STACKED JOB SCHEDULING ON VIRTUAL MACHINES WITH CONTAINERS IN CLOUD COMPUTING SYSTEMS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Mustafa Akın

June 2016

Stacked Job Scheduling on Virtual Machines with Containers in Cloud Computing Systems
By Mustafa Akın
June 2016

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

İbrahim Körpeoğlu (Advisor)

Özgür Ulusoy

Adnan Yazıcı

Approved for the Graduate School of Engineering and Science:

Levent Onural
Director of the Graduate School

ABSTRACT

STACKED JOB SCHEDULING ON VIRTUAL MACHINES WITH CONTAINERS IN CLOUD COMPUTING SYSTEMS

Mustafa Akın
M.S. in Computer Engineering
Advisor: İbrahim Körpeoğlu
June 2016

Virtualization and use of virtual machines (VMs) is important for both public and private cloud systems and also for users. The allocation and use of virtual machines can be optimized by using knowledge about expectations of users, such as resource demands, network communication patterns, and total budget. However, both public and private cloud providers do not expose advanced configuration options to make use of custom needs of users. Adding upon to previous research, we propose a new approach for allocating and scheduling user jobs to virtual machines by use of container technologies like Docker, so that VM utilization can be increased and costs for users can be decreased. In our approach, by predicting resource demands, we can schedule different kinds of jobs on a single virtual machine without jobs affecting each other and without degrading performance to unacceptable levels. We also allow cost-performance tradeoff for users. We verified our approach in a real test-bed and evaluated it with extensive simulation experiments. We also adapted our approach into a real web-based application we developed, called PAGS (Programming Assignment Grading System), which enables efficient and convenient testing, submission and evaluation of programming assignments of a large number students in an interactive or batch manner in identical and isolated system environments. Our approach effectively schedules requests from teachers and students so that the system can horizontally scale in a cost efficient manner.

Keywords: cloud, scheduling, containers, virtual machine, allocation.

ÖZET

BULUT BİLİŞİM SİSTEMLERİNDE SANAL MAKİNELER ÜZERİNDE TAŞIYICILAR ILE YIĞIN İŞ ÇİZELGELEMESİ

Mustafa Akın
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: İbrahim Körpeoğlu
Haziran 2016

Sanal makine çizelgeleme problemi bulut sağlayıcıları ve özel bulutlar için açık uçlu bir problemdir. Dağıtmada, kullanıcının ağ iletişimi, kaynak ve toplam bütçe gibi gereksinimleri bilinerek optimizasyon sağlanabilir. Fakat herkese açık bulut sistemlerinde böyle bir seçenek bulunmamakla birlikte özel bulutlarda da gelişmiş dağıtma seçenekleri bulunmamaktadır. Kullanıcı programlarının çalışmasını planlayacağı zamanda, sağlayıcılardan kaynak isteklerini bilinçli kararlar sonucunda istemekte ve dağıtmayı da kendi yapmaktadır. Bu tezde, geçmiş araştırmaları genişleterek, elimizdeki kaynakları daha verimli bir şekilde kullanan yeni bir çizelgeleme yöntemi sunmaktayız. Tezimizde yapılacak işlerin kaynak kulanımlarının önceden analizi ile farklı karakteristiklere sahip işlerin aynı sanal makinelerde planlanmasıyla, işlerin birbirini etkilemeden kaynaklardan yararlanımının arttırılabildiğini gösteriyoruz. Buna ek olarak, kaynakları kapasitelerinin üzerinde kullandığımız zaman da toplam süreden feragat ederek maliyetin düşebildiğini gösteriyoruz. Bu planlama yöntemini tezin 2.ci parçası olan Programlama devleri Notlandırma Sisteminde'de (PAGS) kullanıyoruz. Programlama ödevleri, programlama derslerinin önemli bir parçası olmakla beraber, notlandırmada yapılan, dosyaların indirilmesi, güvenliğin sağlanması ve çalıştırılması gibi tekrarlayan işler bu süreci zorlaştırmaktadır. PAGS, Bilkent Üniversitesi İşletim Sistemleri dersinde öğrenciler ve asistanlar üzerinde denenmiş olup, bu tezde öğrencilerin davranışsal sonuçları incelenmiştir. Ayrıca, öğrenci kodlarının çalıştırılma isteklerinin çizelgelenmesi de ilk kısımda anlatılan methodlarla yapılmıştır.

Anahtar sözcükler: bulut, sanal makine, kaynak ayrımı, çizelgeleme.

Acknowledgement

I would like to take this opportunity to express my gratitude to my supervisor, Assoc. Prof. Dr. İbrahim Körpeoğlu, for his motivation, guidance, encouragement and support throughout my studies.

I would also like to thank to Prof. Dr. Özgür Ulusoy and Prof. Dr. Adnan Yazıcı for kindly accepting to spend their valuable time to review and evaluate my thesis.

I express my graditute to my father Erol, my mother Meral and my sister Melek, my brother-in-law Nüfer, and finally my fiance Buket for always being motivating and supportive. None of this would have been possible without their love, selflessness and sacrifices they made on my behalf.

I also thank TÜBİTAK (The Scientific and Technological Research Council of Turkey) for supporting this work with project 113E274.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With recent advances and trends in computation and communication, cloud computing has emerged as a prominent technology and has drawn the attention of different types of users and disciplines. Offloading work and storage to cloud for a certain price is a sustainable work-flow for many people and companies. For instance, one does not have to buy thousands of servers to run map-reduce programs, but just rent computing and storage resources in cloud, use them when needed and give back to the cloud when not needed. This also applies to scientific computing applications, web servers and web applications, image processing, data analytics, etc.

However, cloud computing can be unnecessarily expensive if not performed carefully. In most public clouds, users are charged hourly for the time they are using the virtual machines. This can yield to increased costs for relatively short-lived applications, while it could simply be avoided by smart packing and scheduling strategies applied automatically.

In this thesis, we show that allocating a separate virtual machine per job is not a good approach in terms of cost, and propose a job consolidation and scheduling method that can pack several jobs into each virtual machine without degrading their expected performance. Depending on the jobs, their characteristics and

requirements on CPU, RAM, I/O resources, it is possible to stack more than one job to a single machine and still let them run near to full performance, instead of allocating a dedicated virtual machine per job.

Additionally, for jobs not to interfere with each other, we propose a job isolation approach using operating systems containers. Containers allow processes to benefit from operating system level virtualization. Unlike full virtualization, however, hardware is not emulated and access to common hardware and physical resources in the host is both accounted and monitored via the single hosting operating system. This allows total isolation of jobs and helps to freely schedule independent jobs wrapped by containers into different virtual machines. Use of containers has other advantages besides providing VM-like isolation. Since they can be booted up in sub-second time-scales, it is more viable to launch containers than launching virtual machines for relatively short jobs.

As part of our approach, we provide scheduling algorithms to schedule user jobs by use of containers into VMs allocated for a user. Our algorithms can consider various objectives while scheduling the jobs and can also do cost-performance tradeoff. In public cloud environments, a user can get only some predefined virtual machines with certain resource settings from a provider. Our scheduling methods aim to make best use of the resources of these predefined virtual machines rented by a user. Our methods consider the resource demands of jobs while bundling and scheduling the jobs into virtual machines, so that performance is not degraded to unacceptable level while trying to utilize VMs better and save costs. New virtual machines can be allocated when needed.

We did extensive simulation experiments to investigate the effects of our scheduling approach on performance and cost. In our simulations, we observed that scheduling different types of tasks in terms of resource usage to the same virtual machine can be profitable for reducing total cost, since in this way we can avoid allocating a new virtual machine and paying unnecessarily by properly stacking the jobs into already running virtual machines. As mentioned above, we stack the jobs into the same virtual machine by considering the resource usage patterns of the jobs, which leads to better and effective utilization of the already

running virtual machines. As a result of our approach, even if we over-utilize a virtual machine, it still can be profitable since we avoid initialization and boot-up costs of a new virtual machine creation.

We verified our stacked job scheduling method in a real testbed environment. We used Docker [1] as the container technology. Our testbed experiments have shown that naive allocation where a virtual machine is allocated per task is very costly compared to our stacked-job scheduling method. The experiments also show that the way we stack jobs reduces the total cost of running the jobs. Our method decreases the total cost by properly consolidating jobs in a single rented machine, which may decrease the speed of each job but not at a level that will violate service agreements.

Additionally, we adapted our approach in a real web-based application, called PAGS (Programming Assignment Grading System), that requires creation and running of many short-lived tasks belonging to different users over virtual machines. We have seen that even the tasks are very short, same methodology helps to reduce the cost and in the mean-time increase system throughput to allow many users to use the system.

With our PAGS system, we tackle a problem that exists in computer science and programming courses. In teaching of many computer science courses, considerable part of the course work consists of programming exercises, homeworks, or large projects. As the programming course classes may vary from a few students to hundreds, grading process can be exhausting. One of the reasons is that students might upload or e-mail their submissions to the teachers or graders of the course and normally graders would have to download, organize, and run them all in a repetitive manner. If students are not told explicitly, their submission format might vary and the job of the grader might include modifying student submissions. Also grading assignments requires huge manual labor that can be prone to human errors.

To solve the afore-mentioned problems, we designed and developed our PAGS

system. PAGS eases up grading programming assignments. It allows the assignments to be defined via a web-based interface. Students attempt the defined assignments through the web-based interface as well. PAGS allows submissions to be run in independent and isolated lightweight containers and utilizes our job scheduling approach in scheduling and consolidating many containerized student jobs into virtual machines. When necessary new physical machines are utilized automatically, depending on the workload. In this way our scheduling approach allows PAGS to be horizontally scalable while maintaining its cost effectiveness.

The contributions of the thesis can be summarized as follows. 1) We first propose a novel approach to bundle multiple user tasks by using Docker containers into a single virtual machine to save costs. 2) We propose scheduling methods to schedule containerized user jobs into VMs that can allow a user to do cost-performance tradeoff. 3) We evaluate these proposed methods via both simulation and real testbed experiments. 4) Additionally, we adapt our bundling and scheduling approach in a real-world application, called PAGS (Programming Assignment Grading System). 5) As the final contribution, we present the design and implementation of PAGS, a web-based programming and testing environment for students and teachers to improve the grading process of programming assignments in computer science courses.

The rest of the thesis is as organized as follows. Next, we discuss some related work, in both areas of task scheduling and previous automatic grading systems. We also give some background information and discuss the state-of-the-art virtualization techniques, including container based virtualization with full isolation. In Chapter 3, we present our job allocation and scheduling approach and provide our container-based scheduling methods. We verify our approach in a real testbed environment and also evaluate its performance via extensive simulation experiments. In Chapter 4, we present the adaptation of our approach on our PAGS system, which is another real use case of our containerized stacked job scheduling. We also present the design, implementation, and experimental evaluation of PAGS. Finally, in Chapter 5, we gave our conclusions and thoughts about possible future work that might be inspired from this work.

Chapter 2

Related Work and Background

In this chapter, we will be looking into some of the previous work on the topic of bag of tasks scheduling and programming assignment helper tools. We will be also giving information on virtualization technologies, both traditional virtual machine based, and newer container based.

2.1 Related Work

Allocating and scheduling many tasks in a multi-computing environment is often called as bag of tasks scheduling problem, and many prior work has been done in this area, both in cloud computing and high performance distributed computing systems [2].

In [3] the authors focus on task allocation in grid computing environments and try to maximize the utilization of space-shared resources. In [4], Mao et al. proposes scaling up virtual machines by moving the jobs to virtual machines with more resources, or scaling down, to meet required performance criteria, based on budget constraints and current workload of the virtual machines. Although the idea of scaling up or down seems profitable, it can provide an unnecessary delay of transferring the current job state, and some tasks might be just uninterruptible,

atomic tasks. In our work, we assume all jobs are atomic tasks and they cannot be moved to another virtual machine instance without starting from the beginning. The work [5] is similar, and proposes to auto-scale under certain load of the virtual machines, but it also considers the billing period of the IaaS provider [6] and keeps the virtual machines running until the next billing hour approaches. We also use the same idea in our work to reduce cost and keep the machines running, instead of booting a new one to accept new jobs. As it can be seen in [7], boot time of virtual machines can vary over time, location and provider, and this fact can be important to select the most efficient virtual machine for allocation.

Authors in [8] also try to optimize the utilization and completion times in bag of tasks problem under budget constraints, however, they only focus on CPU requirements of the jobs, while in our work we also consider RAM and bandwidth requirements. [9] focuses on heterogeneous platforms for bag of tasks problem, however, the tasks are parallel working jobs and communicate with MPI, and in [10] the jobs share common data, whereas in our work we focus on independent jobs.

Some works such as [11] and [12] aim to estimate budget for bag of tasks by using machine learning on previous runs. The work [13] proposes and evaluates various scheduling scenarios. The described methods suggest cheaper or faster options. In our method, we assume the average run time performance of jobs in each virtual machine type are available or can be predicted. Similar to [11] and [13], which suggest faster or cheaper schedules, our method lets the users to select between individual task completion time performance and total cost. As different from [11] and [13], we assign more than one job to virtual machines to achieve higher levels of utilization, and also we consider the three basic resource types needed by jobs, which are CPU, RAM and I/O, to schedule more intelligently and effectively.

The work [14] allows executing bag of tasks on multiple grids, based on given policies, but it only considers CPU requirements of the tasks, where we consider also RAM and I/O utilization of the tasks besides CPU utilization. Different from

our work, the method in [15] suggests to span a job to many virtual machines, and balance the load between them to meet performance requirements of the tasks. However, as the performance requirements depends on the jobs themselves and can be varying, we did not consider performance requirement as utmost important to keep our method more general, hence a single job does not span to multiple machines in our environment.

The work [16] assumes preemtable tasks, where we consider our tasks as atomic. [17] analyses the scientific tasks on various cloud vendors, but the tasks are also parallel.

Most of the prior work in the scheduling area assume offline optimization of the scheduling the bag of tasks to grids or cloud vendors. However, in our work, we assume a dynamic environment where the tasks can be heterogeneous and may arrive at any time, and our approach schedules them on-the-fly dynamically as soon as they arrive. In this way the system state is to tried to be kept as optimized as possible.

Regarding automatic grading systems, [18] gives information about early assessment systems. [18] signifies the importance of testing the student code in programming course education. One of the earliest automatic evaluation systems can be seen in [19]. The automatic grader of [19] was tried on a class of 20 students and achieved 1/8 of the manual grading time. Although this system was built to assess the punch-cards in 1960s, the foundation still exists as other systems, including ours, still compare the students code output with the expected output. [19] has stated that a student code could cause harm to system and needs to be carefully handled.

In addition to comparing the output of the programs, some studies addresses the issues of code style and software quality, such as reliability, effectiveness, maintainability and readability. In [20], authors define C++ language-specific style rules, and both students and the graders can use the tool they developed which is called `Style++`. The tool helps novice students to pay more attention to their programming assignments and learn better. Most notably, `Style++` allows

removing disparity among the graders. As in our system, having a centralized assessment allows compensating differences among graders.

In [21], authors explain their GUI-based automatic evaluation system. The system controls the execution environment with authentication and execution states, and gives detailed reports about code execution. In a questionnaire, 65.96% of the students stated that they prefer their examinations to be held on computer, which shows a general interest in such a GUI-based automatic evaluation system, as ours.

The Kassandra system [22] is another early automatic grading system where students can interactively check the correctness of their programs. Different than others, it is one of the first that addressed the security issues. [22] describes technical details on how to keep such an automatic grading system safe in Unix environment and prohibit modifying the grades of students. In our system, we also ensure that a student code does not harm the grading system and is resource-limited to maintain system stability.

There are also systems that check assignments for similarity and plagiarism. MOSS, Measure of Software Similarity [23] is a web service that is hosted on Stanford University website. It is based on document fingerprinting algorithms. Although MOSS itself is not completely enough to automatically detect plagiarism, it can be very useful as an online automatic system. We integrated MOSS in our system, and it helped us to catch some serious plagiarism incidents.

In [24], issues which arise in building an automatic assessment system are discussed, such as correctness, efficiency, and maintainability issues for programming assignments. The study states that additional problems arise with human grading, such as subjective judging of correctness and efficiency, difficulty in validating multiple approaches, and distraction by focusing on aesthetics.

[25] shows the importance of Test-Driven Development (TDD) in evaluating programming assignments. Students are divided into two groups. One group did test-driven development using the online grading system and the other group

completed their assignments without using any test-driven technique. [25] showed that test-driven development improved the learning by about 50.88%. Similarly, in [26] students are required to write their test suites with their assignments and as a result it is observed that students wrote their code more effectively. Lastly, in [27], authors discuss their system Codometer based on TDD techniques and their experiences in a 36-students course. Codometer has a Web-UI to collect the assignments from students and send reports to them. It also has manual grading feature, which we found reasonable to be applied in cases where students can trick the system.

With these studies in mind, we created our system, PAGS, providing an efficient, secure and isolated environment for students to both write their code and evaluate it. We greatly used the concepts from test-driven development techniques described above in early systems, however, in our system, programming languages and frameworks are not a limitation and any programming language or framework can be used. Also, in previous studies, the scalability issue was not addressed, which is addressed in our system. We provide experimental results about the scalability feature of our system.

2.2 Virtualization

Virtualization is a widely used technique in both cloud and dedicated server deployments. Off-the-shelf computers with higher cores, larger memory and more storage is preferred in deployments due to lower costs compared to buying more computers with smaller resources. However, leasing all the available resources to one tenant is not feasible, customer might not need all of it and it would not be cost effective from the point of view of the customer. Virtualization helps to create virtual machines inside a physical machine, therefore the resources of the physical machine can be shared among many users.

One of the two virtualization modes, Software-Based virtualization, is usually

done by emulating all the instructions of the virtual machine. It is not preferred due to its obvious performance reasons. As an alternative, with Hardware-Assisted virtualization, recent CPUs with Intel VT-X or AMD-V technologies enabled, processors have become aware of the virtualization and offers some optimizations for CPU and RAM virtualization. There exists hypervisor softwares that benefit from these hardware-assisted virtualization techniques, such as KVM [28], Xen [29], Microsoft HyperV [30] and VMWare ESXi [31]. These hypervisors also emulate the additional devices that virtual machines can use, so that resources such as disks, network devices can be virtualized as well.

Although the performance can be increased with hardware-assisted virtualization and hypervisor-aware operating systems, there is still a considerable overhead in certain workloads [32]. As an alternative, recent operating systems provide container-based virtualization. In contrast to hypervisor based virtualization, devices are not emulated, but underlying operating system is responsible for isolating the containers. Therefore, in container based virtualization, all containers share the same kernel, CPU, and RAM, however, operating system uses technologies like namespaces and control groups to completely isolate them.

Although the container based virtualization is not providing more flexibility, it performs better, because of the avoided emulation. While typical virtual machines take seconds to boot up, containers can start in sub-seconds. Additionally, because there is no additional kernel and operating system running, containers use much less CPU, memory and disk resources compared to virtual machines.

In this thesis, we made use of the performance and quick boot up time characteristics of containers. These allowed us to place jobs into containers and pack multiple containers on a single virtual machine and run them without much overhead and with good performance.

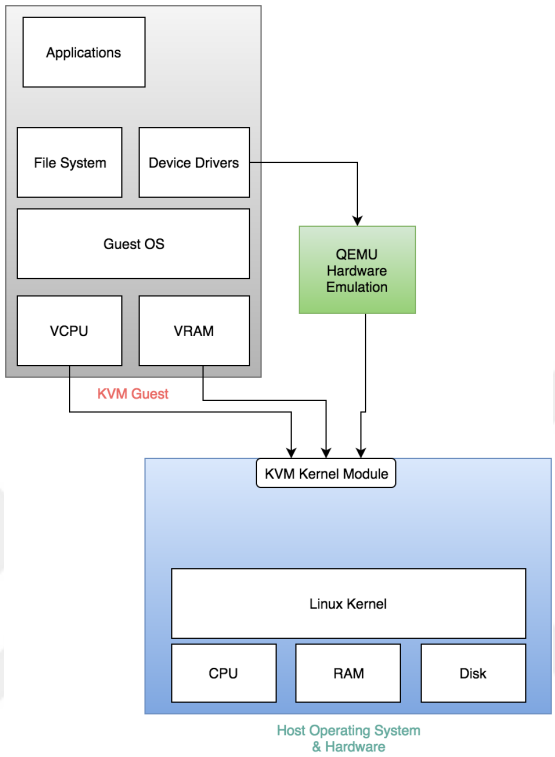


Figure 2.1: Kernel Virtual Machine (KVM) Hypervisor overview.

2.3 KVM and Libvirt

Containers may be run in a physical machine or in a virtual machine. We used virtual machines. To create and manage virtual machines, we used open source Kernel Virtual Machine (KVM) [28] hypervisor during our test-bed implementation and experiments. KVM runs on recent Linux physical servers. KVM can directly boot from executable kernels or from disk images where a bootloader and an operating system is installed. We used the Ubuntu 14.04 Cloud Image from Ubuntu Linux distribution's website and installed our tools and provided a new base image. Therefore, upon each virtual machine creation request, using `qemu-img` tool, we cloned this base image and assigned it to the newly created virtual machine.

To control KVM, we used Libvirt [33]. Libvirt is a library to benefit from the virtualization capabilities of physical machines in a standard way. In our test environment, we use our physical machines to create virtual machines on demand

and schedule tasks to be run on these virtual machines. Libvirt allows specifying all the details needed to define a virtual machine as an XML file, which includes the virtual CPU count, name, hypervisor type, disk images to be exposed as drives in VM and finally a virtual network device. A sample domain (virtual machine) definition is given below.

```
<domain type='kvm'>
  <name>trusty1</name>
  <memory>1048576</memory>
  <os>
    <type>hvm</type>
    <boot dev="hd" />
  </os>
  <vcpu>1</vcpu>
  <devices>
    <disk type='file' device='disk'>
      <driver type='qcow2' cache='none'/>
      <source file='/images/trusty1.img'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <disk type='file' device='disk'>
      <source file='/images/userdata1.img'/>
      <target dev='vdb' bus='virtio'/>
    </disk>
    <network>
      <name>host-bridge</name>
      <forward mode="bridge"/>
      <bridge name="br0"/>
    </network>
  </devices>
</domain>
```

Libvirt [33] is capable of controlling Xen [29], VMWare ESXi [31], Microsoft

HyperV [30], and other hypervisors. It also allows controlling of both local and remote storage, and custom networks. They are, however, not used in this thesis.

2.4 Containers and Docker

As mentioned earlier, different from hypervisor and virtual machine technologies, there are also container technologies that an operating system can support to create multiple isolated environments for multiple applications to run in total isolation. A container in not a virtual machine. A guest OS is not needed. It uses the host OS and duplicates the whole environment of the host OS into a new container. In this way multiple containers can be created that use the same kernel but are totally isolated.

As mentioned above, containerization ability resides in the operating system. Linux, starting from 2.6.24, has support for containers and Microsoft has announced support for Windows Containers starting from the Windows Server 2016 version. In this thesis we used Linux containers.

Linux containers are isolated using *namespaces* and *control groups* (cgroups) features of the Linux kernel. An unprivileged user can create a Linux container, that is completely isolated and independent from the system. Although this seems similar to virtual machines, Linux containers do not emulate a full system, instead all the processes in containers still use the same kernel for its system calls and memory management. However, with proper usage of namespaces functionality, processes in the different containers are not aware of each other and the original system. With this way, we can run untrusted codes without affecting each other or the system.

There are many namespaces available in the Linux kernel. They can be summarized as follows:

- **Mount:** Isolate the set of file system mount points for processes.

- **UTS:** Isolates domain name and host name.
- **IPC:** Inter process communication, such as shared memory, named semaphores, message queues.
- **PID:** Process ID number space. So, init process of each namespace can be different.
- **Network:** Different network devices, routing tables.
- **User:** Different user ids for inside and outside of namespace.

Following control groups allow fine and coarse control:

- **blkio:** set limits and monitor usage of block devices such as disks.
- **cpu:** scheduling, weights of tasks.
- **cpuacct:** usage reports of cpus.
- **cpuset:** assign cpus and memory nodes to tasks.
- **devices:** allow access to devices (webcam, gpu etc.)
- **memory:** limit on memory, and usage reports.
- **net-prio:** priority on network interfaces.

Linux containers, as often referred as LXC, can be used with command-line tools present in recent Linux distributions. However, their usage requires through knowledge of how they work in order to provide full isolation and can be challenging for an average user.

In our thesis, we used Docker [1], a container technology that was originally a wrapper around LXC functionality, which adds now more features such as image management, layered file-systems, mountable volumes and remote API that makes using LXC a better experience. In Docker, one creates an image

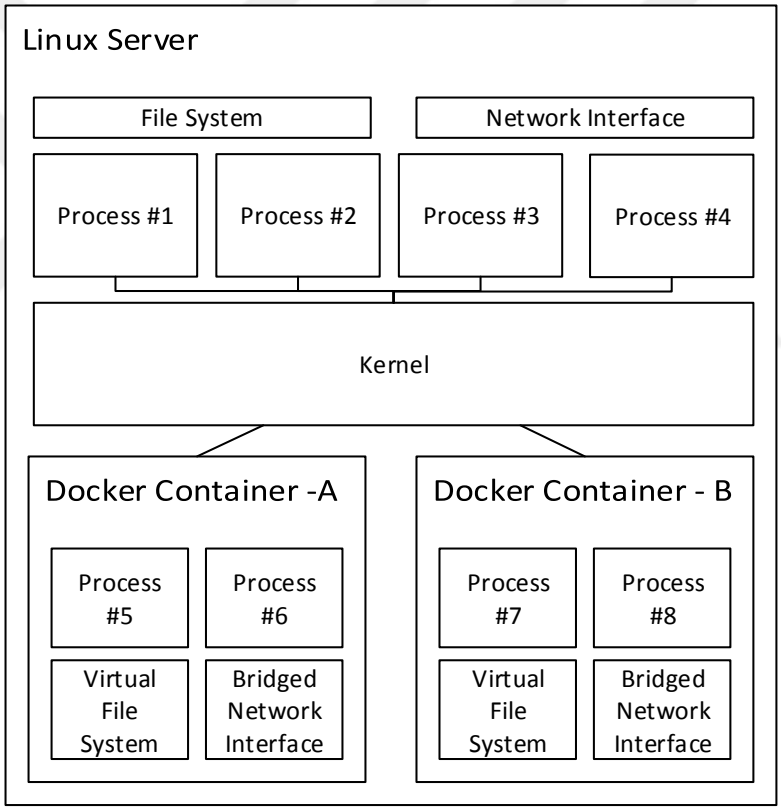


Figure 2.2: Docker Process Overview in a Linux System.

from a set of files, or from another image and commits the changes into another image. Upon each execution of a requested image, a new layer is added to the original image, and changes do not affect the original image. Therefore, different containers can use the same images, and this allows to gain space and performance by caching the same file contents in the file cache.

An example of a system layout using Docker with 2 containers can be seen in Figure 2.2. Processes 1-4 are regular processes in the system and, depending on their users, they have access to file system and network interfaces. Processes 5-6 and Processes 7-8 are in 2 different containers. Processes in the same container are aware of each other, but not of processes in different containers. Additionally, Docker mounts a layered file system as the root file system of the container and the processes in container can only read from and write to that file system. Lastly, Docker also creates a network interface that is originally bridged with the system network interface, and each container has its own network stack with a different local IP address allocated. If desired, those containers can communicate with each other and also with outside world. If desired, their communication among themselves or the outside world can be completely cut by using proper firewall rules.

In our job scheduling method, we used Docker to create images for each task and placed these images in a local repository. When assigning a job to a server with Docker installed, we just executed container run command with the specific job type and collected both its output and resource usage statistics. This allowed us to learn about tasks' resource usage and place them in a better way in future runs. Also, by using Docker, we can run many tasks simultaneously in a single virtual machine in a completely isolated manner.

In our PAGS system, we leverage the functionality of Docker to create a base image for each assignment with the files and execution scripts specified by the teacher. Whenever a student wants to execute his code, we allocate a new container based from the image created by the teacher and place students files into it as well. Since each student has his own container, each execution is independent from each other and can be repeated many times without affecting each other.

In our environment, we installed Docker in our base virtual machine image and enabled its remote API to be used with HTTPS. A Docker container is created with the following request to the API endpoint:

```
POST /containers/create HTTP/1.1
Content-Type: application/json

{
  "Cmd": [
    "sysbench", "--test=cpu",
    "--num-threads=4", "run"
  ],
  "Image": "benchmark_image",
}
```

To access the Docker API in virtual machines, we created a bridged network and made the virtual machines attach to it to ensure they get a publicly accessible IP address. We used an Ubuntu 14.04 image and extended it with installing Docker image and configuring its remote API. We also put our Docker image, and used the resulting image as base image for virtual VMs. Upon each VM creation request, we copied the base image, which was approximately 2.7 GBs. We have written an agent for physical machines, which interfaces with libvirt to create virtual machines and qemu-img for cloning of base disk images.

2.5 Summary

As discussed in this chapter, we see that bag of tasks problem is well researched and a common problem. As a new perspective, we propose stacking the jobs together using container based virtualization as opposed to traditional virtualization. Containers provide ability to quickly schedule and run the tasks. In

the following chapters, we describe our approach in detail and show that it is a feasible solution by providing both simulation and testbed results.

Chapter 3

Job Scheduling to Virtual Machines

As described in previous chapter, bag-of-tasks scheduling problem is often solved by allocating one virtual or physical machine per task, which is most likely to waste available resources. This is because no task would achieve full utilization in terms of all of the resources, such as CPU, RAM, and I/O of a machine. To achieve a better utilization, we propose stacking multiple jobs into the same virtual machine, depending on some quantifiable criteria such as expected resource utilization of tasks on various virtual machine types.

In this chapter, we describe how our bag of tasks scheduling system works and how we use our methodology for scheduling decisions. We also describe how to extract utilization information of tasks, and how we use it in our methodology to achieve an overall greater utilization so that we run the tasks economically. At the end of the chapter we provide our simulation and experimental results that support the idea of stacked job scheduling.

3.1 Proposed System Model

Our proposed system model can be seen in Figure 3.1. The system is composed of one scheduler that has right to allocate and destroy virtual machine instances on one or more IaaS providers. The scheduler receives tasks at various times, and makes an online, immediate decision to either allocate a new virtual machine to place the job in, or assign the job to one of the already running virtual machines. The scheduler can also destroy the allocated virtual machines, if possible, in favor of reducing the total cost.

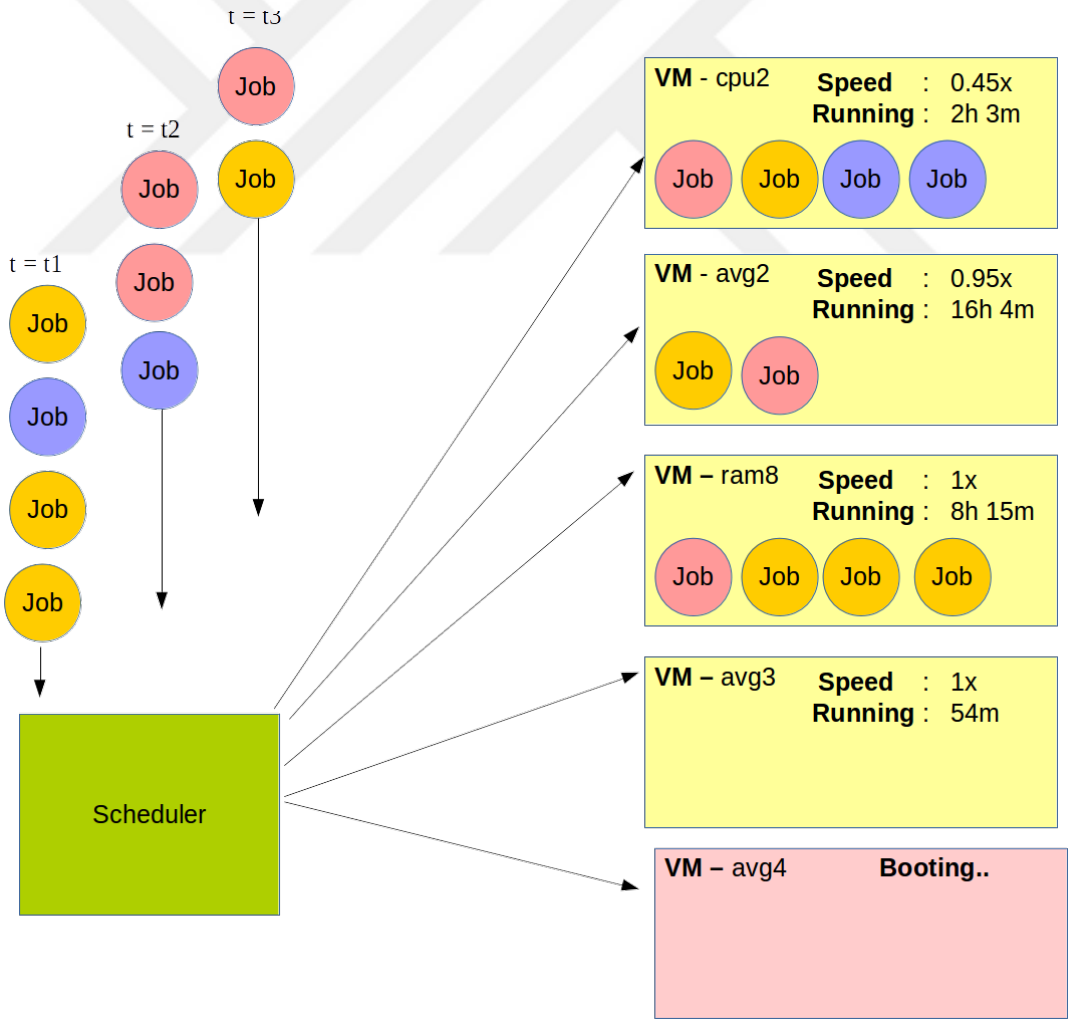


Figure 3.1: Model of the proposed system.

3.1.1 Components

The system allows scheduling and running different types of tasks, where tasks' utilization levels of CPU, RAM and I/O resources on each virtual machine type are known or can be predicted. We will use the terms *job* and *task* interchangeably throughout the thesis.

A task is specified as follows:

$name$: The name/type of job
U : Utilization of resources (ranges from 0 to 1)
$U_{CPU}(vm)$: Utilization of CPU at a given VM type
$U_{RAM}(vm)$: Utilization of RAM at a given VM type
$U_{IO}(vm)$: Utilization of I/O at a given VM type
$t_{completion}(vm)$: Average Completion time of job in a given VM type
$t_{arrival}(vm)$: Arrival time of the job

Also a virtual machine can be specified as follows:

$name$: The name and type of VM
$cost$: Cost incurred at each billing period
$billing$: Billing period, e.g. hourly, 10-min
$bootduration$: Average booting duration
$starttime$: Time that VM started running
$speed$: Current task running speed of a task in VM, ranging from 1 to 0, where 1 means full speed and 0.5 means at half-speed.

3.1.2 Allocation and Running of Jobs

We define an interface for allocation and de-allocation requests that our scheduler accepts. The *add(Jobs, VMs)* and *update(State)* must be implemented in our scheduler. Add method allows the scheduler to accept jobs and place them to the appropriate VM. Update method is called periodically to determine if there is a need for allocating a new VM or deallocating an existing VM. In scheduler, we also have an objective (strategy) parameter, that allows to choose between *Fastest*, *Cheapest* and best *Cost per Performance (cpp)* virtual machine for a given job.

Cost per Performance (*cpp*) is a metric that we define and propose to decide and evaluate VM selection for placing jobs. It indicates the cost we pay per unit performance we get for our jobs. We consider the performance we get for a job to be inversely proportional to the completion time of the job. A larger value of *cpp* indicates that we either used excessive cost or time or both, and a smaller value indicates that we maximized our cost-performance by reducing the cost or the total time or both. We propose the following formula to compute this performance metric:

$$cpp = \log\left(c^2 \times t\right)$$

Here, c is the hourly (or periodic) cost of the virtual machine used to run the job and t is the completion time of the job. The reason that we have chosen the above metric is that we want to find a cost-performance function that takes both hourly cost and time into consideration, however, we desired the affect of cost to be higher compared to time.

The scheduler pseudo-code is given given in Algorithm 1.

We next describe allocation policies that can be applied by our scheduler. In Algorithm 1, the policy is executed as part of the *allocation.add()* method, since it decides how to schedule a task depending on the requested allocation

Algorithm 1 Scheduler

```
1: procedure SCHEDULER(STRATEGY, ALLOCATION)
2:     vms ← list()
3:     pendingJobs ← list()
4:     while true do
5:         arrived ← getNewJobs()
6:         for all arrived, pendingJobs as newJob do
7:             jobs ← jobs + newJob
8:         rejected ← allocation.add(jobs, vms)
9:         pendingJobs ← rejected
10:        allocation.update(VMs)
```

implementation. We first describe naive allocation, then describe our proposed allocation policies.

3.2 Naive Allocation

This methodology is the most basic one, that was used in most of the prior work. We allocate a separate virtual machine for each task. And for each allocation, we have the following three strategies to choose from:

- Allocate the fastest VM for the task.
- Allocate the cheapest VM for the task.
- Allocate the VM that has the minimum cost-per-performance (*cpp*) value.

While this approach allows isolation of tasks, it is not the most efficient. Neither each task can make use of a VM completely in terms of resource usage, nor might we be spending less money than we would otherwise.

Additionally, VMs do not become instantly available. They have booting time which can be a considerable amount of time depending on the requested configuration, and this makes this approach inefficient for many workloads. Opening

Algorithm 2 Strategy VM Finder

```
    procedure FindVMType(Strategy,JobType)
2:      selectedVM ← null
        if Strategy == Fastest then
4:          minTime ← +Inf
            for each VMType, time in JobType.times do
6:              if time < minTime then
                    minTime ← time
8:                  selectedVm ← VMType
        if Strategy == Cheapest then
10:         minCost ← +Inf
            for each VMType, cost in VMTypes do
12:             if cost < minCost then
                    minCost ← cost
14:                 selectedVm ← VMType
        if Strategy == CostPerf then
16:         maxPerf ← +Inf
            for each VMType, time in JobType.times do
18:             cost ← VMType.cost
                perf ← cost * cost * time
20:             if perf < maxPerf then
                    maxPerf ← perf
22:                 selectedVm ← VMType
        return selectedVM
```

a new virtual machine incurs a significant overhead, especially when the VM is used less than one billing period.

3.3 Our Proposed Allocation Methods

In our approach, we allocate more than one job to VMs, depending on the given objective. However, instead of giving tasks randomly to a VM, we consider what the total utilization of the VM would be if we would assign the task to that VM. For instance, if we put a task with 0.8 CPU utilization (CPU load), and a task with 0.7 CPU utilization, we assume that these tasks will slow down linearly, making the total utilization 1.5, 0.5 larger than the VM can handle. Therefore, they work at 0.5 unit speed, instead of 1 unit speed. As another example, if we put two tasks totaling less than 1 utilization, we assume they work at 1 unit speed.

In our methods we define and use the following utilization model:

- If utilization of a VM for a resource does not exceed 1, we assume that resource does not reduce the speed of tasks and each task has a speed of 1. That means running these tasks together will not degrade their performance to an unacceptable level.
- If CPU has over-utilization, we assume all tasks slow down by a factor of CPU over-utilization, as Linux scheduler is capable of fair use among processes.
- If RAM has over-utilization, we assume all tasks slow down exponentially with respect to RAM over-utilization, as over-utilizing may cause trashing and may degrade performance drastically.
- If Disk (I/O) has over-utilization, we assume all tasks slow down by a factor of disk over-utilization divided by 1.5. Since disk blocks can be cached

cached in memory, we assume overutilization will cause less reduction in speed.

3.3.1 Count Limited Allocation Method

In this algorithm, whose pseudo-code is given in Algorithm 3, we stack jobs into virtual machines depending on the strategy type (Algorithm 2), i.e., depending on the objective we prefer. This method only considers the number of tasks already running in the virtual machines to decide whether a virtual machine is a fit. This allocation algorithm takes parameter *DesiredUtil* which acts as an upper limit of the number of tasks running in a virtual machine. If a VM has more tasks then a given *DesiredUtil*, it is discarded and other VMs are checked for a match. If no matching virtual machine is found at the given time, a new VM is allocated by the scheduler to place the new task there.

Algorithm 3 Count-Limited Job Allocation

```
1:  procedure CountLimited(Strategy, Job, DesiredUtil)
2:      vmType ← FindVMType(Strategy, Job.type)
3:      for each VM in currentVMs do
4:          if VM.type == vmType then
5:              VM.addJob(Job)
6:              util = VM.activeJobs
7:              if util > DesiredUtil then
8:                  VM.removeJob(Job)
9:              else return
10:     newVM ← new VM(vmType)
11:     newVM.addJob(job)
12:     VMs.add(newVM)
```

3.3.2 Utilization-Based Allocation Method

In this second method we propose, we consider the VM speed that jobs will perceive in an assigned VM. The pseudocode of the method is shown in Algorithm 4. The *DesiredUtil* parameter given to our algorithm allows to determine whether

to open a new VM or use an already running VM, depending on the given VM strategy type (Algorithm 2). If the total jobs divided by the VM speed a job is perceiving is larger than the *DesiredUtil*, we allocate a new one. Instead of using the VM speed directly, we also consider the currently allocated job count within the VM to ensure not many tasks suffer from low utilization and have longer completion times.

Algorithm 4 Utilization-Based Job Allocation

```
1: procedure UtilAlloc(Strategy, Job, DesiredUtil)
2:     vmType ← FindVMType(Strategy, Job.type)
3:     for each VM in currentVMs do
4:         if VM.type == vmType then
5:             VM.addJob(Job)
6:             util = VM.activeJobs/VM.getJobSpeed()
7:             if util > DesiredUtil then
8:                 VM.removeJob(Job)
9:             else return
10:    newVM ← new VM(vmType)
11:    newVM.addJob(job)
12:    VMs.add(newVM)
```

In Algorithm 4, the method *getJobSpeed()* is returning the speed that a job perceives in the respective VM. The pseudo-code of *getJobSpeed()* method is shown Algoritm 5.

3.4 Simulation Experiments

To test our methods, we have implemented a discrete-time custom simulator, since CloudSim [34] has failed to satisfy our needs. The following are inputs for our simulation experiments.

- **vms**: VM types. Information about VM types, such as name, cost and average boot time, is given as input.
- **jobtypes**: Information about each job type's performance characteristics on each VM type, such as completion time and utilization of resources.

- **jobs**: Set of jobs of a test scenario. Jobs of jobtypes may arrive at different times or at the same time.

Algorithm 5 Calculating individual unit task speed in a VM

```
1:  procedure GetJobSpeed
2:      if Jobs.size <= 1 then
3:          return 1.0
4:      else
5:          U_CPU, U_RAM, U_DISK, U_BW ← 0
6:          for each Job job in Jobs do
7:              U_CPU ← U_CPU + job.U_CPU
8:              U_RAM ← U_RAM + job.U_RAM
9:              U_DISK ← U_DISK + job.U_DISK
10:         speed ← 1
11:         if U_CPU > 1 then
12:             speed ← speed/U_CPU
13:         if U_RAM > 1 then
14:             speed ← speed/exp(U_RAM − 1)
15:         if U_DISK > 1 then
16:             speed ← speed/(U_DISK/1.5)
            return speed
```

3.4.1 Test Scenario

In our simulations, we defined and used 9 different virtual machine types, inspired from Amazon EC2 instances [6]: 3 general purpose computing instances, 2 compute intensive instances, 2 I/O intensive instances, and 2 memory intensive instances. They have different hourly costs and boot times, as it can be seen on 3.1.

We also defined 4 different job types, where their resource demands (utilization) and completion times differ from each other. We defined a web server job (Table 3.2) that uses mostly the disk, but also a considerable amount of CPU and RAM. As another job type we defined a RAM intensive job (Table 3.3) where RAM is used for caching purposes as well, and a fair amount CPU is also used,

Table 3.1: Virtual Machine Types.

VM Name	Hourly Cost ($)	Average Boot Time (min)
avg1	0.070	10
avg2	0.140	12
avg3	0.280	14
compute1	0.420	6
compute2	1.680	4
disk2	1.705	7
disk2	6.820	5
ram1	0.700	7
ram2	2.800	5

but disk usage is minimal. In addition, we defined a CPU-intensive job (Table 3.4) where no matter on which machine it runs, CPU is fully utilized and a fair amount of RAM is used, but again disk usage is minimal. Finally, we defined a long-running, but a low resource utilizing job type (Table 3.5), where each resource is lightly used, however, it takes relatively long time to complete the job, compared to the other three job types. Each table shows how much a job of the corresponding type can utilize the three types of resources (CPU, RAM and disk I/O bandwidth) for different VM instance types. Full utilization of a resource is indicated with 1. Times are expressed as unit time. The tables also show how long it takes on the average to finish a job of certain type in various VM instance types.

Table 3.2: Web Server Job

VM	CPU	RAM	IO	Time (min)
avg1	0.3	0.2	0.9	30
avg2	0.2	0.1	0.8	25
avg3	0.1	0.1	0.8	20
compute1	0.2	0.1	0.6	20
compute2	0.1	0.1	0.5	16
disk1	0.1	0.1	0.1	15
disk2	0.1	0.1	0.1	10
ram1	0.2	0.1	0.6	20
ram2	0.1	0	0.5	16

In our simulation experiments, we defined various workloads to cover possible

Table 3.3: RAM-Intensive Job

VM	CPU	RAM	IO	Time (min)
avg1	0.7	1	0	15
avg2	0.5	1	0	14
avg3	0.4	1	0	13
compute1	0.3	0.9	0	10
compute2	0.2	0.8	0	9
disk1	0.5	0.9	0	14
disk2	0.4	0.8	0	12
ram1	0.3	0.4	0	4
ram2	0.2	0.1	0	8

Table 3.4: CPU-Intensive Job.

VM	CPU	RAM	IO	Time (m)
avg1	1	0.5	0	30
avg2	1	0.4	0	22
avg3	1	0.3	0	18
compute1	1	0.4	0	11
compute2	0.9	0.2	0	3
disk1	1	0	0	20
disk2	1	0	0	16
ram1	1	0.1	0	15
ram2	1	0.1	0	12

use case scenarios. We define the jobs with arrival and required completion times (i.e., service times). In addition, jobs have utilization vectors on each virtual machine type. In our experiments we have synthetically generated jobs with various count and time-windows, whose results can be found in the next section.

3.5 Simulation Experiments Results

In our simulation experiments we observed that our methods perform better than naive allocation of the tasks, where the virtual machines are only responsible for one task at a given time. Our methods, where we allow many tasks to run together

Table 3.5: Long Running, Low Utilization Job

VM	CPU	RAM	IO	Time (m)
avg1	0.2	0.1	0.1	150
avg2	0.1	0.1	0.1	130
avg3	0.1	0.1	0.1	120
compute1	0.1	0.1	0.1	80
compute2	0.1	0.1	0.1	70
disk1	0.1	0.1	0.1	110
disk2	0.1	0.1	0.1	90
ram1	0.1	0.1	0.1	110
ram2	0.1	0.1	0.1	90

in a machine, easily achieve higher levels of utilization in a virtual machine and avoid allocation of extra virtual machines and causing cost.

Despite the fact that our methods are better than the naive methods in many aspects, our scheduler has an option to choose between cost and performance (in this way make a tradeoff) by allowing over-utilization of the machines and causing tasks to run at slower speeds. Running tasks at slower speeds causes the virtual machines to be ON for longer times, but, the overall cost may be reduced because the number of machines used is also reduced. As it can be seen on Figure 3.2 and Figure 3.3, allowing over utilization reduces the cost from $105 to $40, but it increases the total run-time from 1198 minutes to 1678 minutes, which may still be acceptable.

In our experiment with 100 jobs in 30 minutes time window, it can be seen in Table 3.6 that our methods outperform the naive methods very significantly. While our method completes 100 tasks in 1085 minutes with only $13.86 cost, the closest naive method completes these tasks in 794 minutes for $95.06 cost. As one might conclude, by favoring %36 increase in the time, we save 5.85x cost, which is a very significant amount. In terms of our proposed *cpp* metric, naive method performs at 21.6511 unit, whereas our best method performs at 12.2381 unit. Decreasing *cpp* metric indicates that our method is becoming effective when cost rate and performance are considered together.

Table 3.6: Top-performance comparison of 3 methods with 3 strategies.

Allocation	**Strategy**	**Vms**	t_{extra}	**Time**	**Cost**	**cpp**
Intelligent	CostPerf	11	5.41	1075	13.86	12.2381
CountLimited	CostPerf	33	1.57	794	32.34	13.6297
Intelligent	Cheapest	25	5.96	1842	54.25	15.5058
Naive	CostPerf	97	1.14	794	95.06	15.7861
CountLimited	Cheapest	34	3.65	1738	69.02	15.9293
Naive	Cheapest	99	1.08	1079	124.74	16.6363
Intelligent	Fastest	12	5.58	589	429.49	18.5036
CountLimited	Fastest	32	1.61	520	769.77	19.546
Naive	Fastest	87	1.16	508	2231.19	21.6511

Table 3.7: Virtual Machine Speed vs. cpp.

Speed	**Time**	**Cost**	**Extra**	**Vms**	**cpp**
0.1	1878	228.48	5.65	102	18.4009
0.2	1656	211.68	3.41	108	18.1223
0.3	1534	205.65	2.58	113	17.988
0.4	1522	236.59	1.9	130	18.2604
0.5	1466	245	1.53	140	18.2928
0.6	1371	280.14	1.18	174	18.4939
0.7	1371	304.29	1.13	189	18.6593
0.8	1407	386.4	1.06	230	19.163
0.9	1371	516.81	1.02	321	19.7186
1	1371	523.25	1.02	325	19.7434

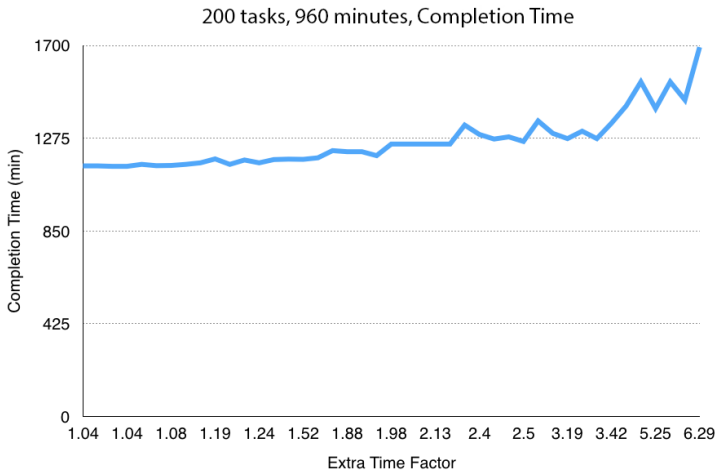


Figure 3.2: 200 tasks/960 minutes completion time vs extra time factor

From the results it is obvious that the fastest method is the naive one, allocating the fastest virtual machine type for each task individually. With this we will reach the minimum time, however, we will have a large cost. As it can be seen from Table 3.6, our methods are performing best according to the *cpp* metric which is considering both cost and completion time. Also note that t_{extra} that indicates the extra time that a task waits to be completed in average increases as we stack the jobs together.

The performance improvement is not a direct goal of our proposed method. However, it can be effected by playing with the *DesiredUtil* variable of our algorithm, which is the variable that decides how slow can a virtual machine run and still be acceptable. We obtained results for our *Utilization-based* method while varying the speed of VM from 0.1 to 1, in 1000 tasks and 240 jobs case, which can be seen in Table 3.7. As results show, although a VM can be stacked with many jobs, it is not always a good idea. For instance, at the time when VM speed was 0.5 cost is 245 and time is 1466, which may be better than full VM speed (1.0) where cost is 523 and time is 1371. Then, we allowed VM to work at 0.3 speed, time increased to 1534 and the cost decreased to 205.65. However, after slowing down further below 0.3 speed, time increases substantially to 1878, and cost also increases to 228.48, as a result of increasing number of context switches. As it can be easily interpreted from these results, there exists a sweet spot for the VM

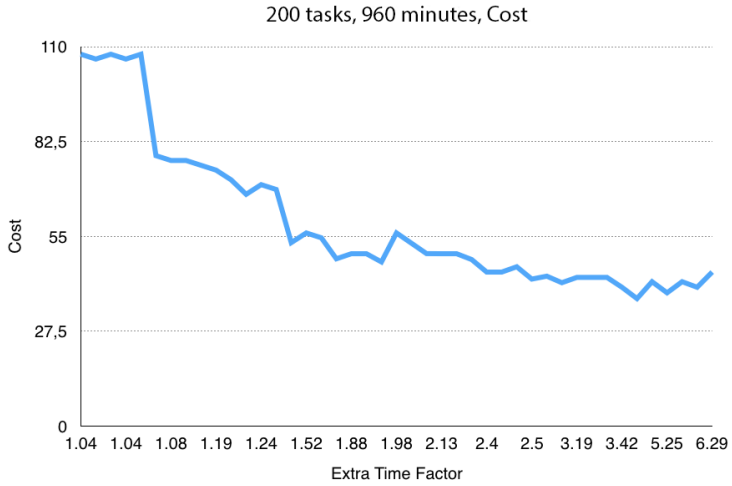


Figure 3.3: 200 tasks/960 minutes completion time vs extra cost factor.

speed in order to achieve a good cost and time performance.

3.6 Testbed Experiments

As the importance of scheduling different tasks together has been shown with simulations, we have taken steps to replicate the same intuition in a real environment. As each job's actual resource utilization is not known before actually running it, we first run each job several times in a set of virtual machines in our test infrastructure. In this way, we try to estimate/predicate their expected resource usage, which is used to schedule the jobs in later runs.

3.6.1 Example of Running Different Tasks Together

First, to have an idea how our approach would work in practice, by synthetically creating two types of tasks with different workloads, we measured task completion times in three types of virtual machines, where VM1 had 1 core, 512 MB RAM, VM2 had 2 cores and 1024 GB RAM, and VM3 had 4 cores and 4096 GB RAM, and a (SSD) solid state disk. After running tasks individually on each of these

virtual machines, we ran some of the tasks together and measured the completion time again. In Table 3.8 shows the results. In the table we have tasks of class c which stands for CPU bound tasks, and class d which stands for Disk I/O bound tasks. The label **c1-c2-d1**, for example, means that tasks **c1**, **c2** and **d1** have been run together.

By examining the completion times from Table 3.8, we can see the benefits of our approach. As we see in the table, in the average running c1 takes 114 seconds in VM1, and d1 takes 1005 seconds. However, when we run them together, d1 takes 996 seconds to complete while c1 takes 112 seconds to complete. Since the values are very close, that means performance is not affected. As these tasks are from different classes, where c1 uses the CPU and d1 uses disk most of the time, they do not affect each other, since most of the time task d1 waits on I/O and has very little work with CPU. As another example, running c1 and c2 together in VM1 causes them to finish in 270 seconds, since they share the same limited resource, CPU. If they would run individually, they would take a total of 231 seconds, however, they took 270 seconds, which is an approximately 16% slow down in total. However, when disk jobs d1 and d2 have been run in VM2, they took approximately 1002 seconds each, but when they have been run together, they took 1591 and 1335 seconds. If they were run one after other they would take 2000 seconds. The reason that time did not increase linearly is that although I/O is a shared resource, multiple I/O requests from multiple processes can be served in a single disk I/O operation. This fact supports our utilization model, where we did not take the disk I/O slowdown linearly. Finally, when we run different types of jobs together, such as running d1-c1-c3 in VM1, job d1 slows down very little, and c1 and c2 finishes almost at exact duration as in c1-c2 run case. In short, we see the benefits of running tasks of different types together in practice.

However, although we have seen an improvement of runtime, we knew the tasks beforehand and scheduled them according to their classes to get an improvement. To apply this idea in a real setting, we need a mechanism to run the tasks together and measure their resource utilization and learn about their

Table 3.8: Example task completion in different VMs.

	VM1	**VM2**	**VM3**
c1	114	109	81
c2	117	76	80
c3	114	34	81
d1	1005	1003	139
d2	988	1002	191
c1-c2	270-269	110-109	161-160
c1-c2-c3	381-384-381	189-190-186	203-202-203
d1-c1	996-112	986-110	142-42
d1-c1-c3	1090-274-275	1013-112-115	179-167-167
d1-d2	1536-1335	1591-1335	223-251

characteristics. After running tasks together, we can finally have a better understanding of their resource needs for a procedural scheduling of them. Next we discuss some methods and tools that we can use to account resource usage.

3.6.2 Identification of Task Resource Utilization

After preparing our testbed environment, we could create VMs on demand and assign jobs to each VM by using Docker containers to isolate the jobs. Each job is run in a different container. First, by using `sysbench` tool and `redis-benchmark` tool, we defined a set of tasks and run them in multiple VM types to observe their resource usage. We considered the fact that, although jobs can be of the same type, they might have different amount of work to do which would cause them to finish in different times.

By using sysbench tool, we have created CPU bound jobs, with 1, 2 and 4 threads with varying number of requests. In addition, we used the sysbench tool's fileio test suite to create randomly filled files and test the disk I/O in random read mode. Lastly, sysbench allows to benchmark a MySQL database. We created a 1.000.000 row test table and performed query benchmark. Lastly, as a different tool, we used redis-benchmark which allows to benchmark the in-memory Redis key-value store and cache database.

We run the above benchmarks against the previously defined VM types (Table 3.9) and collected information about their resource usage. As it can be guessed, fileio stressed the disk mostly, whereas the redis-benchmark and sysbench CPU test suite stressed the CPUs. For instance, the resource utilization of MySQL benchmark that has been run on virtual machine avg1 can be seen in Figure 3.4. MySQL benchmark utilizes 80% of the CPU and uses approximately 180 MB of RAM while there are approximately 250 I/O requests per second. As another example, in Figure 3.5, we see that on average one CPU is used, whereas RAM usage is only around 369 KB and there is no active I/O usage.

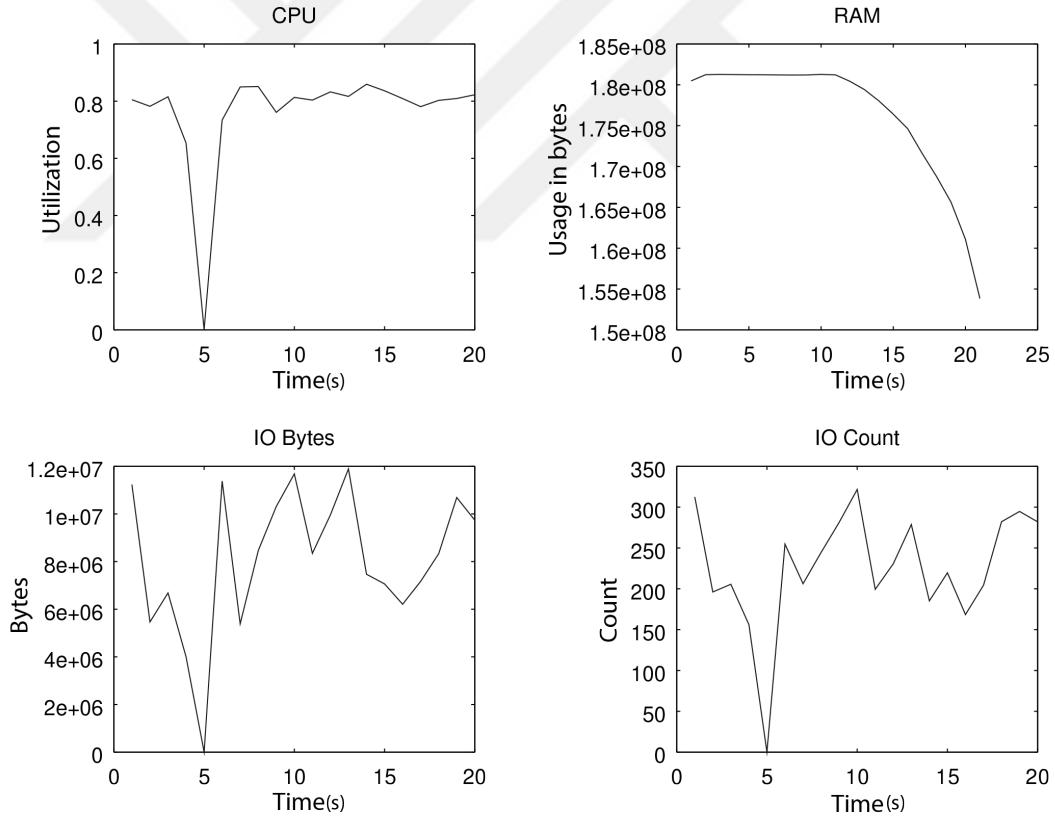


Figure 3.4: MySQL Benchmark Task on Virtual Machine **avg1**.

By using the container resource accounting methods, we measure the CPU, Disk and RAM usages of each task in each virtual machine. Accounting CPU and RAM usage is easier than accounting the I/O usage since they can be easily quantified as a percentage of the maximum resource capacity. However, for I/O,

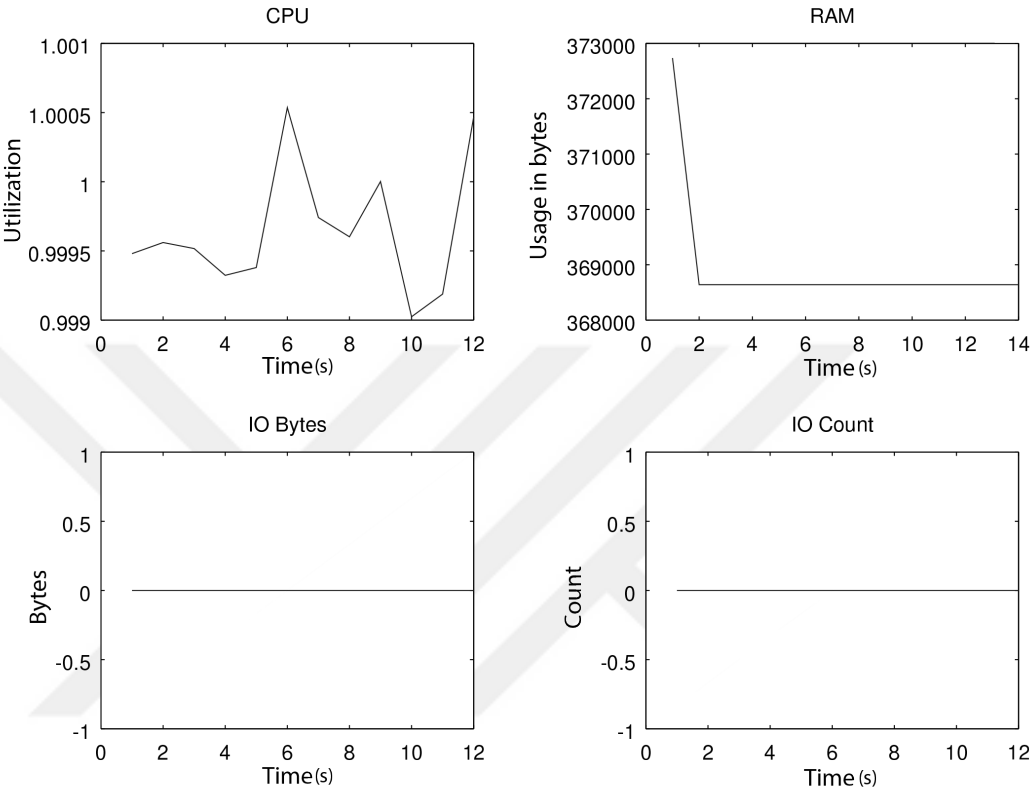


Figure 3.5: CPU Benchmark Task on Virtual Machine. **avg4**.

disks do not have very certain performance specifications. Because, access patterns to the disk affects its performance. Random I/O requests are much slower than sequential I/O requests, even if the underlying disk is a solid-state disk. Considering this, we accounted the CPU as utilization, RAM as total usage in bytes, and Disk as served I/O requests per second.

As it can be seen from Table 3.10, there are some interesting results. Cpu1 task is a simple threaded task and it does not make sense to use a multi-core VM for only that task, since it shows no improvement from avg1 to avg4 both in utilization and average duration. However, since cpu3 is a 4 independent threaded task, we see CPU can be linearly utilized when moving from avg1 to avg4 as utilization is increased from 0.94 to 3.94 while duration is reduced from 134.5 seconds to 18 seconds.

As another example, disk_rndrd is a random read benchmark and we can see

the improvement in disk I/O when an SSD virtual machine is used. However, despite 1.74x increase in I/O performance, the completion time of the benchmark only decreases by 0.25x.

We have run the MySQL task on avg2, ssd2 and ssd4 virtual machines. Although the disk I/O performance of these VMs are close to each other, we see that there has been an improvement in I/O performance when an SSD based virtual machine is used. The increase in I/O performance is also reflected in duration (completion time), which is reduced from 347 seconds to 253 seconds.

Lastly, when we examine the task redis, we see that in avg2, a VM with 1 CPU, the task has 0.94 utilization and in highcpu, a VM with 8 CPUs, the task has 1.75 utilization, which leaves the 78% of the CPU power idle when run alone and it only speeds up by a factor of 17%.

As a conclusion from the above values, we see that not all tasks use the resources completely, and it does not always mean that increasing the resource capacity linearly would grant linear speed up. As it has been shown earlier in the simulations, it is imperative to choose virtual machines wisely to pack and schedule jobs to them. For instance, using redis task in a highcpu VM make it faster, however, causes resources to be underutilized. In addition, we see that cpu3 task benefits from a many core machine in a much better way by fully utilizing the CPU and in this way having a reduced runtime. But even a task utilizes a CPU fully, it may not be using the other resources such as disk and RAM all the time, which leaves room for improvement by bundling different tasks together.

In short, we examined the resource utilizations of various type of tasks and their completion times on various VMs. Based on this, we can determine the best VM type for each different type of task (Table 3.11) and calculate the expected speed of the VMs for bundling tasks together.

3.7 Summary

In this chapter, we have first discussed that naive job allocation method and observed that one virtual machine per task is not very efficient in terms of utilization and cost. Instead, we proposed to stack many jobs into virtual machines to achieve greater utilization and reduce costs. To do this, we made use of container based virtualization to achieve both isolation and resource accounting of tasks. We showed that even over-utilization can be beneficial, since it can lead to decreased costs as long as it is kept on sensible levels. Because, even if the tasks slow down, we avoid allocating a new VM. By this way, we do not wait for VM to be ready, nor we pay for its cost upfront.

Table 3.9: Virtual Machine Types

	CPUs	RAM	SSD?
avg1	1	512	no
avg2	1	1024	no
avg3	2	2048	no
avg4	4	4096	no
ssd1	1	512	yes
ssd2	1	1024	yes
ssd3	2	2048	yes
ssd4	4	4096	yes
highcpu	8	4096	no
highram	4	8192	no

Table 3.10: Resouce utilization of some tasks in different types of VMs.

Job Type	VM Type	CPU Time	DISK I/O	RAM (MB)	Duration (s)
cpu1	**avg1**	0.94	0	0	112
cpu1	**avg4**	1	0	0	119
cpu3	**avg1**	0.94	0	0	134.5
cpu3	**avg4**	3.94	0	0	18
rndrd	**avg2**	0.35	1067.10	0	73
rndrd	**ssd2**	0.38	1858.42	0	55
mysql	**avg2**	0.73	186.22	172.80	347.6
mysql	**ssd2**	0.5	193	170.77	253.6
mysql	**ssd4**	1.03	228	176.65	221
redis	**avg2**	0.93	0	108.74	104
redis	**highcpu**	1.75	0	85.11	66

Table 3.11: Selected VMs for Jobs according to test results.

Job	VM
cpu1	avg1
cpu2	avg3
cpu3	avg4
rndrd	ssd2
mysql	ssd4
redis	avg4

Chapter 4

PAGS: Programming Assignment Grading System with Containers

In the previous chapter we presented our approach and methods to allocate and schedule user jobs into virtual machines of a cloud computing system by use of container technologies like Docker, so that costs are reduced and VM capacities are utilized in a better way. Our approach provides significant improvements in terms of cost and VM utilization as compared to previous naive allocation methodologies, which is shown via both simulation and testbed experiments.

We adapted our solution to a web based cloud computing service that we designed and implemented for a real problem that instructors and students of programming courses are facing. We call our system as PAGS: Programming Assignment Grading System. Our primary goal in implementing PAGS was to ease the execution, testing, evaluation, and grading process of programming assignments in the programming courses. In this chapter we will present the design and implementation of our PAGS system, including its task scheduler, which is adapted from our approach described in the previous chapter.

PAGS provides a web-based IDE and an isolated, scalable execution environment. PAGS runs student submissions, which can be considered as un-trusted

code. Such execution environment would require enormous resources if each student would be allocated a separate virtual machine, which would make a system like PAGS unfeasible. However, since our proposed bag-of-tasks scheduling and allocation methodologies are very efficient and robust, we could apply the same idea to PAGS and achieves a both usable and scalable system.

Next, we describe the problem PAGS solves, the architecture of PAGS, and how it uses our container based task allocation method to achieve robustness, high utilization, scalability and cost effectiveness. We also present analysis of student behaviour on PAGS and the correlation of behavior with the received grade.

4.1 Problem

Programming assignments are essential parts of the programming courses, since they provide hands-on experience to the students. However, as the number of the students grows larger in a course, the grading process of the assignments can be tedious, unfair and can take considerable amount of time. This is mostly caused by the manual labor work needed by a grader. Typically, the students upload their submissions to systems such as Moodle [35], Blackboard [36], or they directly e-mail it to the grader. Hence, even fetching and organizing the assignments can cause a cumbersome process, and is prone to error.

Another problem with grading programming assignments is actually executing the submissions of the students. If a well-defined execution plan is not defined before the deadline, executing each submission can be unique and time consuming process. Even if a perfect execution plan is given, one must consider the security and stability of the execution environment. A running process can leave a trace in the system that may affect the upcoming executions of the submissions. For instance, in Operating Systems course in Bilkent University, students need to use system-wide shared resources, such as named message queues, pipes, sockets, and files in the projects. When all these are not released properly after an execution

of a student project, they can cause the next execution to fail. Alternatively, in the same course, students need to create processes, write and read files, which can be a threat to the system if there is a protection weakness. They can overload the system or destroy important data, both intentionally or unintentionally. To overcome this problem, a grader must setup an isolated and resource-controlled environment, and ideally create and destroy these environments upon each run.

Lastly, due to differences between the teacher's grading environment and the student's development environment, a student submission may fail although the student has a valid submission. This problem can be caused by version differences between software, such as compilers or tools, or the environment itself affecting the execution. This ambiguity can cause a student to object to her/his grade after the grading process. This brings round-trips between students and teacher, and therefore more manual work can emerge for both parties.

Although there exists some platforms that address these issues differently, none of them address the scalability and isolation issues together, where a bunch of programs belonging to many students run on a centralized system. Additionally, most of the solutions require restricting rules, which creates platform or language dependencies.

Our solution PAGS, which includes a scheduling component, allows any type of assignment to be executed in an isolated and secure environment. It has scaling out features, which enables the number of servers to increase when the load, i.e., the number of student programs running in the system, increases. Also by providing a web based IDE, PAGS makes it easy for students to write their code in any machine with Internet connection.

4.2 PAGS System

4.2.1 Architecture

In PAGS, there are users, students and teachers, using web browsers to access the system. The system is running in the cloud as a Software as a Service (SaaS). Backend of PAGS that is deployed on the cloud which consists of a web server, file server, and execution servers as it can be seen in Figure 4.1. Execution servers may be one or more virtual or physical machines, each capable of running Docker containers. Web server saves the files of users to reliable file storage and schedules the run requests to a pool of execution servers available. Since execution servers are independent of the state and gets the required files from an external server, they can be easily scaled on demand.

The PAGS system uses the following files and components.

- **Files**
  - **Required:** The files that students need to write, edit and submit.
  - **Supplied:** Read-only files that are supplied by the teacher. They can be used as providing test-cases or sample code, or the homework explanation such as PDF or text files.
  - **Output:** The files that need to be present at the end of running the execution script. They are persisted and shown in user-interface after the run. They can be used to compare with the supplied files, to see if the output of the students are the expected ones.
- **Script:** A Bash script that runs the assignment. Teacher specifies how the student program should be run. For instance, teacher can choose to compile and execute program a few times and compare the outputs with expected ones. If desired, other languages such as Python can be used as the scripting language.

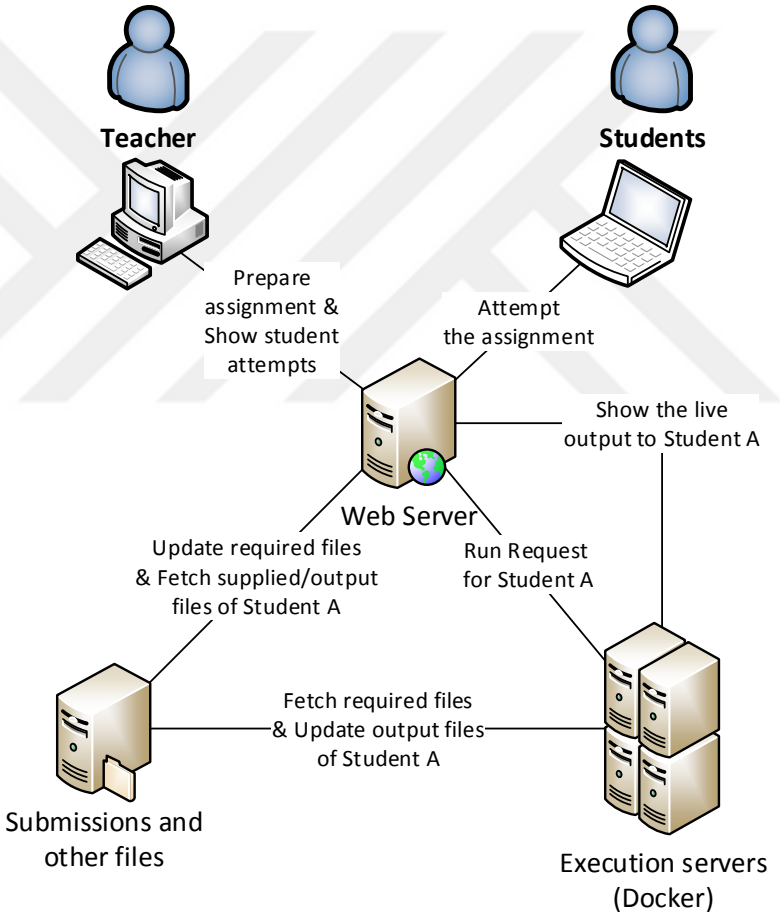


Figure 4.1: Overview of the PAGS architecture.

- **Terminal:** A terminal part of the user interface allows students to provide live interaction such as key strokes (including control signals), or stopping the assignment, while testing their programs.

- **Grading Components:** These define how to grade an assignment for a grader and sets the maximum available point. Therefore, grader looking at the output of the code determines the grade for each component, and assigns it with a text feedback. The automatic grading component is not implemented yet, but it can be easily implemented by comparing the output files with some set of supplied files, if desired.

By properly defining how an assignment to be executed and what it depends on allows us to create a standard among the students. By this way, students know how each assignment is going to be executed and in which environment. We eliminate the round trips between student and teacher after grading process. Also this definition allows the code to be executed in any container on any Linux host, which enables us to have flexibility in scheduling the execution requests, which helps the system to easily scale out horizontally.

4.2.2 Additional Security and Stability Measures

Since we allow executing arbitrary code, we need to have some resource usage control mechanisms for executions. By default, Docker containers have no resource limits and a container starts as root user. First, we use **nobody** user as a starting user. Additionally, by using cgroup limits, we assign a maximum memory of 256 MB to each container. This memory is only for the processes in the container, i.e., operating system utilities are not included in this usage. We also use `ulimit` to restrict the number of processes and number of open files in a container. In addition, for each assignment a timeout value is specified, which causes the execution of the assignment to be stopped after the specified time elapsed.

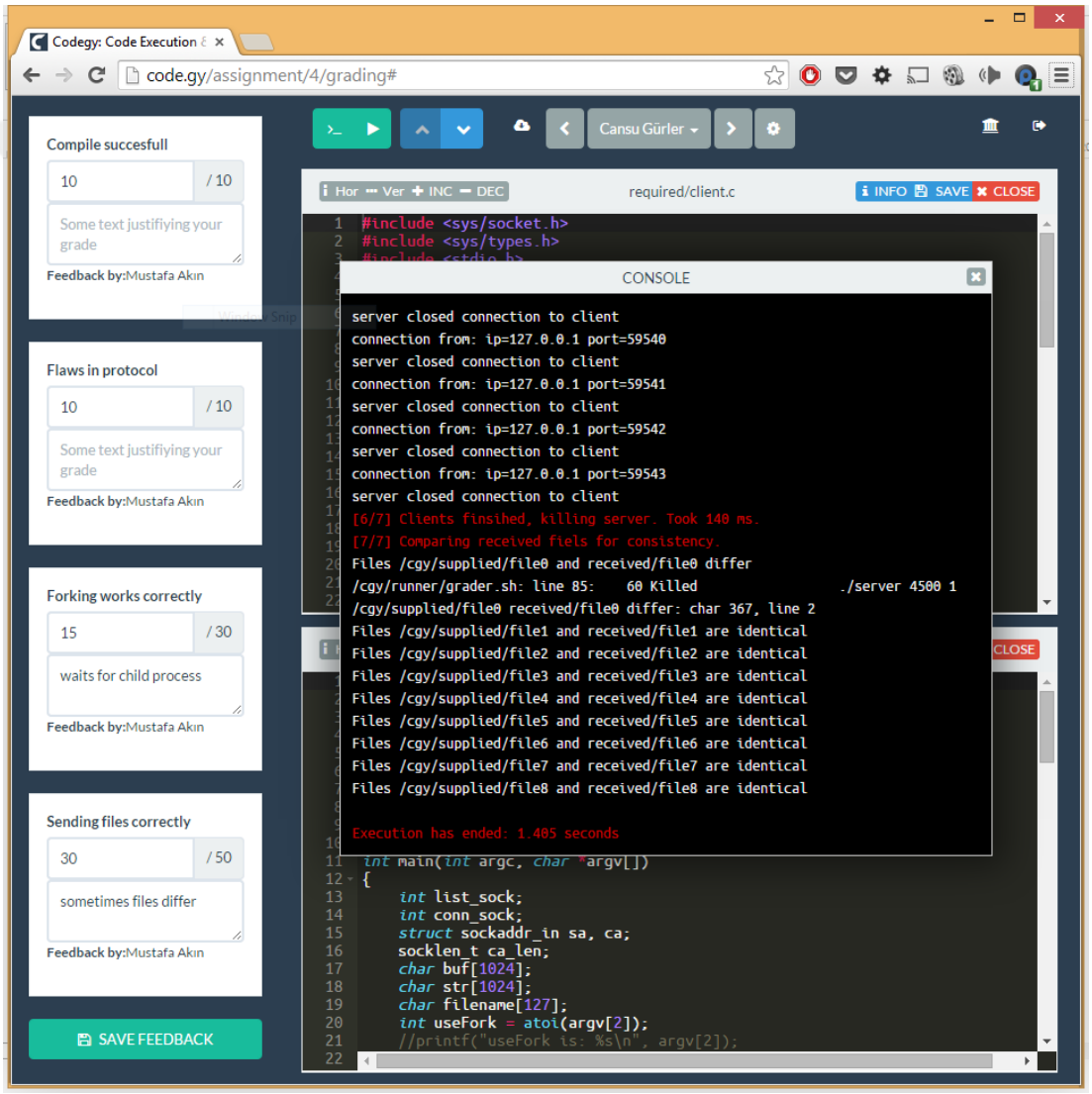


Figure 4.2: Screenshot of the current system.

4.2.3 Scheduling Assignments on Servers

The load on a server can increase beyond an acceptable level while testing and evaluating the assignments of a large number of students. Therefore, it is important to be able select and use a different execution server when we face a lot of students causing high utilization. Since student codes run in independent containers, there is no general restriction in selecting and using a new server.

We assume we have at least one server at hand that is always ready to accept jobs. As the scheduler receives jobs from the PAGS interface, it schedules them to run on the default server immediately.

For educational assignments in our course, almost all of the tasks run under less than 5 seconds, and most of them run under 1 second, as it can be seen in Figure 6. For this reason, considering a typical student assignment execution, a scheduling algorithm that makes complex decisions may be unsuitable for scheduling student codes onto servers. Therefore, we decided to design and implement a simple and practical scheduler that uses simple criteria for scheduling decisions. We simply measure the average tasks completion time in a given time window for each server, and act upon it. The average task completion time for a server can be expressed as follows:

$$t_{avg} = \frac{\sum_{\substack{t_{end}-now \\ <t_{window}}} t_{end} - t_{start}}{n_{tasks}}$$

Here, all tasks that started and ended in a specified time period (time window) are considered and their count is denoted with n_{tasks}. For each such task, t_{start} is the start time of the start and t_{end} is the finish time. Then, t_{avg} is the average job completion time (i.e., average turnaround time).

To simply put, our scheduler has the ultimate goal of minimizing the overall job completion time of the submitted jobs, i.e., student programs submitted for execution. Therefore, the scheduler keeps track of the job completion times for over a predefined window of time for each server, and if within the window the

average job completion time gets over a threshold, scheduler allocates one more server. With a server we mean a virtual machine (VM) that has enough physical capacity for its specification (virtual cpu, memory, etc.). Using a large enough window ensures that temporary spikes do not affect judgment, and a small enough window ensures that the current state is captured.

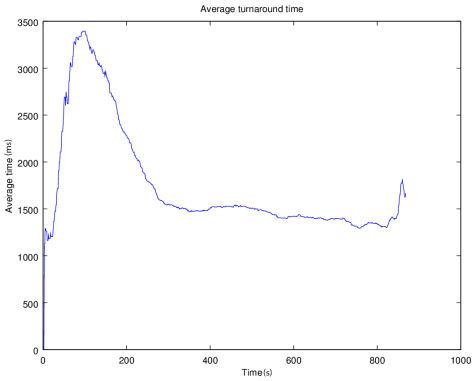
To select from a set of servers, the scheduler assigns a job to the server that has the minimum number of running jobs, as similar to our Algorithm 3. The reason that we did not choose utilization-based stacked scheduling is that since the tasks were very short, they caused utilization to fluctuate in an unreliable manner. A physical server is not overbooked by the virtual machines placed onto it. In this way we allocate new jobs on the newly created servers and we control and reduce the load on the existing servers. In our experiments, the average virtual machine boot time was around 20-25 seconds, and this allowed us to scale in a quick way.

Finally, if we have more virtual machines than needed, we request them to be closed, since we also need to minimize the cost. This may result with a physical machine to be turned off as well if there is no need for it any more.

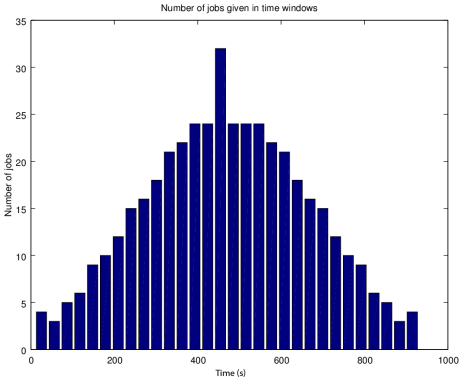
The details of our scheduling algorithm can be seen in Algorithm 6. Our experiments in Section 4.2.4 show that this algorithm can quickly adapt to varying load and can provide a good average job completion time.

4.2.4 Evaluation of the Scheduling Algorithm on Testbed Environment

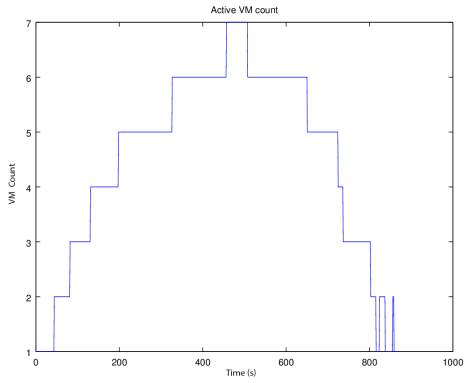
We evaluated our scheduling algorithm (Algorithm 6) in our testbed environment. Because we did not have enough real data to generate enough load for stress testing, we synthetically generated load using `sysbench` [37] tool for a small period of time. The distribution of jobs that have been used in the test environment can be seen in Figure 4.3b. The number of requested jobs starts small. Then it increases linearly and then decreases in the same speed. This increase immediately generates a load in the system that needs to be handled. As it can be seen from



(a) Average duration of executions in 5 second windows.



(b) Job arrival distribution.



(c) Active virtual machine counts.

Figure 4.3: Scheduling algorithm evaluation on synthetic load.

Algorithm 6 Scheduler

```
    procedure SCHEDULER
2:      while true do
            job ← receiveJob()
4:          for each vm in vms do
                if vm.jobs < minVm.jobs then
6:                  minVm ← vm
            minVm.schedule(job)
8:          ratio ← vms.count()/jobs.count()
            if avgCompletion > threshold then
10:             if ratio < 0.3 then
                    requestNewVM()
12:         if avgCompletion < threshold * 1.5 then
                if ratio > 0.75 then
14:                 if vms.length > 1 then
                        vms.pop().close()
```

Figure 4.3c, new VMs are allocated quickly. However, since allocating VMs can require some time, as can be seen from Figure 4.3a, the average job completion time starts to increase for a while, then it settles. When the average job completion time drops, unneeded virtual machines are closed to lower the costs, as it can be seen from Figure 4.3c. Note that the algorithm does not require any information on when the jobs will arrive and adapts to changes in load, as can be seen from our experiment results.

4.2.5 How a Student Interacts with PAGS

We used PAGS to run and grade student assignments in Operating System course given in Bilkent University. This allowed us to collect extensive data about assignment development and testing behavior of students. With this data gathered in PAGS, we have made some analysis on the data collected related with three programming assignments which was given as two-week long projects as part of the course.

The first assignment was to create a multi-threaded word-counter that was

written in a map-reduce style. The assignment was to be done in C language, pthreads usage was crucial, and the data between processes and threads was transferred using files (just for educational purposes). Second assignment is essentially the same, however, the data is transferred using bounded in-memory buffers. Finally, the latest assignment was inspecting a FAT32 file system and finding out the blocks of a given filename.

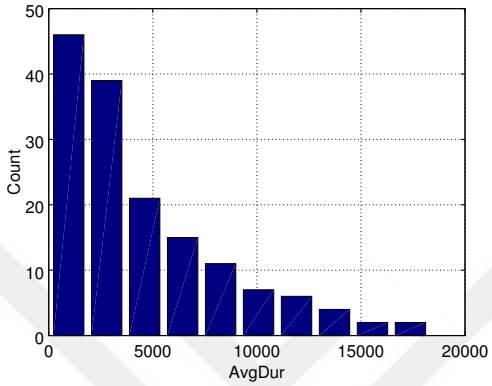
Error	**Count**
variableScope	327
unreadVariable	206
memleak	160
unusedVariable	159
resourceLeak	103
checkCastIntToCharAndBack	52
memleakOnRealloc	51
nullPointer	42
invalidPrintfArgType_sint	32
redundantAssignment	31
uninitvar	31

Table 4.1: Style errors reported by `cppcheck` tool.

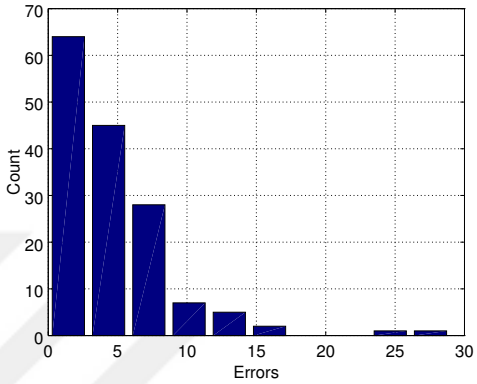
In Figure 4.4a the average execution duration of the submissions can be seen. For most students the execution duration is less than 5 seconds, while a huge portion is under 2 seconds. There are some cases that students have exceeded 10 seconds, but they are rare occasions.

We also made static code analysis on student submissions using `cppcheck` tool (Table 4.1). The analysis results show that majority of the students have less than two errors on average. But a considerable number of students have errors in their submissions. Considering the Figure 4.5b, less number of errors do not mean higher grade for a student, however, as the errors increase we see a slight decrease in the received grade. By using the static analysis tool, we also see the common mistakes and guide the students accordingly, which we find very helpful in teaching programming courses.

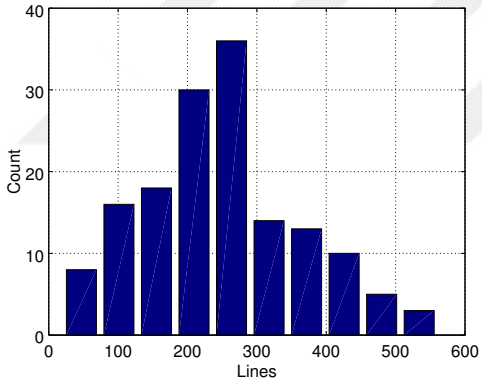
We also measured the lines of code in submissions and see that there is a good



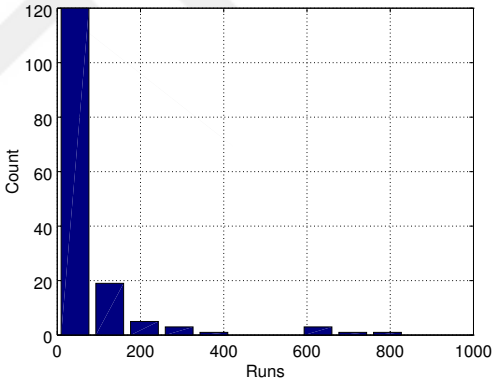
(a) Average duration of execution.



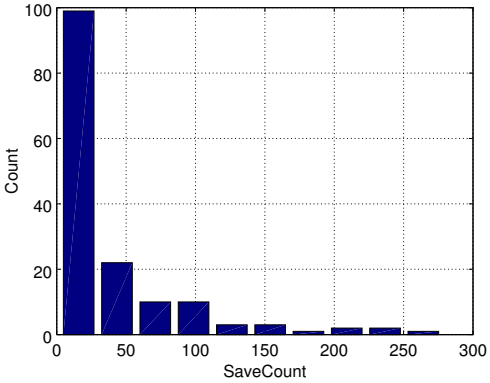
(b) Style mistakes and errors.



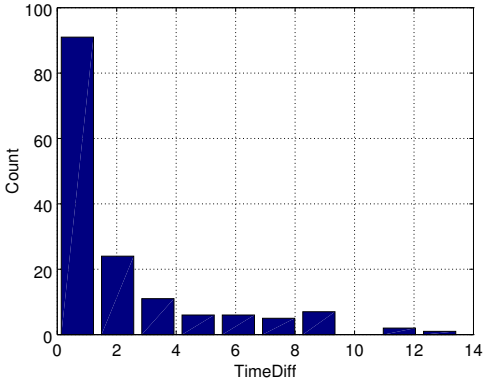
(c) Number of lines in codes.



(d) Number of runs.



(e) Number of saves.



(f) The duration spent on PAGS.

Figure 4.4: Student behavior analysis on PAGS.

relation between number of lines and grades. Higher lines of code in a submitted program implies higher grades.

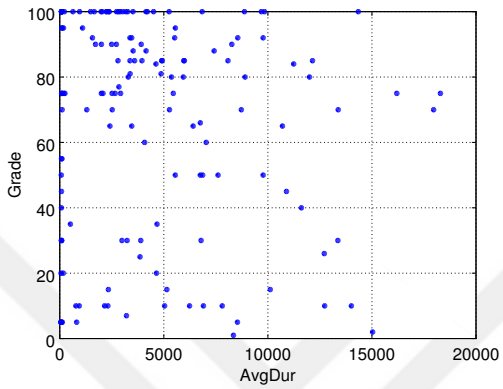
In addition, we measured the number of runs and saves, and see that majority of the students run their code less than 50 times in an assignment. This shows that there is no direct relation between run count and received grade.

Finally, we also measured when a student started and finished working on an assignment. As can be seen from Figure 4.5f, the majority of the students used PAGS in the last day, although the assignments are given with 2 week due. As Figure 4.5f shows, starting early on PAGS does not directly correlate with higher grades, however, it can be deducted that students that started working on PAGS a week earlier have received grades higher than 60. It should be noted that students might have started to work on their local machine and decided to test their code on PAGS later.

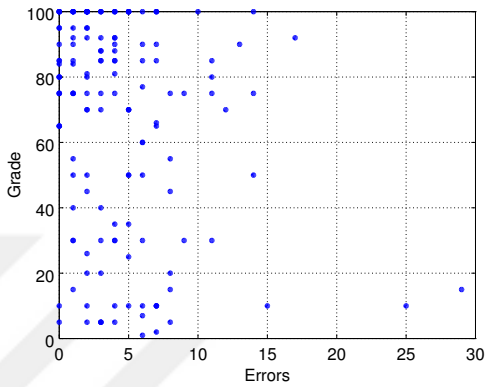
4.3 Student Opinions

A survey conducted among the students that took CS342 Operating Systems course in Spring 2015 semester have shown that most of them find PAGS useful and it saves them time while working on assignments. Although PAGS could assign automatic grades upon their submissions, as we also have foreseen, students prefer semi-automatic grading in which grader sees the results of executing code in PAGS, interferes with grading and gives the grades manually. This consists of examining the code the students wrote and ensuring that they did not cheat by using workarounds to generate the desired outputs.

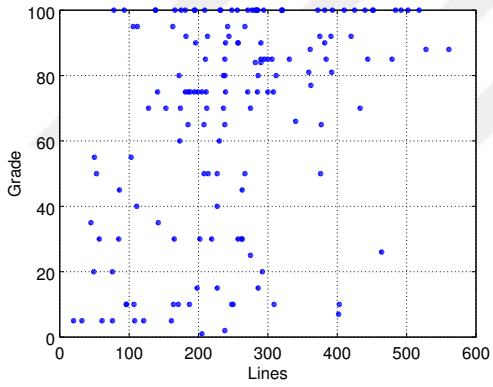
Another feature requested by students is a more flexible environment. Currently, PAGS only runs the script given by assignment creator and it can be hard to debug submissions. Students cannot directly run their own processes, such as debuggers like GDB, and this complicates debugging on the PAGS. We plan to add another terminal that does not run a given assignment script, but instead



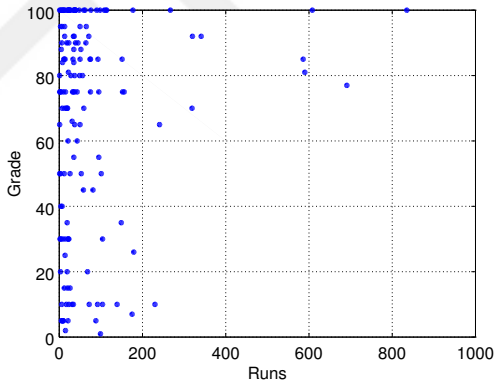
(a) Average Duration of execution.



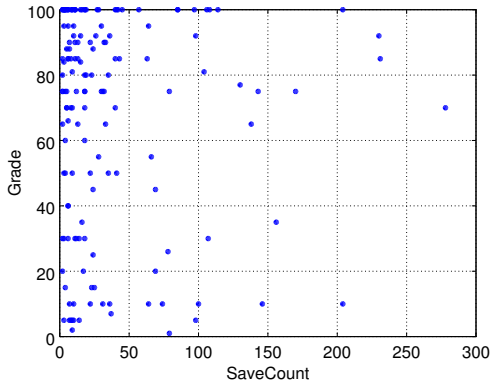
(b) Style mistakes and errors.



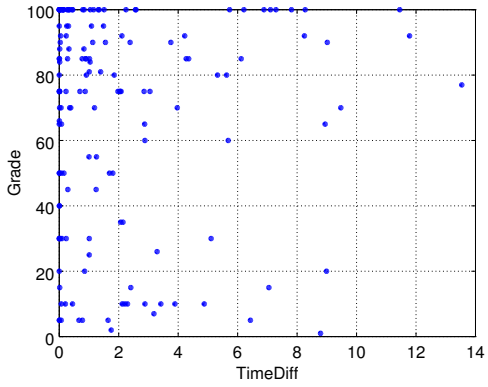
(c) Number of lines in codes.



(d) Number of runs.



(e) Number of saves.



(f) The duration spent on PAGS.

Figure 4.5: Students behavior correlation with grades.

provides an interactive shell that students can freely run their debuggers or other tools that they find convenient.

Lastly, students request PAGS to tell what is wrong with their code, but this is out of the scope of this thesis.

4.4 Summary

In this chapter, we introduced PAGS, that improves the grading process of the courses that has programming assignments. We proposed a semi-automated solution to grading the assignments. Our automation approach relieves both the graders and students from manual labor while providing flexibility in designing and running the assignments. PAGS uses the proposed scheduling approach in the previous chapter for scheduling the run requests from students and teachers. Using containers helps to schedule a request to any available computing resource (VM), and the idea of stacking many jobs to a single resource helps to reduce cost while offering acceptable performance.

Chapter 5

Conclusion

In this thesis we show that containers like Docker provide new ways of allocating resources on cloud computing environments and we provide some methods to schedule jobs on virtual machines in a consolidated manner via the help of containers. Containers as an alternative to virtual machines provide better performance and very fast start times. This allows us to schedule tasks in a more robust, scalable, and cost-effective way. Also, being able to run many containers in a single machine allows us to utilize machines better and develop novel approaches in scheduling tasks over available resources.

Through our work, we have stated that utilizing the VMs fully is important. However, as it has been stated earlier, it might not be possible to guarantee perfect utilization in every case. Tasks have different resource usage characteristics, and a task cannot always effectively utilize a virtual machine over the renting period. Therefore, we show that, in order to better utilize the available virtual machines, we can allocate more than one job to each of them, according to various strategies we propose. This alone allows significant improvement over naive approach: one virtual machine per-task. We may even allow over-utilization of virtual machines to trade time and reduce costs further. As a result of consolidating multiple jobs in VMs, we were able to increase utilizations of VMs and decrease the costs in this manner. This cost decrease is much more than the cost increase caused by

some increased elapsed time resulting from consolidation. In this way, we have shown that paradigm changes allowed by containerization brings new approaches to long existing problems.

We adapted our job scheduling approach in designing and developing a scalable web-based cloud service, called PAGS, that addresses a real problem faced by instructor and students. PAGS is a programming assignment development, testing, submission and evaluation environment, building upon old ideas of automating grading systems. PAGS, however, brings new ideas and solutions to concerns about scalability and isolation. As the assignments are decoupled from any servers, they can run in any allocated server, and this helps to scale as horizontally as needed. PAGS also abstracts the need of defining frameworks and languages, and helps teachers to create any kind of assignment by just specifying a run script. PAGS can easily be adapted to most programming courses as long as the test scenarios can be expressed as scripts in Bash scripting language.

PAGS is highly scalable and can scale horizontally. In our scalability experiments, by building up the previous experience of running different jobs together with containers, we have shown that our approach allows quick response to increasing workloads and allows the system to scale itself and use the resources more efficiently.

The experiments we run on PAGS also give interesting ideas about student behavior on assignments. We have evidence that most of the students start working 2-3 days before the assignments, irrelevant of how many days for the assignment is given. We can also see a correlation between the received grade and starting day of the assignment.

Students also are happier with this new system, as it shows them how their codes are being executed during grading process, and their runs are independent of the system state, and their actions do not leave traces that are hard to find. They can also work on any machine that has access to Internet and has a modern browser. This also helps the graders to give them feedbacks in a more convenient way.

Currently, PAGS is freely accessible to world as a software as a service in http://pags.cs.bilkent.edu.tr.

5.1 Future Work

As future work, the ideas on this thesis can be combined with the Docker's native tools that were announced lately. In addition, our work can be applied to hybrid Docker clusters using docker-machine to allocate virtual machines from public clouds such as Amazon EC2, Microsoft Azure or Google Compute Engine. In addition to virtual machines, since tasks can be isolated using containers, they can be scheduled directly on physical machines as well. There are additional performance metrics that one can use in physical environments, such as cycles per instruction (CPI) metric to evaluate the performance of each physical processor core. This might help better understanding of tasks and VM speeds, therefore might lead to better allocation choices at runtime.

As for PAGS, the ability to debug processes and run arbitrary processes on the code execution environment, such as interactive shells might be useful for students and graders. In addition, students might work on local editors, but ability to have the code compiled and run on PAGS can provide ease.

Bibliography

[1] Docker, "http://www.docker.com," Last accessed on 27 June, 2016.

[2] M. Shalom, A. Voloshin, P. Wong, F. Yung, and S. Zaks, "Online optimization of busy time on parallel machines," in *Theory and Applications of Models of Computation* (M. Agrawal, S. Cooper, and A. Li, eds.), Lecture Notes in Computer Science, Vol. 7287, pp. 448–460, Springer Berlin Heidelberg, 2012.

[3] C. A. F. De Rose, T. Ferreto, R. N. Calheiros, W. Cirne, L. B. Costa, and D. Fireman, "Allocation strategies for utilization of space-shared resources in bag of tasks grids," *Future Generic Computing Systems*, vol. 24, pp. 331–341, May 2008.

[4] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *11th IEEE/ACM International Conference on Grid Computing (GRID)*, pp. 41–48, Oct 2010.

[5] A. Inomata, T. Morikawa, M. Ikebe, Y. Okamoto, S. Noguchi, K. Fujikawa, H. Sunahara, and M. Rahman, "Proposal and evaluation of a dynamic resource allocation method based on the load of vms on iaas," in *4th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–6, Feb 2011.

[6] Amazon, "Elastic compute cloud (ec2) - scalable cloud hosting, http://aws.amazon.com/ec2/," Last accessed on 27 June, 2016.

[7] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pp. 423–430, June 2012.

[8] A. Oprescu and T. Kielmann, "Bag-of-tasks scheduling under budget constraints," in *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 351–359, Nov 2010.

[9] A. Benoit, L. Marchal, J.-F. Pineau, Y. Robert, and F. Vivien, "Scheduling concurrent bag-of-tasks applications on heterogeneous platforms," *IEEE Transactions on Computers*, vol. 59, pp. 202–217, Feb 2010.

[10] Y. C. Lee and A. Zomaya, "Practical scheduling of bag-of-tasks applications on grids with dynamic resilience," *IEEE Transactions on Computers*, vol. 56, pp. 815–825, June 2007.

[11] A. Oprescu and T. Kielmann, "Bag-of-tasks scheduling under budget constraints," in *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 351–359, Nov 2010.

[12] J. O. Gutirrez-Garca and K. M. Sim, "Ga-based cloud resource estimation for agent-based execution of bag-of-tasks applications.," *Information Systems Frontiers*, vol. 14, no. 4, pp. 925–951.

[13] T. Henzinger, A. Singh, V. Singh, T. Wies, and D. Zufferey, "Flexprice: Flexible provisioning of resources in a cloud environment," in *IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pp. 83–90, July 2010.

[14] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster, "Gridbot: execution of bags of tasks in multiple grids," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pp. 1–12, Nov 2009.

[15] Y. Fang, F. Wang, and J. Ge, "A task scheduling algorithm based on load balancing in cloud computing," in *Proceedings of the 2010 International Conference on Web Information Systems and Mining (WISM)*, (Berlin, Heidelberg), pp. 271–277, Springer-Verlag, 2010.

[16] J. Li, M. Qiu, Z. Ming, G. Quan, X. Qin, and Z. Gu, "Online optimization for scheduling preemptable tasks on iaas cloud systems," *Journal of Parallel and Distributed Computing*, vol. 72, no. 5, pp. 666 – 677, 2012.

[17] A. Iosup, S. Ostermann, M. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 931–945, June 2011.

[18] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing.*, vol. 5, Sept. 2005.

[19] J. Hollingsworth, "Automatic graders for programming classes," *Communications of ACM*, vol. 3, pp. 528–529, Oct. 1960.

[20] K. Ala-mutka, T. Uimonen, and H. matti Jrvinen, "Supporting students in c++ programming courses with automatic program style assessment," *Journal of Information Technology Education*, vol. 3, pp. 245–262, 2004.

[21] J. P. Leal, N. Moreira, and L. N. Moreira, "Automatic grading of programming exercises," tech. rep., Kurnia, A.; Cheang, B and Lim, A Online Judge, Computers and Education, 1998.

[22] U. von Matt, "Kassandra: The automatic grading system," *SIGCUE Outlook*, vol. 22, pp. 26–40, Jan. 1994.

[23] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pp. 76–85, ACM, 2003.

[24] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon, "On automated grading of programming assignments in an academic institution," *Computers & Education*, vol. 41, pp. 121–131, Sept. 2003.

[25] L.-R. Chien, D. Buehrer, C.-Y. Yang, and C.-M. Chen, "An evaluation of tdd training methods in a programming curriculum," in *IEEE International Symposium on IT in Medicine and Education*, pp. 660–665, 2008.

[26] S. H. Edwards, "Teaching software testing: Automatic grading meets test-first coding," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pp. 318–319, ACM, 2003.

[27] O. Demir, A. Soysal, A. Arslan, B. Yurekli, and O. Yilmazel, "Automatic grading system for programming homework," *Computer Science Education: Innovation and Technology*, 2010.

[28] A. Kivitya, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: The linux virtual machine monitor," 2007.

[29] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Operating Systems Reviewh*, vol. 37, pp. 164–177, Oct. 2003.

[30] Microsoft, "Hyperv https://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx," Last accessed on 27 June, 2016.

[31] VMWare, "Esxi hypervisor https://www.vmware.com/products/esxi-and-esx/overview," Last accessed on 27 June, 2016.

[32] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *RC25482*, 2014.

[33] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehrster, and A. Brinkmann, "Non-intrusive virtualization management using libvirt," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 574–579, March 2010.

[34] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, pp. 23–50, Jan. 2011.

[35] Moodle, "https://moodle.org," Last accessed on 27 June, 2016.

[36] Blackboard, "http://tr.blackboard.com," Last accessed on 27 June, 2016.

[37] Sysbench, "https://github.com/akopytov/sysbench," Last accessed on 27 June, 2016.