Improving the Efficiency of Cloud Computing

Matthew G. Perrault
University of Minnesota, Morris
perra044@morris.umn.edu

ABSTRACT

This paper discusses various techniques to improve the efficiency of cloud computing. These techniques are aimed at improving cloud storage access time, as well as cloud computational time. Users can store large amounts of data on the cloud and must be able to retrieve their data at all times. The large amount of resources stored on the cloud can result in slow data retrieval times for the user. To decrease the query, or lookup, time for data on the cloud, there are a several strategies introduced in this paper to optimize indexing. Users also use cloud systems for their substantial computational power. Processing large-scale workloads quickly and efficiently is crucial in keeping the user satisfied with computation times, as well as maintaining the lowest possible cost by using the least amount of computational resources. Applications vary in the amount of resources they require so optimizing resource allocation to prevent waste is important. A method called overbooking has been adapted for cloud computing and proves very beneficial in reducing resource waste.

Keywords

Cloud Computing, Virtual Machines, Distributed Databases, Query Processing, Overbooking

1. INTRODUCTION

Cloud computing is a service provided by IT firms, such as Amazon, Google, and Microsoft, that allows users to take advantage of their distributed computing resources. The resources (large amounts of computing power and storage) that are available through cloud providers have proven to be very popular with users. A main feature of cloud systems is their scalability or elasticity. The ability to scale resources up or down with customer demands make cloud structures more reliable than a local user system. These resources would also be quite costly if purchased by users individually; it would mean maintaining many servers at all times, finding the physical space to put these servers, and installing their desired services/applications on many computers. Cloud services are rented out based on specific service-level agreements (SLAs) characterizing their performance, reliability, etc. Each cloud provider aims at providing a cloud system that will be efficient in terms of computing data the fastest as well as using the least amount of their resources as possible, which is the key problem addressed in this paper.

Improving the efficiency of cloud computing can be done in many ways, but this paper summarizes two main approaches. The first approach is about decreasing the physical time it takes for a user to access their data on a cloud storage database. This is done through indexing strategies that lead to reductions in query times as well as cost. The second approach is the use of overbooking to increase computing efficiency. This method relies on the fact that users tend to overestimate the resources they will actually need. Allowing the amount of requested resources to surpass the actual available amount of resources, with careful monitoring, results in less resource waste on the cloud provider's side.

2. BACKGROUND

The approach for cloud storage efficiency that is presented uses databases, so a brief overview of how databases work is provided. A database is a large collection of organized data. The data is organized in a way that it is easy for a process to lookup specific information. A query is the process of selecting and receiving data from a database. Indexes exist so that a query does not have to search through every piece of information in a database. Indexes are equivalent to folders in a file system.

The index manipulations used in this paper use the Extensible Markup Language, or XML. XML is a language that was designed to transport and keep data organized. Its ability to structure data easily has made XML very popular and it is used on most websites found today. XML is structured by tags that are not predefined, which means you can organize data with tags that are self descriptive. A tag opens with <tag> and closes with </tag>. It either encloses specific data or it surrounds another set of tags. For example, a piece of a sample XML document would look like:

```
<painting>
  <id>"1854-1"</id>
  <name>"Olympia"</name>
  <painter>
    <name>
      <first>"Edouard"</first>
      <last>"Manet"</last>
    </name>
  </painter>
</painting>
```

You can see that a painting has attributes consisting of an id, painting name, and the first and last name of the painter. These are in quotations and provide information about the elements. This example uses nested elements, or elements stored within elements.

3. EFFICIENCY THROUGH INDEXING AND QUERIES

While some services that are provided through the cloud are geared towards computational power, this section focuses on the services that allow for large amounts of data storage. In this section, the efficiency of a cloud system is referring to the response time it takes for the user to receive the requested data as well as the monetary costs that go into retrieving that data. The amount of resources used by the cloud to access storage and process queries is what is billed to the user. Therefore, measuring the cost of data requests can be used to determine the actual efficiency of these queries. In order to better understand the basic cloud architecture and examine how exploiting an index can speed up processing to reduce cloud resource consumption, Camacho-Rodríguez et al. [1] used the example the Amazon Web Services (AWS) platform (Figure 1).

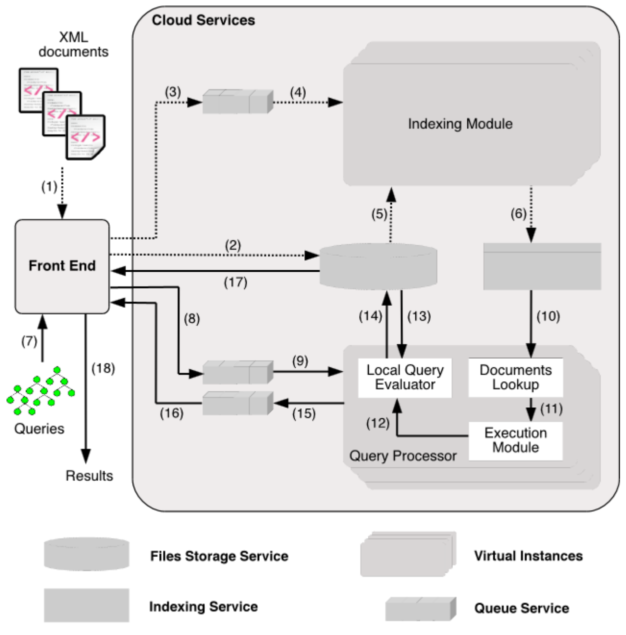

Figure 1: Overview of cloud architecture based on AWS.

User interaction with the system involves the following steps. A document arrives at the front end and is stored in the file storage service (1-2). A message referencing the document is sent through the loader request queue and the document is loaded once the message is retrieved by the indexing module (3-5). The indexing module creates the index data that is then inserted into the index store (6). Once a query arrives it is inserted into the query request queue and sent to the query processor module (7-9). The index data is then extracted from the index store (10).

Index storage services provide an API with simple functionalities typical of key-value stores, such as get and put requests. Thus, any other processing steps needed on the data retrieved from the index are performed by a standard XML querying engine (11), providing value- and structural joins, selections, projections etc. After the document references have been extracted from the index, the local query evaluator receives this information (12) and the XML documents cited are retrieved from the file store (13). Finally, the results are written to a message with a reference to those results (14). That message is read by the front end and the results are retrieved from the file store and returned to the user (15-18) [1].

3.1 Indexing Strategies

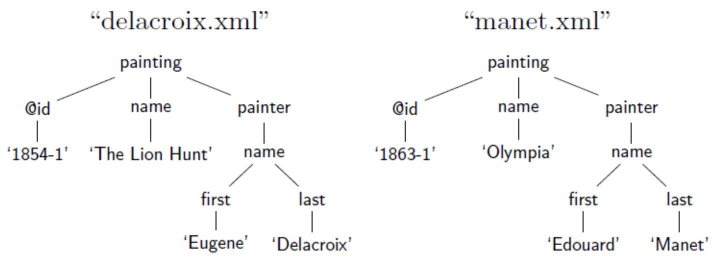

Figure 2: Sample XML documents.

This section explains a few simple XML indexing strategies, as noted by Camacho-Rodríguez et al. [1]. To preface these strategies, it is helpful to understand what a Uniform Resource Identifier (URI) is. A URI can be thought as simply a string of characters that are used to identify or locate a name or ID. To show how these index strategies work on two sample XML documents (Figure 2), it will be useful to show what results are returned with each strategy. For a given node n in a document d, the function *key(n)* computes a string key based on which n's information is indexed. Let e, a and w be three constant string tokens, and || denote string concatenation. Then we define *key(n)* as:

$$key(n) = \begin{cases} \underline{e}\|n.label & \text{if } n \text{ is an XML element} \\ \underline{a}\|n.name & \\ \underline{a}\|n.name\ n.val & \text{if } n \text{ is an XML attribute} \\ \underline{w}\|n.val & \text{if } n \text{ is a word} \end{cases}$$

It is important to note that there are two keys that come from an attribute node. One key is the attribute name and the other is the value of the attribute.

3.1.1 Strategy LU (Label-URI)

This index strategy is the simplest of the four. From a query, it extracts all node names, attributes and element string values. The URI sets that are obtained are then intersected and evaluated on those documents whose URIs are found in the intersection. The table below shows some of what is produced from the LU strategy. Each key that is used returns the attribute name, or document name, of the file. The attribute values column is filled with ∈, which denotes that the values are null. The attribute values column is null in the LU strategy because it only associates the key with the attribute name.

key	attribute name	attribute values
$\underline{e}$name	“delacroix.xml”	ϵ
	“manet.xml”	ϵ
$\underline{a}$id	“delacroix.xml”	ϵ
	“manet.xml”	ϵ
$\underline{a}$id 1863-1	“manet.xml”	ϵ
$\underline{w}$Olympia	“manet.xml”	ϵ

Figure 3: Data extracted from the dataset in figure 3 using LU.

3.1.2 *Strategy LUP (Label-URI-Path)*

This strategy consists of finding, for each root-to-leaf path appearing in a query, all documents having a data path that matches the query path. A root-to-leaf query path is obtained simply by traversing the query tree and recording node keys and edge types [1]. In the table below, you can see that the keys and attribute names are still the same, while the attribute values column is no longer null. The attribute values column is now filled with all paths that can be used to get to the key that was selected.

key	attribute name	attribute values
$\underline{e}$name	“delacroix.xml”	/$\underline{e}$painting/$\underline{e}$name, /$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
	“manet.xml”	/$\underline{e}$painting/$\underline{e}$name, /$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
$\underline{a}$id	“delacroix.xml”	/$\underline{e}$painting/$\underline{a}$id
	“manet.xml”	/$\underline{e}$painting/$\underline{a}$id
$\underline{a}$id 1863-1	“manet.xml”	/$\underline{e}$painting/$\underline{a}$id 1863-1
$\underline{w}$Olympia	“manet.xml”	/$\underline{e}$painting/$\underline{e}$name/$\underline{w}$Olympia

Figure 4: Data extracted from the dataset in figure 3 using LUP.

3.1.3 *Strategy LUI (Label-URI-ID)*

The idea of this strategy is to concatenate the structural identifiers (IDs) of a given node in a document and store them into a single attribute value. This implementation is used because structural XML joins which are used to identify the relevant documents need sorted inputs. When keeping the IDs ordered, it reduces the use of expensive sort operators after the look-up. In the table below, as with the LU and LUP, the key and attribute name columns remain the same. However, now the attribute values column stores the IDs. The IDs are have three values with them, pre, post and depth. Looking at the results of the LUI table (figure 5), the attribute value element that corresponds with $\underline{e}$name has two sets of ordered pairs because there are two name elements found in the example (figure 2).

3.1.4 *Strategy 2LUPI (Label-URI-Path, Label-URI-ID)*

This strategy utilizes two previously introduced index strategies: LUP and LUI. 2LUPI first uses LUP to obtain the set of documents containing matches for the query paths, and second, uses LUI to retrieve the IDs of the relevant nodes. The path labels that are obtained from the LUP strategy are used to form relations that are compared with the ID’s extracted from the root-to-leaf path taken by the LUI strategy.

key	attribute name	attribute values
$\underline{e}$name	“delacroix.xml”	$(3,3,2)(6,8,3)$
	“manet.xml”	$(3,3,2)(6,8,3)$
$\underline{a}$id	“delacroix.xml”	$(2,1,2)$
	“manet.xml”	$(2,1,2)$
$\underline{a}$id 1863-1	“manet.xml”	$(2,1,2)$
$\underline{w}$Olympia	“manet.xml”	$(4,2,3)$

Figure 5: Data extracted from the dataset in figure 3 using LUI.

3.2 Testing Environment

Camacho-Rodríguez et al. [1] experiments ran on Amazon Web Services (AWS) servers from the Asia Pacific region in September-October 2012. They used the centralized Java-based XML query processor, implementing an extension of an algorithm to our larger subset of XQuery. They also used two types of Amazon Elastic Compute Cloud (EC2) instances for running the indexing module and query processor. The first instance is large (L) with 7.5 GB of RAM memory and 2 virtual cores with 2 EC2 Compute Units each. The second instance is extra large (XL), with 15 GB of RAM memory and 4 virtual cores with 2 EC2 Compute Units each.

An EC2 Compute Unit is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. Varying XML documents were generated (20000 documents in all, adding up to 40 GB) to test the indexing strategies. A fraction of the documents were modified to alter their path structure (while preserving their labels), and another fraction were modified to make them different from the original document by rendering more elements optional children of their parents.

3.3 Indexing Results

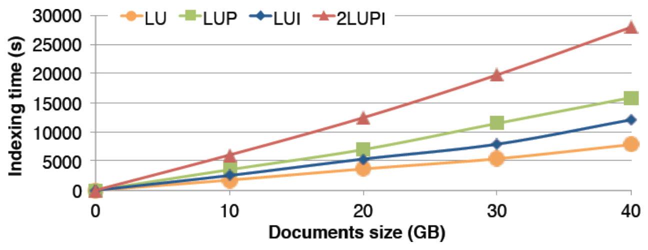

Figure 6: Indexing in 8 large EC2 instances.

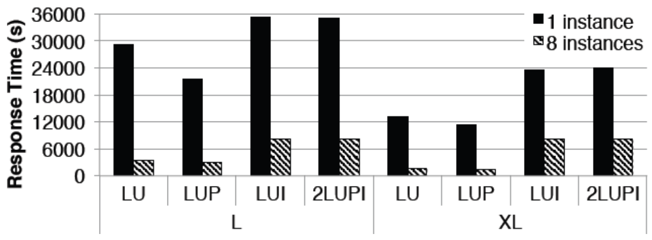

Figure 7: Indexing shown in multiple EC2 instances.

Using the AWS framework from section 3.2, the results from the various indexing strategies are shown in Figures 6 and 7. Figure 6 compares the indexing time it took for each strategy with the increasing document sizes. Using the large EC2 instance, you can see that as the document size gets bigger, 2LUPI is the slowest of the 4 strategies. This is because 2LUPI is essentially LUP and LUI combined to return more accurate results. Figure 7 shows the query response times for both 1 instance, and 8 instances.

The results are split into two sections, one using the large EC2 module and the other with the extra large EC2 module. When running 8 instances versus 1 one instance, the response times were drastically reduced due to the multiple instances working in parallel to return the query. The LUP indexing strategy allowed for the most efficient query processing. Further compression of the paths in the LUP index could potentially make it even more competitive. LUI and 2LUPI strategies were found to behave better on data in which query tree patterns are more multi-branched.

4. OVERBOOKING

Overbooking is the management of resources in such a way that the number of actual available resources is less than the hypothetical number of requested resources. It is a common practice applied to areas such as hotel, airline and concert ticket sales as well network bandwidth allocation and batch scheduling for parallel computers. Users of cloud systems will generally request more resources for applications than they actually need, which means that there will be unused resources on the cloud provider's end. Tomás et al. [4] found a study in which 5000 servers were observed for 6 months and it was noted that average CPU resource usage ranged from 10-50%.

User applications on the cloud can vary drastically in the amount of resources used over time, which makes minimizing resource waste a difficult problem. If a cloud service provider does not provide the requested resources then it can result in a Service Level Agreement (SLA) violation. A SLA is an agreement that states that a service provider will guarantee the availability of their service a certain percentage of the time to the user and can penalize the provider if the agreement is not met.

4.1 Applying Overbooking to Cloud Systems

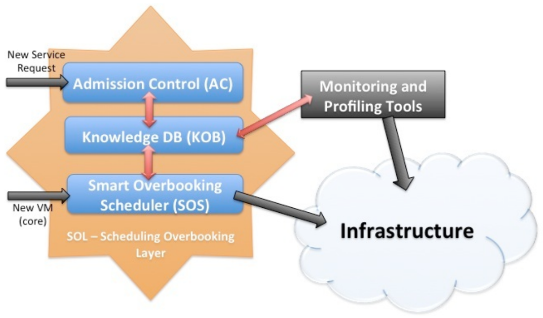
New Service Request
Admission Control (AC)
Monitoring and Profiling Tools
Knowledge DB (KOB)
Smart Overbooking Scheduler (SOS)
New VM (core)
SOL – Scheduling Overbooking Layer
Infrastructure

Figure 8: The relationship between the three main components (AC, KOB and SOS) and the cloud infastructure. AC decides whether to allocate new requests; SOS determines the best placement; and both use the information collected by the KOB.

Each virtual machine is comprised of CPU, memory, and their input/output (I/O). When deciding which resources to overbook it is important to have enough CPU power per VM so that the application being used will not suffer performance loss. Also having a sufficient amount of memory available is needed to prevent the application from crashing. Each VM will have different amounts of CPU and memory allocated to it. The applications that are using these VMs will be constantly changing the amount of resources they need. At some point the amount of resources the application is using will decrease, meaning it will not be efficient to keep the same VMs assigned to the application.

This problem of changing the number of VMs allocated to a given service over time is known as horizontal application elasticity [4]. To improve horizontal application elasticity, a strategy called admission control (AC) is used to allow the maximum number of virtual machines used to surpass the actual number of virtual machines possible, also known as overbooking. AC uses data collected, through a monitoring tool called the knowledge database module (KOB)(Figure 8), to determine how to allocate resources to new incoming application requests. The KOB is used in conjunction with AC to either accept or reject new service application requests based on the data collected.

The issue of modifying the number of VMs assigned to specific services is not the only hindrance to resource utilization on the cloud. Overbooking can also be applied to manipulate the actual resource usage of each virtual machine. Optimizing individual virtual machine resources (CPU, bandwidth, memory, etc.) is called the vertical application elasticity [4]. To handle the vertical application elasticity problem, a resource scheduler, called smart overbooking scheduler (SOS) (Figure 8), is used that can overbook physical resources.

Figure 9: Notations used for Algorithms

App	Incoming Application
$AppProfile$	App Profile
$VMType$	VM Type required by App
N	Set of nodes $\{n_i$ / i in [1..m] $\}$
TC	Total Capacity
TNC	Total Node Capacity
C_{used}	Capacity already booked
RC_{used}	Real Capacity in use
$R_{n_i}C_{used}$	Real node n_i Capacity in use
Min	Minimum # of VMs needed by the App
Max	Maximum # of VMs needed by the App
Avg	Average # of VMs needed by the App
OBF	Overbooking Factor
OBF_{n_i}	Overbooking Factor of node n_i
$ObjFunction$ $(Min, Max, Avrg)$	Returns # of VMs to book depending on the selected objective, Min, Max or Avg
$Prediction(x)$	Predicts future values of x
$Area(r, q)$	Area of line r over line q
$GetSlopeValues(y)$	Obtains final and max slope for time series y

Figure 10: Overbooking Admission Control Algorithm

```
1: if $Prediction(RC_{used}) + AppProfile <- TC$ and
   $OBF > OBF_{threshold}$ then
2:     Accept $App$
3: else
4:     Reject $App$
5: end if
```

The algorithm used for AC (Figure 10) takes into account current and predicted status of the system (real usage, not requested resources), the workload profiles and the overbooking already achieved, but without analyzing the long term impact [4]. There are several parameters that restrict admission control from overpassing the actual capacity. Because AC only uses the current status when making the decision, this means that it is not considering the possibility of having to deploy more VMs to applications that are accepted in the future. Once the AC has decided that an application

Figure 11: Worst-Fit Overbooking Scheduling

```
 1: Allocated = false
 2: NS = Sort Nodes $N$ by $OBF$
 3: for each $n_i \in NS$ do
 4:   $AccumulatedUsage = Prediction(R_{n_i} C_{used}) + AppProfile$
 5:   if $Area(AccumulatedUsage, TNC) < Area_{threshold}$ then
 6:     if $OBF_{n_i} > OBF_{threshold}$ then
 7:       $finalSlope, maxSlope = GetSlopeValues(Prediction(OBF_{n_i}))$
 8:       if $|finalSlope| < Slope_{threshold}$ and $|maxSlope| < Slope_{threshold}$ then
 9:         Allocated = True
10:         AllocateVM at Node $n_i$
11:       end if
12:     end if
13:   end if
14: end for
15: if $Allocated == false$ then
16:   AllocateVM at Node with highest OBF
17: end if
```

has been accepted and will be deployed, the SOS (Figure 11) is then in charge of deciding which is the most suitable node and core(s) for each VM. As physical servers have limited CPU, memory, and I/O capabilities, these have to be carefully considered when performing the overbooking to try to avoid placements that may lead to low performance and possible SLA violations [4]. The worst-fit style algorithm first takes into account the real usage, not what is requested It then predicts what the future expected usage of the physical resources are. This information is used together with the application profile to estimate if accepting the new incoming request would overpass the total real capacity of CPU, Memory or I/O (Figure 11 - Line 4). If the *accumulated usage* (Line 4) is less than the real available threshold of resources (Line 5), then the framework takes into consideration how overbooked the selected node already is (Line 6) and what the trend of that overbooking is (Line 7). Tomás et al. [4] found it useful to measure the overbooking that has already been done so they define the overbooking factor, or OBF, as:

$$OBF_X = \frac{(UsageRequested_X - RealUsage_X)}{min(UsageRequested_X, RealCapacity_X)}$$

where X can be any dimensions of a virtual machine (CPU, memory, I/O), depending on the capacity being measured. The range of $OBFx$ is (0,1) since *RealUsage* cannot be greater than the requested one as it is encapsulated within the requested VM and *RealUsage* cannot be negative. Thus, OBF values represent how overbookable the resources are: the greater the value the higher the potential for overbooking [4]. The OBF is calculated for each node and for each of the VM dimensions on the whole system. The overbooking scheduler is considered a worst-fit style algorithm because it is selecting the node with the largest value that appears in the ranges of $OBFs$ (Line 2). In order to make single comparisons when determining which node should be used through the worst-fit algorithm, the OBF of each dimension is multiplied together to determine the overall OBF:

$$OBF = OBF_{CPU} * OBF_{Mem} * OBF_{I/O}$$

These values are then used to finally determine whether the overbooking action will occur or not. OBF values are predicted for the future based on current and past resource availability, in the form of slope values calculated from OBF values (Line 7). The *finalSlope* is the final expected OBF value and *maxSlope* is the value that corresponds to the worst case possible for that current calculation. Finally, the algorithm ensures that both the *finalSlope* and *maxSlope* values are not greater than the actual resource threshold (Line 8). It can then allocate a VM to the application that initially needed to be overbooked (Line 9-10). Tomás et al. [4] also noted that as AC has accepted the new application, all VMs must be scheduled by SOS, even though this could result in aggressive overbooking of certain hosts.

4.2 Testing Environment

In order to test their overbooking method, Tomás et al. [4] simulated an actual cloud environment. They emulated two different kinds of workloads as their data. The background workload is comprised of web server applications with a varying number of user requests. This workload is interpolated from real available traces, in this case Wikipedia traces. The dynamic workload consisted of applications profiled by using monitoring tools after running the real application and generating a workload through a poisson distribution. Two different type of applications are profiled in the results, one with steady behavior and the other one with bursty usage. Bursty meaning applications that vary in the amount of resources they need. The cloud infrastructure simulated for testing their algorithms consisted of 16 Nodes where each one of them has 32 Cores. The virtual machine sizes they used ranged from small (1 CPU, 2048 MB Memory, 1000 Mbit/s Bandwidth), large (4 CPUs, 4096MB Memory, 4000 Mbit/s Bandwidth) to extra large (8 CPUs, 8192 MB Memory, 8000 Mbit/s Bandwidth).

4.3 Overbooking Results

Figures 12 and 13 show the various results from the using the overbooking method. Figure 12 shows the difference of CPU and memory utilization when overbooking is compared with the standard CPU and memory utilization observed without overbooking. The min, max and avg refers to the minimum, maximum and average use of CPU and memory utilization recorded without using the overbooking method. The ideal CPU and memory lines denote the best possible utilization that could have been achieved. The varying usage of CPU and memory seen in the min, max and avg illustrate the changes that were observed in resource demands over time without using overbooking, while overbooking stayed relatively constant at around 90% utilization. Figure 13 shows the total amount of VMs allocated over time using overbooking and standard methods. Both of them depict the significant improvement obtained by using the overbooking technique, which increased resource usage between 44.7%

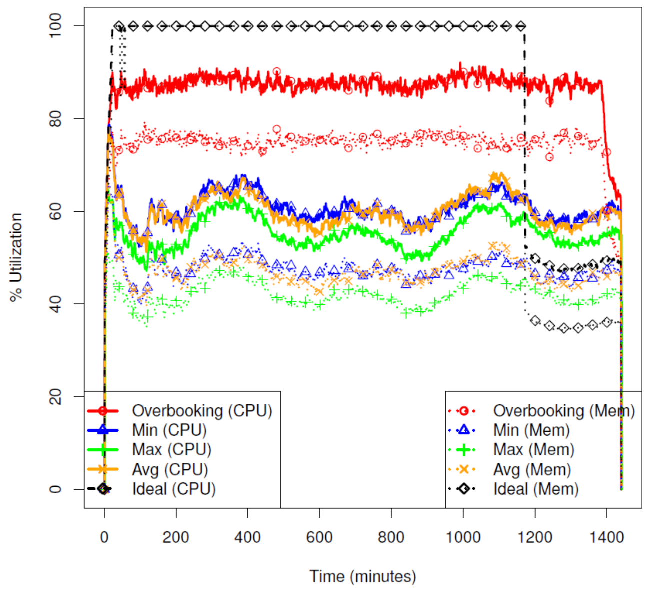

Figure 12: CPU and Memory Usage.

and 56.6% regarding CPU and between 55.8% and 76.1% for memory when compared to non-overbooking. This led to accepting between 3.3 and 6.3 times more dynamic workload applications in the same period of time.

5. CONCLUSION

Improving the efficiency of the cloud, in both terms of storage and computing power, has been shown to be a complex task. Manipulating indexes to minimize the amount of time a query takes, as well as to improve the accuracy of query results was shown in section 3. The results showed that the improvements gained benefited both the user and cloud provider. Applying overbooking to cloud computing resources was another method shown in section 4. The results demonstrated that the general waste of resources due to varying demands of applications was greatly diminished. While these methods are meant to be applied to different areas of cloud systems, storage and computation, it is not too hard to imagine them being used together to improve the total performance of a cloud system. Cloud computing allows users access to such a large amount of resources that it is a service that will continue to grow rapidly in the future. This paper mainly focused on two specific strategies to improve the performance and efficiency of cloud computing. However, the number of techniques and improvements possible to improve performance are endless.

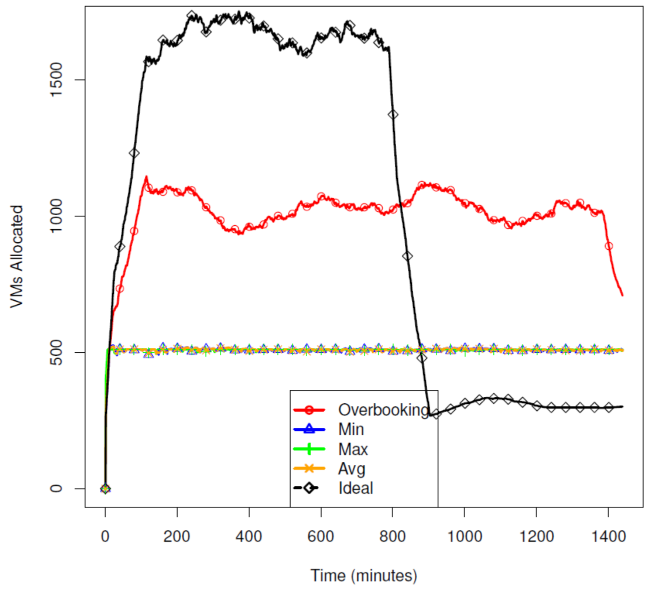

Figure 13: Allocated Virtual Machines.

6. REFERENCES

[1] J. Camacho-Rodríguez, D. Colazzo, and I. Manolescu. Web data indexing in the cloud: Efficiency and cost reductions. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 41–52, New York, NY, USA, 2013. ACM.

[2] E. Casalicchio, D. A. Menascé, and A. Aldhalaan. Autonomic resource provisioning in cloud systems with availability goals. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.

[3] D. Niyato, K. Zhu, and P. Wang. Cooperative virtual machine management for multi-organization cloud computing environment. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '11, pages 528–537, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[4] A. Ruiz-Alvarez and M. Humphrey. A model and decision procedure for data storage in cloud computing. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 572–579, Washington, DC, USA, 2012. IEEE Computer Society.

[5] B. K. Samanthula, G. Howser, Y. Elmehdwi, and S. Madria. An efficient and secure data sharing framework using homomorphic encryption in the cloud. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, pages 8:1–8:8, New York, NY, USA, 2012. ACM.

[6] J. Simão and L. Veiga. A progress and profile-driven cloud-vm for resource-efficiency and fairness in e-science environments. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 357–362, New York, NY, USA, 2013. ACM.

[7] L. Tomás and J. Tordsson. Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 5:1–5:10, New York, NY, USA, 2013. ACM.