size of x thus increases fast as problem instances become larger. This highlights the importance of selecting a suitable m, that is as small as possible while not disturbing the feasibility of the problem. A suitable m can be found e.g. by calculating an approximate solution beforehand using some heuristic method and utilizing that solution as m.

Table 1: Variable and parameter notations.

Symbol	Description
x_{as}	is application a allocated to server s
y_s	is server s utilized in solution
r_{ad}	resource requirement of application a in dimension d
c_d	server capacity limit in dimension d
n	number of applications to allocate
m	upper bound for server to utilize
D	number of resource dimensions

The objective function of the model (1) minimizes the number of utilized servers. Constraint (2) states that each application needs to be assigned to exactly one server, constraint (3) ensures that no capacity limits are violated, and constraints (4) – (5) indicate that all decision variables are binary.

When necessary, we can extend the model to additionally limit migrating applications from one server to another, given an initial allocation is already in place. In that case, we introduce an initial allocation variable, x^I_{as}, denoting the assignment of each application before solving the model. The objective (1) can then be expressed as

$$\text{Min. } \alpha \sum_{s=1}^{m} y_s \; + \; \frac{\beta}{2} \sum_{a=1}^{n} \sum_{s=1}^{m} |x^I_{as} - x_{as}| \tag{6}$$

$$\alpha + \beta = 1, \quad \alpha, \beta \in [0, 1], \tag{7}$$

where the additional constraint (7) restricts the weights of the new objective. These weight parameters can be scaled accordingly depending on how heavily the migrations should be avoided compared to server minimization. As one migration yields two changes in x, we divide β by two.

4.2.1 Optimal solution

It has been shown that there is no polynomial time approximation scheme for the VBPP when there are two or more dimensions. Therefore, the presented model is not only NP-hard as stated in section 3.1.2, it may very well be APX-hard (unless $P = NP$ [77]). In any case, locating optimal or even approximate solutions for the model can be extremely time-consuming even with moderate instance sizes. [78, 79]

Keeping this in mind, we implement a method to reach an exact solution for the problem. With small enough instances, we can locate optimal solutions in reasonable time. For this purpose, we use a popular open-source solver, HiGHS version 1.7.0 [80], which is suitable for integer linear programs. The solver utilizes refined branch-and-cut techniques for reducing the set of potential solutions. HiGHS solver has shown good performance compared to other open source software in industry benchmark tests [81].

For approximating the optimal solution, we implement a particle swarm optimization (PSO) algorithm [82]. PSO is a metaheuristic that iteratively improves candidate solutions until no improvements can be made or other stopping criteria are met. PSO

consists of a swarm of particles that generate interim solutions while searching for the global optimum in the given search space. The method has been inspired by the scattering movement of animals in nature, namely flocks of birds and schools of fish. This method has been utilized in literature for solving resource allocation problems with good results [83]. We implement PSO with ready-made function `pso.m` in the MATLAB Global Optimization Toolbox.

The aforementioned solution procedures are evaluated in section 5.4.1, where their feasibility and computation time are analyzed, and their solutions are compared to those obtained using the heuristics presented in the following section 4.3.

4.3 Heuristics

We explore alternative approaches for calculating the allocations due to the complexity of the problem. This is especially important as the number of applications, or problem dimensions, increases. For this purpose, we introduce five heuristic algorithms for generating the allocations. The heuristics are significantly less computationally demanding compared to methods for determining the global optimum. The algorithms are explained in this section, and their performance is rigorously tested in section 5.

4.3.1 First-fit-decreasing

First-fit is a simple algorithm commonly used for bin packing, which requires no size information regarding the applications beforehand. Its basic function is very simple: pick an application to allocate, and assign it to the first server that has enough capacity remaining to satisfy its resource requirements. If no active servers have the required capacity, a new server is initialized to host the application. This process is performed separately for each application. In case an application cannot be hosted on any server, e.g. due to infeasibly large resource requirements, it is ignored by the algorithm.

First-fit-decreasing (FFD) is a variation of first-fit, in which the applications are sorted in a decreasing order based on size before allocation. This allows the largest, and thus most demanding applications to be allocated first. The pseudocode of the method is in algorithm 1. FFD is an offline algorithm, as it requires the resource requirement information regarding the applications before allocation. If this information is unavailable, the standard first-fit heuristic is applied.

Because the model is multi-dimensional, the sorting of applications based on size is not directly obvious. A decision needs to be made on which resource requirements are used to determine the size of the application and how. The most obvious selection would be to use the stand-alone memory requirement as the metric for sorting, which is arguably the most crucial resource for the usability of applications. However, if memory is not the most competed resource, sorting based on it can yield unsatisfactory results. Another approach for ranking the applications is to identify the most sought-after resources and use those to determine application size. For a more comprehensive view, one can sum together all resource demands to identify which applications require the most resources in total. The best approach depends on the specific characteristics of the servers and applications.

Algorithm 1 First-fit-decreasing

Input: Set of applications $\{a_1, a_2, ..., a_n\}$, with d resource requirements; Set of servers $\{s_1, s_2, ..., s_m\}$, with capacity c_d
Output: Number of servers used and the assignment of each application
Sort applications in decreasing order based on resource requirements
for each application a in the sorted list **do**
    for each server s **do**
        if a fits in s **then**
            Allocate a to s
            Break for
        end if
    end for
    if a has not been allocated **then**
        Initialize a new server and allocate a to it
    end if
end for
Return Number of severs used and the assignment of each application

For the one-dimensional case, the tight lower bound for solutions received with FFD has been proven to be $FFD(I) \leq 11/9 \cdot OPT(I) + 6/9$, where $FFD(I)$ and $OPT(I)$ represent the solution received with FFD and the optimal solution, respectively [84]. This means that in the one-dimensional case, FFD provides solutions that never utilize more than 22% more bins than the optimal solution, excluding small problem instances. This lower bound is not directly applicable to the multidimensional case but provides an insight on the effectiveness of the algorithm.

4.3.2 Best-fit-decreasing

The classic best-fit algorithm has many similarities to the first-fit algorithm. Instead of assigning the application to the first feasible server, an attempt is made to place it on each server, and the one that has the most suitable fit with regard to remaining capacity after the allocation is selected. In best-fit-decreasing (BFD), the applications are first sorted based on size, so that more demanding applications are placed first. The pseudocode of BFD is in algorithm 2. As BFD calculates the fit of each application in all servers, the algorithm can be computationally more expensive compared to some other heuristics such as FFD. This drawback should however not cause major problems unless the number of required servers becomes extremely large.

Similarly to FFD, the sorting of applications needs to be performed in a manner that best suits the system one is investigating. Additional ambiguity comes from the notation of a "best" fit, which is not trivial. In a one-dimensional case, the best fit generally means the tightest fit, i.e. the allocation that leaves the minimum amount of space in the bin after the allocation. In our multi-dimensional case, there is a need to incorporate all resources in the decision of determining the best fit. Therefore, the best fit is the one in which the sum of unutilized resources is the smallest after

Algorithm 2 Best-fit-decreasing

```
1:  Input: Set of applications {a_1, a_2, ..., a_n}, with d resource requirements; Set of
    servers {s_1, s_2, ..., s_m}, with capacity c_d
2:  Output: Number of servers used and the assignment of each application
3:  Sort applications in decreasing order based on resource requirements
4:  for each application a in the sorted list do
5:      BestFitServer ← None
6:      BestFit ← ∞
7:      for each server s do
8:          if a fits in s then
9:              RemainingSpace ← Unused capacity of s after placing a
10:             if RemainingSpace < BestFit then
11:                 BestFitServer ← s
12:                 BestFit ← RemainingSpace
13:             end if
14:         end if
15:     end for
16:     if BestFitServer is not None then
17:         Allocate a to BestFitServer
18:     else
19:         Initialize a new server and allocate a to it
20:     end if
21: end for
22: Return Number of servers used and the assignment of each application
```

potential allocation. To ensure each dimension has equal weight in determining the empty space, the requirements are normalized before calculating the goodness of the fit. The definition of "best" fit may need to be reconsidered in case the number of dimensions increases, as that directly shifts more weight to CPU capacity.

4.3.3 Worst-fit-decreasing

Another approach to the best-fit algorithm is to allocate applications with the goal of maximizing the remaining space on servers after the allocation. This method is generally referred to as the worst-fit algorithm. Whereas best-fit aims to completely fill up the servers as soon as possible, worst-fit has the goal of allowing more options for future allocations due to the maximized remaining space. As with the previously presented algorithms, the applications are sorted in decreasing order based on size before allocation, which yields the worst-fit-decreasing (WFD) algorithm. The pseudocode of the heuristic is in algorithm 3.

Similarly to the BFD algorithm, the goodness of the fit is determined by summing together all unused resources on the server after potential allocation. This time, the most desired destination for the application is the server which has the maximum remaining capacity after placement. WFD is an especially relevant method in situations

Algorithm 3 Worst-fit-decreasing

Input: Set of applications $\{a_1, a_2, ..., a_n\}$, with d resource requirements; Set of servers $\{s_1, s_2, ..., s_m\}$, with capacity c_d
Output: Number of servers used and the assignment of each application
Sort applications in decreasing order based on resource requirements
for each application a in the sorted list **do**
    WorstFitServer ← None
    WorstFit ← -1
    for each server, s, that hosts at least one application **do**
        if a fits in s **then**
            RemainingSpace ← Unused capacity of s after placing a
            if RemainingSpace > **WorstFit then**
                WorstFitServer ← s
                WorstFit ← **RemainingSpace**
            end if
        end if
    end for
    if WorstFitServer is not None **then**
        Allocate a to **WorstFitServer**
    else
        Initialize a new server and allocate a to it
    end if
end for
Return Number of servers used and the assignment of each application

in which the distribution of the load on servers is taken into account. If the pool of servers is large and there is no strict need to minimize the number of utilized servers, the resource requirements may be distributed somewhat evenly to mitigate the possibility of resource overload that could occur e.g. if some applications utilize more resources that have been allocated to them. That being said, WFD is a commonly used method also for bin minimization purposes.

4.3.4 Next-fit

Next-fit (NF) algorithm is the simplest allocation heuristic considered. Contrary to the previously presented algorithms, the applications are not sorted before proceeding to the actual algorithm. With NF, the standard next-fit procedure is carried out to allocate an application onto the current server if there is enough capacity remaining. Otherwise, a new server is initialized to host the application, and the previous server is sealed and no more applications are placed on it. The same process is performed for all applications. In NF, we focus on one server at a time, and after progressing to the next one, we never return to earlier servers. This is the key difference between FFD and NF, as FFD considers all active servers in the fleet as potential destinations for the application. The pseudocode of NF is in algorithm 4.

Algorithm 4 Next-fit

Input: Set of applications $\{a_1, a_2, ..., a_n\}$, with d resource requirements; Set of servers $\{s_1, s_2, ..., s_m\}$, with capacity c_d
Output: Number of servers used and the assignment of each application
Initialize a new server and set it as **CurrentServer**
for each application a **do**
    if a fits in **CurrentServer then**
        Allocate a to **CurrentServer**
    else
        Initialize a new server and allocate a to it
        Update CurrentServer to the new server
    end if
end for
Return Number of servers used and the assignment of each application

Next-fit-decreasing (NFD) is a commonly used variation of the algorithm. It has been shown that the lower bound for NFD is significantly higher than for standard NF [85]. However, the pool of applications in our system contains a relatively high ratio of heavy resource consumers, which would mean that a significant number of applications are allocated to their own servers if the applications are sorted in decreasing order. This effect is smaller if the applications are in random order. Therefore, we do not incorporate sorting in the NF allocation algorithm.

NF can produce lackluster allocations, because there may remain large unused capacities on servers as applications cannot be allocated to earlier servers. However, the benefit of NF comes from its computational efficiency. NF provides results fast even with large problem instances, as the computation time with NF increases linearly with respect to the number of applications, as opposed to exponentially growing calculation times with all of the previously presented algorithms. This is because the algorithm only attempts to fit the applications to the current server instead of optimizing the fit among a larger pool of options.

4.3.5 Random allocation

Finally, we introduce an algorithm that allocates applications to servers randomly. The purpose of this algorithm is to investigate and illustrate the performance efficiency of the other algorithms, compared to a completely arbitrary allocation heuristic. The random allocation algorithm allocates applications to servers one at a time, by gathering all feasible servers that host at least one application and additionally one empty server. Out of the set of feasible servers, one is selected randomly for the allocation. As the set of active servers becomes larger, the likelihood of allocating an application to an empty server decreases since only one empty server is always included in the set of possible assignments. Due to this, one can expect that as the set of applications becomes larger, the performance of the heuristic improves. The pseudocode of this heuristic is in algorithm 5.

Algorithm 5 Random allocation

Input: Set of applications $\{a_1, a_2, ..., a_n\}$, with d resource requirements; Set of servers $\{s_1, s_2, ..., s_m\}$, with capacity c_d
Output: Number of servers used and the assignment of each application
for each application a **do**
    ActiveServers ← list of all servers that host at least one application
    FeasibleServers ← empty list
    for each server i in **ActiveServers do**
        if a fits in i **then**
            Add i to **FeasibleServers**
        end if
    end for
    Add a new empty server to **FeasibleServers**
    RandomServer ← randomly selected server from **FeasibleServers**
    Allocate a to **RandomServer**
end for
Return Number of servers used and the assignment of each application

Despite its simplicity, the random heuristic is computationally more expensive than at least FFD and NF. This yields from the fact that the feasibility of each active server needs to be determined for each application before we can randomly select an assignment for the application. One could also randomly select an iteratively increasing amount of servers without investigating the feasibility until the set of servers contains at least one feasible location. This would, however, complicate the purposely simple heuristic unnecessarily, so we rather tolerate the longer calculation time.

5 Performance evaluation of algorithms

In this chapter, we simulate the resource allocation system using theoretical models of servers and applications that resemble real-life instances. The goal of the tests is to compare the performances of the algorithms in chapter 4.3 and determine which ones are most suitable for the system based on solution quality and computational efficiency. The data used for the tests, testing methods, different scenarios, and their results are presented in this section.

5.1 Data

We define the application models for these tests by extracting historical CPU usage data of actual applications from our target organization. Because each weekday is fairly similar with regard to computational demand, we randomly select one recent regular weekday and utilize the CPU usage data from that date. The CPU data is not continuous. It contains the total CPU seconds that the application has utilized during each minute. To achieve as accurate results as possible, we use single minutes as the time slots for CPU allocation. The applications are therefore modeled as $24 \cdot 60 + 1 = 1441$ dimensional vectors, one dimension for CPU use during each minute of the day, and one for the memory reservation. The amount of memory that is reserved for each application is static and known, and we use it for the application base data as is.

An example of an application CPU usage data is illustrated in figure 8. It is common that the baseline CPU usage is very low for a regular application, while there are some spikes indicating scheduled heavier calculations. These spikes are often scheduled at somewhat similar times – as recurrent calculations are generally performed during the night – which makes it more demanding to consolidate the resources. In addition to these quite moderate CPU utilizers, a few applications have significantly higher baseline CPU demands, and much lengthier spikes often during the night. These applications leave little to no room on their servers for hosting other applications.

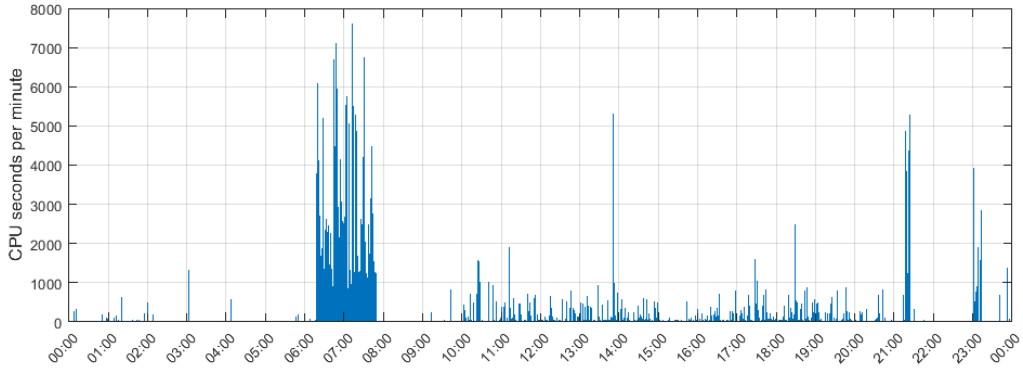


Figure 8: CPU usage profile of one example application.

Even though there is data on real-life applications, we do not utilize the actual applications for the tests as they are. Rather, we create applications for the test scenarios by randomly selecting instances from the pool of actual applications, and varying their resource demands slightly. The selection and modification process allows single applications to be selected multiple times, and any amount of application models can be created with this method, without having multiple identical applications in the set.

For servers, we use multiple different RAM and CPU capacities separately. The servers in the fleet of the organization range from 1 TB to 8 TB, but the most commonly utilized servers have 3 TB or 4 TB of RAM. Here, we utilize 3 TB, 4 TB, and 6 TB servers. For CPU capacities, we consider servers with 64, 128, 192, and 256 virtual cores. The core count is used as the number of simultaneously available calculation threads. In this section, when discussing the number of cores, we always refer to virtual cores. Each combination of the RAM and CPU capacities is separately utilized in the tests. The base application data contains applications that require more CPU capacity that is available with 64, 128, and 196 core servers. Servers with 256 cores and 4 TB or 6 TB of RAM are large enough to host any application. Excessively large applications are left unallocated in the testing scenarios, and their handling in the scenarios varies. In reality, applications that require more computational power than what regular servers can provide, are hosted on more powerful dedicated servers.

An example of resource utilization profile of a server hosting 15 applications is in figure 9. The resource capacities have been normalized so that the maximum capacity is one throughout. The CPU utilization varies significantly and consists mainly of alternatingly long spikes of specific applications. This is a common trend of resource usage on most servers.

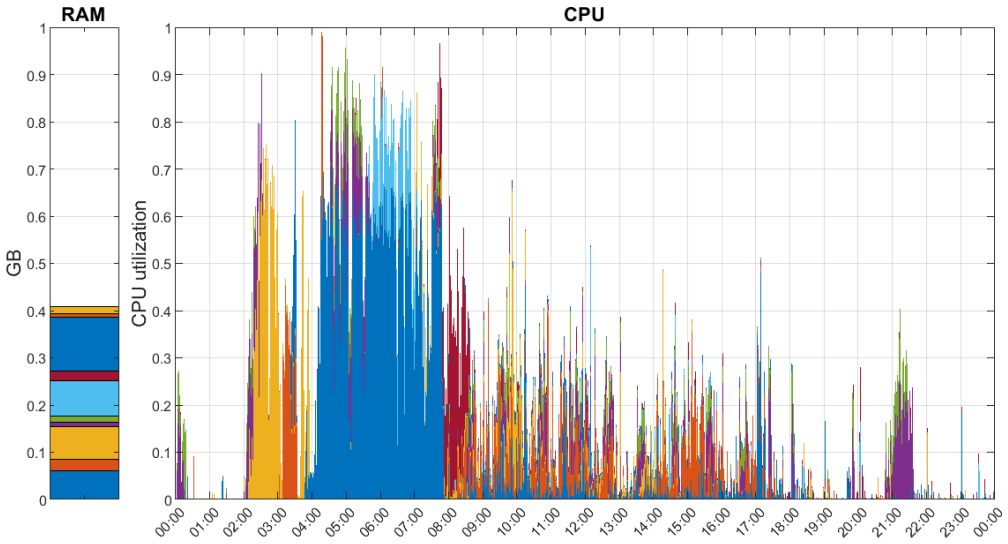


Figure 9: A server hosting 10 applications, with resource requirements of each application cumulatively stacked. The bar on left represents memory utilization, and the temporal graph shows the CPU usage per minute. The color selections that represent individual applications are consistent between the RAM and CPU graphs.

5.2 Methods

The tests in each scenario are performed using the Monte Carlo method. The number of iterations varies between one and one thousand depending on scenario setup and problem instance magnitude.

The heuristic algorithms we evaluate are implemented with MATLAB version R2024b, and the exact solutions are solved and approximated with ready-made MATLAB functions `intlinprog.m` and `pso.m`, respectively. The testing scenarios are executed on the same platform.

All tests are performed with a 10-core Intel i5-12600K processor, with 32 GB of RAM capacity.

5.3 Scenarios

We investigate three separate testing scenarios, and track the performance of the algorithms in each scenario, respectively. Additionally, the results attained with the heuristic algorithms are compared with the global optimal solutions in the first scenario. The results from each scenario are reported in section 5.4.

5.3.1 Allocation to empty servers

In the first scenario, different numbers of applications are allocated to an unlimited set of servers, with the goal of minimizing the number of servers required for hosting the applications. The allocation is performed with all heuristic algorithms and with all combinations of memory capacity and number of cores.

We create the applications to be allocated based on all available real-life applications, including ones with heavy computational requirements. This includes large applications with requirements that cannot be satisfied with the lowest memory and core options. In this scenario, applications that cannot fit on an empty server are left unassigned and thus not included in the allocations. This makes comparing results between different-sized servers non-relevant, as runs with larger servers will contain more successful allocations in total. However, this is not an issue, because the goal is to compare the algorithms within each server category, which remains feasible in this setup.

We perform the allocations starting with 100 applications, and increase the instance size discretely to 100 000 applications to investigate whether the performance of some algorithms are dependent on the instance size. Additionally, we compare the calculation times of the algorithms in this scenario with multiple problem instance sizes.

To further investigate the solution quality of the algorithms, we compare the heuristic results with the exact optimal solutions. The optimal solutions are only calculated for the smallest instance size of 100, as larger instances cannot be solved in a reasonable time. Nevertheless, these results provide insight on how close to the optimum the heuristics can reach.

5.3.2 Allocation of additional applications

In this scenario, we begin with a fleet of servers that host a set of applications. However, additional applications are introduced to the system (e.g. due to organizational growth) and need to be incorporated into the current allocation, while minimizing the number of new servers we need to initialize. Each iteration of the scenario begins with the same server fleet and the same initial allocation. The initial allocation contains 85 servers hosting 699 applications, which have been allocated using the random allocation algorithm introduced in section 4.3.5. Migrating the initially allocated applications between the servers is not allowed. In this scenario, we have strictly 4 TB and 128-core servers.

The number of additional applications is varied in the tests, ranging from 10 to 1000. The allocation is performed with each heuristic for the set of applications with two different approaches. The first approach assumes that all additional applications are allocated at once. This allows us to sort the applications before they are allocated. The second approach assumes that the applications are received one at a time without information of the subsequently incoming applications beforehand. Thus, they cannot be sorted before the allocation. The second approach essentially transforms our BFD, FFD, and WFD algorithms to their standard unsorted versions.

5.3.3 Removing servers from the fleet

The third scenario begins from a similar initial setup as the second scenario, with a fleet of servers hosting a set of applications. In this scenario, a variable number of servers is removed from our fleet due to some unavailability. This could mean predictable occurrences, such as maintenance or cleanups, or unforeseen circumstances like power outages, malfunctions, fires, or acts of terror. The applications that were hosted on the servers removed from the fleet are left without a host, and they need to be reallocated to servers that are still active. The initial allocation contains a fleet of 84 servers hosting 692 applications, allocated with the random algorithm. Similarly to the previous scenario, we utilize only 4 TB and 128-core servers.

In this scenario, we assume no servers are available for initialization, in addition to the ones already included in our fleet. Our goal is to allocate as many of the hostless applications to a running server, to ensure that as few customers as possible suffer system outages for their end users. We begin by removing one server and increase that number until all but one server is left unusable. The performances of the algorithms are compared based on the percentage of hostless applications that were successfully reallocated to the server fleet.

We do not allow migrations between servers in this scenario either, even though this would very likely allow us to allocate a higher percentage of applications on the remaining servers. Migrations always require time and manual labor, and if servers become suddenly unavailable, the hostless applications should be reassigned as fast as possible.

5.4 Results

5.4.1 Allocation to empty servers

The main metric for evaluating the allocation of applications to empty servers from scratch is the number of servers needed to host the set of applications. The number of servers used with each heuristic algorithm for three selected server sizes are reported in table 2. These server sizes were selected to best represent the data. The full data with all memory-core combinations is presented in appendix A. It is worth reiterating that comparing the results between different server capacities is not sensible, since the amount of successfully allocated applications varies significantly between different server sizes. The average allocation success rate with each server size is explained in the caption of table 2.

Table 2: The number of used servers with different algorithms while utilizing servers with three selected capacities. The mean ratio of successful allocations was 29,8% with 3 TB and 64 cores, 69,6% with 4 TB and 128 cores, and 99,8% with 6 TB and 256 cores. The results with all server capacities are shown in appendix A.

Apps	RAM	Cores	FFD	BFD	WFD	Random	NF	Iter
100	**3 TB**	**64**	3,9	3,9	4,0	8,0	5,7	**1000**
1000	**3 TB**	**64**	24,7	24,7	25,4	36,5	52,6	**100**
10000	**3 TB**	**64**	241,0	241,6	243,6	255,6	527,6	**7**
100000	**3 TB**	**64**	2342	2331	2344	2365	5326	**1**
100	**4 TB**	**128**	8,6	8,6	8,8	14,9	16,7	**1000**
1000	**4 TB**	**128**	59,4	59,3	63,0	76,9	162,1	**100**
10000	**4 TB**	**128**	544,0	539,4	582,0	575,4	1623,5	**8**
100000	**4 TB**	**128**	5389	5333	5744	5427	16277	**1**
100	**6 TB**	**256**	9,4	9,4	9,6	17,3	16,5	**1000**
1000	**6 TB**	**256**	77,0	76,7	77,6	94,8	160,1	**100**
10000	**6 TB**	**256**	766,7	766,9	763,9	773,7	1614,4	**7**
100000	**6 TB**	**256**	7687	7690	7590	7562	16154	**1**

The results indicate clearly that the two algorithms that provide the best allocations with nearly all settings are FFD and BFD. These algorithms yield very similar results, with BFD perhaps slightly outperforming FFD with larger problem instances. WFD is not far behind. The number of servers used with WFD is consistently slightly higher than with BFD and FFD, with the exception of large instances with a 6 TB and 256-core server, in which WFD in fact illustrates the best performance out of all algorithms. The random allocation algorithm provides significantly worse results with smaller instances, but interestingly, the gap between its results to the more sophisticated algorithms reduces significantly for large instances. The performance of the NF algorithm seems completely unaffected by the number of applications, as the number of servers used increases linearly while the number of applications increases. These results are further illustrated in figure 10, which presents the mean number of applications allocated to one server with each heuristic.

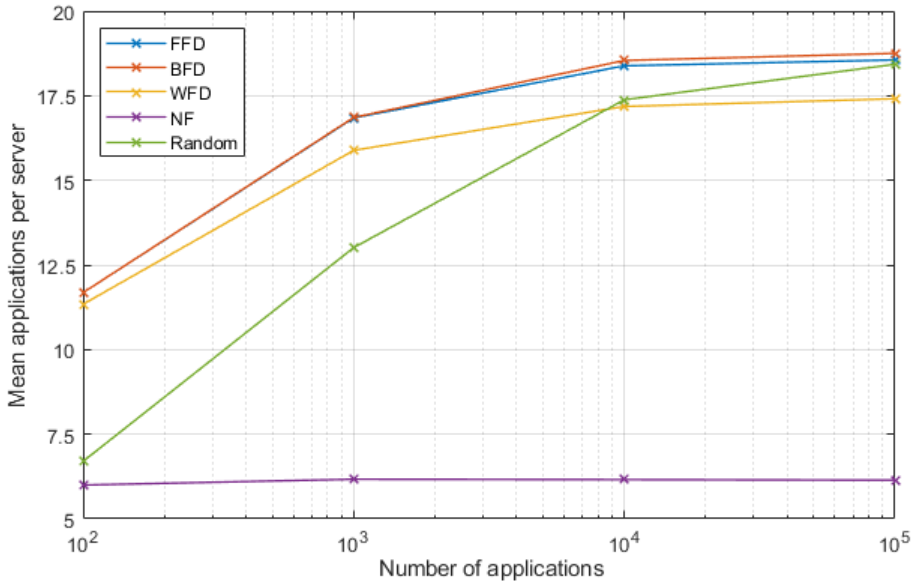


Figure 10: Applications per server with each algorithm on a logarithmic scale, with a 4 TB and 128-core server.

The calculation times of one allocation from scratch with each algorithm, given different sizes of application sets, are presented in figure 11. The figure shows that clear differences in calculation times begin to emerge quite early, and those differences are highlighted as the number of applications increases. Calculation times increase exponentially, with the exception of next-fit, which has a linear increase, and its calculation time is under two seconds even with ten thousand applications. Another algorithm that performs well in comparison is FFD, whose calculation times increase less for larger instances.

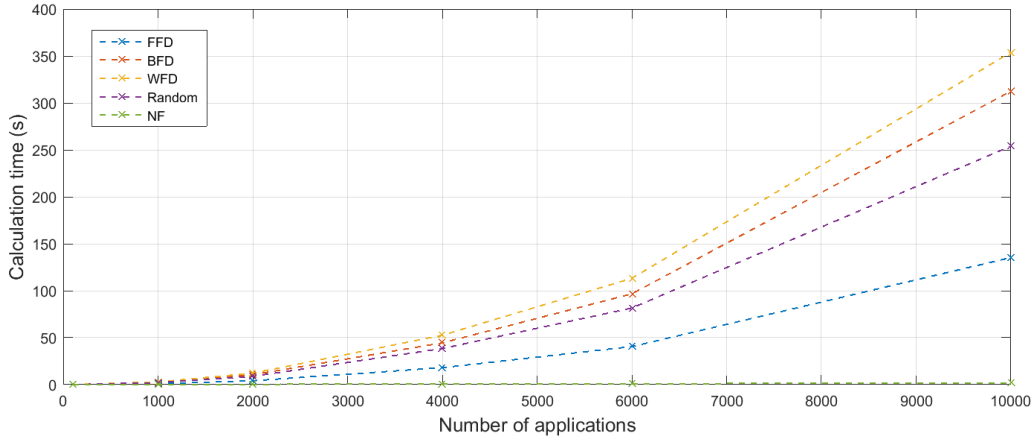


Figure 11: Calculation times with each algorithm with respect to the number of applications allocated. Applications are allocated to empty servers, one iteration.

We solve the allocation problem using HiGHS optimizer to locate the optimal solution, allowing us to estimate the quality of our heuristic solutions compared to global optimal solutions. The optimization proved infeasible even with very few applications when operating in 1441 dimensions. We compress the problem to consider CPU usage with 15-minute time slots, reducing the problem dimensions to 97. With this simplification, the optimizer was generally able to locate optimal solutions fairly quickly with small instance sizes, up to around 30 applications. However, when the number of applications approached 100, the optimizer was rarely able to converge to a global optimum within a cut-off time of two minutes. A summary of these calculations can be seen in table 3. The server fleet size required for hosting the applications is generally higher here than in the previous table due to the dimension reduction, as now the CPU peaks of the applications are generally more long-lasting.

Table 3: The performance of FFD and BFD heuristics compared to the optimal solutions obtained with HiGHS optimizer. A set of 100 applications was allocated with 100 iterations for each server type.

Server capacity (RAM (TB), cores)	**3, 64**	**4, 128**	**6, 192**
Optimum found	100 / 100	21 / 100	4 / 100
Heuristic found optimum	84 / 100	4 / 21	0 / 4
Heuristic required 1 additional server	16 / 100	11 / 21	4 / 4
Mean optimal used servers	4,9	9,7	11,8
Mean used servers, FFD	5,0	11,1	13,0
Mean used servers, BFD	5,0	11,0	12,8
Mean number of successful allocations	29,8	69,6	95,1

With a 64-core server, on average 29,8% of the 100 applications were successfully allocated. Here, the optimizer was able to locate the optimal solution in seconds nearly every time. In these cases, at least one of the heuristics found the optimal solution in 84 out of 100 attempts and required only one additional server in the remaining 16 iterations. Increasing the server capacity to 4 TB and 128 cores, the optimizer found the optimum only 21 times out of 100, due to the larger amount of successful allocations. Out of these successes, the heuristics were able to locate the optimum four times and used one additional server in 11 attempts. With the heuristics, the mean solutions utilized on average 13% more servers than the optimum. With the largest server capacity in these tests, the optimal solution was found within the two-minute cut-off time only four times out of 100, as the average number of successful allocations increased to 95,1. The solutions found with the heuristics were one server away in each of these iterations, while the total mean distance from the optimum was 8% with BFD.

The implemented PSO method was unable to locate feasible solutions near the optimum in any reasonable time frame in nearly all cases. This was most likely due to the immensely large number of local optima in our problem, which causes the algorithm to get overwhelmed with the sea of possibilities. Heavily increasing the swarm size allowed the algorithm to generate better solutions with the expense of

increased runtime, but even then, the solutions proved suboptimal. The only cases in which the PSO algorithm found the optimal solution in an almost feasible manner, was with a server upper limit, m, that was initially within one server of the optimum.

The exact optimizer provided better results than the heuristics as expected, but our other assumption that the PSO metaheuristic could yield near-optimal solutions in a reasonable time proved faulty, as the solutions were in fact far from optimal and run times were often longer than those of the exact optimizer. Thus, we can state at this point that our implementation of the PSO algorithm is not a suitable method for our allocation system.

5.4.2 Allocation of additional applications

In this scenario, we are focused on the number of additional servers needed as new applications need to be allocated on top of the existing set already hosted on our server fleet. We investigate two separate instances: one in which the applications are allocated at once, and another in which the applications appear sequentially one at a time. The average number of additional servers required with each set of applications and with each algorithm is presented in table 4.

Table 4: Additional servers needed with different amounts of new applications to allocate. Each instance size has been allocated both sorted (S) and not sorted (N). Server capacity is 4 TB and 128 cores. Each instance size was iterated 1000 times with each allocation method.

	FFD		BFD		WFD		Random		NF
Applications	**S**	**N**	**S**	**N**	**S**	**N**	**S**	**N**	**N**
10	0,0	0,0	0,0	0,0	0,0	0,1	0,4	0,5	0,1
50	0,3	0,3	0,3	0,3	0,8	1,1	2,6	2,5	4,1
100	1,4	1,4	1,4	1,4	2,1	3,1	5,5	5,4	12,1
200	4,7	4,8	4,7	4,8	5,1	7,9	11,2	11,5	28,3
500	18,3	18,7	18,4	18,8	20,3	24,8	29,4	30,0	77,1
1000	43,7	43,0	44,2	43,8	46,8	54,4	58,8	60,2	158,4

The results show that FFD and BFD provide the best allocations with each application set, similarly to the first scenario. These two algorithms manage to allocate nearly all applications to the current server fleet with 10 and 50 new applications. The expected value of new servers will never be exactly zero even with one application, as there is always a chance that there is a substantially large application among the additional applications, requiring an empty or a nearly empty server. WFD is able to reach allocations that are quite close to FFD and BFD, especially with sorting, but again shows that its performance is slightly worse with all application sets. Random allocation algorithm performs better as the number of applications increases, similarly to the first scenario. However, it cannot compete with the other heuristics. NF provides the worst allocations of the five algorithms. This is to be expected especially in this scenario, as the behavior of the heuristic can cause a large number of servers with some empty space to be discarded when an unsuitable application presents itself.

Interestingly, the sorting seems to have little to no effect on the results in this scenario for FFD and BFD. In fact, the sorting seems to degrade the performance of the algorithms with 1000 additional applications. This shows that the advantage given by the sorting when allocating to empty servers is diminished, when there are smaller spaces where to allocate the incoming applications from the start, as is the case in this scenario.

5.4.3 Removing servers from the fleet

In the scenario, we remove servers from the current fleet and reallocate the applications that are left hostless to the remaining server fleet. The metric we are investigating here is the percentage of applications that were successfully reallocated back to the servers after their original host had been deactivated. These results are presented in table 5 for all algorithms with different numbers of servers removed from the fleet. The number of server removals ranges from one to 83, meaning all but one server became unavailable.

The results show that with all algorithms except NF, on average over 90% of the hostless applications managed to get reallocated up until 20 servers were removed. This means that even if one fourth of the fleet becomes unavailable, a large majority of applications can be kept running even without any rearrangement of the initial allocations. In fact, even if three fourths of our fleet is removed, over 50% of the hostless applications can still be reallocated. This can be achieved as the large majority of our applications do not require significant portions of server resources, while there are some applications that take up the majority of the resources of a server. The largest applications remain without a host nearly without exception in this scenario.

The FFD, BFD, WFD, and Random algorithms show very similar performance in this scenario. The variances and worst-case values in the results also showed no significant differences between these heuristics. The efficient performance of FFD and BFD is not surprising, as they have provided the best allocations in the earlier scenarios as well. WFD yields similar or slightly worse results than FFD and BFD, which is also in line with the previous tests. However, the relative success of the random allocation seems exceptional. This shows that in a scenario in which there are a lot of smaller gaps in servers, finding a specific fit for an application does not seem as relevant. Rather, it is more important to identify some feasible assignment for the application. To achieve better results with the FFD algorithm, one could sort the servers in decreasing order before allocation, starting with those with the most capacity.

Table 5: Expected value of the percentage of applications successfully reallocated after different numbers of servers were removed from the fleet. Initial fleet size is 84 servers. One through 10 server removals were iterated 1000 times, and the rest were iterated 500 times. Server capacity is 4 TB and 128 cores.

Servers removed	FFD	BFD	WFD	Random	NF
1	98,7%	98,7%	98,5%	98,7%	92,1%
2	98,4%	98,4%	98,2%	98,3%	74,3%
3	98,3%	98,3%	97,6%	98,1%	54,7%
4	98,1%	98,1%	97,1%	97,7%	42,2%
5	97,9%	97,9%	96,6%	97,4%	32,9%
6	97,5%	97,6%	96,0%	96,9%	27,4%
7	97,2%	97,3%	95,3%	96,5%	24,2%
8	97,0%	97,1%	95,0%	96,2%	20,7%
9	96,6%	96,8%	94,7%	95,7%	17,2%
10	96,3%	96,5%	94,2%	95,3%	15,9%
15	94,8%	95,0%	92,8%	93,7%	11,0%
20	93,1%	93,2%	91,3%	92,0%	7,4%
25	90,6%	90,7%	89,1%	89,6%	5,2%
30	87,6%	87,9%	86,5%	86,9%	4,2%
35	84,4%	84,6%	83,6%	83,7%	3,1%
40	80,7%	80,9%	80,1%	80,3%	2,4%
45	76,9%	77,0%	76,5%	76,7%	2,0%
50	73,2%	73,4%	73,0%	73,1%	1,6%
55	69,0%	69,1%	68,9%	69,0%	1,2%
60	63,8%	63,9%	63,9%	64,0%	1,0%
65	53,7%	54,1%	54,7%	54,9%	0,8%
70	36,2%	36,1%	36,0%	36,7%	0,5%
75	20,5%	20,3%	20,9%	20,8%	0,3%
80	7,9%	8,1%	7,8%	7,9%	0,1%
83	1,8%	1,8%	1,8%	1,8%	0,0%

5.5 Algorithm selection

The results from the three presented scenarios and the computational efficiency investigated in the first scenario indicate that FFD is the most suitable algorithm. The performance of BFD is very similar to FFD with regard to the performed allocations in each scenario, but its clearly longer running times make it less appealing. Other evaluated heuristics provided subpar results. Thus, we select the first-fit-decreasing algorithm for investigating allocations in a real data center in section 6.

6 Capacity allocation in a private cloud infrastructure

In this section, we apply the first-fit-decreasing algorithm described in section 4.3.1 to the server fleet of one DC of our target organization. We use the existing allocation of applications at the DC as a starting point and examine several reallocation scenarios to investigate the possibility of improving the utilization of servers, and the resilience of the fleet come possible outages.

The server fleet consists of approximately one hundred servers that are located in the same DC. The servers are hosting multiple types of applications, including customer production applications, other applications used by the customers, and several types of internally used applications. We exclude internal applications from our analysis and only consider the customer applications as usability problems with internal applications are not as critical. Internal applications can be considered a separate entity within the DC, as they are hosted on their own servers. Removing them from the analysis essentially just excludes a subset of the server fleet from the investigation. However, we keep in mind that in case of disasters, there is potential room to be utilized in the servers hosting internal applications. If necessary, these servers can be entirely emptied to host customer applications.

From a resource utilization standpoint, the different customer application types are identical. However, production applications are the most critical from a business point-of-view, and outages in other types of applications are more tolerable. Therefore, our priority needs to be in every situation to have the customer production applications hosted at all times, if possible. To simplify the investigation, each customer is considered equally important.

6.1 Initial allocation

In the initial state, we consider 89 servers hosting 520 customer applications. The applications are not at all evenly distributed among the servers, as the number of applications per server varies from one to 26. The total resource load on the servers also has significant variation. The server fleet in the DC is not homogeneous, because there is a large collection of different server models. The servers hosting customer applications range from 2 TB to 4 TB and contain between 88 and 128 physical cores. However, as each server supports hyper-threading (explained in section 2.2), the number of available virtual cores is double the number of physical cores. Thus, the CPU capacity of the servers, which relates to the number of simultaneous calculation tasks that can be performed, is equal to the number of virtual cores. From here on, the number of cores each server has refers to the virtual core count. The most common server type we have contains 4 TB of memory and 256 virtual cores. For safety reasons, and because the operating system and other mandatory programs require memory on the servers, we initially limit the memory usage on each server to a maximum 80% of total capacity.

We know the initial allocation of each application in the DC, as well as the

capacities of each server. The applications are modeled similarly to the previous section, by taking the minute-wise CPU usages from a regular weekday and using those as the requirements for CPU seconds at each minute-long time slot. RAM requirements are static, and they have been manually determined based on historical memory use.

Based on the DC server loads reflecting the application resource requirements and the initial allocation, some servers are being overutilized with respect to their capacity limits. We however know that in reality, the servers are running generally with no issues given the current allocation. Thus, we suspect that there are some inaccuracies in the data. These issues were handled by performing manual modifications to the application requirements for roughly 10 applications. The modifications were slight, mainly including scaling down single suspicious CPU usage peaks; limiting the memory reservation for one application that had over 80% of total memory capacity reserved; and lowering very high CPU usages just after zeros, as that indicates additional computation required for starting an application, which can be considered not part of the general daily CPU usage patterns. With these small modifications, the set of applications respects the capacity limits on all servers.

The applications use CPU throughout the day, but there is clear variation in the total demand for computational load depending on the time of day. The total cumulative CPU requirements of all applications are illustrated in figure 12. The figure clearly indicates that the most demanded time period for computation is roughly between 04:00 and 08:00 (UTC+2) in the morning. This occurs as all of the applications are Europe-based, and the daily recurring calculations are commonly performed during the night when end users do not need to access the application. The afternoon and especially the evening are correspondingly less crowded. The noticeable peaks on many even hours are the result of lighter scheduled runs throughout the day. The total CPU resource utilization is 13% compared to continuous full CPU utilization. This might seem quite low, but the high and short peaks in CPU usage that many applications have, make continuous high utilization rather infeasible. Most servers are utilizing their CPU capacities at high levels at some points during the day, and the

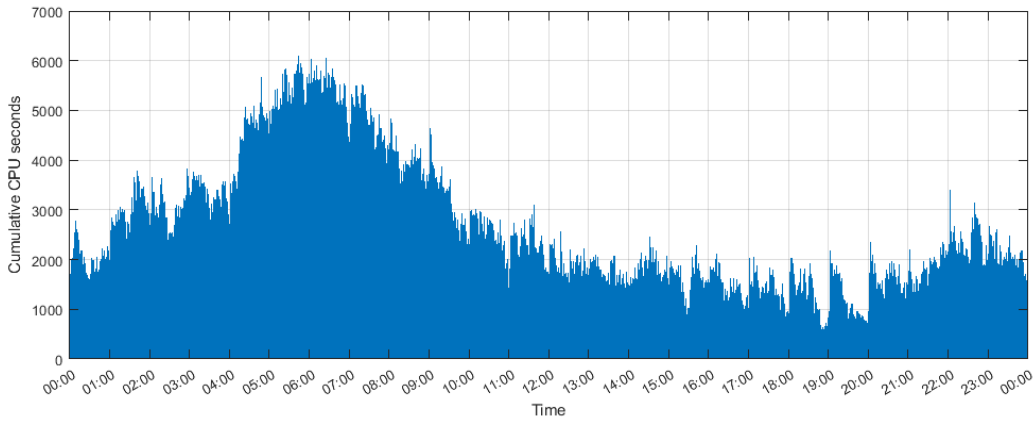
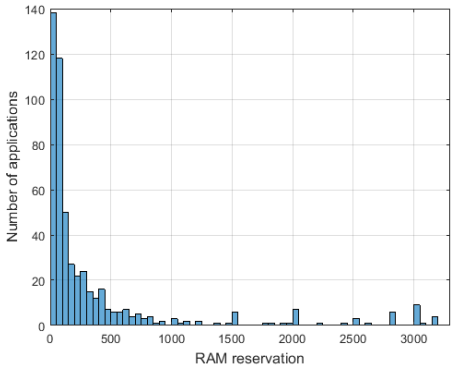


Figure 12: The combined CPU requirements of all 520 applications in the DC.

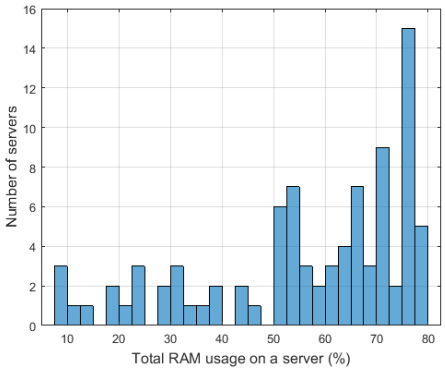
median peak CPU utilization of servers is 83% of maximum capacity.

The memory requirements of applications range from 5 GB to 3,2 TB. As our largest servers have 4 TB of RAM capacity and we only allow 80% of that to be used, this means that the 3,2 TB applications require their own dedicated servers. However, only a handful of applications demand 3 TB of memory or more, while most applications require much less, as depicted in figure 13a, which shows the distribution of RAM reservations for the set of applications. From the histogram, it is clear that the vast majority of applications are on the smaller side of the scale, roughly half requiring 100 GB of memory or less.

The total memory utilization of all servers in the DC is roughly 57% given the initial allocation. This load is not evenly distributed among the 89 servers, as some utilize the full 80% and a few run with less than 10% utilization. The distribution of memory utilization among servers is shown in figure 13b. All five servers that utilize the full 80% host only one large application respectively. Many servers with low memory utilization host applications with heavier computational needs. Thus, even though the memory utilization is low, the servers are not necessarily totally underutilized.



(a) Memory requirements in GB of all 520 applications in the DC.



(b) Memory utilization of each server in the DC.

Figure 13: Memory requirements of applications and the memory utilization on each server in the DC presented as histograms.

6.2 Optimal allocation

Next, we perform a reallocation for the set of 520 applications with the available servers in the DC, using the FFD allocation heuristic presented in section 4.3.1.

This allocation is performed as in previous sections, except that the servers are heterogeneous. Our goal is again to minimize the number of servers, regardless of server type. Thus, we assume that the cost savings gained from reducing the server fleet by one is equal for all server types. With this assumption, it is most beneficial for us to allocate applications to the largest servers first, before initializing servers with less capacity. We achieve this by sorting the servers in decreasing order based on

their resource capacities before commencing the allocation procedure. The sorting is conducted primarily based on memory capacity and secondarily based on CPU capacity.

Reallocation with FFD yields a significant reduction in the number of required servers. We manage to host the 520 applications on 60 servers, cutting fleet size by almost a third, while respecting the limit that we utilize 80% of memory capacity at most. In the achieved allocation, we only utilize the largest server type with 4 TB and 256 cores, as we have 69 of those servers available. The total memory utilization with 60 servers is 79,8%, and the total memory requirement of the applications is only some 500 GB less than 80% of the total memory capacity of the servers. This means that decreasing fleet size from 60 with the current memory requirements cannot be achieved. Thus, the result would be the same even if we ignore the CPU capacity limits. However, if we relax the 80% memory limit to 95%, we can further reduce fleet size to 51. The relevant metrics of the server fleet after reallocation are presented in table 6, with a comparison to the initial situation.

Table 6: Server fleet metrics after reallocation with FFD compared to the initial situation.

	Initial allocation	**Reallocation**
Number of applications	520	520
Active server fleet size	89	60
Server memory capacities	2 TB - 4 TB	4 TB
Server number of virtual cores	176 - 256	256
Total RAM utilization	57,2%	79,8%
Total CPU utilization	12,5%	17,5%
Peak CPU utilization	28,7%	40,2%
Median server peak CPU utilization	83,2%	96,2%
Mean server peak CPU utilization	78,5%	85,4%
Servers hosting only one application	32	4

The allocations of individual servers have significant variations, even though the memory usage is similar on all servers. Many larger applications that are allocated first are hosted on their own servers, or they have some smaller applications hosted with them. Another common allocation pattern contains a few moderately large applications, along with a variety of small applications. Examples of servers with these types of allocations are in figure 14, with further examples shown in appendix B. Perhaps the most noticeable change in the single server assignments, is that only four applications remain hosted on dedicated servers after reallocation, whereas the number was 32 originally. This suggests that the decision to provide a dedicated server for a customer application is performed overly liberally, from a resource utilization perspective.

The total CPU utilization of the fleet remains low, only reaching 18%. However, the median peak utilization of individual servers now eclipses 96%, indicating that the majority of servers are achieving high utilization during some time periods. There are

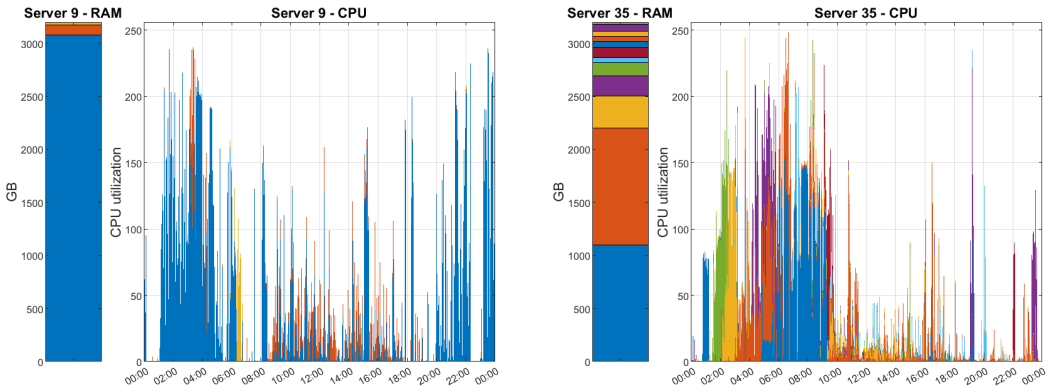


Figure 14: Two example servers illustrating common resource allocations. Server 9 is hosting three applications and server 35 is hosting 15 applications.

however some servers in which the CPU utilization is very low throughout. This is caused by applications with large memory requirements, but very limited CPU usage.

6.3 Disaster recovery

We investigate a situation in which one or more servers become unusable due to some disturbance. The disturbance can be known in advance, such as a shutdown for maintenance, or it may be unpredictable, such as a power outage. We begin with removing one server from the fleet and move on to shutting down full racks of servers. The goal in each case is to reallocate the newly hostless applications to some of the still active servers without conducting any migrations within the undisturbed servers.

We begin each scenario with the initial allocation as described in section 6.1. If possible, we aim to respect the 80% memory utilization cap. However, if the reallocation is not possible with the limit, we relax the limit to 95% at most. Additionally, if we cannot allocate all hostless applications, we can attempt to allocate only the production applications, as their availability is more important than for the other applications.

6.3.1 One server disabled

We remove each one of the 89 servers separately and attempt to reallocate the applications that were hosted on the removed server. We perform four allocation rounds with each server, each with different limitations. First, we respect the original 80% memory limit in the allocations. Second, we relax the limit to 90% and attempt to allocate again. Third, we further raise the memory limit to 95% and allocate. Last, we maintain the 95% memory limit, but this time we only allocate production applications and ignore the rest. As we were able to allocate the set of applications to 29 fewer servers than the initial allocation in the previous section, we can hypothesize that removing one server from the fleet should not cause any applications to remain without a host.

The test results, also summarized in table 7, show that while respecting the 80% memory limit, all applications previously hosted on the removed servers can be

reallocated in 73 out of 89 cases. This means that there were 16 servers, in which we are unable to find enough room for the hosted applications. Interestingly, in each of the 16 unsuccessful reallocations, the server removed from the fleet hosted only one application. This clearly indicates that these applications would require the majority of the memory capacity of one server, and there is not enough room on any server to accommodate their needs. The number of unsuccessful reallocations reduces as we relax the memory limit. With 95% of memory in use, we can successfully reallocate all but one application. This application requires 3,2 TB of memory, and a very high portion of the CPU capacity throughout the day. It cannot be run unless it has a dedicated server.

Table 7: Results from reallocating hostless applications after separately disabling one of the 89 servers at a time. A success means that all applications were reallocated successfully.

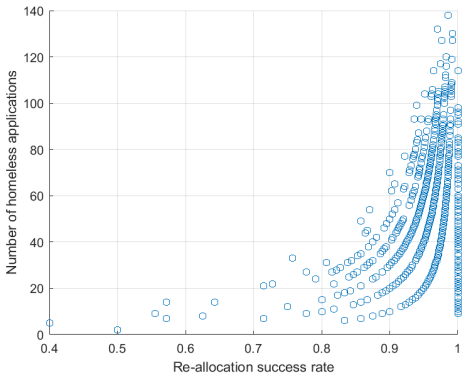
Limitation	Successes	Total hostless applications
80% memory	73 / 89	16
90% memory	83 / 89	6
95% memory	88 / 89	1
95% memory, only production	88 / 89	1

6.3.2 One rack disabled

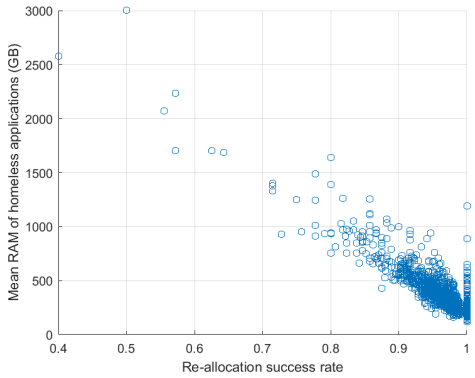
It is possible that a rack full of servers becomes unusable at once, given some malfunctions or issues with the rack itself. In this case, all servers within that rack are taken out of the fleet, and the applications hosted on the servers need to be reallocated. In our model, the DC has 10 racks, each containing on average nine servers. The servers, however, may not be equally distributed among the racks. We simulate the rack distribution here by randomly generating a rack assignment, in which each server has an equal chance of being located on any rack. However, we restrict the number of servers on one rack to 25 due to physical limitations.

We generate 100 separate rack assignments, and with each of them, we separately remove each rack one at a time. Then, we attempt to reallocate the applications previously hosted on the servers belonging to the rack. To track the success of the reallocations, we measure the number of successful allocations compared to the total number of hostless applications in each iteration. Additionally, we maintain other relevant indicators, such as the size of hostless applications, which proved very impactful in the previous tests.

The success rate of reallocations compared to the total number of hostless applications and the mean memory requirement of hostless applications are shown in figure 15. These figures indicate that the factor lowering the success rate of reallocations is not the number of hostless applications, but their size. In fact, figure 15a clearly shows that the more applications there are, the more likely it is that most of them can be successfully allocated. This means that single large applications are again the ones that cause difficulties in reallocations.



(a) Reallocation success rate with different amounts of hostless applications.



(b) The effect of average application memory requirement on reallocation success rate.

Figure 15: After one rack of servers is disabled, we attempt to reallocate the hostless applications to other servers that are still running. The success rate of reallocations reflecting the share of hostless applications that found a new host with the 80% memory cap is shown.

Furthermore, we investigate the specific individual applications that have been left hostless in these tests. Out of the 520 applications, only 41 remained hostless even once when performing 100 iterations of each 10 racks separately removed from the fleet. Furthermore, 16 applications were never reallocated. These difficult-to-allocate applications are presented in figure 16, which illustrates the relation of large memory requirement with the number of failed reallocations for these applications.

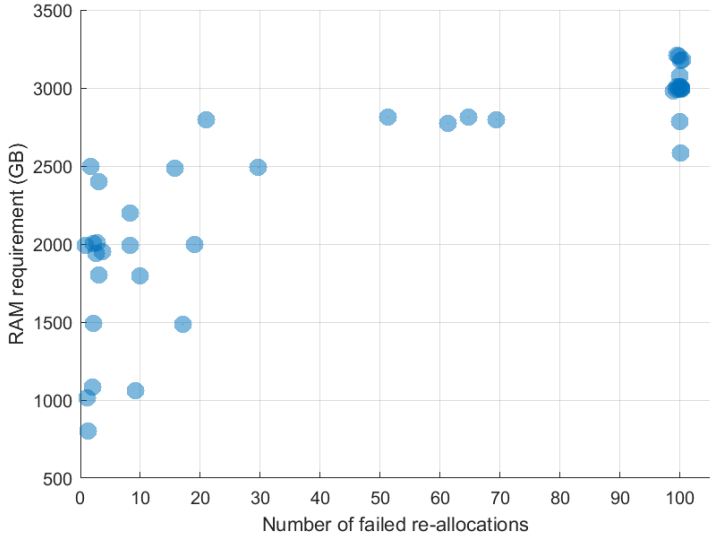


Figure 16: The effect of memory requirement on the difficulty of reallocating an application is illustrated. Each point represents a distinct application. The points include slight jitter for better visualization.

6.3.3 Multiple racks disabled

If the DC faces larger disturbances, multiple racks may become unavailable simultaneously. Additionally, it is common that two server racks utilize several shared components necessary for the functioning of the racks. If these components are damaged, both racks may shut down. Major disruptions, such as fires or power outages, can also make multiple or even all racks unusable. We therefore investigate how to reallocate the hostless applications efficiently in scenarios in which multiple server racks are disabled.

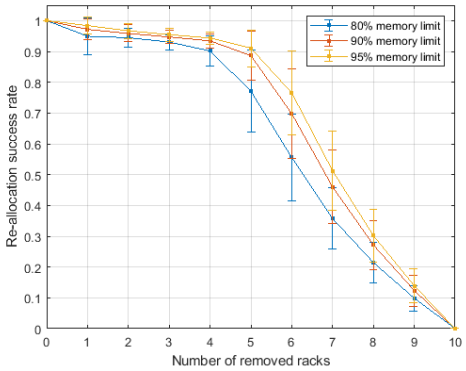
We utilize 10 server racks in the DC, and similarly to the previous scenario, the servers are randomly assigned to the racks. The number of servers on each rack is between zero and 25. The initial allocation of the applications is as described in section 6.1. We simulate the disabling of racks by generating 1000 random rack assignments for each number of server removals and attempting to allocate the hostless applications to the remaining servers. The main results from these iterations, including the mean success rate of reallocation, the mean number of unsuccessful allocations, and minimum and maximum values of these metrics from the iterations, are presented in table 8. As expected, the success rate of reallocations decreases as the number of disabled racks grows. Interestingly though, the decline in mean success rate is very gentle from one to four racks. This again relates to the large applications, because when there are fewer applications to allocate, the success rate decreases drastically. Another notable finding is the high maximum success rates with the majority of racks being disabled. For instance, if we disable seven out of our 10 racks, there is still a possibility that we can reallocate 85% of the hostless applications. The differences between the iterations are however distinctively high, as depicted by the large standard deviations across the board, and especially between five and seven disabled racks.

The success rates and the number of unsuccessful allocations are illustrated in figures 17a and 17b. The figures include the standard deviations of the mean values as

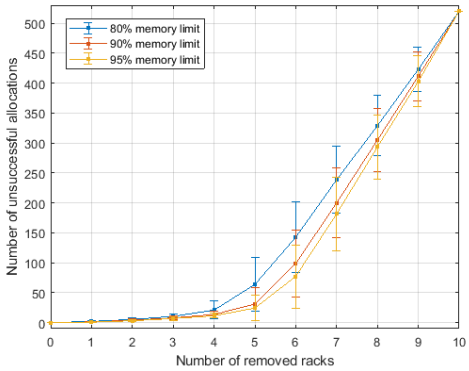
Table 8: Mean reallocation success rate and mean number of hostless applications after reallocation with different numbers of disabled racks and their respective minimum and maximum values. The results were obtained with 1000 iterations for each number of disabled racks.

Racks removed	Success rate	Std dev	Max	Min	Hostless	Max	Min
1	94,9%	5,8%	100%	0%	2,0	12	0
2	94,5%	3,0%	100%	68,0%	5,3	16	0
3	93,0%	2,4%	98,7%	82,8%	10,4	23	2
4	90,2%	4,9%	97,1%	52,4%	20,9	158	6
5	77,1%	13,3%	95,3%	27,3%	63,8	274	10
6	55,6%	14,0%	91,8%	21,5%	142,7	333	18
7	35,7%	10,0%	85,1%	14,0%	238,5	387	36
8	21,4%	6,7%	51,1%	6,6%	328,8	471	150
9	9,9%	4,1%	31,0%	0,8%	423,0	515	276

the error bars and the comparison between the standard 80% memory limitation and the less restrictive limits of 90% and 95%. The difference between the different memory limits is clearly noticeable throughout. The distinction becomes more significant after four racks, as the success rate with an 80% limit remains quite stable until the four-rack mark, while the more relaxed limits hold the higher rates still at five disabled racks. The standard deviations are also significantly larger around five to eight racks. Thus, the means do not with a high probability represent the actual success rates if a real-life disturbance disables roughly half of the fleet.



(a) Reallocation success rate with three different memory utilization limits.



(b) The number of applications unsuccessfully reallocated with the different memory limits.

Figure 17: Results from attempting to reallocate applications back to servers as an increasing number of racks are taken out of use. The error bars represent one standard deviation in both directions.

In conclusion, these tests indicate that some applications require special attention due to their large resource requirements. Locating these applications and the servers hosting them is crucial for proper capacity management and disaster recovery. If we can locate the difficulties and bottlenecks in our system, we can increase the resilience of the system, e.g. by deviating the difficult servers to different racks, or perhaps by attempting to reduce the memory requirements of certain applications with more frequent database cleanups. The removal of several racks furthermore paints a rather positive picture of the persistency of our server fleet. Some racks can be lost with most likely a small fraction of applications becoming unavailable. Additionally, because there are internal servers available in the DC, one may be able to generate the required resources on those servers, even if the large applications require their own dedicated servers.

7 Conclusions

In this thesis, we investigated the resource allocation process of a private cloud environment. We modeled the system with the vector bin packing problem with the objective of minimizing the number of required servers and implemented five heuristics for solving the problem. The performance of the heuristics was thoroughly evaluated in several scenarios simulating real-life settings, and the most suitable algorithm, first-fit-decreasing (FFD), was selected and applied to a real DC of the target organization.

The formulated resource allocation problem proved extremely large and complex, even with rather limited instance sizes and static allocations. Locating exact optimal solutions became infeasible quickly as the number of applications increased, even with significant simplifications made to the system. The computational difficulty of the problem was known from the literature, as such problems have been proven NP-hard at minimum, and essentially all papers on DC resource allocation utilize heuristic methods for solving the problem. The problem is further complicated by including the dynamic movement of applications and additional resource restrictions.

The modeling of CPU utilization of applications was eventually performed with one-minute time intervals. The decision to narrow down the intervals from the initial one hour was necessary due to the major overlapping of CPU demand peaks during rush hours, which would have caused excessive CPU reservations for many applications, thus yielding lackluster core utilization. A downside of this decision came from the decreased sensitivity of the model, as with tighter resource allocation, smaller changes in the actual CPU usage can cause runs to overlap and to slow down. Additionally, the added resolution further complicated the system, yielding longer calculation times. To address this issue, the time intervals could be lengthened during off hours while maintaining the desired peak-hour accuracy.

Some of the implemented heuristics proved to be very usable in the evaluations in section 5. FFD and BFD algorithms performed best, providing competitive allocations in all investigated scenarios and in comparison with the exact results received with the HiGHS optimizer given the simplified problem instances. With regard to calculation times, the performance of the heuristics that scan through the entire active server fleet before performing the allocation degraded in larger problem instances. Due to its superior computational efficiency, FFD was chosen over BFD to be applied to the realistic server fleet in section 6.

We were able to improve the fleet utilization significantly with simple heuristic allocation in section 6.2, reducing the fleet size by almost a third in comparison to the current prevailing allocation in the DC. With the reallocation, nearly the entire memory capacity of the active server fleet was utilized, indicating that computational power is the less sought-after resource and has less weight when determining the allocations. The most significant change in the allocation regarded the number of servers that host only one application, which decreased from 32 to four with the reallocation. There is no binding reason for this many applications to demand their individual servers, even if they utilize the majority of available resources on the server. This finding suggests that dedicated servers are perhaps excessively liberally allocated

to customers to reduce the risk of resource overutilization. Balancing between this risk and the cost reduction received from server fleet minimization requires careful consideration when conducting the allocation decisions.

The server fleet of the DC demonstrated surprisingly effective resilience toward outages in individual servers and full server racks. We were able to reallocate on average over 90% of the hostless applications back to active servers if up to 40% of the fleet was disabled due to some malfunction. The reallocation success rate also improved clearly once we relaxed the 80% memory utilization limit we generally have on the servers. The most difficult applications to reallocate were the heavy memory consumers, which accounted for nearly all failed reallocations when one rack was disabled. This comes as no surprise, as some of these applications require the large majority, or in some cases, the entire, RAM capacity of one server, and these memory shares are rarely available on active servers. To mitigate the damages from many large applications becoming hostless, servers hosting these large applications should be identified and placed rather evenly among available racks. This reduces the number of difficult applications requiring reallocation when individual racks become unavailable.

While the results point to massive room for improvement in the utilization of server resources, the simplifications in the application modeling must be considered before celebrating any major successes in fleet size reduction. The CPU utilization requirements of the applications were approximated based on data from a single date, which is unlikely to represent the actual demand for computational power at all times. There is slight unpredictable variation in CPU usage every day, and there can be larger spikes at unexpected times for many reasons. If the server resources are more tightly allocated, deviations from predicted use have a higher probability of causing service disruptions for the customers. Nevertheless, the scale of the evidenced server fleet reduction clearly indicates that improvements to resource allocation are achievable with systematic application placement within the data center.

7.1 Future research

This thesis leaves room for further improvement and expansion in many directions regarding the capacity management of the private cloud infrastructure. The most evident future research topic concerns the CPU requirement modeling for the applications. In this thesis, the CPU profile was modeled based on a single date, but this can easily be extended to include a weekend, which has a different usage pattern, or a full week of data. The data range can be further extended to a significantly longer time period, and a dynamic CPU requirement forecast can be calculated based on the data, including possible seasonal variation and trend regressors.

Working with increased amounts of data in order to extend the model call for more scalable and automated methods for gathering and handling the data. Even with the CPU data used for the model in this thesis, plenty of manual labor was required to handle the outliers and gaps in the dataset. The reasons behind the data exceptions demand further investigation, in case the data is taken into regular use.

An additional point of interest concerning the application resource demands arises from the correlation between RAM and CPU utilization. For most cases, the

correlation between memory and computation is positive, but there is a significant minority of applications that demand an extensive amount of CPU power although having miniscule memory reservations. The reason behind this phenomenon is not evident. Additionally, cases in which the CPU need increases heavily while RAM use remains constant have been noticed. Pinpointing the drivers of this behavior possibly allows us to mitigate the increase or at least prepare for it in advance.

We have assumed strict capacity limits in all resource dimensions in this thesis. This is mostly sensible, as the capacities cannot be exceeded due to physical hardware limitations. If a CPU interval is overallocated, the system will not completely fail, but the execution of active runs is prolonged. This can be perilous at times, as delayed scheduled runs can cause customers to miss business-critical deadlines. However, a slight deceleration in short but heavy runs during off-peak hours often has little to no adverse effects for the customer users. These short spikes in CPU demand can still prevent applications from being hosted on the same server if the spikes are directly aligned. Locating such scenarios in which capacity breaches are less critical or can be avoided with slight run schedule modifications, can be used for further improving the resource consolidation of our fleet.

In the empirical sections of this thesis, we did not consider the possibility of migrating applications between servers during the allocation processes to simplify the investigation. Currently, the process of migrating an application from one server to another is rather laborious and slow, and it is avoided whenever possible. However, investigating how migrations can help reduce the fleet size is valuable in determining roadmaps for future capacity management process development. If the relative cost of one migration compared to the expenses of server acquisition and maintenance can be calculated, it is possible to measure the added value that can be created with seamless, or at least more elastic, migrations.

The privacy and security given by the private cloud infrastructure comes with the downside of reduced scalability because computational resources cannot be extended simply with a higher subscription fee. Combining the resource allocation model with the growth forecast of the organization, predictions can be generated on the future demand for new servers, racks, and additional DCs. These predictions can assist the organization in scaling the computational resources proactively and systematically, rather than reacting to expansions as they come. Further research is therefore required to build reliable resource demand forecasts.

References

[1] D. C. Marinescu, *Cloud Computing: Theory and Practice*. Cambridge, MA: Morgan Kaufmann, 2. ed., 2018.

[2] B. Jennings and R. Stadler, "Resource Management in Clouds: Survey and Research Challenges," *Journal of Network and Systems Management*, vol. 23, pp. 567–619, July 2015.

[3] A. Hameed, A. Khoshkbarforoushha, R. Ranjan, P. P. Jayaraman, J. Kolodziej, P. Balaji, S. Zeadally, Q. M. Malluhi, N. Tziritas, A. Vishnu, S. U. Khan, and A. Zomaya, "A Survey and Taxonomy on Energy Efficient Resource Allocation Techniques for Cloud Computing Systems," *Computing*, vol. 98, pp. 751–774, July 2016.

[4] G. Scheithauer, *Introduction to Cutting and Packing Optimization: Problems, Modeling Approaches, Solution Methods*. Springer, Oct. 2017.

[5] S. Sulyman, "Client-Server Model," *IOSR Journal of Computer Engineering*, vol. 16, pp. 57–71, Jan. 2014.

[6] N. Jain and S. Choudhary, "Overview of Virtualization in Cloud Computing," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pp. 1–4, Mar. 2016.

[7] A. Goel, *Computer Fundamentals*. Pearson Education India, Sept. 2010.

[8] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, Feb. 2002.

[9] G. B. Mathews, "On the Partition of Numbers," *Proceedings of the London Mathematical Society*, vol. s1-28, pp. 486–490, Nov. 1896.

[10] J. S. K. Ang, C. Cao, and H.-Q. Ye, "Model and Algorithms for Multi-Period Sea Cargo Mix Problem," *European Journal of Operational Research*, vol. 180, pp. 1381–1393, Aug. 2007.

[11] S. Zhou, X. Li, K. Zhang, and N. Du, "Two-Dimensional Knapsack-Block Packing Problem," *Applied Mathematical Modelling*, vol. 73, pp. 1–18, Sept. 2019.

[12] H. Shachnai and T. Tamir, "On Two Class-Constrained Versions of the Multiple Knapsack Problem," *Algorithmica*, vol. 29, pp. 442–467, Mar. 2001.

[13] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin: Springer, 2004.

[14] M. Assi and R. A. Haraty, "A Survey of the Knapsack Problem," in *2018 International Arab Conference on Information Technology (ACIT)*, pp. 1–6, Nov. 2018.

[15] V. Cacchiani, M. Iori, A. Locatelli, and S. Martello, "Knapsack Problems — an Overview of Recent Advances. Part II: Multiple, Multidimensional, and Quadratic Knapsack Problems," *Computers & Operations Research*, vol. 143, p. 105693, July 2022.

[16] D. Pisinger, "Where Are the Hard Knapsack Problems?," *Computers & Operations Research*, vol. 32, pp. 2271–2284, Sept. 2005.

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco CA: Freeman, 1979.

[18] S. Martello and P. Toth, "Algorithms for Knapsack Problems," in *North-Holland Mathematics Studies* (S. Martello, G. Laporte, M. Minoux, and C. Ribeiro, eds.), vol. 132 of *Surveys in Combinatorial Optimization*, pp. 213–257, North-Holland, Jan. 1987.

[19] J. Zhang, "Comparative Study of Several Intelligent Algorithms for Knapsack Problem," *Procedia Environmental Sciences*, vol. 11, pp. 163–168, 2011.

[20] G. Wäscher, H. Haußner, and H. Schumann, "An Improved Typology of Cutting and Packing Problems," *European Journal of Operational Research*, vol. 183, pp. 1109–1130, Dec. 2007.

[21] A. Lodi, S. Martello, and M. Monaci, "Two-Dimensional Packing Problems: A Survey," *European Journal of Operational Research*, vol. 141, pp. 241–252, Sept. 2002.

[22] C. Chekuri and S. Khanna, "On Multidimensional Packing Problems," *SIAM Journal on Computing*, vol. 33, pp. 837–851, Jan. 2004.

[23] F. Gzara, S. Elhedhli, and B. C. Yildiz, "The Pallet Loading Problem: Three-Dimensional Bin Packing with Practical Constraints," *European Journal of Operational Research*, vol. 287, pp. 1062–1074, Dec. 2020.

[24] C. Paquay, M. Schyns, and S. Limbourg, "A Mixed Integer Programming Formulation for the Three-Dimensional Bin Packing Problem Deriving from an Air Cargo Application," *International Transactions in Operational Research*, vol. 23, no. 1-2, pp. 187–213, 2016.

[25] U. Elııyı and D. T. Elııyı, "Applications of Bin Packing Models Through the Supply Chain," *International Journal of Business and Management Studies*, vol. 1, pp. 11–19, June 2009.

[26] C. Bassem and A. Bestavros, "Multi-Capacity Bin Packing with Dependent Items and its Application to the Packing of Brokered Workloads in Virtualized Environments," *Future Generation Computer Systems*, vol. 72, pp. 129–144, July 2017.

[27] S. Ali, A. G. Ramos, M. A. Carravilla, and J. F. Oliveira, "On-Line Three-Dimensional Packing Problems: A Review of Off-Line and on-Line Solution Approaches," *Computers & Industrial Engineering*, vol. 168, pp. 108–122, June 2022.

[28] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, "Approximation and Online Algorithms for Multidimensional Bin Packing: A Survey," *Computer Science Review*, vol. 24, pp. 63–79, May 2017.

[29] N. Bansal, M. Eliáš, and A. Khan, "Improved Approximation for Vector Bin Packing," in *Proceedings of the 2016 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Proceedings, pp. 1561–1579, Society for Industrial and Applied Mathematics, Dec. 2015.

[30] A. Caprara, H. Kellerer, and U. Pferschy, "Approximation Schemes for Ordered Vector Packing Problems," *Naval Research Logistics (NRL)*, vol. 50, no. 1, pp. 58–69, 2003.

[31] M. Delorme, M. Iori, and S. Martello, "Bin Packing and Cutting Stock Problems: Mathematical Models and Exact Algorithms," *European Journal of Operational Research*, vol. 255, pp. 1–20, Nov. 2016.

[32] E. J. A. Virmavirta, *Optimization Models for Allocating Memory Capacity in Server Applications*. B.Sc. thesis, Aalto University, Espoo, Oct. 2023.

[33] P. C. Gilmore and R. E. Gomory, "A Linear Programming Approach to the Cutting-Stock Problem," *Operations Research*, vol. 9, no. 6, pp. 849–859, 1961.

[34] E. Malaguti, R. Medina Durán, and P. Toth, "Approaches to Real World Two-Dimensional Cutting Problems," *Omega*, vol. 47, pp. 99–115, Sept. 2014.

[35] J. Błażewicz, K. H. Ecker, E. Pesch, G. Schmidt, M. Sterna, and J. Węglarz, *Handbook on Scheduling: From Theory to Practice*. International Handbooks on Information Systems, Cham, Switzerland: Springer, 2. ed., 2019.

[36] K. E. Stecke, "Design, Planning, Scheduling, and Control Problems of Flexible Manufacturing Systems," *Annals of Operations Research*, vol. 3, pp. 1–12, Jan. 1985.

[37] M. Ji, J. Fang, W. Zhang, L. Liao, T. Cheng, and Y. Tan, "Logistics Scheduling to Minimize the Sum of Total Weighted Inventory Cost and Transport Cost," *Computers & Industrial Engineering*, vol. 120, pp. 206–215, June 2018.

[38] W. Herroelen and R. Leus, "Project Scheduling Under Uncertainty: Survey and Research Potentials," *European Journal of Operational Research*, vol. 165, pp. 289–306, Sept. 2005.

[39] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, "Sequencing and Scheduling: Algorithms and Complexity," in *Handbooks in Operations Research and Management Science*, vol. 4 of *Logistics of Production and Inventory*, pp. 445–522, Elsevier, Jan. 1993.

[40] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, "United States Data Center Energy Usage Report," Tech. Rep. LBNL–1005775, 1372902, Lawrence Berkeley National Laboratory, Dec. 2024.

[41] J. Zong, Y. Peng, B. Zhang, and X. Yin, "Research on Energy Consumption in Data Centers," in *2024 3rd International Conference on Energy, Power and Electrical Technology (ICEPET)*, pp. 1732–1738, May 2024.

[42] C. Jin, X. Bai, C. Yang, W. Mao, and X. Xu, "A Review of Power Consumption Models of Servers in Data Centers," *Applied Energy*, vol. 265, p. 114806, May 2020.

[43] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, pp. 68–73, Dec. 2008.

[44] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing Energy and Server Resources in Hosting Centers," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, (Banff Alberta Canada), pp. 103–116, ACM, Oct. 2001.

[45] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano-SLA Based Management of a Computing Utility," in *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*, pp. 855–868, May 2001.

[46] A. Roytman, A. Kansal, S. Govindan, J. Liu, and S. Nath, "PACMan: Performance Aware Virtual Machine Consolidation," in *10th International Conference on Autonomic Computing (ICAC 13)*, pp. 83–94, 2013.

[47] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 3. ed., Aug. 2022.

[48] M. Kumar, S. Sharma, A. Goel, and S. Singh, "A Comprehensive Survey for Scheduling Techniques in Cloud Computing," *Journal of Network and Computer Applications*, vol. 143, pp. 1–33, Oct. 2019.

[49] K. L. Krause, V. Y. Shen, and H. D. Schwetman, "Analysis of Several Task-Scheduling Algorithms for a Model of Multiprogramming Computer Systems," *Journal of the ACM*, vol. 22, pp. 522–550, Oct. 1975.

[50] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, Mar. 2002.

[51] I. Mahmood, S. Zeebaree, H. Shukur, K. Jacksi, H. Yasin, A. Radie, and Z. Najat, "Task Scheduling Algorithms in Cloud Computing: A Review," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, pp. 1041–1053, Apr. 2021.

[52] R. Ghafari, F. H. Kabutarkhani, and N. Mansouri, "Task Scheduling Algorithms for Energy Optimization in Cloud Environment: A Comprehensive Review," *Cluster Computing*, vol. 25, pp. 1035–1093, Apr. 2022.

[53] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris, *Fundamentals of Queueing Theory*. John Wiley & Sons, 5. ed., Apr. 2018.

[54] K. J. Åström and R. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2. ed., Feb. 2021.

[55] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam, "Managing Server Energy and Operational Costs in Hosting Centers," in *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, (Banff Alberta Canada), pp. 303–314, ACM, June 2005.

[56] Y. C. Lee and A. Y. Zomaya, "Energy Efficient Utilization of Resources in Cloud Computing Systems," *The Journal of Supercomputing*, vol. 60, pp. 268–280, May 2012.

[57] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic Placement for Clustered Web Applications," in *Proceedings of the 15th International Conference on World Wide Web*, (Edinburgh Scotland), pp. 595–604, ACM, May 2006.

[58] F. Semchedine, L. Bouallouche-Medjkoune, and D. Aïssani, "Task Assignment Policies in Distributed Server Systems: A Survey," *Journal of Network and Computer Applications*, vol. 34, pp. 1123–1130, July 2011.

[59] T. Prem Jacob and K. Pradeep, "A Multi-objective Optimal Task Scheduling in Cloud Environment Using Cuckoo Particle Swarm Optimization," *Wireless Personal Communications*, vol. 109, pp. 315–331, Nov. 2019.

[60] X. Wei, "Task Scheduling Optimization Strategy Using Improved Ant Colony Optimization Algorithm in Cloud Computing," *Journal of Ambient Intelligence and Humanized Computing*, Oct. 2020.

[61] Z. Cui, T. Zhao, L. Wu, A. K. Qin, and J. Li, "Multi-Objective Cloud Task Scheduling Optimization Based on Evolutionary Multi-Factor Algorithm," *IEEE Transactions on Cloud Computing*, vol. 11, pp. 3685–3699, Oct. 2023.

[62] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[63] W. Wang, B. Liang, and B. Li, "Multi-Resource Fair Allocation in Heterogeneous Cloud Computing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 2822–2835, Oct. 2015.

[64] J. Khamse-Ashari, I. Lambadaris, G. Kesidis, B. Urgaonkar, and Y. Zhao, "An Efficient and Fair Multi-Resource Allocation Mechanism for Heterogeneous Servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 2686–2699, Dec. 2018.

[65] M. Masdari, S. S. Nabavi, and V. Ahmadi, "An Overview of Virtual Machine Placement Schemes in Cloud Computing," *Journal of Network and Computer Applications*, vol. 66, pp. 106–127, May 2016.

[66] M. Mishra and A. Sahoo, "On Theory of VM Placement: Anomalies in Existing Methodologies and Their Mitigation Using a Novel Vector Based Approach," in *2011 IEEE 4th International Conference on Cloud Computing*, pp. 275–282, July 2011.

[67] W. Lin, B. Peng, C. Liang, and B. Liu, "Novel Resource Allocation Model and Algorithms for Cloud Computing," in *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pp. 77–82, Sept. 2013.

[68] J. Xu and J. A. B. Fortes, "Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments," in *2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*, pp. 179–188, Dec. 2010.

[69] D. Wilcox, A. McNabb, and K. Seppi, "Solving Virtual Machine Packing with a Reordering Grouping Genetic Algorithm," in *2011 IEEE Congress of Evolutionary Computation (CEC)*, pp. 362–369, June 2011.

[70] S. Kumaraswamy and K. N. Mydhili, "Bin Packing Algorithms for Virtual Machine Placement in Cloud Computing: A Review," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 1, pp. 512–524, 2019.

[71] M. Abdel-Basset, L. Abdle-Fatah, and A. K. Sangaiah, "An Improved Lévy Based Whale Optimization Algorithm for Bandwidth-Efficient Virtual Machine Placement in Cloud Computing Environment," *Cluster Computing*, vol. 22, pp. 8319–8334, July 2019.

[72] B. Speitkamp and M. Bichler, "A Mathematical Programming Approach for Server Consolidation Problems in Virtualized Data Centers," *IEEE Transactions on Services Computing*, vol. 3, pp. 266–278, Oct. 2010.

[73] T. C. Ferreto, M. A. Netto, R. N. Calheiros, and C. A. De Rose, "Server Consolidation with Migration Control for Virtualized Data Centers," *Future Generation Computer Systems*, vol. 27, pp. 1027–1034, Oct. 2011.

[74] K. Pradeep and T. Prem Jacob, "A Hybrid Approach for Task Scheduling Using the Cuckoo and Harmony Search in Cloud Computing Environment," *Wireless Personal Communications*, vol. 101, pp. 2287–2311, Aug. 2018.

[75] J. Zhang, H. Huang, and X. Wang, "Resource Provision Algorithms in Cloud Computing: A Survey," *Journal of Network and Computer Applications*, vol. 64, pp. 23–42, Apr. 2016.

[76] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource Allocation Algorithms for Virtualized Service Hosting Platforms," *Journal of Parallel and Distributed Computing*, vol. 70, pp. 962–974, Sept. 2010.

[77] L. Fortnow, "The Status of the P Versus NP Problem," *Communications of the ACM*, vol. 52, pp. 78–86, Sept. 2009.

[78] G. J. Woeginger, "There Is No Asymptotic PTAS for Two-Dimensional Vector Packing," *Information Processing Letters*, vol. 64, pp. 293–297, Dec. 1997.

[79] A. Ray, "There Is No APTAS for 2-Dimensional Vector Bin Packing: Revisited," *Information Processing Letters*, vol. 183, p. 106430, Jan. 2024.

[80] Q. Huangfu and J. A. J. Hall, "Parallelizing the Dual Revised Simplex Method," *Mathematical Programming Computation*, vol. 10, pp. 119–142, Mar. 2018.

[81] H. Mittelmann, "Decison Tree for Optimization Software – Benchmarks for Optimization Software." https://plato.asu.edu/bench.html (accessed Mar. 16, 2025).

[82] R. Eberhart and J. Kennedy, "A New Optimizer Using Particle Swarm Theory," in *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pp. 39–43, Oct. 1995.

[83] Y.-J. Gong, J. Zhang, H. S.-H. Chung, W.-N. Chen, Z.-H. Zhan, Y. Li, and Y.-H. Shi, "An Efficient Resource Allocation Scheme Using Particle Swarm Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 16, pp. 801–816, Dec. 2012.

[84] G. Dósa, "The Tight Bound of First Fit Decreasing Bin-Packing Algorithm," in *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies* (B. Chen, M. Paterson, and G. Zhang, eds.), (Berlin, Heidelberg), pp. 1–11, Springer, 2007.

[85] B. S. Baker and E. G. Coffman, Jr., "A Tight Asymptotic Bound for Next-Fit-Decreasing Bin-Packing," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, pp. 147–152, June 1981. Publisher: Society for Industrial and Applied Mathematics.

A Algorithm performance evaluation results

Allocations in the scenario presented in section 5.3.1 are performed with five separate heuristic algorithms and the number of servers required for hosting the set of applications is compared. The size of the set varies from 100 to 100 000 applications. The heuristics are described in section 4.3, and the results of these tests are presented and analyzed in section 5.4.1. The full data from the allocations is shown in tables A1 and A2.

Table A1: Expected value of servers used with 100 and 1000 applications allocated. All different server types included. The allocation of 100 applications was iterated 1000 times, and the allocation of 1000 applications was iterated 100 times.

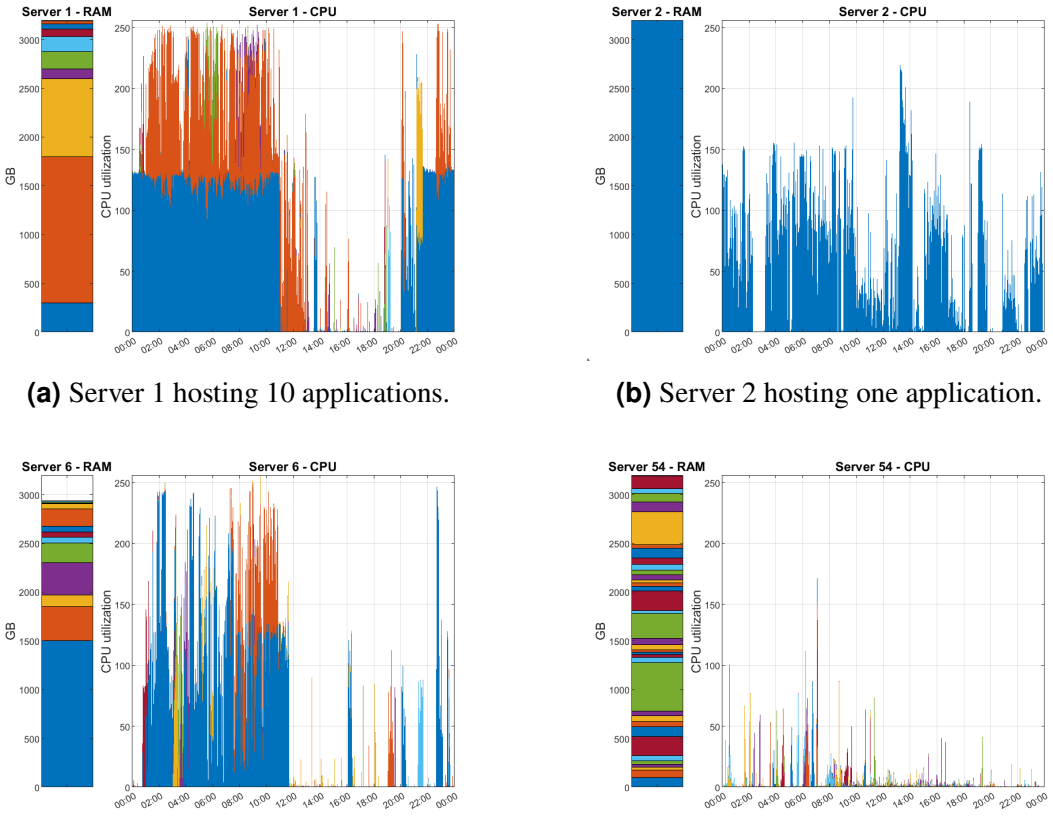
Applications	**100**					**1000**				
Algorithm	**FFD**	**BFD**	**WFD**	**Random**	**NF**	**FFD**	**BFD**	**WFD**	**Random**	**NF**
3 TB / 64 core	3,9	3,9	4,0	8,0	5,7	24,7	24,7	25,4	36,5	52,6
3 TB / 128 core	8,6	8,5	8,8	14,8	16,7	60,0	59,7	63,2	76,4	164,1
3 TB / 192 core	12,1	12,1	12,4	19,1	22,1	98,7	99,2	101,2	114,7	217,4
3 TB / 256 core	10,8	10,7	10,9	18,2	17,2	97,5	97,4	97,0	111,3	168,1
4 TB / 64 core	3,9	3,9	3,9	8,0	5,7	24,6	24,5	25,2	36,4	52,6
4 TB / 128 core	8,6	8,6	8,8	14,9	16,7	59,4	59,3	63,0	76,9	162,1
4 TB / 192 core	12,0	11,9	12,2	18,9	22,1	96,7	96,9	99,7	113,2	216,5
4 TB / 256 core	9,7	9,7	9,8	17,5	16,5	82,1	81,2	81,9	98,3	161,1
6 TB / 64 core	4,0	4,0	4,0	8,1	5,7	25,0	25,0	25,7	36,3	53,1
6 TB / 128 core	8,5	8,5	8,8	14,9	16,7	59,8	59,5	63,7	77,0	164,0
6 TB / 192 core	11,9	11,9	12,1	18,8	21,9	96,4	96,2	99,1	112,9	216,1
6 TB / 256 core	9,4	9,4	9,6	17,3	16,5	77,0	76,7	77,6	94,8	160,1

Table A2: Expected value of servers used with 10 000 and 100 000 applications allocated. All different server types included. The allocation of 10 000 applications was iterated seven or eight times depending on server capacity, and the allocation of 100 000 applications was iterated once for each server type.

Applications	**10 000**					**100 000**				
Algorithm	**FFD**	**BFD**	**WFD**	**Random**	**NF**	**FFD**	**BFD**	**WFD**	**Random**	**NF**
3 TB / 64 core	241,0	241,6	243,6	255,6	527,6	2342	2331	2344	2365	5326
3 TB / 128 core	537,6	538,9	579,4	580,0	1643,9	5382	5291	5568	5311	16230
3 TB / 192 core	927,6	938,6	970,6	961,4	2162,4	9183	9408	9605	9334	21722
3 TB / 256 core	995,8	994,0	981,4	1009,6	1705,0	9924	9908	9689	9734	16873
4 TB / 64 core	223,4	222,5	227,8	140,4	513,6	2343	2330	2344	2366	5326
4 TB / 128 core	544,0	539,4	582,0	575,4	1623,5	5389	5333	5744	5427	16277
4 TB / 192 core	927,0	934,4	952,1	954,5	2166,6	9213	9397	9496	9370	21563
4 TB / 256 core	804,3	797,8	786,1	801,9	1615,4	8229	8165	7939	7815	16280
6 TB / 64 core	230,7	230,9	234,4	246,6	525,7	2341	2330	2343	2379	5326
6 TB / 128 core	541,1	539,9	567,0	567,9	1620,4	5392	5355	5752	5426	16241
6 TB / 192 core	907,7	915,9	930,4	941,1	2143,1	9138	9376	9322	9288	21461
6 TB / 256 core	766,7	766,9	763,9	773,7	1614,4	7687	7690	7590	7562	16154

B Example server allocations

A set of 520 applications is allocated to a set of servers using the FFD heuristic. The set of servers and applications belong to one DC, which is described in section 6.1. The allocation required 60 servers in total, of which four examples are presented in figure B1.



(a) Server 1 hosting 10 applications.

(b) Server 2 hosting one application.

(c) Server 6 hosting 13 applications.

(d) Server 54 hosting 35 applications.

Figure B1: Four example server allocations after the set of applications has been reallocated using FFD heuristic. Each server has 4000 GB of memory capacity, of which 3200 GB is used at most, and 256 virtual cores.