Department of Electronic and Computer Engineering
Technical University of Crete

THESIS

Investigation of Resource Allocation and Bandwidth Pricing for Cloud Computing Architectures

---

Author: Georgia KOUTSANDRIA

CHANIA
August 2012

ABSTRACT

The dramatic increase in cheap broadband wireless and wired network bandwidth along with the drop in storage pricing has lead to the practical fulfillment of computing as utility.

Our work investigates resource allocation strategies for different types of time-varying traffic. A service provider has to assign a request/service from a cloud consumer to the virtual machine that maximizes the profit. Our study focuses on multiple requests/services from a number of consumers and a set of services offered by a particular provider. The revenue of a provider for a particular time frame is defined as the total of values charged to consumers for processing their applications (service requests) during that frame.

We first implement two-profit driven scheduling algorithms from the literature (MaxProfit and MaxUtil) on the well-known and widely used CloudSim simulation tool. We discuss the minor differences in the algorithms' implementation that were dictated by the CloudSim structure. Then, we propose a new profit-driven algorithm, based on achieving the minimum service delay, which we compare with MaxProfit and MaxUtil, and we discuss the results of this comparison.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my supervisor, Polychronis Koutsakis for giving me the opportunity to work with him, for the continuous support of my final thesis study, for his guidance, motivation and enthusiasm.

Besides my supervisor, I would like to thank the members of my thesis examination committee, Prof. Michael Paterakis and Prof. Minos Garofalakis, for their participation.

I thank Amir Sayegh who participated in this thesis with interest, for offering suggestions for improvements, for his guidance and insightful comments. Also my sincere thanks goes to my colleague Manolis Skevakis for the interesting discussions about our theses and for sharing his ideas concerning the code implementation part, my brother Panos Koutsandrias and Dimitris Papailiopoulos for their continuous encouragement and support.

Last but not least, I would like to thank my parents for supporting me spiritually throughout my life and my friends for their encouragement in every possible way.

List of acronyms

VM virtual machine

CPU central processing unit

MaxUtil maximum utilization

MaxProfit maximum profit

MinDelay minimum delay

aad application-wise allowable delay

alft actual latest finish time

alst actual latest start time

sad service-wise allowable delay

aft actual finish time

asad aggregate service-wise allowable delay

clft current latest finish time

Contents

1 Introduction

The dramatic increase in cheap broadband wireless and wired network bandwidth along with the drop in storage pricing has led to the practical fulfillment of the dream of computing as utility.

Cloud computing is a recent advancement wherein IT infrastructure and applications are provided as services to end-users under a usage-based payment model. Cloud Computing can be defined as a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned, and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.

The cloud providers are willing to provide large-scale computing infrastructures at a price based primarily on usage patterns. The use of Cloud computing eliminates the initial high-cost, for application developers, of setting up an environment for the application deployment. Furthermore, there are large-scale software systems providers, who develop applications such as social networking sites and e-commerce, which are gaining popularity on the Internet. These applications can benefit greatly of Cloud infrastructure services to minimize costs and improve service quality to end users. There are three types of Cloud computing [1]:

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)

The level on which computing services are offered to consumers varies according to the abstraction level of the service. At the lowest level, Infrastructure as a Service, services are supplied in the form of hardware where consumers deploy virtual machines, software platforms to support their applications, and the application itself. Using IaaS, users rent the use of servers

(as many as needed during the rental period) provided by one or more cloud providers.

At the next level, Platform as a Service, Cloud consumers do not have to handle virtual machines. Instead, a software platform for hosting applications (typically, web applications) is already installed in an infrastructure and offered to consumers. Then, consumers use the platform to develop their application. Using PaaS, users rent the use of servers and the system software to use in them.

Finally, in Software as a Service, an application is offered to consumers, who do not have to handle virtual machines and software platforms that host the application. Using SaaS, users rent the application software and the databases. The cloud providers manage the infrastructure and platforms on which the applications run.

Some of the traditional and emerging Cloud-based application services include social networking, web hosting, content delivery, and real time instrumented data processing. Each of these types has different composition, configuration and deployment requirements. Some of the examples of Cloud computing infrastructures/platforms are Microsoft Azure [2], Amazon EC2, Google App Engine, and Aneka [3].

One implication of Cloud platforms is the ability to dynamically adapt (scale-up or scale-down) the amount of resources provisioned to an application in order to attend to variations in demand that are either predictable and occur due to access to patterns observed, or unexpected and occurring due to an increase in the popularity of the application service. These applications often exhibit transient behavior and have different QoS requirements depending on how time critical they are and on their users' interaction patterns. Hence, the development of dynamic provisioning techniques to ensure that these applications achieve QoS under transient conditions is required.

The cloud model is composed of four deployment models: private cloud, community cloud, public cloud, and hybrid cloud.

- *Private cloud:* The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and

may exist on premise or off premise.

- *Community cloud:* The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.

- *Public cloud:* The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

- *Hybrid cloud:* The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability.

A Cloud computing system (owned and operated by an infrastructure service provider) consists of a set of physical resources (server computers) in each of which there are one or more processing elements/cores that are fully interconnected in the sense that a route exists between any two individual resources. A service provider rents resources from cloud infrastructure providers and prepares a set of services in the form of virtual machine (VM) images. The underlying cloud computing infrastructure service is responsible for dispatching these instances to turn on physical resources. A running instance is charged by the time it runs at a flat rate per time unit.

Users (Cloud clients) access Cloud computing using networked client devices, such as desktop computers, laptops, tablets and smartphones. Many cloud applications do not require specific software on the client and instead use a web browser to interact with the cloud application. With Ajax [4] and HTML5 these Web user interfaces can achieve a similar or even better look and feel as native applications. Some cloud applications, however, support specific client software dedicated to these applications (e.g., virtual desktop clients and most email clients). A Cloud client sends a service request for an application, consisting of one or more services, to a provider specifying two main constraints, time and cost.

Similar to the work in [5] on profit-driven scheduling for cloud services, our work focuses on the scheduling of the consumer's service requests on service instances made available by service providers after taking cost into account. More specifically, our work follows the steps of the work presented in [5], where two scheduling algorithms were presented as an attempt to maximize profit while providing a satisfactory level of service quality as specified by the service consumer. The authors in [5] found that by incorporating the profit into scheduling decisions, utilization and response rate have a significant impact on cost. Therefore, profit is strongly influenced by utilization and response time *collectively*, not independently.

In this work, we first implement the two profit-driven scheduling algorithms (MaxProfit and MaxUtil) on the well-known and widely used CloudSim [6] simulation toolkit. Both algorithms were implemented via C/C++ simulation in [5]. We discuss the minor differences in the algorithms' implementation that were dictated by the CloudSim structure. Then, we propose a new profit-driven scheduling algorithm, based on the minimization of service delay, which we compare with MaxProfit and MaxUtil, and we discuss the results of this comparison. A service provider has to assign a request-service from a cloud consumer to the virtual machine that maximizes the profit. Our study focuses on multiple requests-services from a number of consumers and a set of services offered by a particular provider. The revenue of a provider for a particular time frame is defined as the total of values charged to consumers for processing their applications (service requests) during that time frame.

The thesis is organized as follows: Chapter 2 gives a brief overview of CloudSim. Chapter 3 presents the three algorithms, as well as their implementation details. Chapter 4 presents our results and the discussion on them and Chapter 5 includes our conclusions.

2 Simulation Tool

CloudSim [6] is an extensible simulation toolkit that enables modeling and simulation of Cloud computing systems and application provisioning environments. CloudSim offers the following novel features:

- Support for modeling and simulation of large scale Cloud computing environments.
- A self-contained platform for modeling Clouds, service brokers, provisioning, and allocations policies.
- Support for simulation of network connections among the simulated system elements.
- Facility for simulation of federated Cloud environment that inter-networks resources from both private and public domains.
- Availability of a virtualization engine that aids in creation and management of multiple, independent and co-hosted virtualized services on a data center node.
- A flexibility to switch between space-shared and time-shared allocation of processing cores to virtual services.

Figure 1 shows the multi-layered design of the CloudSim software framework and its architectural components. The CloudSim simulation layer provides support for modeling and simulation of virtualized Cloud-based data center environments including dedicated management interfaces for virtual machines (VMs), memory, storage, and bandwidth. The fundamental issues such as provisioning of host VMs, managing application execution, and monitoring dynamic system state are handled by this layer. Cloud providers, who want to study the efficiency of different policies in allocating its hosts to VMs, would need to implement their studies at this layer by extending the core VM provisioning functionality.

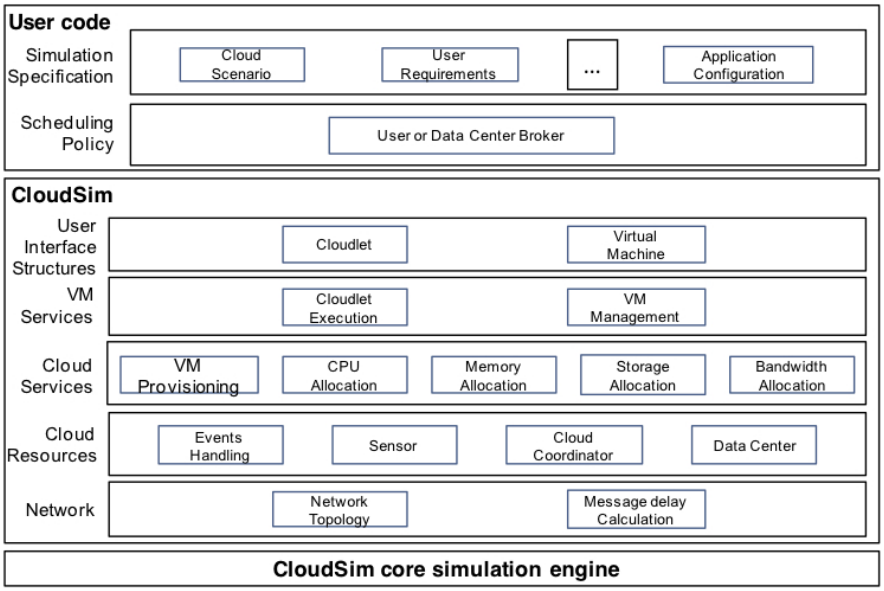

Figure 1: Layered CloudSim architecture.

The top-most layer in the CloudSim stack is the User Code that exposes basic entities for hosts (number of machines, their specification, etc.), applications (number of tasks and their requirements), VMs, number of users and their application types, and broker scheduling policies. By extending the basic entities given at this layer, a Cloud application developer can perform the following activities:

(i) Generate a mix of workload request distributions, application configurations

(ii) Model Cloud availability scenarios and perform robust tests based on the custom configurations

(iii) Implement custom application provisioning techniques for clouds and their federation.

As a result, by extending the basic functionalities already exposed with CloudSim, researchers will be able to perform tests based on spesific scenarios and configurations, thereby allowing the development of best practices in all the critical aspects related to Cloud Computing. Some of the fundamental

classes of CloudSim, which are also the building blocks of the simulator are:

BwProvisioner: This is an abstract class that models the policy for provisioning of bandwidth to VMs. The main role of this component is to undertake the allocation of network bandwidth to a set of competing VMs that are deployed across the data center. The BwProvisiongSimple allows to VM to reserve as much bandwidth as required, however this is constrained by the total available bandwidth of the host.

Cloudlet: This class models the Cloud-based application services (such as content delivery, social networking, and business workflow). CloudSim orchestrated the complexity of an application in terms of its computational requirements. Every application service has a pre-assigned instruction length and data transfer (both pre and post fetches) overhead that it needs to undertake during its life-cycle. This class can be extended to support modeling of other performance and composition metrics for applications such as transactions in database-oriented applications.

CloudScheduler: This abstract class is extended by implementation of different policies that determine the share of processing among Cloudlets in a virtual machine. As described, two types of provisioning policies are offered: space-shared and time-shared.

Datacenter: This class models the core infrastructure level services (hardware) that are offered by Cloud providers (Amazon, Azure, App Engine). It encapsulated a set of compute hosts that can either be homogeneous or heterogeneous with respect to their hardware configurations (memory, cores, capacity, and storage). Furthermore, every Datacenter component instantiates a generalized application provisioning component that implements a set of policies for allocating bandwidth, memory, and storage devices to hosts and VMs.

DatacenterBroker: This class models a broker, which is responsible for mediating negotiations between SaaS and Cloud providers that are driven by QoS requirements. The broker acts on behalf os SaaS providers. It discovers suitable Cloud service providers by querying the Cloud Information Service (CIS) and undertakes on-line negotiations for allocation of resources/services that can meet the application's QoS needs.

DatacenterCharacteristics: This class contains configuration information of data center resources.

Host: This class models a physical resource such as a compute or storage server. It encapsulates important information such as the amount of memory and storage, a list and type of processing cores (to represent a multi-core machine), an allocation of policy for sharing the processing power among virtual machines, and policies for provisioning memory and bandwidth to the virtual machines.

RamProvisioner: This is an abstract class that represents the provisioning policy for allocating primary memory (RAM) to the VMs. The execution and deployment of VM on a host is feasible only if the RamProvisioner component approves that the host has the required amount of free memory. The RamProvisionerSimple does not enforce any limitation on the amount of memory a VM may request. However, if the request is beyond available memory capacity then it is simply rejected.

Vm: This class models a virtual machine, which is managed and hosted by a Cloud host component. Every VM component has access to a component that stores the following characteristics related to a VM: accessible memory, processor, storage size, and the VM's internal provisioning policy that is extended from an abstract component called the CloudletScheduler.

VmmAllocationPolicy: This abstract class represents a provisioning policy that a VM Monitor utilizes for allocating VMs to Hosts. The chief functionality of the VmmAllocationPolicy is to select an available host in the data center that meets the memory, storage, and availability requirement for a VM deployment.

VmScheduler: This is an abstract class implemented by a Host component that models the policies (space-shared, time-shared) required for allocating processor cores to VMs.

3 Profit-driven service request scheduling

In this section, we first present the two profit-driven service scheduling algorithms (MaxProfit, MaxUtil) which were proposed in [5]. Then, we present and discuss our own algorithm, which is based on minimizing service delays and will be compared against the MaxProfit and MaxUtil algorithms. All three algorithms attempt to assign services to VMs, aiming to maximize the profit for providers without violating time constraints associated with consumer applications.

The scheduling model focuses on service requests and a set of VMs offered by a provider. The revenue of a provider for a particular time frame is defined as the total of values charged to consumers for processing their applications during that frame. Since a virtual machine runs constantly once it is created, the provider needs to strike a balance between the number of virtual machines it provides, and the service request pattern to ensure its profit is maximized. The effectiveness of this balancing should be reflected in the average utilization of those virtual machines.

In our work, we assume that the requests arrive as an exponential process and that every request can be processed by one VM. Furthermore, each request for an application consists only of one service. As in [5], a consumer application is associated with two types of allowable delay in its processing, i.e., application-wise allowable delay and the service-wise allowable delay. For a given consumer application, there is a certain additional amount of time that the service provider can afford when processing the application. This application-wise allowable delay is possible due to the fact that the provider will gain some profit as long as the application is processed within the maximum acceptable mean processing time *TMAX*.

In our implementation, the minimum processing time *TMIN* of an application is its processing time based on its length and we define the maximum acceptable mean processing time as

$$TMAX = k * TMIN. \tag{1}$$

The value of k is discussed in Section 4. Therefore, the application-wise allowable delay aad of an application is

$$aad = TMAX - TMIN. \tag{2}$$

For each service, the service-wise allowable delay time is calculated based on the actual latest start time from all the running services on the same VM and its finish time. Thus, the actual latest finish time for a service is defined as

$$alft = alst + w \tag{3}$$

where $alst$ is the start time of the service that was processed last and w the estimated process time of the service the current time. The service-wise allowable delay time sad of a service is defined as

$$sad = alft - aft \tag{4}$$

where aft of a service is its estimated finish time. Based on the delays previously defined, we derive one more metric that directly correlates with profit and attempts to ensure maximum profit gain by focusing on each individual service. The aggregate service-wise allowable delay $asad$ of a service is defined as:

$$asad = sad + \frac{aad}{TMIN}(sad + w). \tag{5}$$

The aggregate service-wise allowable delay of a service, in addition to its estimated finish time, indicates the upper bound processing time of the service without loss.

Furthermore, we assume that the virtual machines are heterogenous and geographically distributed. For the implementation of this case, we define one delay metric for each virtual machine, the transmission delay to remote VM.

3.1 Maximum Profit algorithm

MaxProfit(Fig. 2) [5] takes into account the profit achievable from the current service, and the profit from other services being processed on the same service instance. Specifically, the service is assigned to a virtual machine only if the assignment of that service onto the service instance yields some additional profit.

Each service is associated with a profit decay rate, and therefore the assignment of a service to a virtual machine on which some other services are being processed may result in a certain degree of profit loss. As the changes happen frequently to a dynamic environment, *MaxProfit* intends to run quickly to allocate VM instances for a service to optimize profit in a greedy manner.

MaxProfit maintains a service queue containing a number of services. When a new application arrives, its entry is the only one to be processed. *MaxProfit* checks all the VMs (Steps 4-25) and selects the best VM based on additional profit incurred by the current service (Step 19).

For each service running on the current VM (Step 7), the profit difference between the two schedules (one considering and the other one not considering the current service) is computed, and this is denoted as profit index (p_i). In the case of considering the current service, we assume that this service has been assigned to the the current virtual machine.

Then, the profit indices for those two schedules are computed using the current latest finish time of each service (Steps 14-15). The current latest finish time of a running service may be different from the actual latest finish time due to the fact that it is computed based on the current schedule, hence new services may be assigned to the current virtual machine before the current service completes its processing.

1. Let $max\_pi = \emptyset$
2. Let $s_{j,*} = \emptyset$
3. Let s_j^* = the first service to be scheduled
4. **for** $\forall s_{j,k} \in I_j$ **do**
5. Let $pi_k = \emptyset$
6. Let $pi_k^* = \emptyset$
7. **for** $\forall s_{j,k}^l$ running on $s_{j,k}$ **do**
8. Let $aft_{j,k}^l = aft$ of $s_{j,k}^l$ without considering s_j^*
9. Let $aft_{j,k}^{l*} = aft$ of $s_{j,k}^l$ with considering s_j^*
10. Let $clft_{j,k}^l = aft_{j,k}^l + asad_{j,k}^l$
11. **if** $aft_{j,k}^{l*} > clft_{j,k}^l$ **then** // *possible loss*
12. Go to Step 4
13. **end if**
14. Let $pi_k = pi_k + clft_{j,k}^l - aft_{j,k}^l$
15. Let $pi_k^* = pi_k^* + clft_{j,k}^l - aft_{j,k}^{l*}$
16. **end for**
17. Let $pi_k^* = pi_k^* + clft_{j,k}^* - aft_{j,k}^*$ // include s_j^*
18. **if** $pi_k^* > pi_k$ **then**
19. Let $\Delta pi_k = pi_k^* - pi_k$
20. **if** $\Delta pi_k > max\_pi$ **then**
21. Let $max\_pi = \Delta pi_k$
22. Let $s_{j,*} = s_{j,k}$
23. **end if**
24. **end if**
25. **end for**
26. **if** $s_{j,*} = \emptyset$ **then**
27. Create a new service instance $s_{j,new}$
28. Let $s_{j,*} = s_{j,new}$
29. **end if**
30. Assign s_j^* to $s_{j,*}$

Figure 2: MaxProfit Algorithm.

The current service instance is disregarded if the actual finish time of any of the running services considering the current service is greater that its current latest finish time. The reason is that this implies a profit loss (Step 11). If there is no possibility of profit loss, then the profit index of the current service is included (Step 17).

After each iteration of the inner *for* loop (Steps 7-16), *MaxProfit* checks if the current virtual machine delivers the largest profit increase (Step 20) and keeps track of the best instance (Steps 21-22). If none of the current virtual machines is selected, in our implementation of the algorithm the service is rejected, i.e., our implementation does not create new VMs, as *MaxProfit* does in Step 27. The final assignment of the current service is carried out in Step 30 onto the "best" virtual machine.

3.2 Maximum Utilization algorithm

The main focus of *MaxUtil* (Fig. 3) [5] is on the maximization of service instance utilization. This approach is an indirect way of reducing costs to rent resources and decrease the number of instances the service provider creates/prepares. Specifically, for a given service, *MaxUtil* selects the instance with the lowest utilization by taking into account the profit.

As in *MaxProfit*, the service with the earliest arrival time (Step 3), in the service queue, is selected for scheduling and the *MaxUtil* algorithm is activated. *Maxutil* checks all the service machines (Steps 4-18) and selects the "best" based on the maximum utilization incurred by the current service (Step 13). Each VM is checked if it can accommodate the current service without incurring loss by calculating the profit (Steps 4-11).

The current virtual machine is disregarded (Step 9) if the actual finish time of the running service when taking the current service into consideration is greater than the current latest finish time. After each iteration of the inner *for* loop (Steps 5-12), *MaxUtil* checks if the current virtual machine delivers the minimum utilization (Step 14) and keeps track of the best instance (Steps 15-16).

As in *MaxProfit*, if none of the current VMs is selected (no profitable VM assignment), the service is not processed further and is rejected. Thus, in our work the *if* statement (Steps 19-22) does not create a new VM, as in [5], for ensuring the assignment of the current service, but instead rejects it.

1. Let $min\_util = 1.0$
2. Let $s_{j,*} = \emptyset$
3. Let s_j^* = the first service to be scheduled
4. **for** $\forall s_{j,k} \in I_j$ **do**
5. **for** $\forall s_{j,k}^l$ running on $s_{j,k}$ **do**
6. Let $aft_{j,k}^l = aft$ of $s_{j,k}^l$ without considering s_j^*
7. Let $aft_{j,k}^{l*} = aft$ of $s_{j,k}^l$ with considering s_j^*
8. Let $clft_{j,k}^l = aft_{j,k}^l + asad_{j,k}^l$
9. **if** $aft_{j,k}^{l*} > clft_{j,k}^l$ **then** // *possible loss*
10. Go to Step 4
11. **end if**
12. **end for**
13. Let $util_{j,k}$ = utilization of $s_{j,k}$
14. **if** $util_{j,k} < min\_util$ **then**
15. Let $min\_util = util_{j,k}$
16. Let $s_{j,*} = s_{j,k}$
17. **end if**
18. **end for**
19. **if** $s_{j,*} = \emptyset$ **then**
20. Create a new service instance $s_{j,new}$
21. Let $s_{j,*} = s_{j,new}$
22. **end if**
23. Assign s_j^* to $s_{j,*}$

Figure 3: MaxUtil Algorithm.

The utilization $util_{j,k}$ of a VM $s_{j,k}$ is defined as:

$$util_{j,k} = \frac{\tau^{\text{used}}}{\tau^{\text{cur}} - \tau^{\text{start}}}, \tag{6}$$

where τ^{used}, τ^{cur} and τ^{start} are the amount of time used for processing services, the current time and the creation time of the VM, respectively.

3.3 Minimum Delay algorithm

The main focus of the algorithm that we propose is on the minimization of service delay (Fig. 4). As we already described above, the *Maxutil* algorithm [5] calculates the utilization based on the percentage of time that a VM is used. But does not take into account the delays incurred to each service in the queue. Therefore, for a given service, *MinDelay* selects the virtual machine which will lead to the minimum service delay, but also takes into account profit.

One of the main assumptions in the work presented in [5] is that the VMs are homogeneous and cost equally to operate. In our work, we also investigate the case of having heterogeneous virtual machines. More specifically, we consider the case of geographically distributed VMs, which are associated with variable transmission delays.

MinDelay is activated when the service with the earliest arrival time (Step 3) is selected for scheduling. *MinDelay* checks all the VMs (Steps 4-23) and selects the one that leads to the minimum delay (Step 27). As in *MaxUtil*, we calculate the profit in order to conclude if a virtual machine can accommodate the current service without profit loss (Step 9).

If the actual latest finish time of the running service when taking into account the current service is greater than the current latest finish time, the current VM can not be a candidate for assigning the current service to it and the algorithm continues by checking the next virtual machine (Step 10).

1.Let $upper\_bound = \infty$
2.Let $s_{j,*} = \emptyset$
3.Let s_j^* = the first service to be scheduled
4.**for** $\forall\ s_{j,k} \in I_j$ **do**
5. **for** $\forall\ s_{j,k}^l$ running on $s_{j,k}$ **do**
6. Let $aft_{j,k}^l = aft$ of $s_{j,k}^l$ without considering s_j^*
7. Let $aft_{j,k}^{l*} = aft$ of $s_{j,k}^l$ with considering s_j^*
8. Let $clft_{j,k}^l = aft_{j,k}^l + asad_{j,k}^l$
9. **if** $aft_{j,k}^{l*} > clft_{j,k}^l$ **then** // *possible loss*
10. Go to Step 4
11. **end if**
12. Let $Tmin = \frac{length_{j,k}^l}{\text{mips}_{j,k}}$
13. Let $Tmax = k * Tmin$
14. **if** $clft_{j,k}^l > Tmax$ **then**
15. Go to Step 4
16. **end if**
17. **end for**
18. Let $del_{j,k}$ = total delay of$s_{j,k}$
19. if $del_{j,k} <$ upper bound **then**
20. Let $upper\_bound = util_{j,k}$
21. Let $s_{j,*} = s_{j,k}$
22. **end if**
23.**end for**
24.**if** $s_{j,*} = \emptyset$ **then**
25. Reject s_j^*
26.**end if**
27.Assign s_j^* to $s_{j,*}$

Figure 4: MinDelay Algorithm.

We incorporated in our proposed algorithm one more control procedure. If there is no profit loss, we check if the processing time of the running service when taking into account the current service is greater than $TMAX$ (Step 14). If this is true, then we suppose that the assignment violates the acceptable processing time of a service ($TMAX$) and we check the next VM (Step 15).

Then, after the end of the inner for loop, *MinDelay* checks the time that will be needed for the completion of the service (Step 19). If the current virtual machine delivers the minimum delay, *MinDelay* considers the current

virtual machine as a candidate and keeps track of it (Steps 18-22).

If there is no profitable assignment, the service is rejected (Steps 24-26) and the algorithm proceeds to the next service in the service queue. In the case of geographically distributed virtual machines with variable capabilities (MIPS) the total delay $del_{j,k}$ of a virtual machine $s_{j,k}$ is defined as:

$$del_{j,k} = \frac{\sum_{l=1}^{L} length_{j,k}^{l}}{\text{mips}j,k} + transmDelay_{j,k} \tag{7}$$

where L is the total number of running services, $length_{j,k}^{l}$ is the length of the l-th service, mipsj,k is the MIPS capacity and $transmDelay_{j,k}$ is the transmission delay due to the distance to the remote VM.

4 Simulation Studies

In our work we investigated the setup of a cloud computing environment using the CloudSim[6] simulation tool under different types of time-varying traffic. Several experiments followed our implementation of the three algorithms presented in Section 3.

Our experimental results are plotted, analyzed and discussed using the following performance metrics:

- Average utilization
- Average response rate
- Average profit
- Average accepted services
- Average rejected services

The average utilization of a VM is defined as:

$$\overline{util} = \frac{\sum_{j=1}^{L} util_j}{L}, \tag{8}$$

where L is the number of VMs.

The response rate of a service s_i is defined as:

$$rr_i = \frac{TMIN_i}{t_i} \tag{9}$$

where t is the actual processing time of the service. Then the average response rate for all the requests serviced by the provider is defined as:

$$rr = \frac{\sum_{i=1}^{N} rr_i}{N} \tag{10}$$

where N is the number of accepted services.

The provider's profit for serving a service s_i is defined as:

$$p_i = \begin{cases} p_{\max}, & \text{if } d = \text{TMIN} \\ p_{\max} - a \cdot d, & \text{if TMIN}<d\leq \text{TMAX} \\ 0, & \text{if } d > \text{TMAX} \end{cases} \tag{11}$$

where $p_{max} = 100\%$. The decay rate a is defined as:

$$a = \frac{p_{\max}}{TMAX - TMIN} \tag{12}$$

and the delay d is defined as:

$$d = t - TMIN. \tag{13}$$

Thus, the average profit for the provider is defined as:

$$\overline{p} = \frac{\sum_{i=1}^{N} p_i}{N}, \tag{14}$$

where N is the number of services.

Each simulation point, in our results, is the average of 100 independent simulations runs. Seven distinctive simulation scenarios were selected and the three algorithms presented in Section 3 were implemented for each scenario, in order to conduct a performance evaluation of their efficiency.

The selected scenarios used a diverse set of services and settings, such as different number of services' length, different number of VMs, various services characteristics and various VMs characteristics. We start by using a low load for the VMs, and we steadily increase the load of the system to observe the performance of each algorithm under a diverse set of loads. This will help us understand the differences and similarities in their performance, depending on the load scenario. Moreover, each scenario consists of two parts, as we executed all the scenarios for two different values for the variable k (Eq. 1). This variable defines the maximum processing time $TMAX$ (in relation to $TMIN$), thus we wanted to evaluate each algorithm for a low and a high $TMAX$ by defining the variable k to be equal to 2 and 6, respectively, for each simulation scenario.

4.1 Scenario 1

Scenario 1 focuses on the performance of each algorithm for a low VM load. The system parameters used were the following:

- Services = 100
- VMs = 5
- Service length (MI) = 200000
- CPU per VM = 1
- VM capacity = 10000 MIPS
- Arrival rate $\lambda = \frac{1}{3.0} \frac{service}{sec}$
- Transmission delay = U(2,5), where U is the uniform distribution.

In this scenario all the services have the same length and all the VMs have the same capacity. Thus, the $TMIN$ for each service is 20sec (length/MIPS). We set that the arrival rate $\lambda = 3.0$, which means that on average every 3sec a new service arrives in our system. This means that a small delay will occur during the processing time of each service due to the fact that during $TMIN$ the number of concurrent service requests will on average be higher than the number of available VMs (5).

The results obtained from Scenario 1 are summarized in Tables 1-3 which show the performance of the three algorithms over the performance metrics used in our study. Figures 5-9 graphically present this performance for each algorithm individually, and their comparison.

metric	$k = 2$	$k = 6$
Avg Util	80%	83%
Avg response rate	91%	56%
Avg profit	82%	60%
Avg accepted	44%	73%
Avg rejected	56%	27%

Table 1: MaxProfit algorithm

metric	$k = 2$	$k = 6$
Avg Util	49%	66%
Avg response rate	51%	24%
Avg profit	50%	21%
Avg accepted	100%	100%
Avg rejected	0%	0%

Table 2: MaxUtil algorithm

metric	$k = 2$	$k = 6$
Avg Util	90%	94%
Avg response rate	100%	85%
Avg profit	100%	95%
Avg accepted	25%	62%
Avg rejected	75%	38%

Table 3: MinDelay algorithm

Regarding the *MaxProfit* algorithm, by increasing the value of the variable k, we observed that the average percentage of accepted services increased. This result can be explained by the fact that the bound on the profit loss check (Fig. 2 Steps 11-13) is loosed via the increase in $TMAX$. Thus, the algorithm accepts more services by assuming that there will not be a profit loss due to the additional delays experienced by the users.

The average utilization also increases, since more services use the system and each VM handles a higher number of services than in the case of $k = 2$. As a result, the average response rate is decreased since each service has to wait until a VM is free in order to be processed, and new services have to be delayed until the current running service on the VM is finished.

Finally, the average profit decreases. This result can be explained by the fact that even though the number of accepted services is higher in the case of $k = 6$, the average response rate decreases. Eq. (12) shows us that when a service finishes after the maximum processing time, the profit for the provider is equal to zero.

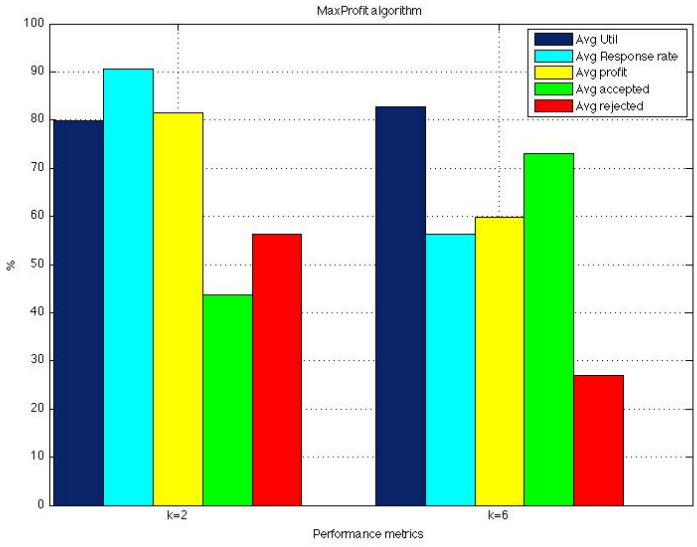

Figure 5: MaxProfit Algorithm

Moving on the *MaxUtil* algorithm, we observe that for both $k = 2$ and $k = 6$ all services are accepted. For $k = 6$, the average utilization reaches up to 66%, a 17% increase in comparison to the $k = 2$ case. Again, this higher utilization is combined with a dramatic drop in profit and in the average response rate.

The reason is that *MaxUtil* suffers from an important drawback: when all VMs are busy, a newly arriving service will be assigned to the VM that started last, as it has been utilized for the least amount of time in comparison to the other VMs. If the other VMs continue to be busy (e.g., they are processing long services) all newly arriving services will pile up on the same VM, thus leading to excessive delays and very low profit.

This is the case with *MaxUtil* when k increases from 2 to 6. More specifically, the increase in the value of k increases $clft$, which in turn leads (through Step 9 of the algorithm) to fewer cases of possible loss; hence, more VMs receive services to process and this leads, as shown in Table 2, to the increase in utilization.

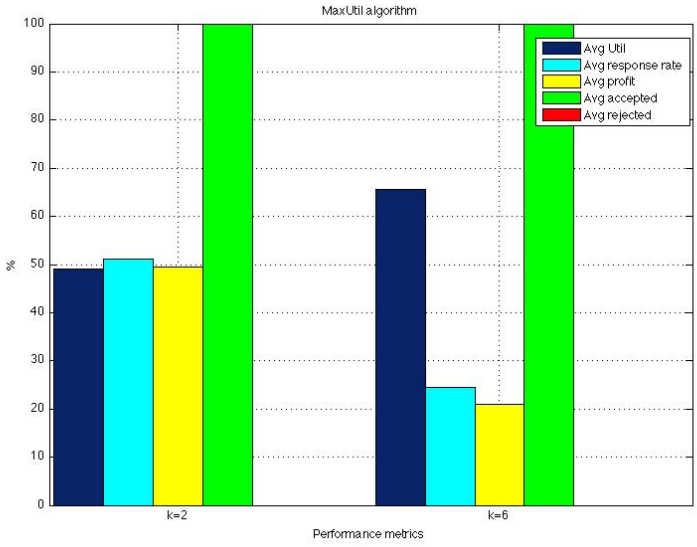

Figure 6: MaxUtil Algorithm

However, longer services keep certain VMs busy while others pile up on the VMs that were lately activated. This leads to significant delays, which affect negatively the average response rate and the average profit of *MaxUtil*.

The performance of our algorithm, *MinDelay*, is presented in Table 3 and Figure 7. When we increase the value of the variable k, we notice again that the average percentage of accepted services is increased (together with the average utilization) as the increase in $TMAX$ loosens the services' requirements.

However, due to the fact that our algorithm's focus is on the minimization of the services' delay, the increase in accepted services leads to a minor decrease in response rate and a very minor decrease in profit. These results are a strong first indication of the better performance of our algorithm.

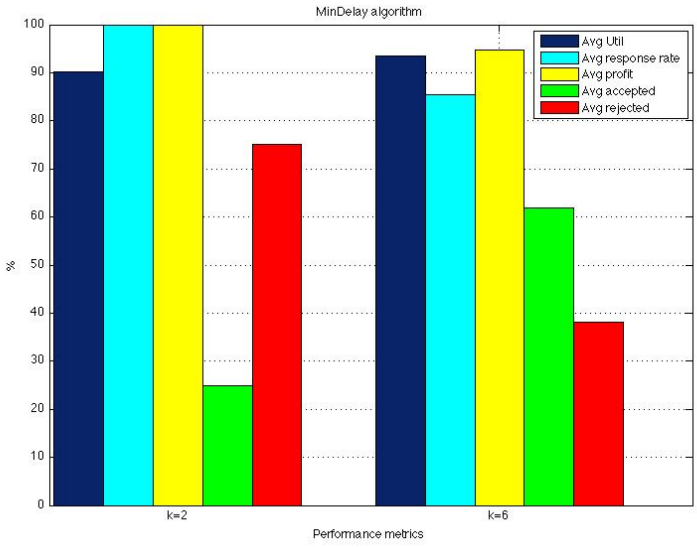

Figure 7: MinDelay Algorithm

We continue to a head-to-head comparison of the performances of the three algorithms. Figures 8 and 9 provide such a graphic comparison for $k = 2$ and $k = 6$ and show that our algorithm, *MinDelay*, clearly outperforms

both *MaxProfit* and *MaxUtil* over all performance metrics with the exception of the percentage of accepted services.

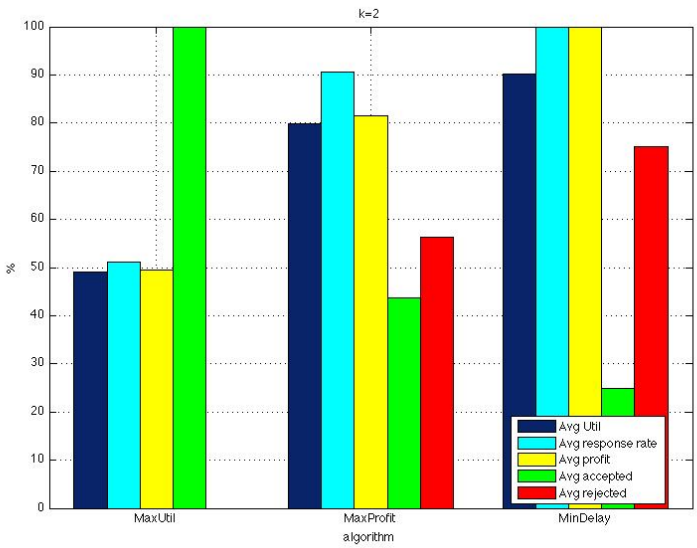

Figure 8: Algorithms' comparison for $k = 2$

This indicates that, for a given number of VMs, our algorithm provides a good tradeoff: by accepting less services (which would theoretically lead to user irritation) it offers a much higher satisfaction (response rate) for services that are accepted and a much higher profit rate to the provider.

The comparison between *MaxUtil* and *MaxProfit* shows that *MaxUtil* accepts more services. The reason is that *MaxProfit* checks if the current VM delivers the largest profit increase (Fig. 2 Steps 19-23). In order to achieve this, the average response time of service should be the minimum possible. By accepting more services, the processing time of a service may be large and thus, a profit loss is possible.

MaxProfit outperforms *MaxUtil* over all the performance metrics, except the percentage of accepted services; this indicates that, among the three algortihms, *MaxUtil* is the one that accepts as many services as possible, but this leads to the lowest profit to the provider due to the large delays.

It should be noted, however, that these delays do not violate, on average, the quality of service (QoS) requirements of the services: the response rate remains above $1/k$, for both the $k = 2$ and $k = 6$ values.

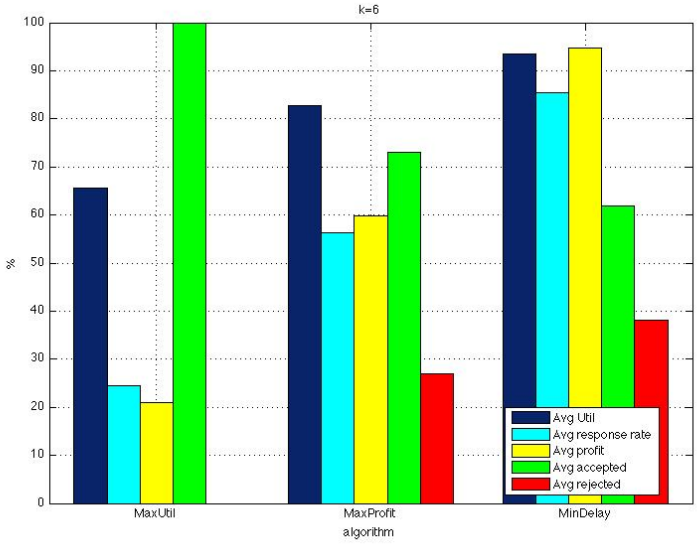

Figure 9: Algorithms' comparison for $k = 6$

4.2 Scenario 2

In Scenario 2 we try to understand how some small changes in the parameters affect the performance metrics. The parameters for the specific scenario are the following:

- Services = 200
- VMs = 15

- Service length (MI) = U(200000,300000)
- CPU per VM = 1
- VM capacity = 10000 MIPS
- Arrival rate $\lambda = \frac{1}{2.8} \frac{service}{sec}$
- Transmission delay = U(2,5), where U is the uniform distribution.

In this scenario, all the VMs have once again the same capacity, but services vary in length. The average $TMIN$ for a service is 25sec.

The results for the Scenario 2 are summarized in Tables 4-6 and in Figures 10-14. All three algorithms exhibit the same performance, qualitatively, in terms of how their performance is influenced by the increase in the value of k.

metric	$k = 2$	$k = 6$
Avg Util	69%	87%
Avg response rate	76%	38%
Avg profit	58%	42%
Avg accepted	60%	72%
Avg rejected	41%	28%

Table 4: MaxProfit algorithm

metric	$k = 2$	$k = 6$
Avg Util	39%	60%
Avg response rate	100%	72%
Avg profit	100%	73%
Avg accepted	100%	100%
Avg rejected	0%	0%

Table 5: MaxUtil algorithm

metric	$k = 2$	$k = 6$
Avg Util	88%	77%
Avg response rate	98%	100%
Avg profit	97%	100%
Avg accepted	47%	95%
Avg rejected	53%	5%

Table 6: MinDelay algorithm

However, one significant difference that we came across in this scenario is that *MaxUtil* outperforms both of the other algorithms for $k = 2$ not only in terms of the average percentage of accepted services (as in Scenario 1) but also in terms of the response rate and the average profit. Our algorithm, *MinDelay*, has a very slightly worse performance in response rate and profit but achieves a 49% higher utilization; our algorithm also clearly outperforms the other two algorithms for $k = 6$.

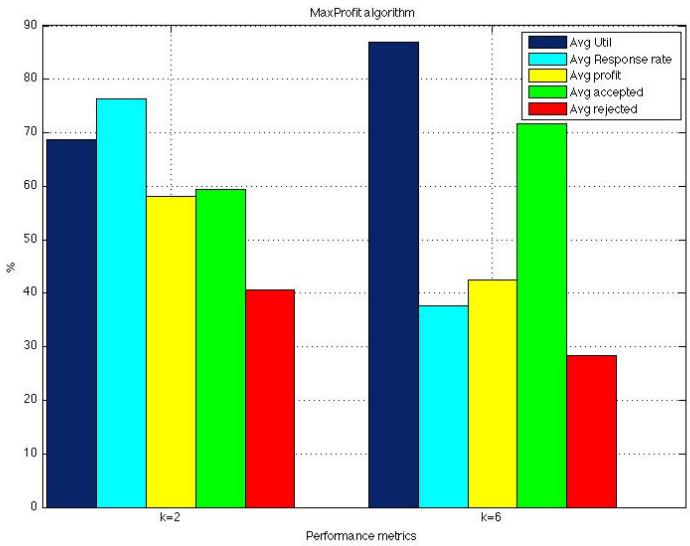

Figure 10: MaxProfit algorithm

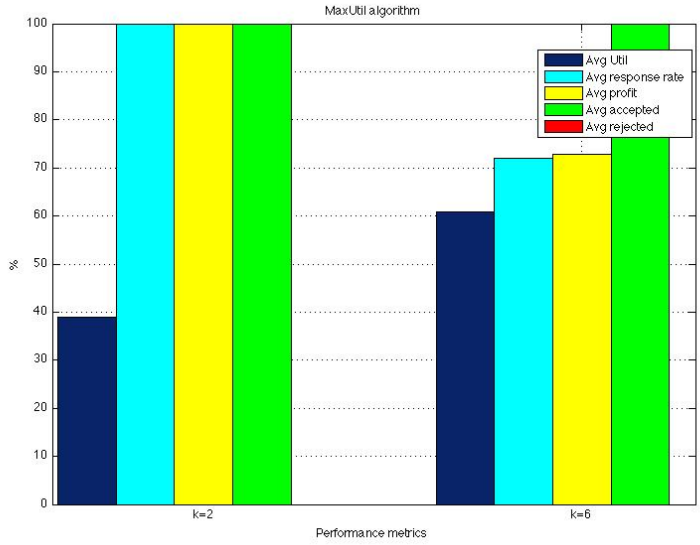

Figure 11: MaxUtil algorithm

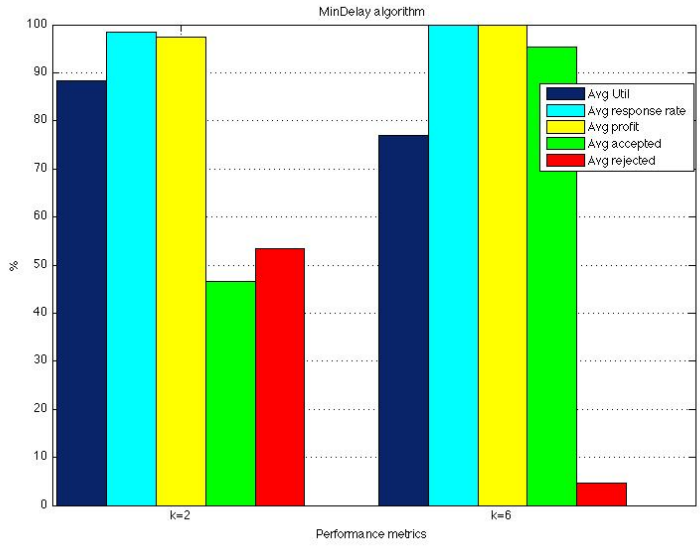

Figure 12: MinDelay algorithm

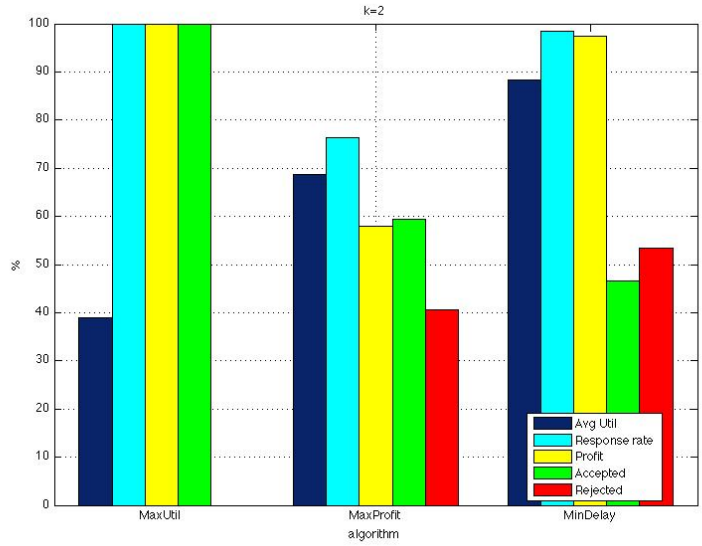

Figure 13: Algorithms' comparison for $k = 2$

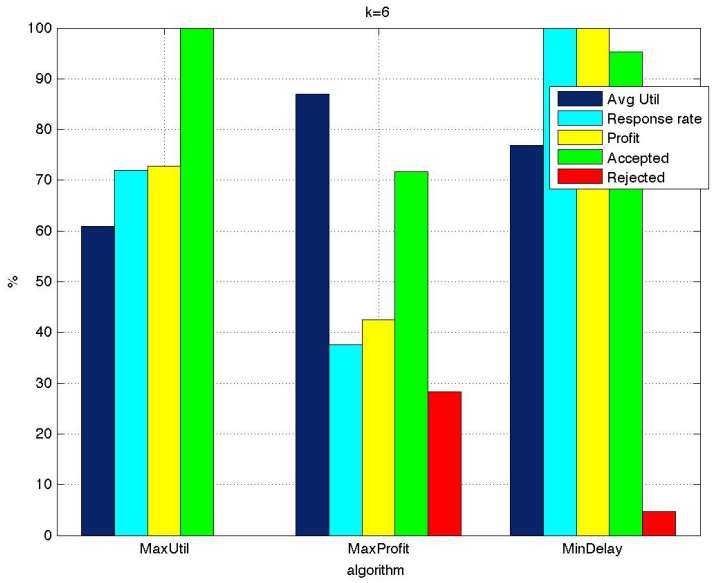

Figure 14: Algorithm's comparison for $k = 6$

The reason that *MaxUtil* excels (even slighly) for $k = 2$ is that the load (which is lower than that of Scenario 1) can be handled well by few VMs. When more VMs are activated, because of the increase in k (as explained in our discussion of Scenario 1) the performance of *MaxUtil* is clearly worse than that of *MinDelay*.

4.3 Scenario 3

In this Scenario, more services arrive at the system and their length is larger. Moreover, the capacity of a VM follows the uniform distribution and it is between 10000 MIPS and 20000 MIPS. We defined the parameters as follows:

- Services = 500
- VMs = 10
- Service length (MI) = 400000
- CPU per VM = 1
- VM capacity = U(10000, 20000) MIPS
- Arrival rate $\lambda = \frac{1}{3.0} \frac{service}{sec}$
- Transmission delay = U(2,5), where U is the uniform distribution.

The average $TMIN$ for a service is 27sec. The results for Scenario 3 are summarized in Tables 7-9 and in Fig. 15-19. All three algorithms are shown to exhibit the same behavior as in Scenarios 1, 2 for $k = 2$ and $k = 6$, and *MinDelay* is shown to outrperform *MaxProfit* and *MaxUtil* over all the performance metrics except the percentage of accepted services. However we need to point out that because of the high load studied in this scenario, the percentage of accepted services by *MinDelay* is very low.

metric	$k = 2$	$k = 6$
Avg Util	88%	80%
Avg response rate	75%	57%
Avg profit	53%	64%
Avg accepted	28%	76%
Avg rejected	72%	24%

Table 7: MaxProfit algorithm

metric	$k = 2$	$k = 6$
Avg Util	61%	74%
Avg response rate	64%	54%
Avg profit	61%	55%
Avg accepted	100%	100%
Avg rejected	0%	0%

Table 8: MaxUtil algorithm

metric	$k = 2$	$k = 6$
Avg Util	88%	78%
Avg response rate	100%	68%
Avg profit	96%	88%
Avg accepted	16%	29%
Avg rejected	84%	71%

Table 9: MinDelay algorithm

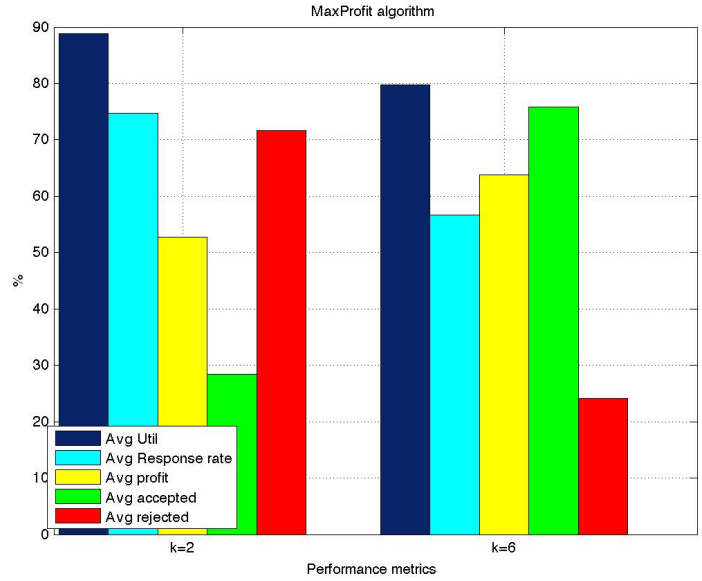

Figure 15: MaxProfit algorithm

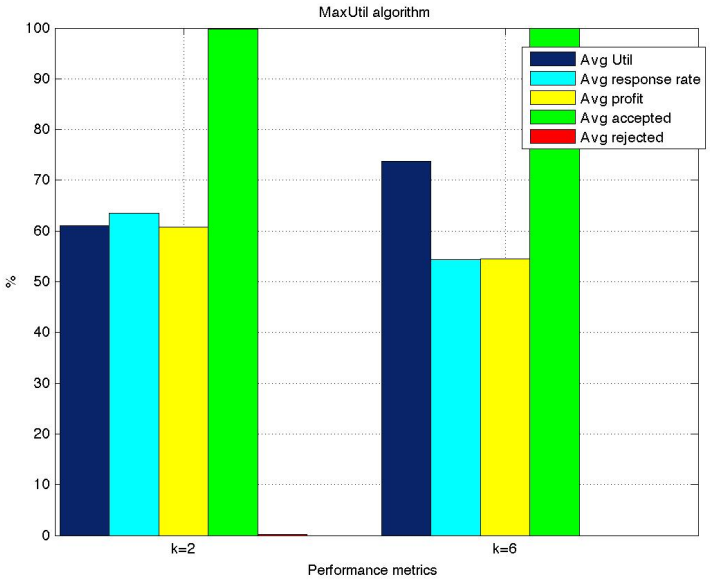

Figure 16: MaxUtil algorithm

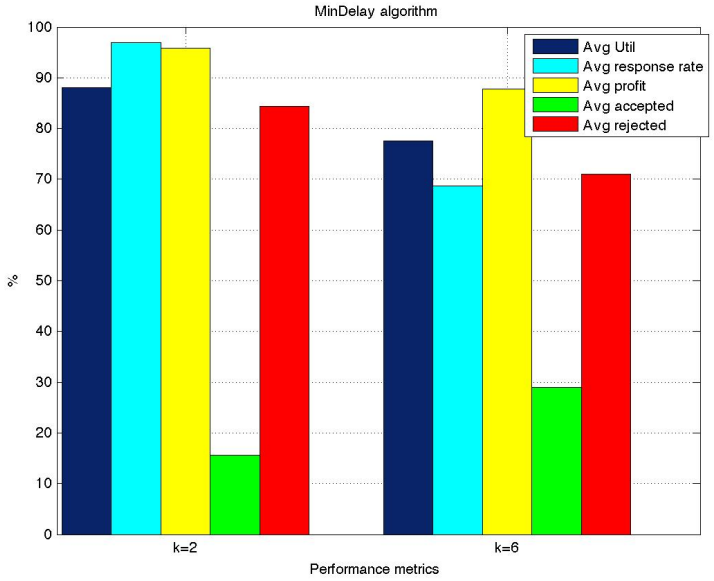

Figure 17: MinDelay algorithm

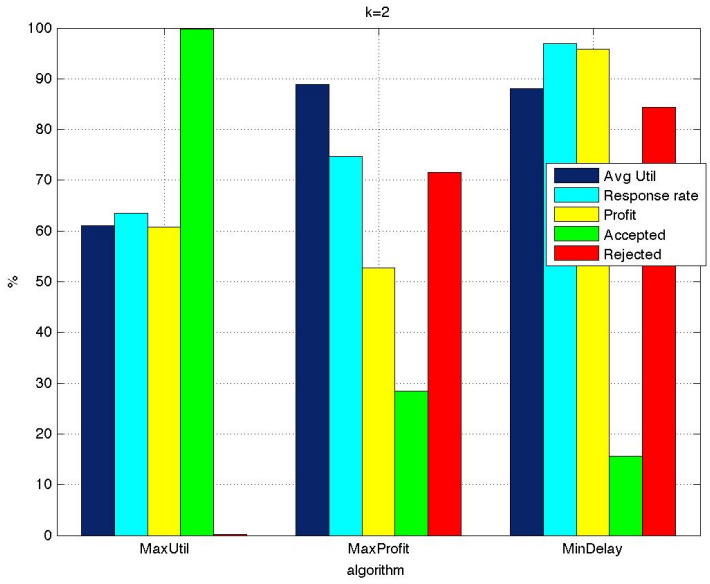

Figure 18: Algorithm's comparison fo $k = 2$

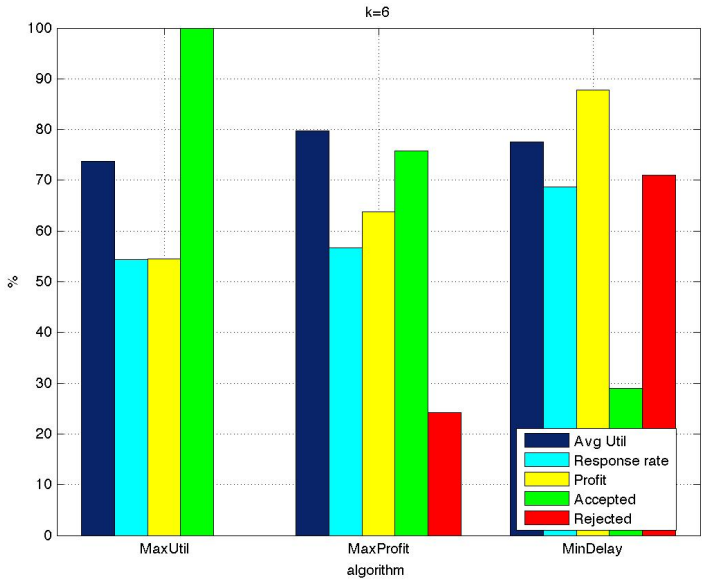

Figure 19: Algorithm's comparison for $k = 6$

The rest of the seven scenarios that were used in our study showed that, regardless of the changes in the system parameters (including an increase in the number of CPU per VM) the behavior of the three algorithms remained the same. To avoid repetition, we present the results for these scenarios in the Appendix A.

5 Conclusions

In this thesis, we implemented two algorithms from the literature and proposed one more algorithm for scheduling service requests in a Cloud architecture with the main objective of maximizing the profit. Our proposed algorithm, *MinDelay*, attempts to maximize the profit while guaranteeing the highest possible level of user satisfaction through delay minimization.

We simulated different scenarios in order to understand the performance of each algorithm and the similarities or differences that they exhibit. Our results have shown that our proposed algorithm exhibits better performance than *MaxUtil* and *MaxProfit* based on several performance metrics, with the tradeoff of accepting a smaller number of services into the system.

Our future work will focus on taking into account energy minimization in Cloud architectures, by proposing resource allocation algorithms that will seek a tradeoff between pricing, user QoS requirements and "greener" scheduling at the Cloud.

References

[1] S. Bhardwaj, L. Jain, S. Jain, "Cloud Computing: A study of Infrastructure as a Service (IaaS)", International Journal of Engineering and Information Technology, Vol. 2, No. 1, 2010.

[2] D. Chappell, "Introducing the Windows Azure platform", White paper, http://www.davidchappell.com/writing/white_papers, Oct. 2008.

[3] C. Vecchiola, X. Chu, and R. Buyya, "Aneka: A Software Platform for NET-based Cloud Computing", IOS Press, High Speed and Large Scale Scientific Computing, W. Gentzsch, L. Grandinetti, G. Joubert (Eds.), 2009, pp. 267-295.

[4] JJ. Grrett, "Ajax: A new approach to Web applications", Adaptive path, http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications, Feb. 2005.

[5] Y. C. Lee, C. Wang, A. Y. Zomaya, B. B. Zhou, "Profit-driven scheduling for cloud services with data access awareness", Journal of Parallel and Distributed Computing, Vol. 72, No. 4, 2012, pp. 591-602.

[6] 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, 2011, pp. 23-50.

A Appendix

A.1 Scenario 4

- Services = 500
- VMs = 20
- Service length (MI) = 500000
- CPU per VM = 1
- VM capacity = U(20000, 40000) MIPS
- Arrival rate $\lambda = \frac{1}{0.8} \frac{service}{sec}$
- Transmission delay = U(5,10), where U is the uniform distribution.

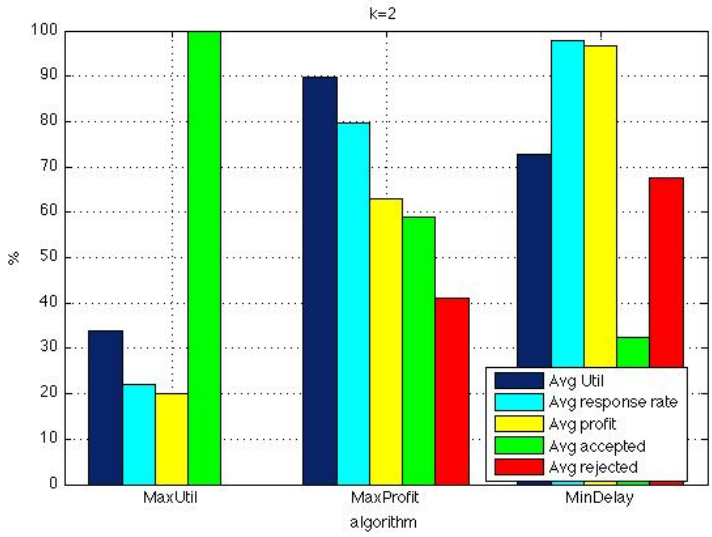

Figure 20: Algorithms' comparison for $k = 2$

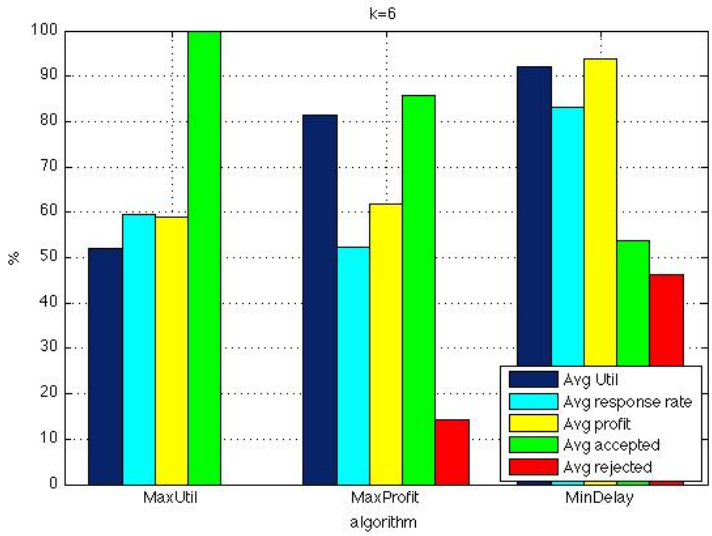

Figure 21: Algorithms' comparison for $k = 6$

A.2 Scenario 5

- Services = 500
- VMs = 20
- Service length (MI) = 500000
- CPU per VM = 2
- VM capacity = 10000 MIPS
- Arrival rate $\lambda = \frac{1}{0.8} \frac{service}{sec}$
- Transmission delay = U(5,10), where U is the uniform distribution.

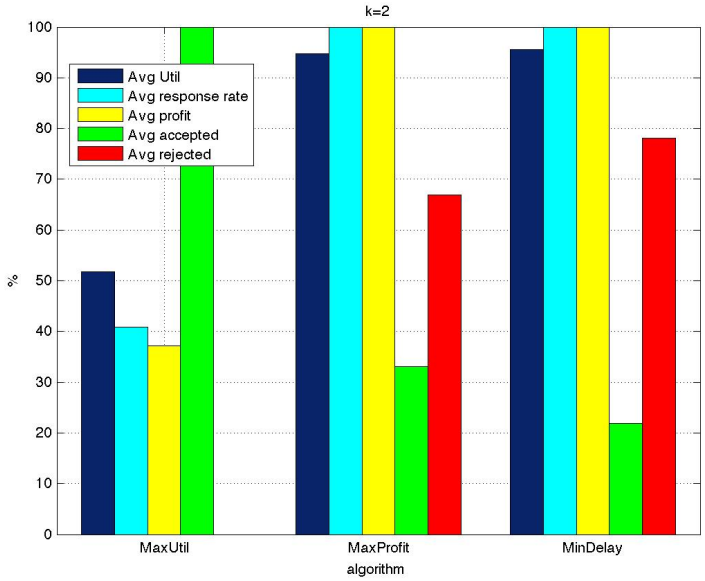

Figure 22: Algorithm's comparison for $k = 2$

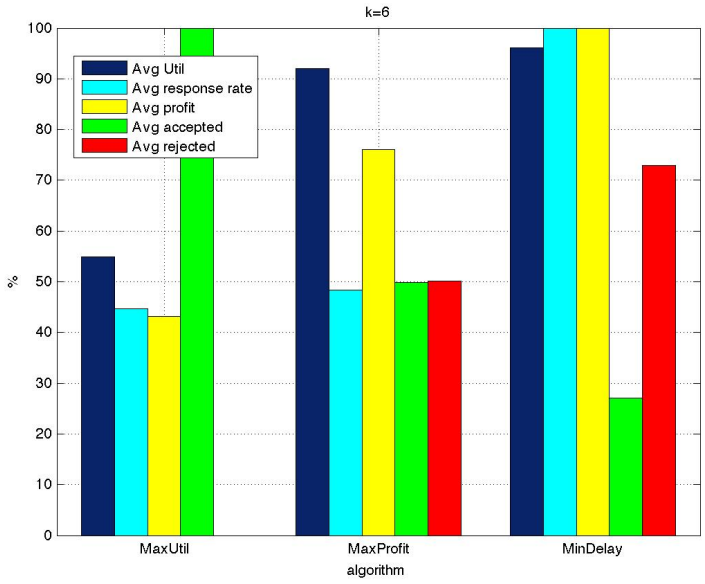

Figure 23: Algorithms' comparison for $k = 6$

A.3 Scenario 6

- Services = 500
- VMs = 15
- Service length (MI) = 600000
- CPU per VM = 2
- VM capacity = 20000 MIPS
- Arrival rate $\lambda = \frac{1}{1.2} \frac{service}{sec}$
- Transmission delay = U(5,10), where U is the uniform distribution.

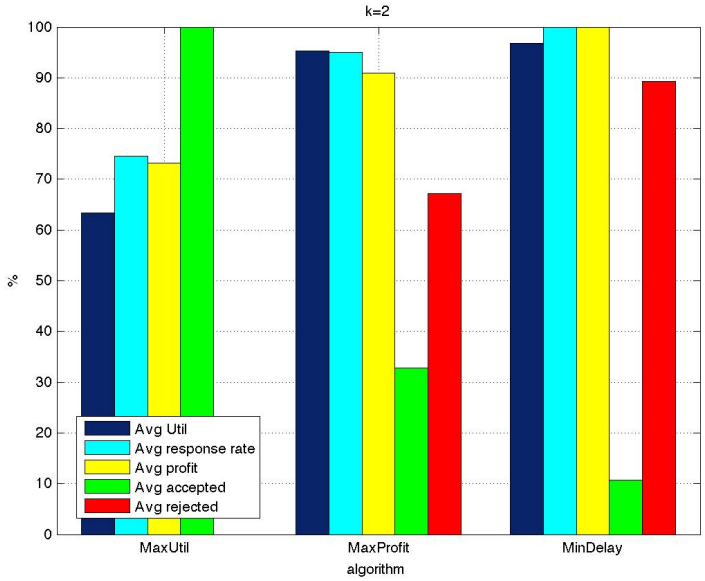

Figure 24: Algorithm's comparison for $k = 2$

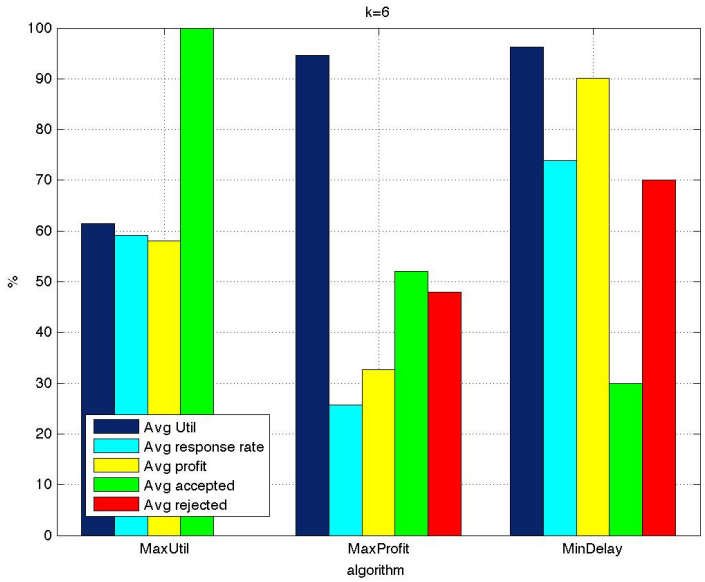

Figure 25: Algorithms' comparison for $k = 6$

A.4 Scenario 7

- Services = 1000
- VMs = 40
- Service length (MI) = 700000
- CPU per VM = 2
- VM capacity = 20000 MIPS
- Arrival rate $\lambda = \frac{1}{1.0} \frac{service}{sec}$
- Transmission delay = U(5,10), where U is the uniform distribution.

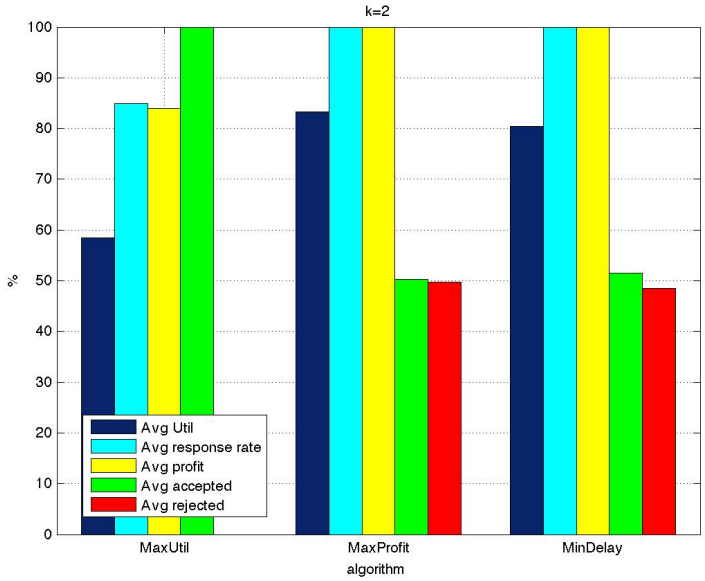

Figure 26: Algorithms' comparison for $k = 2$

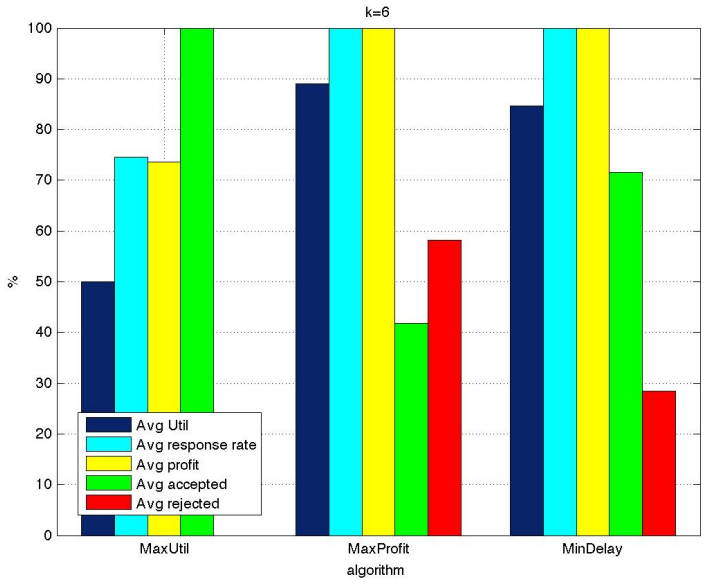

Figure 27: Algorithms' comparison for $k = 6$