NATIONAL TECHNICAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

COMPUTING SCIENCE DIVISION

COMPUTING SYSTEMS LABORATORY

Analysis and optimization of Cloud software in shared environment with adaptive resource allocation

Diploma thesis

Rigas A. Papathanasopoulos

Supervisors: Nectarios Koziris
N.T.U.A. Professor

Athens, February 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ, ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Ανάλυση και βελτιστοποίηση της επίδοσης cloud εφαρμογών σε διαμοιραζόμενα περιβάλλοντα με προσαρμοστική ανάθεση πόρων

Διπλωματική εργασία

Ρήγας Α. Παπαθανασόπουλος

Επιβλέπων Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από τριμελή εξεταστική επιτροπή την 13η Φεβρουαρίου 2015.

....	....	....
Νεκτάριος Κοζύρης	Παναγιώτης Τσανάκας	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2015

....

Ρήγας Α. Παπαθανασόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Το cloud computing είναι ένα από τους σημαντικότερους κλάδους της επιστήμης των υπολογιστών στις μέρες μας. Οι cloud υποδομές έχουν ήδη υιοθετηθεί ευρέως από μία πληθώρα μοντέρνων εφαρμογών, ενώ συνεχώς αυξάνεται ο αριθμός και τα είδη των εφαρμογών που χρησιμοποιούν cloud συστήματα για την αποδοτικότερη εκτέλεση τους. Στα πλαίσια της συγκεκριμένης διπλωματικής εργασίας θα μελετηθεί η συμπεριφορά αντιπροσωπευτικών οικογενειών από cloud όταν εκτελούνται σε cloud περιβάλλοντα. Ιδιαίτερη έμφαση θα δοθεί στον προσδιορισμό της αλληλεπίδρασης που δημιουργείται λόγω της «συνύπαρξης» διαφορετικών εφαρμογών κατά το διαμοιρασμό κοινών πόρων στο data center και στη μελέτη της επίδρασης που έχει στην επίδοση τους. Η ανάλυση της συμπεριφοράς και της επίδοσης των εφαρμογών σε διαμοιραζόμενα περιβάλλοντα θα διεξαχθεί λαμβάνοντας υπόψη τις διαφορετικές ανάγκες κάθε εφαρμογής για χρήση των διάφορων πόρων του συστήματος (π.χ. επεξεργαστική ισχύ, μνήμη, δίκτυο κ.τ.λ.). Τα συμπεράσματα που θα προκύψουν θα αξιοποιηθούν για την ανάπτυξη ενός αλγορίθμου δρομολόγησης των εφαρμογών σε διαφορετικούς συνδυασμούς, ανάλογα με τις απαιτήσεις τους για πόρους, με τρόπο που να επιτρέπει την αποδοτικότερη αξιοποίηση των πόρων των server και καλύτερη συνολική επίδοση.

Λέξεις Κλειδιά

Νέφος, πληροφορική νεφών, εικονικές μηχανές, εικονικοποίηδη, δρομολόγηση με βάση τη χρήση πόρων, δρομολογητής, διαμοιρασμός πόρως, χρησιμοποίηση εξυπηρετητή

Abstract

Cloud computing is one of the most important computer science parts of nowadays. A wide range of modern applications have already become cloud-based, while more and more applications of all kinds are planning to do the same aiming to a more efficient operation. In this thesis we will study the behavior of some representative families of cloud applications whenever they are executed in cloud environments. We will especially focus on determining the interference that is caused by the "coexistence" of different application during they share common resources on a data center, and study how this affects their performance. The analysis of the behavior and the performance of the applications in shared environments will take place considering the different needs each application has for different resources (like processing power, memory, network etc.). The conclusions that will be drawn by this analysis will be used to create a scheduling algorithm that will place the applications in different combinations, based on their needs for resources, in a way that will allow better server utilization and a good overall performance.

Keywords

Cloud, Cloud computing, virtual machines, virtualization, resource aware scheduling, scheduler, resource sharing, server utilization

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή Νεκτάριο Κοζύρη, ο οποίος ήταν ο επιβλέπων της διπλωματικής μου εργασίας και μου έδωσε την ευκαιρία να την εκπονήσω στο χώρου του εργαστηρίου υπολογιστών συστημάτων. Επέλεξα να εργαστώ στο συγκεκριμένο εργαστήριο διότι κατά τη διάρκεια των σπουδών μου κέντρισε τον ενδιαφέρον μου με τα άψογα οργανωμένα μαθήματά του, ενώ μου έδωσε και την ευκαιρία να έρθω σε επαφή με μερικούς εξαιρετικούς ανθρώπους και επιστήμονες. Θέλω να ευχαριστήσω ιδιαίτερα την Αθανασία Ασίκη και το Γεώργιο Γκούμα, μέλη του εργαστηρίου, που καθ'όλη τη διάρκεια της διπλωματικής μου εργασίας, μου παρείχαν γνώσεις, καθοδήγηση και εμπιστοσύνη, όντας πάντα πρόθυμοι και συνεργάσιμοι οποτεδήποτε τους χρειάστηκα. Επίσης, θα ήθελα να ευχαριστήσω συνολικά το προσωπικό του εργαστηρίου, το οποίο μου παρείχε χρήσιμη βοήθεια και τεχνική υποστήριξη όταν αντιμετώπισα δυσκολίες.

Θα ήθελα ακόμη να ευχαριστήσω θερμά την οικογένειά μου για τη στήριξη που μου προσέφερε κατά τη διάρκεια των σπουδών μου και που ακόμα μου προσφέρει. Ευχαριστώ τους φίλους μου και την κοπέλα μου που σταθήκαν δίπλα μου όποτε το χρειάστηκα, τόσο στις ευχάριστες όσο και στις δυσάρεστες στιγμές.

Table of Contents

1 Introduction

1.1 Cloud computing

Cloud computing, is one of most rapidly developing and evolving computer technologies at moment. Cloud computing, or just “*the cloud*”, as we simply refer to it, has quickly evaded peoples’ lives in the last few years. Although cloud computing is so widespread, there is no strict definition about it. Cloud is strongly connected with the idea of centralized computing, which refers to computing power at a central location, that has multiple terminals attached to a main computer. A simplified definition of cloud computing is the following:

> ***Cloud computing*** *is computing in which large groups of remote servers are networked to allow centralized data storage and online access to computer services or resources*
>
> *Wikipedia.org*

Cloud computing can vary a lot. There are different service models, deployment models, architectures and an extendible variety of possible uses. Users can have different and conflicting requirements too. We will try to explain the main concept of cloud computing, and how important it is for the future of technology.

1.1.1Evolution

Since cloud computing cannot be strictly defined, it is impossible to determine when its concept was conceived. The term “cloud” has not been widely used before 2006. Cloud has its origins back to the 1950s, when the first large scale mainframe computers were deployed, and became accessible via terminal computers without computing capabilities. The first forms of cloud computing were inseparable with the terms of supercomputer and HPC (high performance computing), while its first uses were mostly scientific, financial and academic.

It was during the 1990s when scientists created the first large scale computers that could be shared among multiple users. Telecommunication companies used these infrastructures to provide VPN services with a lower cost and good Quality of Service

(QoS). Time-sharing became a field of research and optimization algorithms were developed to provide better efficiency to users.

At the late 2000s the first cloud platforms were created, providing tools for developers, for making centralized computers easier to be managed and shared. In the last few years, cloud computing quickly became mainstream or even a trend. A high number of the world’s most powerful technological companies developed their own clouds, providing services that started to vary and extend rapidly. New products and client needs were created, and many widely used services moved to the cloud. Today, cloud users are not limited to scientists, developers or people who need high-performance computing power, but almost everyone has access to some very popular cloud services. File storage, file and settings synchronization across multiple devices and backup creation, are some tasks people’s PCs, smartphones and tablets do every day.

Some very popular cloud-based services

Today, cloud computing enjoys days of glory. Hundreds of new services are developed every year, managing to attract millions of users, while many of them change the way we communicate, socialize or even the way we live. The majority of the most used online services have already moved to the cloud or have plans to do so. There is no doubt that cloud is the future of computing, and it will attract our attention for the next years.

1.1.2 Characteristics

Cloud’s popularity and success, relies on some very important characteristics.

- On-demand self-service:
  Users must be able to change the resources provided to them alone. Programming APIs, and web user interface are some common ways achieving this.

- Broad network access:
  Cloud services can be accessed remotely, without demanding user's physical presence.

- Resource pooling:
  Physical resources are able to be shared among multiple users, without conflicts. Virtualization is usually the best solution to achieve this.

- Rapid elasticity:
  Each user's resources can be altered either on demand or automatically, so every client has the necessary resources he needs.

- Measured service:
  Users' usage of resources is monitored and reported to both clients and service providers. This makes services transparent and better controlled, allowing service providers to alter their clients' resources according to their needs.

1.1.3 Advantages

Compared to the classic computing model, the cloud has some serious advantages that make it special. The most important are mentioned below.

High performance
Tasks like scientific experiments, physics simulation or data analysis, can usually be very demanding and seriously costly. Universities, scientific research centers or even governments are not always able to acquire, maintain and use their own high performance computer systems that are capable of satisfying their needs. Cloud computing offers a solution for *immediate* access to *high performance computing*. Clients can rent computing power from cloud providers. The provided computing

power can be as *powerful* as clients need it, for as *long* as they need it, helping them save time, money or even space.

Scalability

As cloud resources can be extended on demand, clients can scale their applications efficiently as much as they need to. Cloud systems usually allow their users to see unlimited resources, although in fact there are limits depending on the actual computing hardware. Virtualization makes resource expansion an easy task, allowing virtual resources (that are visible to the clients) to surpass the physical ones.

Reliability

Modern cloud systems usually guarantee some minimum performance expectancy to their clients, usually called Quality of Service (QoS). There are no standard metrics for measuring a system's performance capabilities. The most commonly used unit of high performance computing power is FLOPS (FLoating-point Operations Per Second).

Independency

Virtualization, allows clients to design, implement and run applications on virtual environments without dealing with cloud's platform characteristics. The same applications can be reused to different clouds, with different architecture, or even different operating systems. Clients can always use the operating system and the applications they are familiar with, while the virtualization level will allow execution on every server.

Expandability

Modern cloud infrastructures are designed to be easily expandable, allowing the physical resources to be adjusted in order to satisfy clients' needs with the minimum operating cost. Cloud's architecture allows easier changes infrastructure changes compared to classical computing model. There are various techniques and technologies in development to make cloud cluster's expansion work with a plug-and-play feature.

1.1.4 Virtualization

Virtualization, in computing, refers to executing something in a virtual environment, instead of an actual one, using virtual resources, such as processors, memory and network. A *virtual machine* (VM), is a virtual computer system. Virtual machines, are also mentioned as *hardware virtualization.* They emulate a real computer with an operating system installed and virtual resources and devices attached to them.

Creation and execution of virtual machines, requires the appropriate software, called *hypervisor*. A computer that executes one or more virtual machines, is called *host*, while the virtual machines are called *guests*.

Cloud computing has been taking advantage of virtualization since its very first years. The most important advantages of virtualization are mentioned below:

Failsafe
When a guest system fails, the host is not affected. Virtual machines are seen as processes from the host and they are not affected by the failure of the guests' virtual resources.

Resource management
A virtual machine creates and uses its own virtual resources, which make it strongly limited. Physical resources can be shared and simultaneously used from multiple guests, achieving one of the most important goals of the cloud computing.

Independency
A guest operating system, can be hosted on any physical machine that is capable of virtualization. Cloud computing provides independency from physical machine's characteristics by using virtualization. Users are usually able to choose the operating system they want to work with and they do not have to deal with any of the characteristics of the host machine.

Snapshotting
The state of a virtual machine and its attached devices and resources, is called snapshot. The snapshot of a specific moment can be created, so the virtual machine can revert back to this state at any time. This is an extremely useful technique for creating

backups and making the machines *failsafe*, allowing them to perform risky operations without limits. On case of failure, the latest snapshot is reverted, minimizing the risk of losing data.

Migration

Snapshotting allows a virtual machine to be saved, stopped, moved and resumed to another physical machine, making migration an easy task. This is a very useful property for the cloud computing, as resource management in centralized computing usually requires migration of guests. Migrating virtual machines is much easier than migrating processes.

1.1.5 Challenges

While the cloud is rapidly developing, it needs to accomplish some business goals. Operating and maintaining a cloud cluster with the best cost efficiency is a parameter that can make a cloud product either profitable or injurious. Everything that takes part in the cloud can affect the cost and the final product's outcome accordingly. A cloud operator seeks a way to provide the best quality of service, with the minimum cost, consuming the minimum energy and computer resources possible. To achieve this, a suitable architecture must be deployed, using the best software and hardware optimizations possible. Design, optimization, implementation, and maintenance of a cloud system is a brand new field of research, offering many ways to provide solutions for a better cost efficiency, creating business opportunities.

1.2 About this thesis

1.2.1 Motivation

With a cloud cluster given as resource, a service provider wants to achieve the optimal results for their clients' needs. To achieve this, they must ensure they will operate with:

- Minimum cost
- Minimum energy consumption
- Best optimization for their computing resources

- Maximum server utilization
- Best quality of service for their clients

In this thesis, we are about to focus on *server utilization* and *quality of service*. We believe, that different types of tasks, that require different types of resources, can have different interactions when they are placed in physical machines differently. By scheduling virtual tasks on physical machines depending on their behavior, we can come up with better server utilization without damaging the clients' application performance, and therefore the quality of service.

Let's take a look at an example of this theory. Consider two virtual tasks, one disk-bound and one network-bound. We have a choice to make. We can either place them and execute them on different physical machines, or place them on one physical machine so they run alongside. An execution on the same host could possibly affect both negatively, but it might not do so. Each one of these two tasks, performs some input/output operations, reserving different virtual and physical resources. These resources can be the same or independent. In this example, it is likely that the two virtual machines will not have a serious slowdown on their performance, because they rely on two almost independent hardware resources.

1.2.2 Short description

At first, we will define some basic types of applications that we want to focus on, and choose a specific and representative application of each type. We will measure, study and analyze the interactions between these different applications as they happen when hosted on virtual machines, so we have a basic knowledge about the interference between them.

After that, we will define different types of problems, depending on difficulty. On each one of these cases, we will try to create an algorithm for efficient and resource-aware scheduling. The results of each problem will be analyzed and evaluated separately. We compare our algorithm to other state-of-the-art competing decision systems and evaluate it using our starting goals for the system.

1.2.3 Goal

Our goal is to create an algorithm that takes a sequence of virtual machines (guests), hosting different types of tasks, and a set of physical servers (hosts), and schedules the guests to the hosts with an efficient way. To create this decision making system, we need a base of knowledge. Our basic knowledge will be the set of the possible interactions between the available types of applications, expanding it as much as needed in more difficult problems. Our algorithm can be consider as resource-aware, since we categorize the guests, schedule them with respect on what resources they use.

Cloud systems that do not use a resource-aware algorithm for scheduling usually use some classic techniques, like round-robin and random schedulers. We will compete with these techniques, and evaluate our results compared to theirs. We will analyze the average performance of the guests' hosted applications, and define some metrics of fairness and quality of service.

We intend to prove that scheduling guests according to their type can lead to better overall performance, better overall fairness between the clients, and a better quality of service. We also believe that combining guests by respecting their resource usage will lead to better utilization of the physical resources and less interference between the guests. By better server utilization, cloud providers can achieve better results with the same or even less physical resources, reducing their operation cost and achieving a better performance-to-cost ratio.

Since the research of this thesis focuses on the virtualization level, the results of this research can be used on most cloud infrastructures that include a virtualization level on their services. The prerequisites for such use, are for the service provider to have information about the behavior of each guest and also be able to schedule them freely at their cloud infrastructure. Clouds that follow the "platform as a service" (PaaS) or the "software as a service" (SaaS) service model, are potential users of a resource-aware scheduling algorithm like the one we want to create. The PaaS and SaaS service models provide to the customers either a platform (Virtual Machine) or a software (that is hosted on virtual machines), allowing the service provider to schedule those virtual machines according to their needs.

1.2.4 Related work

With cloud computing developing quickly, there is big interest in developing optimization techniques in many possible levels. Optimizations can be either on hardware level or software level. Let's ignore hardware optimization techniques, and focus on software. Profiling applications and scheduling them depending on their behavior is the most common software optimization technique. A profiler, is a program that takes an executable as input and analyzes its interaction with the hardware, from basic resource usage (CPU, memory, disk etc.), to advanced information (like cache miss or hits). These data can be very valuable for a scheduler that is resource-aware, since it can predict some behavior patterns and come up with better results.

Some relevant published work, that outlines related schedulers to our own, includes:

- **DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments παραπομπή**
  This paper describes an algorithm that is executed alongside virtual machines and detects interference between them. It can also store information about interference between different types of applications, so it can predict potential problems in the future.

- **PACMan: Performance Aware Virtual Machine Consolidation παραπομπή**
  This paper describes an algorithm that takes a set of virtual and physical machines as input, and by profiling the guests in a first phase, it schedules them to the hosts efficiently in a second phase. The algorithm can run in two different modes, achieving either the best average slowdown or the best maximum slowdown.

In this thesis, we won't develop or use a profiler, since the information about each virtual machine's behavior is known in advance. We focus on how we can use these information combined with already known interactions to create an efficient scheduling algorithm.

2 Experimental environment

2.1 Physical servers and hypervisor

We had two physical server available for the following experiments.

Server 1	**Server 2**
Intel(R) Xeon(R) E5335 @ 2.00GHz	Intel(R) Xeon(R) E5-2650 @ 2.00GHz
8 cores	8 cores
1 thread / core	2 threads / core
8 GB RAM	64 GB RAM
64 GB SWAP	No swap

Both of those servers were located on the same local network. They were connected on the exact same way with a third server which was used from the network benchmarks.

The first one was used for the two first experiments, and the second one for the last experiment. Further information about resources management and sharing between the guests are provided at each experiment's description.

We selected QEMU for hypervisor, as it was preinstalled and it can be used with KVM, allowing virtual machines have a near-native performance. QEMU is a free and open-source hypervisor software that performs *hardware virtualization*. It is very popular across Linux users because and it provides support for many different CPU architectures and allows easy management of virtual resources. We define our requirements for virtual resources that each VM will use from the command line, when we launch the hypervisor. KVM can be used as a QEMU's module, straight from the command line. The use of KVM, allows the virtual machines' applications to run almost like they were running natively, and it is a part of software optimizations we use to boost virtual applications' performance.

2.2 Virtual Machines

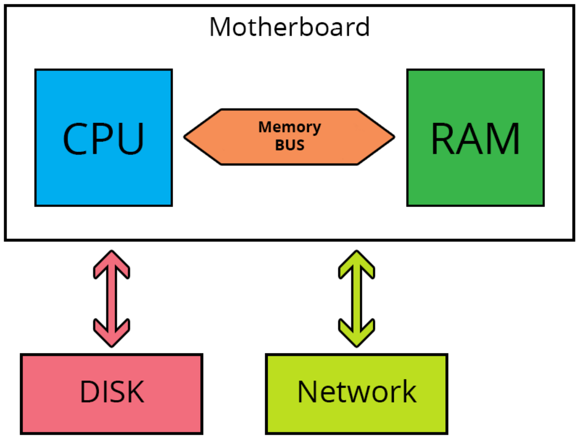
All the virtual machines used on the following experiment have Ubuntu Server 12.04 installed. We created a virtual machine using the services of Okeanos, which provides free cloud services to the Greek academic community. We also installed the benchmarks and the scripts that were necessary for our tests. Using the SNF image creator software, we created a disk image file, representing the hard disk of the above virtual machine. This file was a raw copy of the whole virtual hard disk data and it is compatible with QEMU. The chosen file format allowed us to create new virtual machines by just copying the original image into a new file. The ease of this procedure reveals virtualization's elasticity that we mentioned before.

2.3 The applications

The most important part of our experiment is to choose the appropriate applications that will allow us to safely conclude about interference between different application types.

Our first step, is to define the different application categories that we want to examine. We assume that any application can belong to one of these possible families. We consider that every physical machine has the following basic structure.
There are five basic components:

- The central processing unit (CPU)
- The main system's memory (RAM) that is connected with the CPU through a data bus
- The disk that is connected with the motherboard through an independent bus (or cable)
- The network card that connects the server with the rest of infrastructure and furthermore, the internet.



From the perspective of operating systems theory, the five different colored components are able to work *almost* independently. For example, a CPU-bound application needs as much computing power is possible, small amounts of memory (slightly affects the memory bus and the memory availability) and usually, no disk or network resources. A memory-bound application, which needs large amounts of memory, affects the RAM availability and creates data traffic on the memory bus (not necessarily a big one). A memory-bandwidth-bound application, causes lots of data traffic in the memory bus, but it does not consume large amounts of memory. Disk and network bound applications, use the desired resources without seriously affecting any other. Every application needs processing power and memory to operate, but we assume that when an application is not CPU-bound or memory-bound it cannot affect these resources in a harmful way. So from now on, we will categorize applications into these five families: *CPU, memory bandwidth, memory size, disk, network*.

Interference between applications running on virtual machines within the same physical host might differ with interference the same applications cause to each other when running in the same computer directly. An ideal application of each family is the one that does not affect other resources at all, but this is not really possible. So we have to work with applications that cause a minimal and ignorable workload to the virtual resources that are not targeted.

2.3.1Why use benchmarks

In our experiments we chose to work with a benchmark from each application family. Benchmarks are extreme examples of an application family that measure the limits of a specific part of the computer. This will help us understand interference between applications, as we want each application to isolate and stress only one virtual or physical resource that matches the categorization we made above. For example, an application that measures how quickly the computer calculates a specific range of magic numbers, is a CPU benchmark that evaluates CPU's performance and it will slightly affect the memory, the disk or the network.

Using benchmarks as applications will also help us test our algorithms under more stressful conditions. An *average* application that is CPU-bound does not stress the CPU as much as a benchmark, an *average* memory-bound application does not consume the whole memory, and the same applies for the rest of the application types.

Interference prediction between any two applications is a difficult job. Applications that belong to the same family and use the same resources can have very different behavior, and therefore different interference. So we want to study what happens on the *worst case scenario*. We suppose that the use of benchmarks as applications will cause the maximum interference possible between the virtual machines. We also surmise the stressing the virtual resources as much as possible will also cause the maximum possible stress on the physical resources too.

2.3.2 Benchmarks choice and explanation

We present the benchmarks that we selected to use and describe their jobs.

CPU benchmark: Sysbench

This program calculates a specific amount of prime numbers passed as parameter and measures the total runtime as a metric of performance. This application has multithread support, while the total amount of threads created is passed as parameter. We chose to create as many threads as the amount of virtual cores that exist in each VM, so we make sure that the program takes advantage of all the CPU cores that it has and CPU utilization is the maximum possible.

Network benchmark: Netperf

We used netperf to measure the network throughput, using the TCP stream test between the virtual machine and a server located in the same local network. The results were measured in Megabits / second. This application is executed in a single thread.

Memory bandwidth benchmark: MBW

This benchmark performs three tests that measure system's memory bandwidth. The only parameter we defined the amount of repeats that each test performs. The metric of performance used was the average memory throughput of these three tests measured in Megabytes / second. This application is executed in a single thread.

Memory size benchmark: Custom

We created a custom Python script that performs the following:

1. Allocates the maximum amount of memory allowed from system
2. Loops N times writing one random character on each page. It also calculates and writes a random hash every 8th page.

Let's explain the logic behind the procedure. This application manages to be memory-size-bound, by reserving the maximum possible virtual memory. Memory allocation through a system call does not necessarily mean that system's memory has reserved the space for all of these data on the main system's memory. By writing each page we make sure that the system moves every one of them to the system's RAM, forcing real allocation of memory.

Consecutive page writes might lead to a memory bandwidth application, as pages are transferred between CPU and RAM without a break. This is why we need to have a considerable CPU time too. We achieve this by using hashing, a calculation that takes up to 1 or 2 seconds (on average) of CPU time, allowing memory data bus to be disengaged. As a metric, we use the total runtime, because the script's operation are depended on the system's memory availability. Both CPU and memory bus can affect the runtime too, but not as much as the page availability. If memory fills up and pages start moving to the system's SWAP, then when a page that cannot be found on system's main memory will be brought back from the SWAP which is located to the hard drive. Fetching the page will result a time penalty that is incomparable bigger than the time penalty that might be added from CPU or memory bus stress.

This application is single threaded. The full Python code can be found on the appendix.

Disk benchmark: HDparm

Hdparm can be used as a benchmarking tool for measuring the performance of a hard disk. It performs both read and write speed tests, with both cached and non-cached data. We will ignore the cached data tests because we want to measure the real disk performance only. The average speed of both read and write tests will be used as a metric, measured in Megabytes / second. Hdparm runs in a single thread.

3 Problem one: Placing couples of Virtual Machines into physical servers

Before we start with problem description, we need to explain the process we will follow.

What we deal with, is interference and performance slowdown. We want to experiment, study and -if possible- predict interference between different combinations of VMs, running a specific benchmark from those defined above. Before we proceed with predictions, we first have to create our knowledge base that will be used to predict more complex combinations' interference. The first step is to create a table with the possible combinations between two VMs running a benchmark. We will use labels on the virtual machines based on the benchmark that they are assigned to execute. For example, a disk-VM, is a virtual machine that executes hdparm inside, which is a disk-bound benchmark, so we expect the virtual machine to behave as a disk-bound application too.

3.1 Problem description

Ideally, in a cloud cluster, we have as many physical servers as the number of the virtual machines we need to host. Each VM is hosted by one physical machine, allowing optimal performance for the virtual machines and their applications, without any interference from other applications that run one the same server. This is a scenario that mostly fits HPC applications, where the performance of the guests is the most important parameter of the service's success.

We want to start by solving a simple problem:
We are given N physical machines and 2N virtual machines that need be scheduled and hosted.
Each server will host exactly 2 VMs after the scheduling algorithm runs, so we can call this problem as "coupling VMs to servers". We suppose that all scheduling algorithms cannot have any information about future input, so the VM that will be the next one

coming in the queue is passed as input. The scheduling algorithms will decide about what server the provided VM will be hosted on, based on the current state of the cloud cluster and the type of VM given. This means that the algorithm that we will design will be *deterministic*, having the same output every time that is given the same input, independently from possible future input. This is useful because:

1. A deterministic algorithm is easier to be understood, having similar outputs when given similar inputs.
2. It is compliant with the nature of the real problem. In real cloud applications the future input is normally unknown and it depends on the clients' behavior and needs.

3.2 Measuring slowdown between possible couples

As mentioned before, the base of our knowledge that we will use to predict interference between VMs, is the slowdown of each application caused by interference when another application runs alongside. After running all the possible combinations, the following table is formed:

	CPU (sysbench)	*Network (netperf)*	*Bandwidth (mbw)*	*Mem.Size*	*Disk (hdparm)*
CPU (sysbench)	1,00	0,97	0,96	1,00	1,00
Network (netperf)	0,99	0,95	0,85	0,92	0,99
Bandwidth (mbw)	1,00	0,93	0,57	0,87	0,96
Mem.Size	1,02	0,92	0,82	0,60	0,95
Disk (hdparm)	0,82	0,76	0,84	0,91	0,51

Let's explain the table. Each cell has one number, which refers to *performance slowdown.* By performance slowdown we refer to the following ratio:

Slowdown = $\frac{performance\ metric\ with\ interference\ from\ another\ VM}{performance\ metric\ when\ running\ alone}$

Each cell refers to slowdown caused to the *application indicated by its row, when it is running alongside the application indicated by its column*. Let's see some examples:

- The cell on row 1 and column 1 has the value 1.00, meaning that when a CPU benchmark runs alongside another, there is no performance slowdown.

- The cell on row 3 and column 2 has the value 0.93, meaning that when a memory bandwidth benchmark runs alongside a network benchmark, the first one has a performance slowdown of 7%, or equally, it performs slower than when it is executed isolated, by 0.93 times.
- To find out what happened to the network benchmark when combined like above, we must take a look at the cell on row 2 and column 3. Its value is 0.83, meaning that the network benchmark had a bigger performance slowdown of 17%.

Colors are relative to performance, with red being a sign of bigger slowdown (worse performance), and green being a sign of less slowdown (better performance). A full row shows us how much a benchmark affects the others, while a column shows how a benchmark gets affected when combined with others.

Overall, we observe some *behavior patterns*. The more obvious are the following:

- CPU benchmark is less affected by other VMs running on the same physical machine, no matter their type.
- Disk benchmark is more vulnerable to interference caused by other VMs, especially when it is combined with itself.
- Combing application types with themselves is generally a bad idea, causing more slowdown.

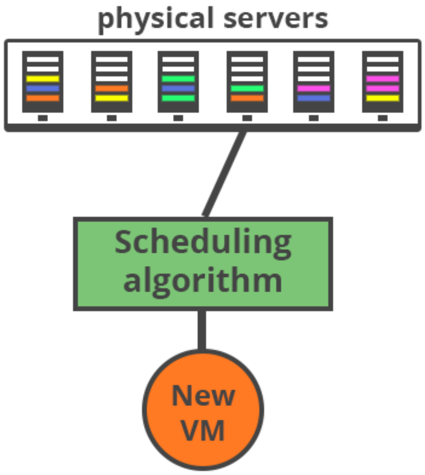
3.3 Running model

The physical machines are equipped with an 8-core CPU and 8GB of RAM, as mentioned on a previous section. Every virtual machine we used on this experiment is equipped with 4 virtual cores and 4GB of virtual memory, exactly half of the physical resources. All the applications run in a single thread, except the CPU benchmark (sysbench) that we set to run with 4 threads, exactly as many as the virtual CPU cores. We chose to create 4 threads with sysbench to have the best CPU utilization. If we create more threads that the virtual CPU cores, then the application will start to slowdown, as different threads will have to compete for the same virtual cores.

In this scheduling problem the virtual threads do not exceed the physical cores. The worst case scenario is when two CPU benchmarks run alongside, with 4 threads each. This means that the applications that are executed inside the virtual machines will be running almost without interruption. We make sure that the physical machines have no other intensive tasks running except from our tests. The fact that each VM has CPU and memory resources exactly half of the physical ones, means that the two VMs can run in parallel with the best utilization possible.

3.4 Algorithm design

We need to create an algorithm that will place the VMs in the physical servers. It is important to remind that we will have N servers and 2N virtual machines given in this problem. The algorithm will accept as input the next VM that needs to be scheduled and the current status of the cloud, meaning the list of the physical servers and their state. The state of a physical server can be described with the list of VMs that it hosts.



The scheduler decides where the new VM will be placed

The implementation of the scheduling algorithm requires some basic design decisions to be taken first. A scheduler’s goals can vary, but the most important parameters that we need to take care of are the *average slowdown*, the *maximum slowdown* and the *performance variance*. The offered quality of service can be measured on the value of these metrics. Ideally, we should be able to provide a minimum value for each parameter that we can guarantee it will not be violated. We preferred to specify a minimum value for each metric that the algorithm will *try* to respect, but if it is impossible to abide by, the algorithm should continue by placing the VM on the server that each time has the minimum average slowdown. We believe that this tactic will lead us to the minimum violation of the QoS values that we have initially planned to keep.

We also need to define how the algorithm will choose a server so the final combinations are as near to optimal as possible. We used a very simple procedure that can be summed up in these steps:

1. Scan the servers serially

2. If a server hosts the maximum number of VMs allowed (2 on this case), then continue
3. If a server is empty, place the VM there and exit
4. Check if a possible insertion of the next VM on this server will cause violation of standards
5. If there are no violations put the VM on this server and exit
6. If there are violations, calculate the average slowdown of the server's VMs after the insertion of the new one. If this score is better than the best-so-far, replace it and mark this server as the best. Continue looping through the server list
7. If the loop exits with a server marked as best, it means we failed to find a server without violations. Then we place the VM to that server which will have minimum average slowdown.
8. If the loops exits without any server marked as best, it means that all servers reached the maximum number of VMs allowed, so we dismiss the VM. This will not happen unless we provide more than 2N VMs as input.

The scheduling algorithm written in pseudo-language:

```
algorithm place_vm:
        input: array of servers, vm /* to be added */
        output: destination_server
destination_server = -1 /* none */
max_performance = 0

for i from 0 to server.length do
        if  ( server[ i ] has the maximum number of VMs )
                continue
        else if ( server[ i ] is empty )
                destination_server = i
                return destination_server /* We choose this server and exit */

        score_i = combine( i, j ) // combine( i, j ) is the slowdown added to i by j
        score_j = combine( j, i ) // combine( j, i ) is the slowdown added to j by i
        /* We check for average or distinct scores that violate our QoS */
        has_violations = check_for_violations_of_QoS( score_i, score_j )

        if ( not has_violations )
                destination_server = i
                return destination_server /* We choose this server and exit */
        /* We always keep the best score so far, even if it violates standards of QoS */
        else if ( scores_after.average > max_performance )
                destination_server = i
                max_performance = scores_after.average
done
/* This happens only when every server is full */
if ( destination_server == -1 )
```

```
        dismiss vm and exit
else
        return destination_server
end
```

In the above algorithm there is a procedure called *has_violations* that checks for quality of service violations. This procedure has two parameters that are common for all the servers:

Minimum score: *a number between 0 and 1.0*

A server that hosts a VM with excepted performance slowdown less than this score violated the standards.

Minimum average: *a number between 0 and 1.0, usually bigger than the minimum score.*

A server that has an average expected slowdown less than this number violates the standards.

These values need to be balanced between the minimum and the maximum interference that occurs in the slowdown map we created. The *minimum average* can be greater than the average slowdown of all the possible combinations, as we have the goal to create a mix of pairs that have an overall performance better than the average. The *minimum score* can be a value lower than the average, but not by much, so we avoid big variance on applications' performance. After some experiments with various values, we chose to set the minimum score to 0.75 and the minimum average to 0.8.

3.5 Other algorithms

To evaluate our scheduling algorithm we need to compare it with other popular schedulers that are used in real cloud systems and are not aware of the resources a VM uses. In our scheduling problem we have not different priority levels between the guests. The most common schedulers of this category are the round-robin, the random and the least-used one (which places the next VM to the server that has the minimum amount of guests at that point). Here is a detailed description of the algorithms we used and tested in this problem.

Round robin

Destination server is initially set as 1 (meaning the first server). On every call the destination is increased by 1, so the VMs are placed on next to another until every server is full (with 2 VMs each). If the last server is reach, then the destination server I set as 1 again, leading to cycles.

```
destination = 1 //global variable

placeRoundRobin:
        destination = (last_destination++) % servers

        while ( destination is full )
                destination = (last_destination++)  % servers;

        last_destination = destination
        return destination
```

Round robin (full first)

Destination server is initially set as 1. This algorithms start placing the VMs on the same server until it reaches the maximum amount allowed, and then it moves to the next one.

```
destination = 1 //global variable

placeRoundRobinFull:
        if (destination is full )
                destination++
        return destination
```

Random

Destination server is chosen randomly each time, as long as it is not full.

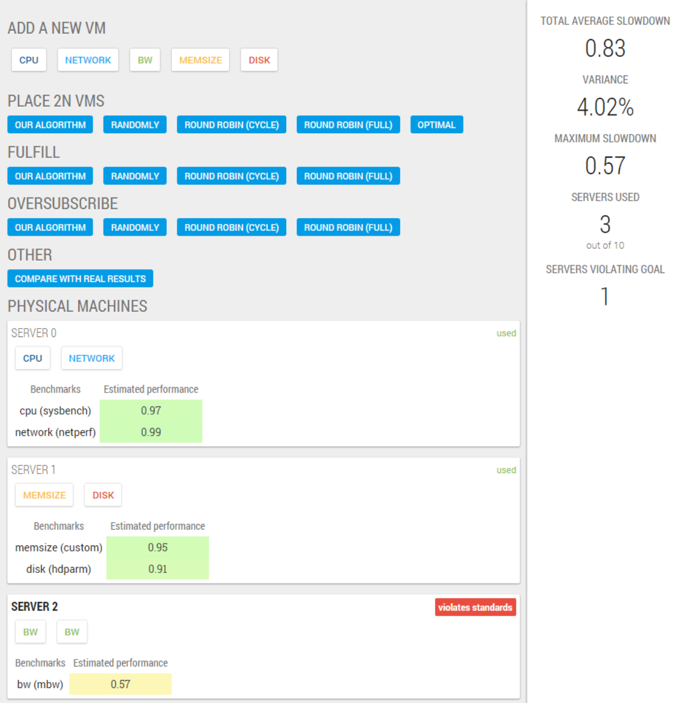
```
placeRandom:
        var destination = -1
        do {
                /* Random selection of server */
                destination = random( 0, servers )
        }
        while ( destination is full and not all servers are checked )
        return destination
```

Optimal

The complexity of this problem is relatively small, allowing us to search for the optimal combination of VMs that offers the best results for a specific metric. We chose to have the best average performance as goal.

3.6 Testing platform

In order to test the different schedulers, we created a platform that emulates a cloud cluster in Javascript. We also developed an HTML user interface for visualizing the cluster's servers, the VMs hosted in them, and the expected performance of each application that is executed.



TOTAL AVERAGE SLOWDOWN
0.83

VARIANCE
4.02%

MAXIMUM SLOWDOWN
0.57

SERVERS USED
3
out of 10

SERVERS VIOLATING GOAL
1

SERVER 0

Benchmarks	Estimated performance
cpu (sysbench)	0.97
network (netperf)	0.99

SERVER 1

Benchmarks	Estimated performance
memsize (custom)	0.95
disk (hdparm)	0.91

SERVER 2

Benchmarks	Estimated performance
bw (mbw)	0.57

This is the main window on the left. In this use case, the cloud cluster consists of three servers. The first server has two guests (VMs), one running the CPU benchmark (sysbench), and another running the network benchmark (netperf). Below the VMs there is a table that the expected performance slowdown is calculated and colorized accordingly. Each server has a label on the top right corner that informs us if it is used, unused or if it violates the standards of quality of service. On the right, there is a sidebar with statistics that are useful for evaluating the cloud's total performance. On the top of the page there are some action buttons we created for controlling the content of the servers and for choosing between different schedulers.

3.7 Tests & results

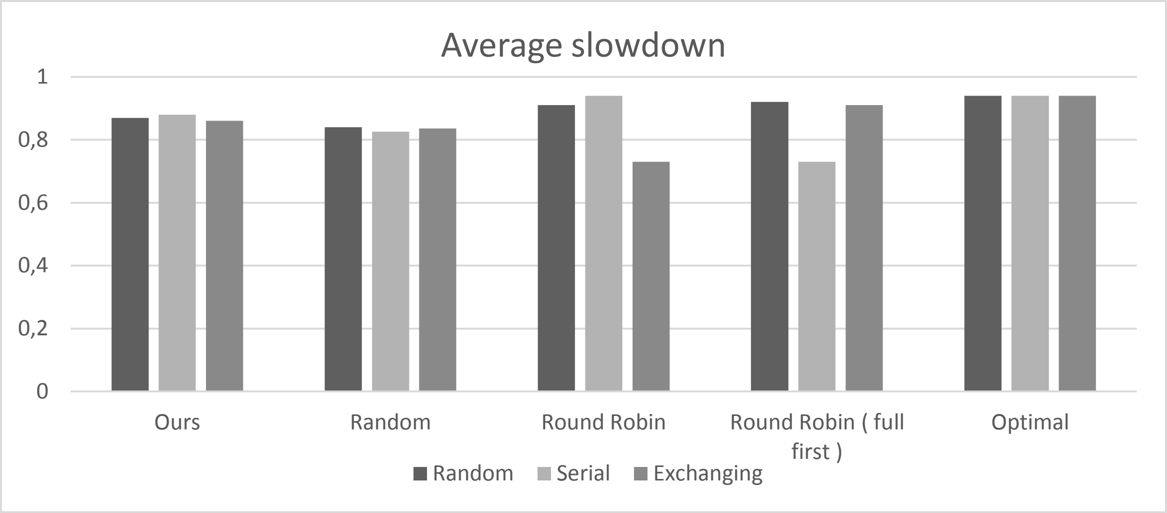
We emulated a cloud with 50 physical servers and 100 virtual machines to test the performance of the schedulers. We created three different test cases that we will analyze below. For each test case, we read the queue of virtual machines one by one and call the scheduler every time. For example, consider the following array as input:
[cpu, memory_bw, disk]
This means that there are 3 virtual machines that need to be hosted, arriving as input to the scheduler on that exact order. The following functions will be executed:
Schedule(cpu)
Schedule(memory_bw)
Schedule(disk)
Each call of the Schedule function (no matter which algorithm is selected to handle it) will return a destination server that the virtual machine will be hosted on.

The 3 different input types are the following:
Serial
In this case, there are 20 virtual machines of each application family that are coming sorted by their type:
[*cpu, cpu, cpu,… network, network,… memory_bw, memory_bw,… memory_size,…, disk, disk, …*]

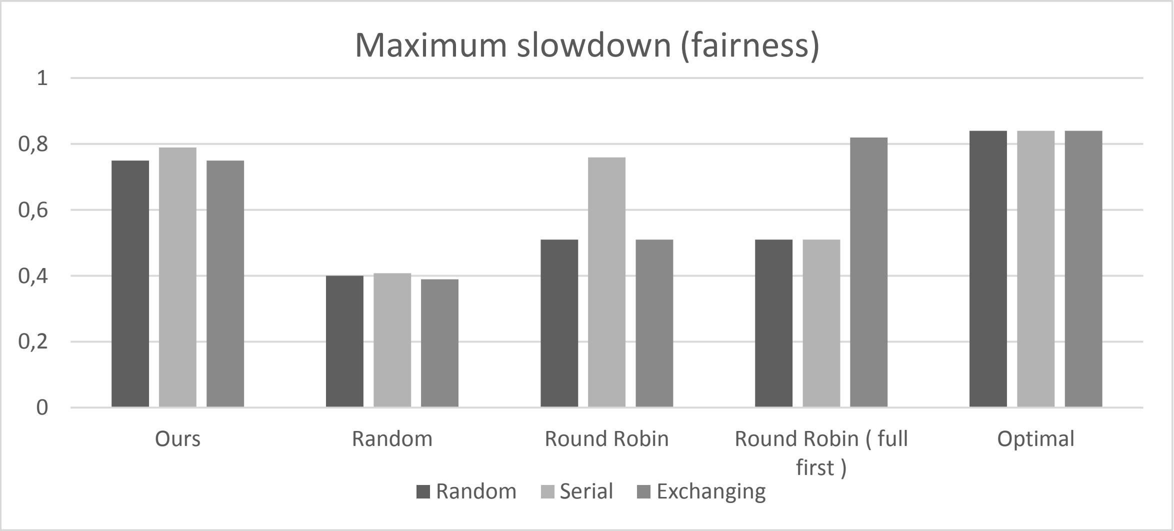
Exchanging
There are 20 virtual machines of each application family in this case too. This time the input changes in same order every time:
[*cpu, network, memory_bw, memory_size, disk, cpu, network, memory_bw, memory_size, disk, …*]

Random
A totally random array of VMs that is common for all schedulers.

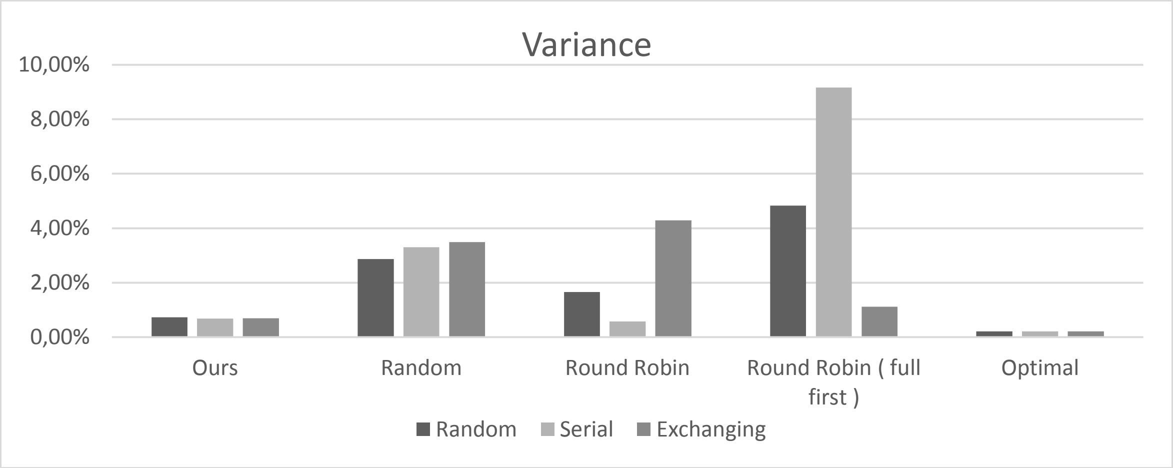
Graphs



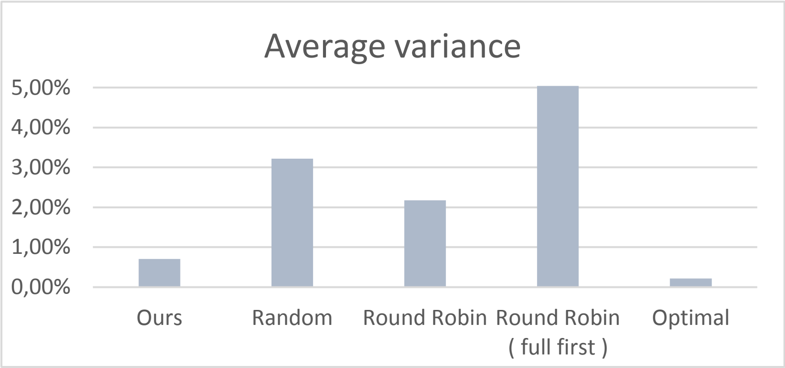
This graph shows the average performance of all the VMs after they are all placed in servers. The vertical axis shows the average performance of each scheduler and for all possible inputs (grouped by the schedulers).



The *maximum slowdown* is the first metric of a scheduler's *fairness*. The value of this metric is the worst performance of all the tests that ran in the servers. The bigger the maximum slowdown it is, the more fair the algorithm is.



The variance measures how far a set of numbers is spread out. As the variance grows, bigger differences occur between the virtual machines' performance. The smallest the variance is, the better fairness we achieve, because each application's performance will be near the average.



This graph shows the average variance for each scheduler, as it is calculated from all inputs.

3.8 Evaluation

The average slowdown graph shows no big differences between the different schedulers. The round robin techniques have the worst results, with big difference between different inputs. Our algorithm has the best results with all three inputs, and without big variance when the input changes. It also satisfies our initial goal of quality of service on all cases, by keeping the average slowdown above the value of 0.8 that we set as parameter. The random scheduler has similar results to ours, but slightly worse.

Our algorithm achieves to be very near the optimal values of *maximum slowdown*. This means that we achieved to satisfy our goals about the quality of service, as on both three cases the maximum slowdown is high than 0.75. This is a good indicator of an overall fair scheduler, because there are no big fluctuations between applications' performance. The other schedulers fail to be fair with any input, scoring an average maximum slowdown value by far less than ours.

Variance, is the most important indicator of fairness, as a low value means that all the applications have about the same amount of slowdown. Our algorithm achieves to have a very low variance that is very near the optimal value and by far lower than the other algorithms result.

Overall, we achieved to create an algorithm that respects the clients' needs by offering a quality of service, and we did not violate our starting goals on any case. Of course our algorithm will not be able to satisfy these standards for every case, as the client's applications start to be more intensive. If we have input with more disk-bound applications that cause more interference, the overall performance will start to descent. We study the behavior of all the schedulers when input is balanced between application types and there is no oversubscription. Additionally, we managed to create a very fair scheduler compared to the others, without sacrificing the overall average performance. So finally, we completely achieved our goals on this problem.

4 Problem two: Packing physical servers with Virtual Machines

4.1 Problem description

Moving on, we want to test our algorithm under more stressful situations. This time we will test the performance when the physical machines have as many virtual threads as their physical cores, and therefore the maximum utilization. Here is the problem's description:
Given N physical servers and M virtual machines we need to schedule the guests to the hosts so every server hosts as many virtual threads as its physical cores.

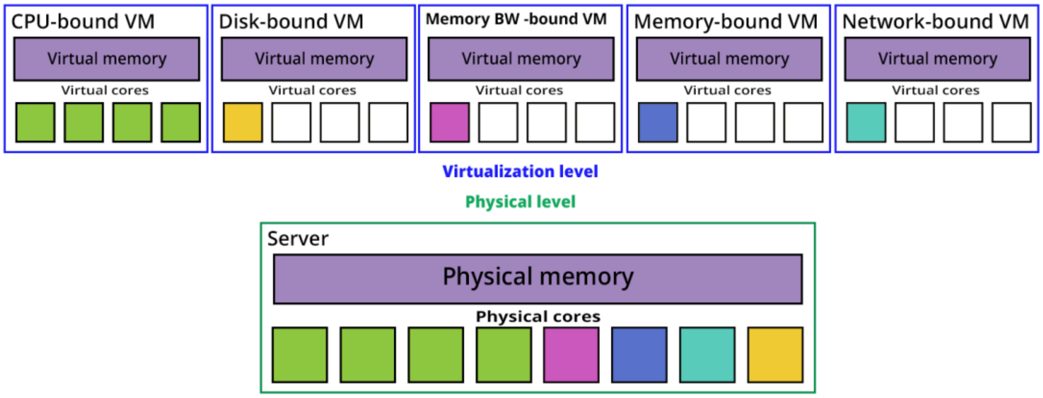
Contrary to the previous problem, this time we will need to place more than two virtual machines on each server. In this case we do not know how our applications will perform and interact between each other, as we have not measured more complex combinations than couples.

Having in mind that each application is single threaded, except the CPU benchmark that runs in 4 threads, then to fill up a server with 8 virtual threads we might need 8 different virtual machines (in case we do not have a CPU-bound one). This makes the complexity of the problem too big. It is impossible to measure every possible combination of applications, so we need to develop a prediction system.

Our goals it to use the interference map that we created in the previous problem in order to predict the expected slowdown of each virtual machine that runs alongside with more than one other VM.

4.2 Running model

To proceed with the development of the prediction system, we must understand what happens inside the server first. The servers have 8 CPU cores and 8 GB of memory as they had in the previous problem. Each virtual machine has 4 virtual CPU cores and 4 GB of memory. A virtual machine that hosts a single threaded application will keep busy only one virtual and physical core at a time, or so we suppose. For example, when we place a CPU application alongside 4 other types of benchmarks, we suppose that all the applications will run in parallel in the physical cores with a minimal interruption. The virtual cores that are not used will not cause any interference as they remain inactive.



The above schema shows what we believe that happens inside a server that hosts one guest from each application family. The active virtual threads are exactly equal to the number of the physical cores, so all the active threads can run in parallel. Some applications mostly perform input / output (IO) operations, so they may not remain in running mode during their execution. What is important about their performance is that the CPU has available cores whenever they need to return in running mode. The rest of the physical resources, such as memory, are equally shared too, when needed.

This time, interference between the applications is bigger and more complex, as everyone affects its “neighbors” in a different way each. We cannot know in which way each application affects the other. Every commonly used resource between two applications is a possible threat for slowdown. For example, a network benchmark can be affected from a CPU-bound by competing for processing time, and by a memory-bandwidth-bound application by competing for the memory data bus, causing a slowdown value each. If these three applications run in parallel, the network benchmark will be affected from both of its “neighbors” in two different ways, causing a slowdown that is a combination of the two different slowdown values.

Let’s consider a server that hosts one application of each family, named A, B, C, D and E. We want to guess what happens to the A-type application when we know what happens to it when it runs with each one of the rest. We already have this information from the interference map that we created in the previous problem. So let's assume that in our interference map we have the following row that provides us the information about what happens to application A when combined with others:

	B-type	C-type	D-type	E-type
A-type	X	Y	Z	W

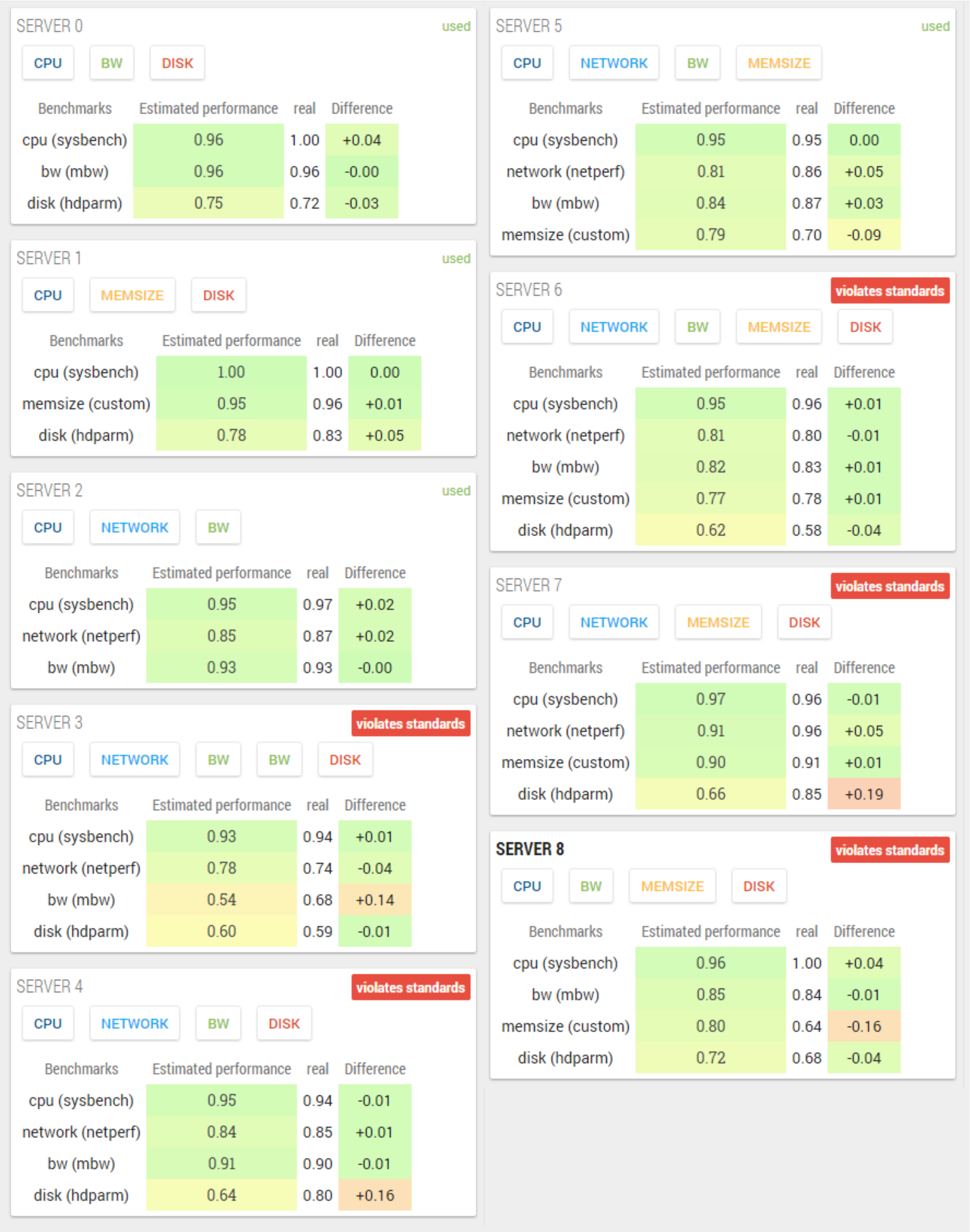
The worst case scenario about A-type application's performance, is that it gets affected from any other application in a completely independent way. This means that application B causes a slowdown equal to X, then the C application adds a slowdown equal to Y, and moving on with the rest of the application we end up having a total slowdown equal to X*Y*Z*W. Let's call this predicted value as ***worst***, in order to refer to it easier.

On the other hand, the best case scenario, is that application A is affected on the same way from each other. This could possibly happen if every application stresses only one physical resource, allowing greater independency between each other. Nevertheless, every application will use some CPU and memory resources, causing at least a minimal interference between each other. In this case, application will be slowed down no more than the minimum value between the different combinations of the above table, which will be the combination that causes the maximum interference. This time, let's call this performance expectancy for application A as ***best***, which is be equal to max(X, Y, Z, W).

At last, we believe that the real performance of the application we study, will finally be a value somewhere between the values of the two scenarios we defined above. Let's simply say the average of these two values. So we predict that the final performance expectancy of application A will be:

slowdown = (best + worst) / 2

In order to either accept or reject this prediction model, we ran some random combinations of VMs and measured their actual performance. Then we compared the real results with our predictions to see how close they were.



SERVER 0 used

CPU BW DISK

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.96	1.00	+0.04
bw (mbw)	0.96	0.96	-0.00
disk (hdparm)	0.75	0.72	-0.03

SERVER 1 used

CPU MEMSIZE DISK

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	1.00	1.00	0.00
memsize (custom)	0.95	0.96	+0.01
disk (hdparm)	0.78	0.83	+0.05

SERVER 2 used

CPU NETWORK BW

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.95	0.97	+0.02
network (netperf)	0.85	0.87	+0.02
bw (mbw)	0.93	0.93	-0.00

SERVER 3 violates standards

CPU NETWORK BW BW DISK

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.93	0.94	+0.01
network (netperf)	0.78	0.74	-0.04
bw (mbw)	0.54	0.68	+0.14
disk (hdparm)	0.60	0.59	-0.01

SERVER 4 violates standards

CPU NETWORK BW DISK

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.95	0.94	-0.01
network (netperf)	0.84	0.85	+0.01
bw (mbw)	0.91	0.90	-0.01
disk (hdparm)	0.64	0.80	+0.16

SERVER 5 used

CPU NETWORK BW MEMSIZE

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.95	0.95	0.00
network (netperf)	0.81	0.86	+0.05
bw (mbw)	0.84	0.87	+0.03
memsize (custom)	0.79	0.70	-0.09

SERVER 6 violates standards

CPU NETWORK BW MEMSIZE DISK

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.95	0.96	+0.01
network (netperf)	0.81	0.80	-0.01
bw (mbw)	0.82	0.83	+0.01
memsize (custom)	0.77	0.78	+0.01
disk (hdparm)	0.62	0.58	-0.04

SERVER 7 violates standards

CPU NETWORK MEMSIZE DISK

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.97	0.96	-0.01
network (netperf)	0.91	0.96	+0.05
memsize (custom)	0.90	0.91	+0.01
disk (hdparm)	0.66	0.85	+0.19

SERVER 8 violates standards

CPU BW MEMSIZE DISK

Benchmarks	Estimated performance	real	Difference
cpu (sysbench)	0.96	1.00	+0.04
bw (mbw)	0.85	0.84	-0.01
memsize (custom)	0.80	0.64	-0.16
disk (hdparm)	0.72	0.68	-0.04

The “difference” column indicated how close we got with our predictions and it is colored based on the difference between the prediction and the real result. Green colored difference means we got a more accurate prediction, while the red color means a less accurate one.

Overall, this model of prediction seems to have pretty accurate results, although there are some big deviations. Predicting each combination with a high accuracy is not an easy algorithm to implement and it might require a deeper and more detailed analysis of the problem. We do not necessarily need high accuracy predictions to proceed further. Ideally, we want our prediction model to have the ability to decide which combination will have the best results between a set of different possible ones. We do not need to know *how much better* that will be, but it is important to have the ability to recognize the good combinations over the bad ones, or if it possible to decide which is the best one.

The prediction model will be an important parameter for our algorithm. The final evaluation will judge if it was able to make good decisions and if it needs any improvements.

Below is the algorithm that predicts the interference between combinations of 3 or more VMs in the same server. It is written in pseudo-language.

```
algorithm predict_performace:
        input: server[ array of VMs ]
        output: predicted_delay // An array with the predicted performance for each VM
for each VM in server as i:
best = 1.0
        worst = 1.0

        for each VM in server as j:
                /* We avoid combining with itself */
                if ( i == j )
                        continue

                /* We add up the delay of each possible combination of VMs */
                worst = worst * combine( i, j ) // combine( i, j ) is the slowdown added to i by j

                if ( combine( i, j ) < best )
                        best = combine( i, j )

        predicted_delay[i] = (best + worst) / 2

return predicted_delay
end
```

4.3 Algorithm design

We will extend the previously designed algorithm for this problem. The main idea remains the same.

Our main goal is to preserve a quality of service but this time we can have up to 8 guests on one host, as we need to fulfill the physical machines' cores. This makes it more difficult to remain in the initial goal, especially after the third VM is added, because interference is becoming much bigger than it was on the previous problem. As soon as we violate the QoS values for the average and the maximum slowdown, the algorithm will continue on, choosing each time the server that will have the best performance after the VM that has been given as input is inserted there.

The algorithm written in pseudo-language:

```
algorithm place_vm:
        input: array of servers, vm /* to be added */
        output: destination_server
destination_server = -1 /* none */
max_performance = 0

for i from 0 to server.length do
        if  ( server[ i ] has the maximum number of VMs )
                continue
        else if ( server[ i ] is empty )
                destination_server = i
                return destination_server /* We choose this server and exit */

        /* This will return the array of scores that the server will have after we add the VM */
        scores = predict_performance( vm, server[ i ] )
        /* We check for average or distinct scores that violate our QoS */
        has_violations = check_for_violations_of_QoS( scores )

        if ( not has_violations )
                destination_server = i
                return destination_server /* We choose this server and exit */
        /* We always keep the best score so far, even if it violates standards of QoS */
        else if ( scores.average > max_performance )
                destination_server = i
                max_performance = scores.average
done
/* This happens only when every server is full */
if ( destination_server == -1 )
        dismiss vm and exit
else
        return destination_server
end
```

This problem is more difficult version of the previous one. We will compete the same algorithms again, but this time, it is impossible to search for the optimal combinations. Problem's size is exponentially bigger. If an equal number of VM from each family is given as input, then the servers will host one VM from each family on average, in order to fulfill their physical cores. This means that we have about 5N guests to schedule to N hosts. Each guest has N possible options that need to be evaluated by our algorithm, so its complexity will be $O(5N^2)$.

Choosing a guest's destination needs to be a very quick task. When a client's application is ready to start execution we should be able to provide the immediate start. A late placement can introduce *latency* on our services, negatively affecting our quality of service. The search of the optimal solution would introduce the following problems:

1. To calculate the optimal combination of the guests we need an algorithm that will migrate the guests to another host when a better set-up is discovered. We have chosen to avoid migrations on our implementation, since they add an extra latency on runtime that we cannot measure.
2. On every new guest given as input, we will need to re-calculate the optimal solution that might differ a lot from the previous one. This means that we might need to migrate VMs on every step of the scheduler's process, which is something unwanted that will add too much latency on the guests' performance.

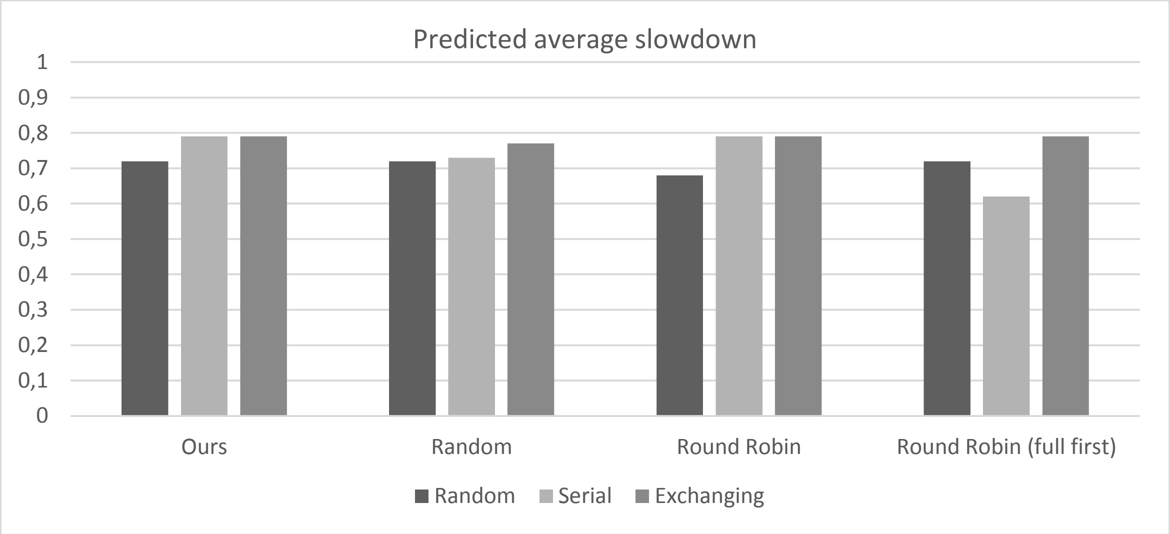
As mentioned before, the scheduler needs to be deterministic and *quick*. The complexity of our scheduler allows it to run quickly even with a high number of servers. We need to consider that the rest of the schedulers (random and round-robin) make their decisions immediately, and this is part of the competition and the quality of service too. A further analysis is made on chapter 7.

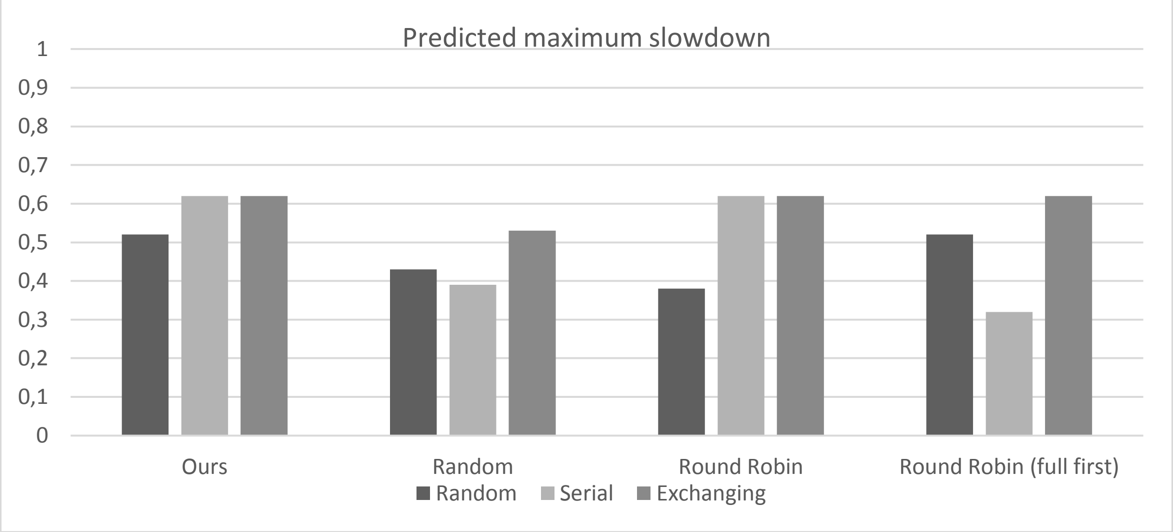
4.4 Tests & results

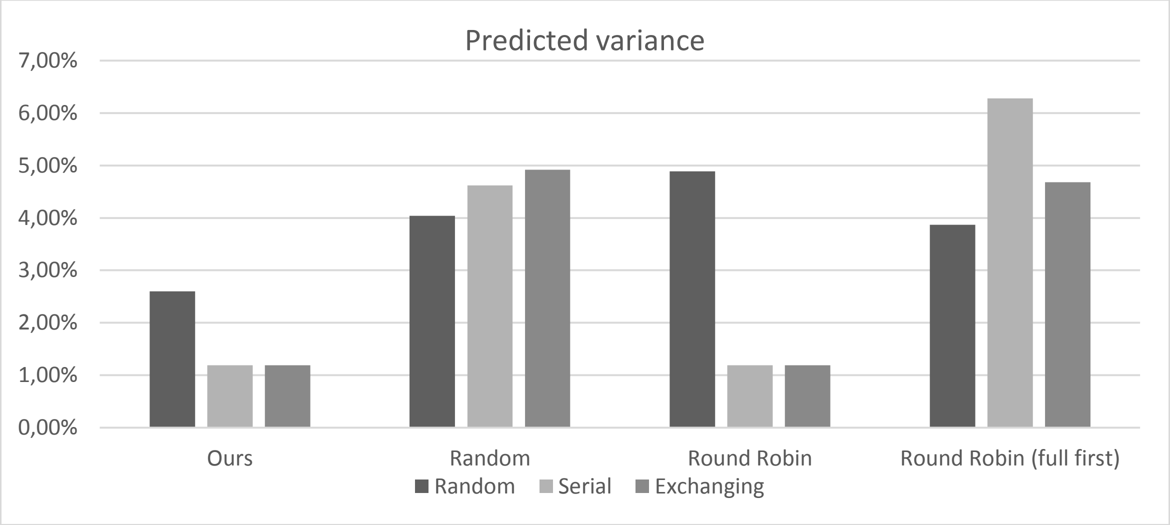
In this problem, we emulated a cloud with 3 physical servers and 15 virtual machines that need to be scheduled. We kept the same input tests we had on the previous problem: random, serial and exchanging.

In the previous problem, every server had two guests, which means that we already knew how they will react and interfere. This time, every combination that is created by the scheduler needs to be really executed so we receive the real results and compare them with the predicted ones. The prediction system has to be evaluated too.

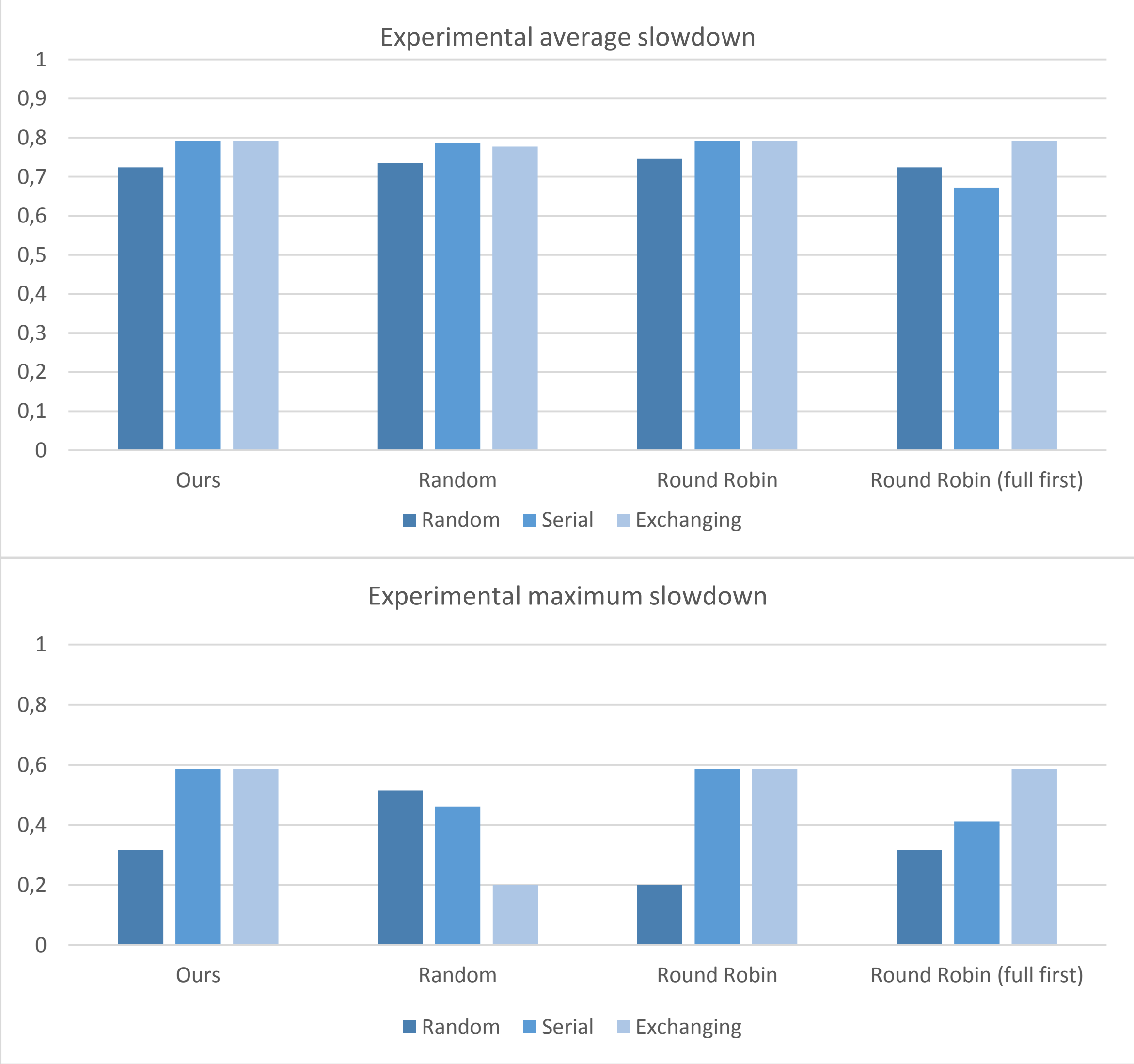
Predicted results

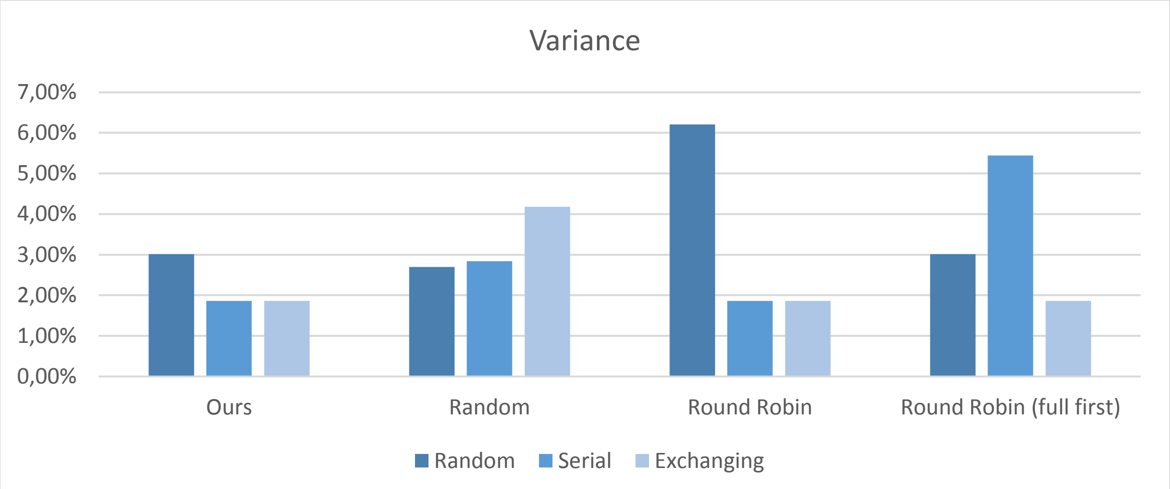


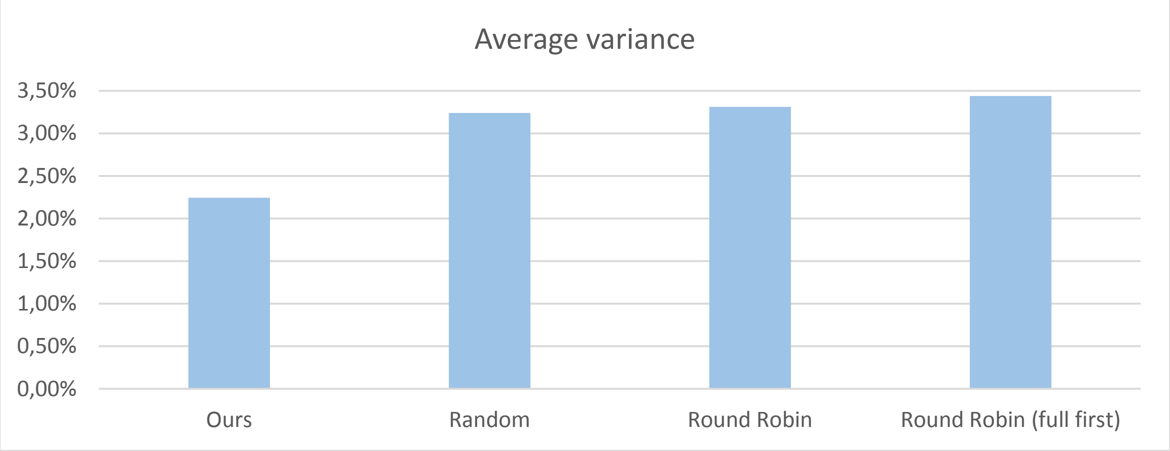




Real results (experimental)



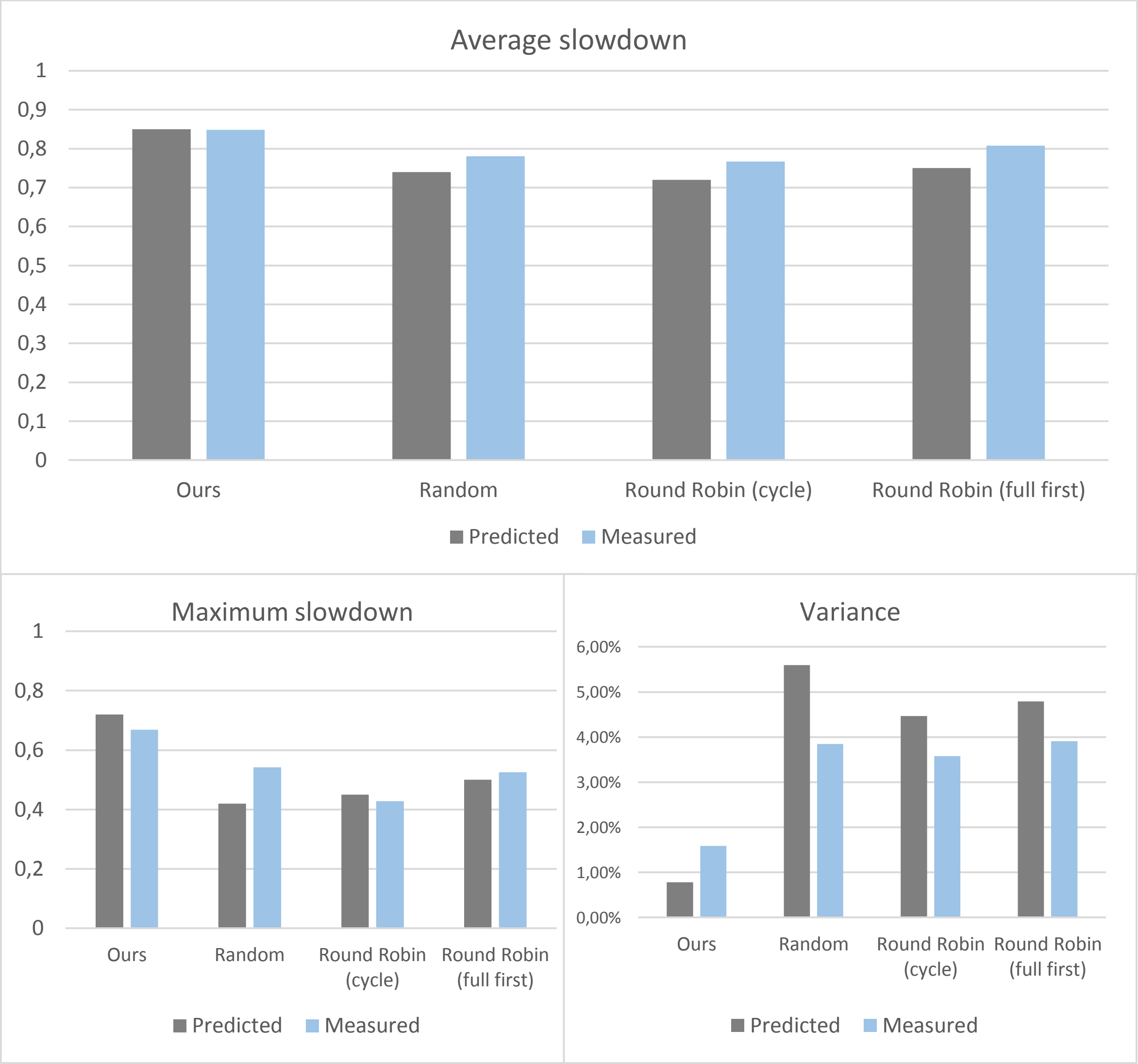




Those graph do not reveal a great improvement of performance with our scheduler, so we wanted to try another input, without balanced numbers of VMs from each family. We created the following input array:

[cpu, network, disk, cpu, memory_bandwidth, disk, cpu, memory_size, memory_bandwidth, memory_bandwidth, memory_size]

These virtual machines have not enough virtual threads to fulfill the physical cores of each server. So it can be consider a problem with lower difficulty. We want to examine what happens if we stress the cloud a little bit less than before. This test has less network-bound guests and more CPU and memory-bandwidth-bound ones so we see how the different schedulers react to an unbalanced input.



4.5 Evaluation

According to the prediction system, we expected the average performance of the guests to be slightly better or at least equal to the other schedulers' results. We expected a bit better performance on average of all input tests. Maximum slowdown is expected to be better on average too, while variance will be greatly lower on every case.

In the experimental results, numbers do not quite match exactly the predictions. The prediction system is responsible for this. Our prediction model is quite primitive. What we did, was to define the limits of the performance of each application, and arbitrary, predict that the application's performance will be in the average of the two limits. This might work well sometimes, but the real interference that causes the applications to slow down is much more complex and difficult to be predicted. We propose some ideas for a better prediction system on the "future extensions" section.

Let's ignore the last test for now.

The average performance of the guests does not fluctuate a lot between the different schedulers. Our algorithm failed to achieve better results, even at minimum, as we had predicted before the experiments. On the other hand, it does not have worse results that the other schedulers either. The problem's difficulty adds some performance limits, but without an optimal combination calculation we do not know how close we got to those.

Although on average we failed to make a noticeable improvement, we still need to evaluate our fairness. The maximum slowdown (or equally, the minimum performance) has an average value slightly bigger (and better) that the others, but not enough to become noticeable from the client's perspective. We also had a lower average variance, which is the most important fairness metric. The difference between our variance percentage and the rest of the schedulers' is not enough to advertise it as a success, but it proves that our algorithm still respects that need, even if it cannot maintain the initial goal.

The quality of service in this case is almost always violated, so the algorithm just ignores it after a specific point, choosing the best server on every step. We could possibly lower down the QoS values, in order to try to maintain some standards in our guests' performance, even if these are not good enough. It is important to be able to guarantee to our clients the minimum expected performance. If we set the QoS variables at a very low value, then our scheduler would act like the round robin that fulfils each server serially, as it would keep adding the incoming virtual machines to the first server that would have free space since it does not violate these low standards. Since we kept the values that we had set at the previous problem, the algorithm will most probably place couples of VMs on each server, and after that it will start to fail finding server that do not violate the QoS, so it will simply choose the best option on each step.

Moving on to the last input, we want to see what changed when a less stressful input was provided. First of all, the average slowdown of our scheduler has an important different from the others, achieving the highest score. Once again, the experimental results differ from the predicting ones, confirming that the prediction model needs improvements. We also managed to be fairer again this time, having a greatly higher value of the minimum performance, both on predictions and experiments. Variance is incomparable better too, concluding that we were overall fairer than our competitors. Keeping in mind that this problem is categorized somewhere between those two first problems, based on the difficulty and the number of virtual tasks, we cannot claim that we scored better on this problem, but that we make it to be more fair when the physical limits allow it. We perform a more detailed analysis and evaluation of our algorithm on chapter 7.

5 Problem three: Oversubscription

5.1 Problem description

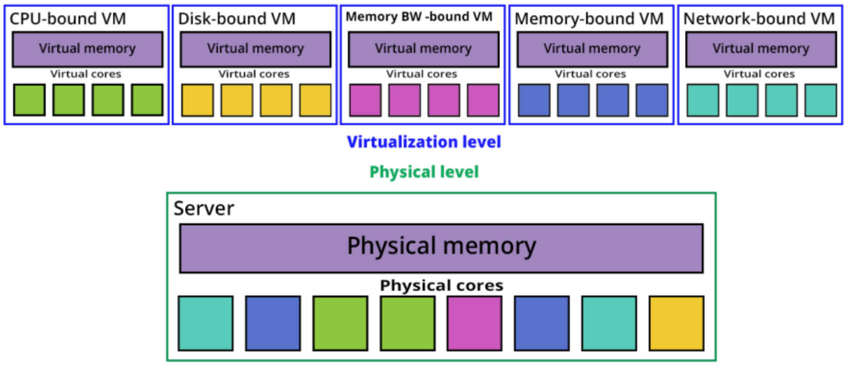
To understand how our scheduler works and whether it provides better results than its competitors, we have to test it with a variety of different kinds of inputs. We started from the easier ones and we keep moving to more and more difficult. This time, we want to see what happens when the virtual cores that the guests use surpass the physical ones, meaning that the server will be oversubscribed.

In this problem we used the second server, due to some technical issues, and because it is more powerful. As the problem get more difficult and stressful for the physical machines, a more powerful server will help us save time. The virtual machines have the same virtual cores (4) and memory (4GB), but this time we used the *virtio* drivers (part of the KVM module) for the virtual hard drive to enhance the guest's performance.

Our first try is to create test cases with virtual threads twice the number of the physical cores. We need to have 16 virtual threads on each server, which is a big amount of VMs that need to be spawned considering that only the CPU-bound ones run multithreaded. It would be convenient for us to run each benchmark in 4 threads, so we achieve better utilization with a smaller number of guests. To achieve this, we made a major change this time. On every guest that is not CPU-bound, we created 4 instances of the benchmark they run. This means we had to rerun every benchmark family in this new environment and having 4 instances of each benchmark - except the CPU one which was already multithreaded - so we have a new comparison basis to calculate the slowdown from now on. This means that we are not able to compare results of this problem directly with those of the previous problems, but only compare the results of the different schedulers to each other.

5.2 Running model

We now have full utilization of all virtual cores and also an equal number of virtual threads from each VM, no matter what benchmark is executed inside. Because of the oversubscription, the virtual threads cannot run all together in parallel, but only the half of them will be running at every moment, while the rest will wait their turn for CPU-time.

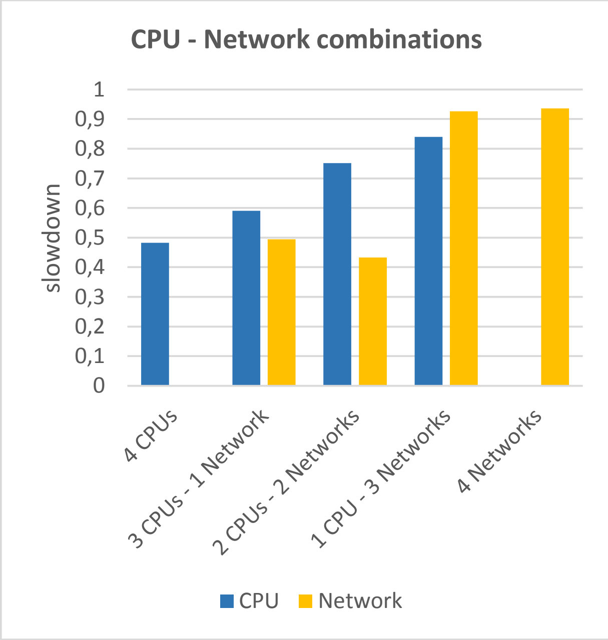


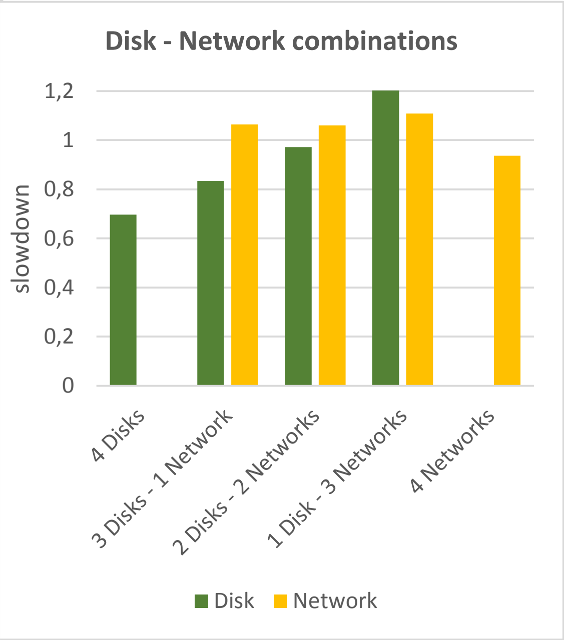
Interference keeps getting more complex as the problems get more difficult. Our previously deployed prediction model is clearly irrelevant with this running model. That model could give an estimation of performance when all of the virtual processes shared the CPU without interruptions, or at least without serious interruptions caused from other applications than the benchmarks. This time something very different happens. Combinations of applications that run together at a time are formed in a completely random way and they keep changing very quickly as the server's scheduler will recycle the CPU space between the virtual processes that compete each other. While the virtual tasks start and stop execution, the interference keeps changing with the same frequency.

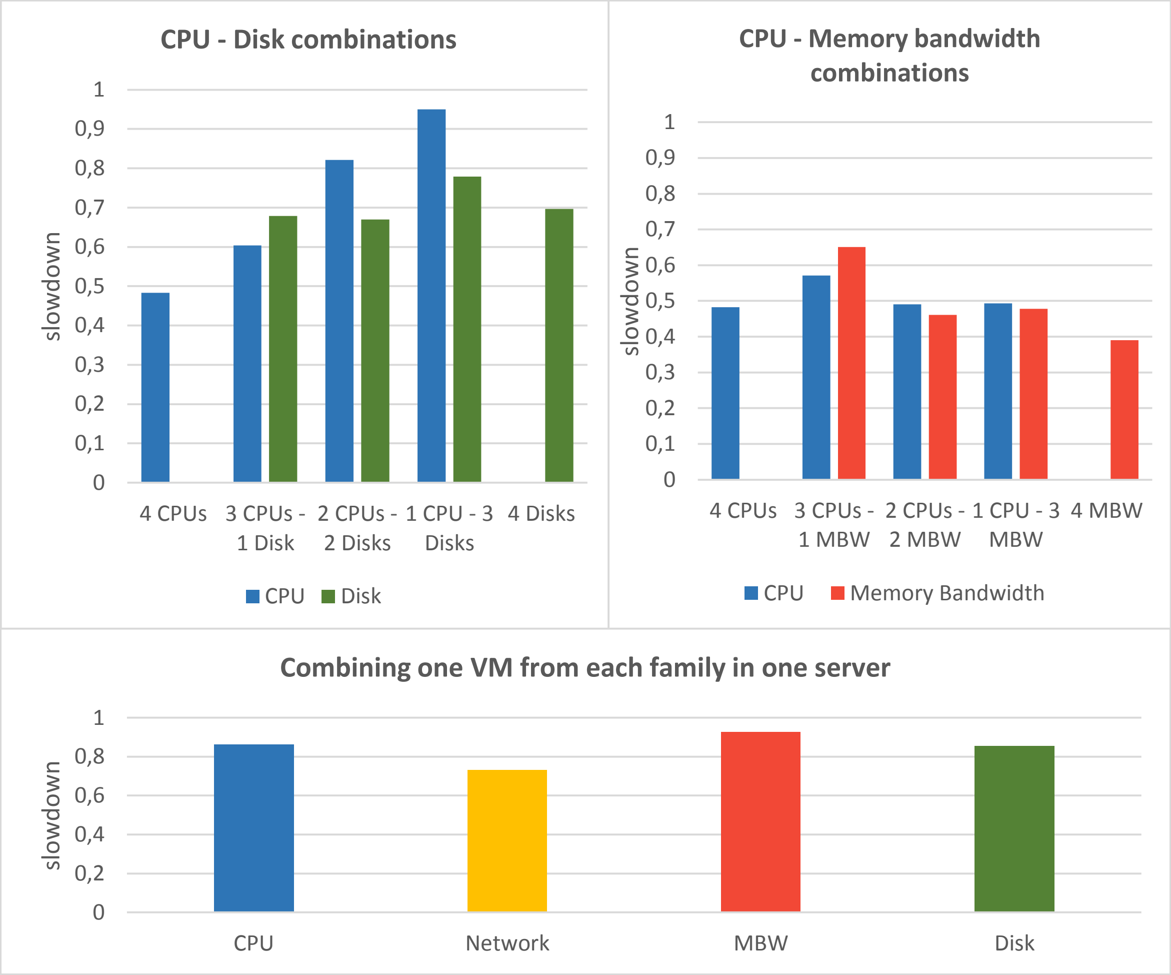
A very primitive practice to develop a new prediction model, could be the following: form every possible combination of the virtual processes that can run along in the CPU at a specific time, predict their performance depending on the previous problem's prediction model, and then calculate the average slowdown for each benchmark family. This is based to the idea that each one of the possible combinations has the same probability to be formed within the server's CPU. Depending on that fact, and assuming the previous prediction model was accurate enough, we could predict the estimated average slowdown. But this is not going to work well. First of all, the previous prediction model was not accurate enough to predict results and produce a major improvement of quality of service. This will affect the results of this hypothetical predictor in a negative way. Additionally, it is a blind guess to assume that all possible combinations between the virtual tasks can form with the same probability. For example, an operating system that uses a gang scheduling algorithm will schedule tasks from the same virtual machine (and therefore the same system process) together with a higher probability, without being complete random.

5.3 Example tests & algorithm design

Instead of developing a prediction system, we went on and began testing some very basic combinations of VMs that will help us understand what is happening. We ignored the memory size benchmark at this first testing, in order to focus on how the CPU benchmark behaves when combined with the input/output benchmarks. Both memory bandwidth, disk and network benchmarks perform input/output operations to a specific computer's resource. We started by combining CPU with these three benchmarks, in every possible ratio:







Let's try to expound the graphs.

First of all, we see that the CPU benchmark has about a 50% slowdown when there are four VMs running it alongside. This is exactly what we expected, as the virtual threads are exactly twice as the physical cores of the system. When we combined CPU benchmark with the disk and network ones, the less CPU VMs we had, the better their performance was. This can be easily explained, as the less CPU-bound VMs we have, the less virtual processes compete for CPU time. Both disk and network benchmarks are input/output bound, and they do not stress the processor, so they affect the CPU benchmark less that it does to itself.

The CPU-Network combination graph reveals that network benchmark is affected by the lack of CPU time. The best combination between these two types, is when we have 1 CPU and 3 network bound VMs. Network has less self-interference than the CPU. This is why the combination of 4 network-bound VMs has a good total performance.

The Disk-Network combination graph reveals some very good combinations. 2 disks and 2 networks have an average performance near 1.0, meaning almost no slowdown at all, and the combination of 1 disk and 3 network reveals an overall performance boost! The fact that network benchmark is improved when there two other VMs that run the same one, can be explained with better network utilization. Given the fact that there are 4 hdparm threads running in parallel in each disk VM and they are all using the same disk with virtio drivers, we can conclude that the interference is caused between disk VMs that saturate the number of concurrent IO workers on the physical disk. We can assume that the number of spindles is between 4 and 8.

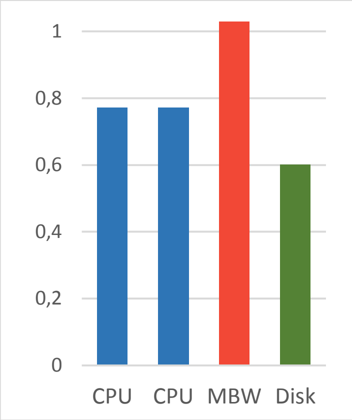
The CPU-Disk combination graph shows the same behavior pattern with the CPU-Network combination, having the same best case scenario when we have 1 CPU guest.

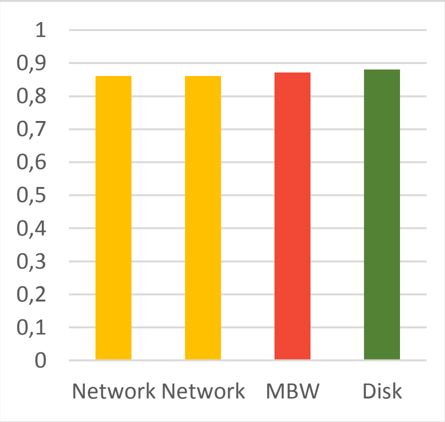
The CPU-Memory bandwidth combination graph informs us that this is a bad mixing at any ratio, having very bad performance on all cases.

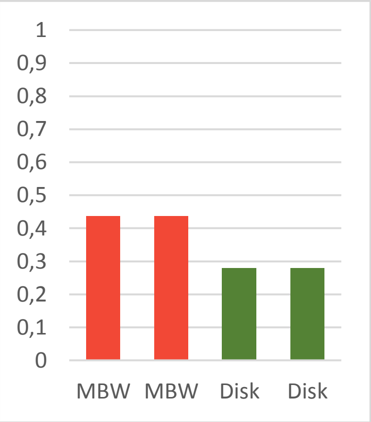
The combination of the four different families of benchmarks graph reveals a good average performance. This is very useful information for creating a scheduling algorithm for this problem.

We have some very basic and useful information on our hands about how different benchmarks behave and interact with each other, but not enough to come up with a scheduling plan that will guarantee us good results. We decided to run some more combination tests to have a better understanding of the overall behavior patterns.

Each graph from those below presents a different combination:







We will use those results as base of knowledge, as we will try to create a scheduler for this problem.

After observing the additional combination graphs, we can create some basic rules that we can follow to either have a better performance or avoid the worse combinations. Below we collect and sum up those rules:

Try	**Avoid**
Combine different types of VMs	CPU combined with Memory bandwidth
Combine Disk with Network VMs	Two or more CPU-bound VMs
Combine many Network-bound VMs	Two or more MBW-bound VMs

We do not want to deploy a full prediction system, as we did in the past problem. As we have already mentioned, prediction is a very difficult task in a complex running model like the one we have here. We simplified our algorithm design by following the behavior patterns of our tests, and turning the above rules into a scheduling algorithm. Of course these rules must be ranked in a series of significance, so the scheduler will try to satisfy all rules, starting from the most important and moving on to the next rule as long as it fails. Let's sum up those rules in order:

1. Combine different types of VMs (place the VM on a server that does not include already one of this type)
2. No more than two CPU-bound VMs in one server
3. No more than two memory-bandwidth-bound VMs in one server
4. Do not combine CPU with memory bandwidth
5. Place disk and network VMs together
6. Place many network VMs together

We hope that the behavior patterns we noted on the tests we made will be followed on the average scenario, so our scheduler will lead to better results. The possible combinations of VMs on this problem is huge, so we cannot explore those behavior patterns deeply. We ran some basic combinations and we achieved to generalize the behavior patterns on a higher scale. The success of the scheduler will judge how well or bad aimed our conclusions were.

5.4 Results

We created 3 different input cases with 16 guest machines each. In all cases, the guests arrive sorted by their category, meaning that first will arrive all the CPU ones, then all the network ones, and so on.

Before we proceed with the results, we need to describe our competitors once again. We kept the same schedulers again, but this time we renamed them so they are recognized by the patterns they create within the physical machines:

Packed

This is the round robin algorithm that first fulfills each host. Combined with a sorted-by-category input, the result will be that each host will have as many guests of one family as possible. It seems a bad idea, but we still need to see its results.

Separable

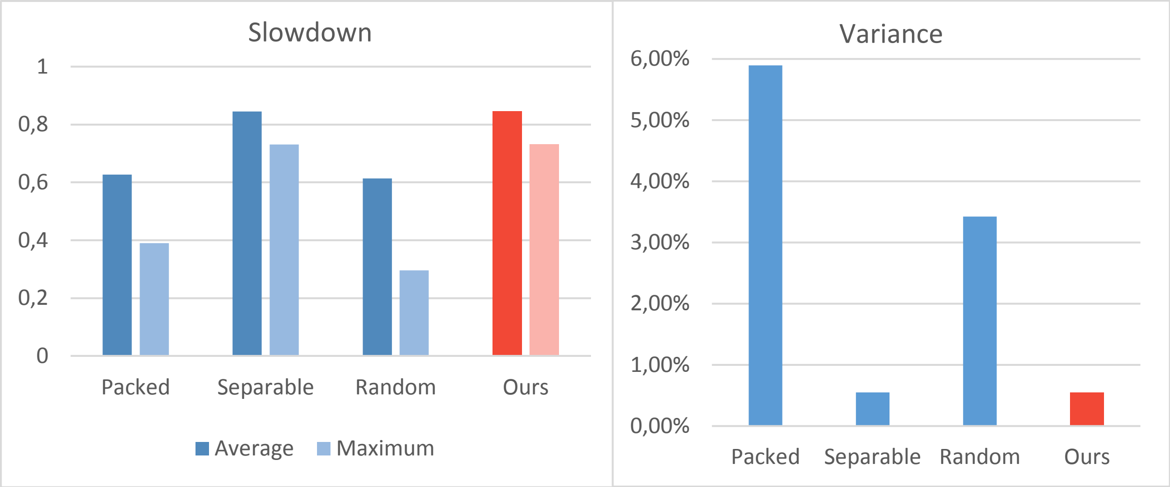
This is the classic round robin algorithm. Combined with the sorted-by-category input it will result a separation of same VMs to different servers, depending on the ratio of the different guests.

Random

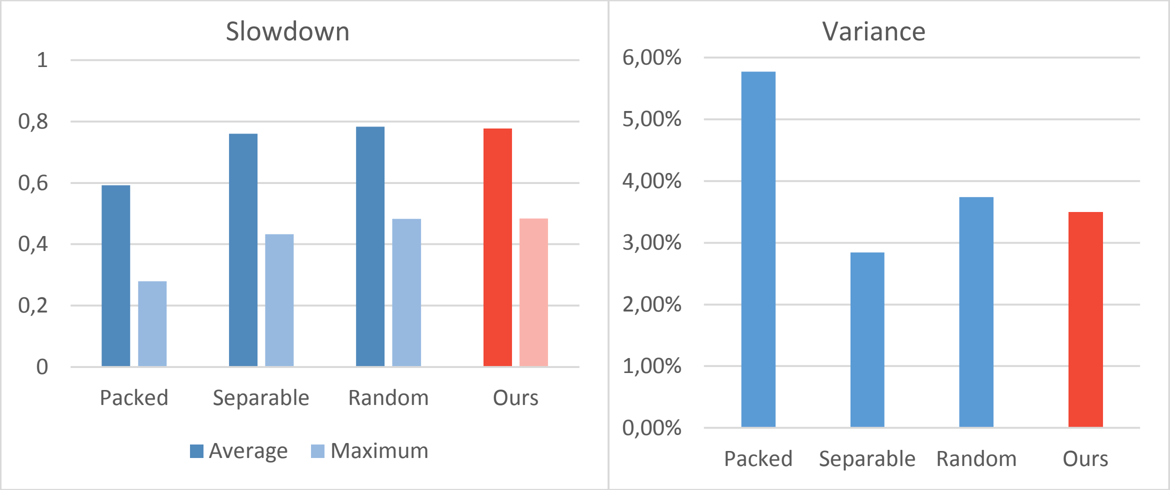
A completely random scheduler, with random results. It does not create any patterns at all.

So let's proceed with the test cases and the results. We marked the results of our algorithm with red to be clearer.

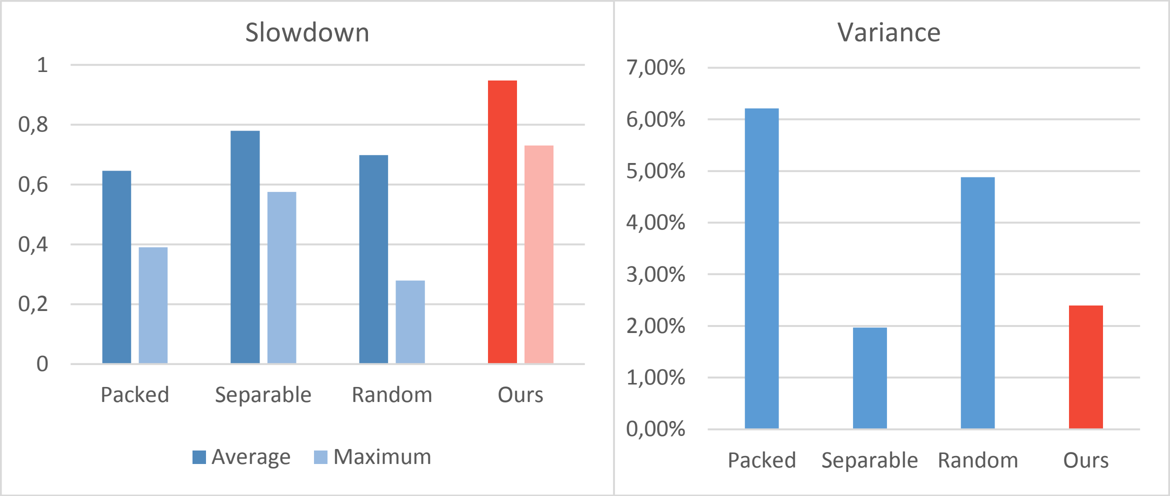
Test case one: 4 VMs of each one of the four categories



Test case two: 6 CPU, 6 network, 2 memory bandwidth, 2 disk



Test case three: 4 CPU, 2 network, 6 disk, 4 memory bandwidth



5.5 Evaluation

The *packed technique* was proven to be a very bad idea, as we expected, having both low performance and big variance. Our algorithm seems to behave steadily, avoiding performance fluctuations. The *separable technique* has mostly similar results, as it will create similar combinations to the servers, but in the last case we surpass both its performance and variance. The *random scheduler* results an overall average performance worse than ours, with a big variance too, but it is not too bad, like the packed one. We also notice that the *maximum slowdown* value of our algorithm is the best on both three cases, which is a very important metric of fairness. Once again we made it to be fairer than our competitors, and also have a stable behavior with different inputs.

At the beginning of the tests, we mentioned that four CPU-bound VMs running alongside result an average performance of 0.5, meaning that the performance of each benchmark is exactly the half of what is was when it ran isolated. Some test have an average performance much greater than 0.5, even if we have a number of virtual threads twice the number of the physical cores. This strengthens our theory that different types of applications can run with high independency and a low interference, as they occupy different physical resources. The processor is not stressed from the input-output-bound applications, allowing them to run alongside with a small slowdown caused by resource sharing.

The important part of the results is not the difference we made. The problem's size is huge and we cannot be sure that this behavior will be met on the average case. What is the most important information from those experiments is that we have a *serious profit margin*. Although the problem got harder, and the stress of the physical machine grew accordingly, we still have a chance to optimize performance and fairness. Our scheduler is still primitive and it might need serious improvements to have a stable behavior and a stable performance gain, but it is still a very good start to be based on. We suggest some possible extensions of our scheduler on the "future extensions" paragraph of the next chapter.

6 Conclusions

6.1 Total evaluation

Having in mind the goals we set on the beginning of this thesis, as we mentioned them on the first chapter, under the ”goal” paragraph, we want to evaluate what we achieved by our experiments.

The first and most important goal was to prove that a resource-aware scheduler can lead to a better average performance. During the tests we competed to some classic schedulers which take decisions being unaware of the resources a guest uses. In most of the experiments we made we had better results than our competitors’, and when we failed to be better, we had approximately similar slowdown. This pretty much proves our initial theory that scheduling based on the resources each guest stresses will create more efficient guest combinations.

We need to examine the results from different perspectives to have a full image of what we managed to achieve and whether we did it good enough to satisfy our goals.

Average performance

The simplest metric of our scheduler’s efficiency and possibly the most important. We did not manage to make a big difference in most of the problems, but we also did not have much worse values. If we compare our results with the best one of the rest of the schedulers, we notice that in most cases we had only a little performance gain, which is a very good sign.

Fairness (maximum slowdown & variance)

When we implemented our scheduler in both three problems, we wanted it to be *fair*. We are judging fairness of the schedulers by two metrics: maximum slowdown and variance. We managed to have a very slow value of variance throughout all the experiments, and in most cases it was much lower than the rest of the schedulers. This fact confirms that our scheduler was fairer and more respectful to the client’s performance. A lower variance value means that different clients will have less performance difference between each other, which is a very important parameter in modern clouds. The maximum slowdown revealed much higher values for our scheduler on the first problem, and a similar behavior like the average slowdown on the other two. So as a metric of fairness, maximum slowdown was useful to prove the superiority of our algorithm on the first problem.

Stability

Our algorithm did not manage to make big difference in average performance values mainly because we compared it with *the best* of our competitors each time. If we compare each metric of efficiency with each one of the other schedulers alone, we will notice that we have a much more stable behavior. While the other algorithms have a very different behavior with a different input, we manage to have similar behavior when the input changes. The results of the round-robin schedulers are based on the order that the guests arrive. The random scheduler creates very different combinations each time it is called, even with the exact same input, and this is why it has very big variance values. On the other hand, our scheduler manages to reserve good results even with much different inputs. This is a very important characteristic, because it allows us to predict our results based on the size of the input and we can avoid big and unwanted fluctuations on performance.

6.2 Future extensions

The purposes of the thesis were to prove that a resource-aware scheduler for a cloud cluster can enhance its total performance and boost the quality of service without additional physical resources. As we mentioned before, this is important for a cloud service provider in order to increase the service quality-to-cost ratio. We analyzed the performance of the system we developed in the previous paragraph, but we also need to examine the limits of the scheduling system and its possible extensions, so it can become cleverer and more efficient. Below are some possible extensions that can be attached to what we already developed:

Migration support & optimal scheduler

The simplest but really important feature that needs to be added is migration support. As we mentioned at the beginning of the experiments, we did not use migrations of guests along the servers as it requires additional software, but most importantly, because it would add an overhead that would be impossible to measure in terms of performance. While guests were arriving in the cloud cluster our scheduler placed them to the best server *without moving those guests that have been already placed*. While the number of the guests grows, the optimal combination might differ a lot from what we have formed, but since migration was not supported, there was no point searching for this optimal combination. Support for migration, combined with an algorithm that searches for better combinations given the set of the guests that exist in the cloud as input, but without restrictions in moving them around, could lead to better results.

It is important to mention that migration has a serious penalty, since it is a task that requires some time that cannot be ignored when calculating a guest's performance. So migrating the guests cannot take place every time a new VM arrives and a new better combination of the guests is discovered. We should introduce a new feature to our algorithm. We want to search for better combinations whenever the overall performance starts getting *too bad.* We could tolerate performance penalty until a specific point, and when this is violated, we could start searching for better combinations every time a new guest arrives. If we find what we searched for, we should decide if it is worth to try moving the guests around to create the desired setup. If the new setup provides only a small performance boost, it might not worth it. The same will happen if it requires lots of migrations, because the penalty of migrations might balance the performance gain. We believe that this algorithm could lead to better results, especially when we have many guests like we had at the second problem, where we failed to make a big difference from our competitors.

Better prediction system

During the second problem we experienced unexpected performance in experiments that caused our algorithm to have results close to its competitors' ones. We mentioned that the prediction system should be able to recognize which is the best combination between a set of guests. To achieve this, we can try to create a technique that provides more accurate predictions for the expected performance, but this is most probably a very difficult and complex task to deal with. We could make a more clever prediction system by adding memory to it. While the clients run in various different combinations, the scheduler can get information about their performance. This might require some software installation on client's side, so the client can inform the server about its slowdown, but we do not want to engage with the design issues on this thesis. This information will be stored to a database, so the next time that the prediction system will be called to predict a previously created combination of VMs, it will no longer take a guess, but it will provide the experimental measurements. This will allow our prediction system to get wiser as the time passes and it keeps storing useful information. The more information it stores, the more accurate it will be in the future, allowing us to make the best decisions between the possible combinations. While the database grows and our knowledge expands, we can also make improvements to our prediction model's mathematical formula, based on the results.

Profiler

We have already described why we chose to experiment with benchmarks instead of some more common applications. In a real cloud system, the average client's needs are not as intensive as the ones of the benchmarks we chose to run. The majority of applications use most of the computer's resources, so they won't behave like our benchmarks did. In order to find out what resources an application (or a VM) uses, we need to use *profiling software*.

A profiler is a program that performs analysis to another program, monitors the resources it uses and recognizes its behavior patterns. We can use a profiling software that analyzes the virtual machines that run on a server and provides us information about the resources it uses. We can either categorize VMs to the one family from those we have defined, based on the resource that the VM stresses the most, or categorize it to more than one family with a percentage of relevance. For example, an application can act 80% of the time as a CPU-bound process, and 20% as a disk-bound process. We should also extend our scheduler's abilities so it can work with VMs that act varyingly, which is tough and complex task, but it still is a perspective that can boost the system's flexibility and efficiency.

Overall, there are many possible ways to extend our system, even in different directions and with different approaches. The main idea remains the same: clients are treated with respect to whatever resources they need. What we developed was a primitive version of a system that satisfies this need. Different cloud systems, with different characteristics might lead to a very different behavior and results when our scheduler is used. Additionally, different kinds of guest applications can lead to unpredictable results.

The most imperative need for this system is intelligence. An intelligent system can adapt at different cloud clusters, study their behavior, and configure its decision making system accordingly. A system like this can be called *plug-n-play*. For example, the interference map that we used for the first two problems changes when we move to a new cloud. The scheduling system should be able to recalculate it on a new environment. An intelligent system can also have memory and profiling tools, as we have already proposed. As a result, intelligence will add elasticity to it, being platform-independent, providing better results while it keeps getting wiser and also being able to work with any client.

The results of this thesis and the studies about interference between different types of applications can have various uses. For example, as long as we have spotted some very good combinations during our experiments that have a minimal interference between each other, we can suggest those to a cloud provider for a long-term use. Imagine a cloud service provider that hosts 2 different services, one CPU-bound and one network-bound service. If we have ensured that those two types of applications have a minimal interference, we can suggest to this cloud provider to use the same physical machines to host both of the services on a permanent basis. This will allow a near-optimal performance without any additional servers at all.

What we have developed already is just the core of an intelligent resource-aware scheduling system that can be extended in multiple ways, depending on our needs. In this thesis we intended to prove that a system like this can lead to better results. As long as we were encouraged by our results, we can continue the development with a new direction, creating a system that provides efficient scheduling in cloud environments.

7 Appendix

7.1 Code

Python script used for memory size benchmark:

```
#!/usr/bin/python
import random
import time
import hashlib

memstep = 128 #in Megabytes
pagesize = 4096

#start of execution
start = time.time()

#The huge string is initialy empty
huge_str = ''

while True:
        try:
                huge_str += bytearray( memstep * 1000000 )
        except MemoryError:
                break

memsize = len( huge_str )

print "Allocated %d MBs in %.2f seconds" % ( memsize/1000000 , time.time() - start )

edit_start = time.time()

times = 4

hashes = 0

#We loop for "times" times
for j in range(0, pagesize, pagesize/times):
        cycle = time.time()

        #Every loop edits every page once, so our step is the pagesize
        for i in range(0, memsize, pagesize):
                #A standard random edit, one character long
                huge_str[ i + j ] = str( random.choice( 'abcdefghijklmno' ) )

                #We want some hashes to
                if ( i % (memsize/pagesize)/8 == 0 ):
                        hash_t = time.time()
```

```
            m = hashlib.sha1()
            m.update( random.choice( 'abdcefghijklmnopqrtsuvxyz' ) )
            hash = m.hexdigest()
            hashes += 1
            print( "\tHash %d, got %.4f seconds" % (hashes, time.time() - hash_t) )

            huge_str[ i : m.digest_size ] = hash


    print( "Cycle time: %.2f seconds" % (time.time() - cycle) )

print( "Edited every single page within %.2f seconds" % (time.time() - edit_start) )
print( "Total execution time: %.2f seconds" % (time.time() - start) )
```

7.2 Bibliography

[1] The NIST definition of cloud computing, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[2] Cloud computing, Wikipedia.org, https://en.wikipedia.org/wiki/Cloud_computing

[3] Supercomputer, Wikipedia.org, https://en.wikipedia.org/wiki/Supercomputer

[4] Virtualization, Wikipedia.org, https://en.wikipedia.org/wiki/Virtualization

[5] Centralized computing, Wikipedia.org, https://en.wikipedia.org/wiki/Centralized_computing

[6] DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments, http://infoscience.epfl.ch/record/183449/files/TRDeepDive.pdf

[7] PACMan: Performance Aware Virtual Machine Consolidation, http://research.microsoft.com/pubs/192861/pacman2013.pdf

[8] About Okeanos, https://okeanos.grnet.gr/about/what/

[9] About Synnefo, https://www.synnefo.org/

[10] Snf-image-creator software documentation, https://www.synnefo.org/docs/snf-image-creator/latest/

[11] Sysbench, https://github.com/dallasmarlow/sysbench/

[12] Netperf, http://www.netperf.org/netperf/

[13] MBW, https://github.com/raas/mbw

[14] hdparm, https://wiki.archlinux.org/index.php/hdparm

[15] Live migration, Wikipedia.org, https://en.wikipedia.org/wiki/Live_migration

[16] Profiling, Wikipedia.org
https://en.wikipedia.org/wiki/Profiling_%28computer_programming%29

[17] Gang scheduling, Wikipedia.org, https://en.wikipedia.org/wiki/Gang_scheduling