C-Meter: A Framework for Performance Analysis of Computing Clouds

Nezih Yigitbasi, Alexandru Iosup, and Dick Epema
{M.N.Yigitbasi, D.H.J.Epema, A.Iosup}@tudelft.nl
Delft University of Technology

Simon Ostermann
simon@dps.uibk.ac.at
University of Innsbruck

***Abstract*—Cloud computing has emerged as a new technology that provides large amount of computing and data storage capacity to its users with a promise of increased scalability, high availability, and reduced administration and maintenance costs. As the use of cloud computing environments increases, it becomes crucial to understand the performance of these environments. So, it is of great importance to assess the performance of computing clouds in terms of various metrics, such as the overhead of acquiring and releasing the virtual computing resources, and other virtualization and network communications overheads. To address these issues, we have designed and implemented *C-Meter*, which is a portable, extensible, and easy-to-use framework for generating and submitting test workloads to computing clouds. In this paper, first we state the requirements for frameworks to assess the performance of computing clouds. Then, we present the architecture of the C-Meter framework and discuss several resource management alternatives. Finally, we present our early experiences with C-Meter in Amazon EC2. We show how C-Meter can be used for assessing the overhead of acquiring and releasing the virtual computing resources, for comparing different configurations, for evaluating different scheduling algorithms and for determining the costs of the experiments.**

I. Introduction

Cloud computing has emerged as a new technology that lets the users host and deploy their applications in a secure environment with a promise of increased scalability, availability, and fault tolerance. As the use of cloud computing environments increases [1], it becomes crucial to understand the performance of these environments in order to facilitate the decision to adopt this new technology, and understand and resolve the performance problems. In this paper, we present C-Meter, which is a framework for generating and submitting test workloads to computing clouds. By using C-Meter, users can assess the overhead of acquiring and releasing the virtual computing resources, they can compare different configurations, they can evaluate different scheduling algorithms and they can also determine the costs of their experiments.

Many big vendors like Amazon, Google, Dell, IBM, and Microsoft are interested in this technology and they invest billions in order to provide their own cloud solutions [1]. The cloud providers are responsible for maintaining the underlying computing and data infrastructure while at the same time reducing the administration and maintenance costs for the users. This is commonly known as *Infrastructure as a Service* (IaaS).

Today's cloud environments make use of virtualization technologies for both the computing and networking resources. Virtualization is an abstraction between a user and a physical resource which provides an illusion that the user could actually be interacting directly with the physical resource [2]. These resources, which have access to large and efficient data storage centers, are interconnected together and are provisioned to the consumers on-demand. The cloud computing environment is open to its users via well defined interfaces over well known Internet protocols enabling anytime and anywhere access to the resources, similar to other common utilities like electricity and telephony. The users can deploy their software by creating their customized virtual machines and running these virtual machine images on the resources in the cloud.

To assess the performance of computing clouds, we have designed and implemented C-Meter, which is a portable, extensible and easy-to-use framework for generating and submitting workloads and analyzing the performance of cloud computing environments. C-Meter is designed as an extension to GrenchMark [3], which is a framework for generating and submitting synthetic or real workloads to grid computing environments. Our contribution is threefold:

- We state the requirements for frameworks to assess the performance of computing clouds (Section II-B) and we design and implement the C-Meter framework which adheres to these requirements (Section III).
- We address the problem of resource management in computing clouds by discussing the pros and cons of various alternatives (Section III-C).
- We present how C-Meter can be used in practice through several experiments with Amazon EC2 and we assess the performance and cost of EC2 with scientific workloads (Section IV).

II. Problem Statement and Background

In this section, we describe the problem statement, and then we state the requirements that we have identified for frameworks to assess the performance of computing clouds. After that, we give an overview of Amazon EC2, which we

have used in our experiments. Finally, we present an overview of GrenchMark, within which we have implemented C-Meter.

A. Problem Statement

Big vendors and users are getting more and more interested in the cloud computing technology [1]. As a result, it becomes crucial to assess the performance of computing clouds in order to adopt this emerging technology, and identify main performance problems when using this technology. Currently there is a lack of frameworks for assessing the performance of computing clouds. The requirements for such frameworks should be identified and a framework adhering to these requirements should be implemented in order to conduct performance studies with computing clouds. Since no resource management components and no middleware exist for accessing and managing cloud resources, the problem of resource management in computing clouds should also be addressed by identifying various resource management alternatives and by discussing the advantage and disadvantage of these alternatives.

B. Requirements for Cloud Performance Analysis Frameworks

To assess the performance of cloud computing environments researchers need to have a framework which should satisfy various requirements. We have identified three main requirements for framework to assess the performance of computing clouds:

1) The framework should be able to generate and submit both real and synthetic workloads. It should gather the results and extract various statistics specific to computing clouds such as the detailed overheads of resource acquisition and release. It should also provide performance analysis reports to the users as well as a database of obtained statistics in order for the users to perform offline analysis for their own needs. The framework should also be able to determine the costs of the experiments.
2) It should let the users compare computing clouds with other environments such as clusters and grid. It should also let the users perform experiments with different configurations such as with different type and amount of resources.
3) Since currently no resource management components and no middleware exist for accessing and managing cloud resources, the framework has to provide basic resource management functionalities. The framework should be extensible in the sense that new resource management algorithms and support for new cloud environments can easily be added. It should also be platform independent and easy to use.

C. Amazon EC2

Amazon EC2 (Elastic Compute Cloud) is a web service that opens Amazon's cloud computing infrastructure to its users [4]. It is elastic in the sense that it enables the applications to adapt themselves to their computational requirements either by launching new virtual machine *instances* or by terminating virtual machine *instances* which are running. By using the EC2 web service, users of the cloud can launch, monitor and terminate the virtual machine instances. EC2 uses its own image format called AMI (Amazon Machine Image) for virtual machine images allowing the users to create their own virtual computing environments containing their software, libraries and other configuration items. To launch instances of these virtual machine images, the user has to upload the AMI to the Amazon S3 (Simple Storage Service) storage service. After the AMI is launched on a physical computing resource, the resulting running system is called an *instance*.

TABLE I
AMAZON EC2 INSTANCE TYPES AND RELATED COSTS FOR LINUX/UNIX INSTANCES.

Instance Type	Capacity (EC2 Compute Unit)	Cost (US $ per hour)
m1.small	1	0.10 $
m1.large	4	0.40 $
m1.xlarge	8	0.80 $
c1.medium	5	0.20 $
c1.xlarge	20	0.80 $

There are many instance types in Amazon EC2 environment which are grouped into two families: the standard and High-CPU [4]. Standard CPUs are suitable for general purpose applications whereas High-CPU instances have more computational resources and so they are more suitable for computationally intensive applications. The users pay per hour according to the instance type they have used. The instance types and the related costs for Linux/Unix instances are given in Table I.

D. GrenchMark

GrenchMark is a framework for generating and submitting synthetic or real workloads to grid computing environments. Over the past three years, GrenchMark has been used in over 25 testing scenarios in grids (e.g., Globus based), in peer-to-peer systems (e.g., BitTorrent-based), and in heterogeneous computing environments (e.g., Condor-based). By using GrenchMark, users can perform functionality and performance tests, system tuning, what-if analysis, and compare various grid settings. GrenchMark supports unitary and composite applications and allows the users to determine the job interarrival time distribution, letting them generate various workloads for their analysis. GrenchMark can also replay real traces taken from various grid environments by converting them into the standard workload format, hence it can help to perform realistic test scenarios on a large scale. However, GrenchMark can not satisfy the requirements stated in Section II-B since it does not have support for managing cloud resources and hence it can not be used for experiments with computing clouds. So, to satisfy these requirements, we have designed and implemented C-Meter as an extension to GrenchMark.

III. THE C-METER FRAMEWORK

In this section we present the architecture of the C-Meter framework. Then we explain the C-Meter experimentation

process and describe the various alternatives for resource management in clouds.

A. Overview

C-Meter is a portable, extensible and easy-to-use framework for generating and submitting both real and synthetic workloads to analyze the performance of cloud computing environments. It is designed as an extension to GrenchMark. It is portable in the sense that it is implemented in Python, which is a platform-independent programming language. It is extensible in the sense that, it can be extended to interact with many cloud computing environments and it can also be extended with different scheduling algorithms.

The architecture of C-Meter is illustrated in Figure 1. C-Meter consists of three subsystems. The *Utilities* subsystem is responsible for providing basic utilities and consists of four modules. The *Configuration Management* module is responsible for the configuration of the experiments. The user can define parameters such as the number and type of resources that should be allocated from the cloud and the credentials needed for authentication. The other modules are the *Statistics* module, the *JSDL Parser* module and the *Profiling* module. The functionalities of these modules are obvious as implied by their names. Thanks to the workload generation capabilities of GrenchMark and functionalities provided by this subsystem, C-Meter satisfies the Requirement 1 that we have identified in Section II-B.

The *Core* subsystem is responsible for providing the core functionalities of C-Meter and it consists of three modules. The *Listener* module is responsible for listening job submissions from the workload generator and commands from the user such as a command to terminate the experiment. After receiving the job descriptions, these descriptions are queued in the *Job Queue* until some resources become available for submitting these jobs. The *Job Submission* module is responsible for copying the executables and stage in files of the job to the HTTP server, and transferring a test proxy to a virtual resource in the computing cloud. This test proxy downloads the executables and stage in files from the HTTP server and executes the job and reports the statistics back to C-Meter.

The *Cloud Interaction* subsystem is responsible for interacting with the cloud environment under test. This subsystem consists of two modules. The *Resource Management* module is responsible for acquiring, managing and releasing the virtual resources from the cloud. The resource scheduling algorithms are also provided by this module. The user configures the resource specification in the configuration file and the *Resource Management* module interprets the specification and allocates resources accordingly. The *Connection Management* module is used to establish connections to the cloud environment in order to submit the jobs by the *Job Submission* module. By using functionalities provided by this subsystem, C-Meter provides basic resource management functionalities which means that C-Meter satisfies the Requirement 3. Users can submit the same workload to different environments like grids or clusters, by using the extensible architecture of GrenchMark, and as a result, they can compare the performance of various different architectures. By using the resource management capabilities of C-Meter, users can also perform experiments with different number of resources of both homogeneous and heterogeneous types, and they can compare different configurations for computing clouds. With these functionalities, C-Meter also satisfies the Requirement 2.

B. C-Meter Experimentation Process

The flow of a typical experiment when using the C-Meter framework is illustrated in Figure 1 (The numbers below correspond to the numbers in the figure):

1) The C-Meter user prepares a workload description file in the format of the GrenchMark workload description format. In this workload description file it is possible to define the type of the jobs (e.g. sequential or MPI jobs) and the statistical distribution for the job interarrival times.
2) The workload generator of GrenchMark uses this workload description for generating the workload in JSDL (Job Submission Description Language) format [5].
3) The workload is submitted to C-Meter.
4) C-Meter parses the job descriptions and copies the necessary executable and stage-in files to the HTTP Server.
5) After copying the files C-Meter launches a test proxy on the virtual resource which is responsible for running the executable.
6) The proxy downloads the executable and stage-in files from the HTTP Server.
7) The proxy runs the executable and reports the gathered statistics to C-Meter.
8) C-Meter stores these statistics in the results database.
9) Finally, when the user concludes the experiments C-Meter generates performance analysis reports for the experiment, and stores them in the results database.

C. Managing Cloud Resources

Since the user has complete control over the cloud resources there is a need to devise resource management schemes for effective management of these virtual resources. We have identified four alternatives for managing these virtual resources, based on whether there is a queue between the submission host and the cloud computing environment and whether the resources are acquired and released for each submitted job. The alternatives are shown in Figure 3.

For alternative A, a queue resides between the submission host and the cloud, and it is managed by a resource management component responsible for managing the virtual resources denoted by 'R' in the figure. The resource manager acquires all the resources at the beginning of the experiment and releases them at the end. For alternative B, a queue resides between the submission host and the cloud, but this time the resources are acquired and released for each job submission, causing overhead for each job. For alternative C,

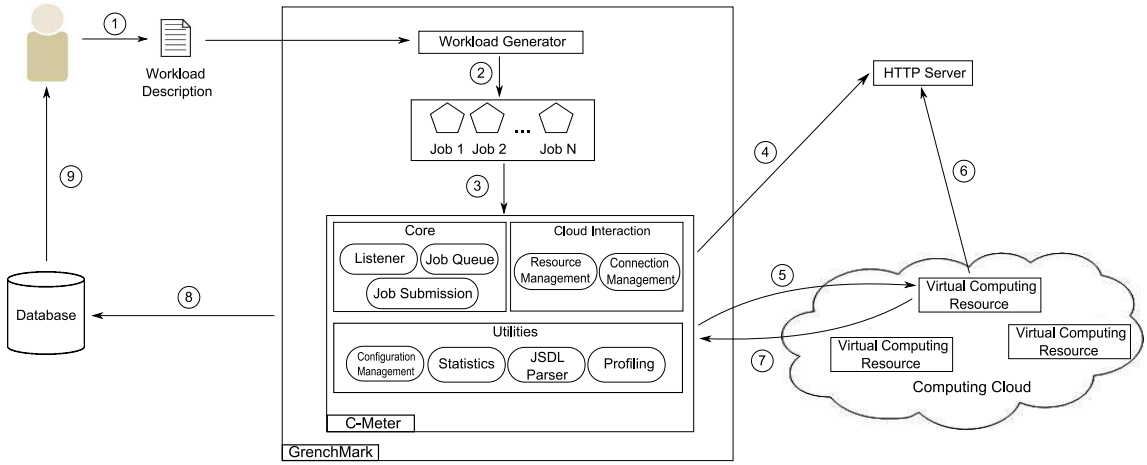
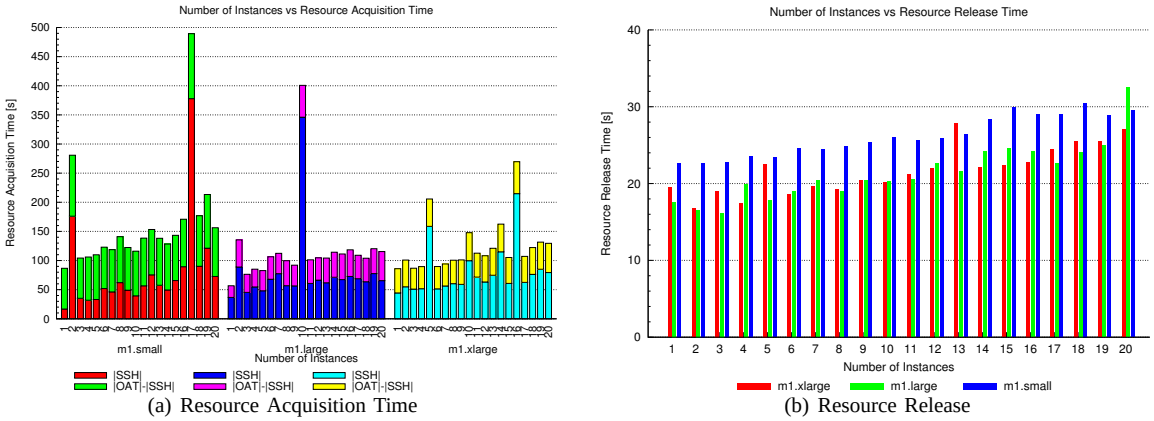


Fig. 1. The architecture of C-Meter and the flow of a typical experiment.



(a) Resource Acquisition Time (b) Resource Release

Fig. 2. Resource acquisition time (a) and resource release time (b) with different number of instances of standard instance types.

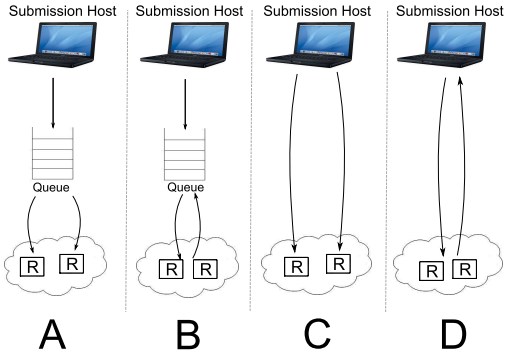


Fig. 3. Resource management alternatives for computing clouds. (R denotes the virtual resources in the cloud.)

there is no queue in the system and the application to be submitted is responsible to acquire the resources at startup and to release them at termination. Finally, for alternative D, there is no queue in the system like in C but the resources are acquired and released for each job submission causing overhead like alternative B. Although alternatives B and D seem inefficient, they may still be useful in practice, for example when there is a possibility of submission of jobs with malicious behavior which is more likely in production environments. After executing a job with malicious behavior on a resource, the resource can be left in an inconsistent state but since a new resource is acquired for each job then the next job will not be affected by the previous malicious jobs. Although not depicted in Figure 3, the resources acquired from the cloud can be managed in a time-shared or space-shared basis.

In C-Meter, we have implemented alternative A because alternatives B and D causes resource acquisition and release overheads for each job submission and in alternative C each application should implement a resource management scheme, which is impractical for most applications. And we have implemented time-shared resource scheduling in order to fully utilize the virtual resources. Assessing the effectiveness of each resource management scheme for computing clouds is in itself an interesting research topic, but it falls outside the topic of this paper.

IV. Experimental Results

In this section we present our early experiences with the C-Meter framework on the Amazon EC2. Three sets of experiments are performed. In the first set of experiments we have analyzed resource acquisition and release overhead for three different standard instance types, namely the *m1.small*, *m1.large* and *m1.xlarge* instance types (see Table I). In the second sets of experiments, we have analyzed the performance of different configurations e.g. configurations with different number and types of resources. In the last set of experiments we have analyzed the performance of two scheduling algorithms, the round robin scheduling algorithm and a predictive scheduling algorithm.

We have performed experiments with at most 20 instances which is the limit of the EC2 environment for testing purposes [4]. During the experiments we have used a workload of one thousand jobs consisting of single processor sequential applications with a Poisson arrival process of mean 500 ms. As performance metrics we have used the response time, the bounded slowdown with a threshold of 1 second [6], the time spent waiting in the queue, and the job execution time.

A. Overhead of Resource Acquisition and Release in Computing Clouds

In this experiment we have acquired and released 1 to 20 instances of standard instance types. The results show that C-Meter is capable of measuring the detailed overheads of acquiring and releasing virtual resources in the computing cloud.

The results of the resource acquisition experiments are given in Figure 2(a). *SSH* is the time for polling the resources using SSH to see whether it is ready to accept job submission. Since it is not enough to look for the state of the instance and check whether it is 'running'. *OAT* is the overall acquisition time for the resource. The summary of the results for Figure 2(a) is given in Table II. The outliers in the results have higher SSH overhead which is due to latency of the virtual machine boot or network latency between the EC2 and S3 to transfer the AMI.

TABLE II
Summary of Results for Resource Acquisition and Release Experiments.

Metric	m1.small	m1.large	m1.xlarge
Avg. Acquisition Time [s]	15.32	11.19	11.77
Avg. Release Time [s]	2.49	2.04	2.06

The result of resource release experiments is given in Figure 2(b) and also summarized in Table II. Releasing resources is a cheap operation that just turns off the virtual machine and triggers other possible housekeeping and cleanup operations. Acquiring resources is quite expensive since it involves transferring the AMI from the S3 store to the EC2 environment and booting that virtual machine.

B. Evaluation of the Performance and Cost of Different Configurations

The user can configure C-Meter to acquire both homogeneous and heterogeneous resources by using the resource specification fields of the configuration file. For example, by using a resource specification like 'm1.small=5', C-Meter can acquire 5 m1.small instances, whereas by using a resource specification like 'm1.small=10,m1.large=5,m1.xlarge=5' C-Meter acquires 10 m1.small instances, 5 m1.large instances and 5 m1.xlarge instances. This shows that C-Meter can be easily used to perform experiments with different configurations.

We have performed two experiments with six configurations. Each configuration consists of a specific number and type of resources. In the first experiment we submit the workload to three different configurations which consist of m1.small instance types of 5, 10 and 20 resources respectively. The results of this experiment are shown in Figure 4(a) and also summarized in Table III.

In the second experiment we submit the workload to three different configurations which consist of 10 instances with m1.small, m1.large and m1.xlarge instance types respectively. Figure 4(b) illustrates the results of our second experiment; these results are summarized in Table IV.

TABLE III
Summary of the results for the experiment with m1.small instance types of 5,10, and 20 instances. (Figure 4(a))

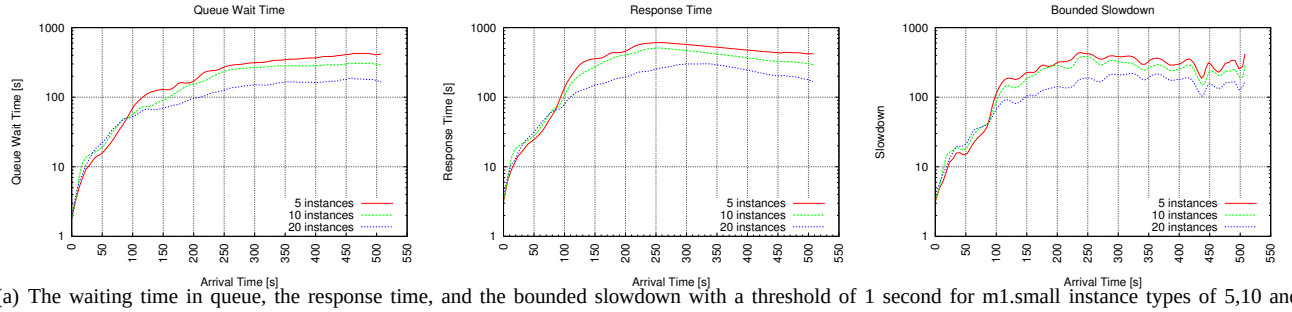
Metric	5 instances	10 instances	20 instances
Avg. Response Time [s]	396.40	313.92	187.27
Avg. Bounded Slowdown	321.87	257.15	155.62
Avg. Wait Time in Queue [s]	242.39	189.10	114.12

From Figure 4(a) we conclude that as the number of instances increases, the relative performance also increases. This happens because the load is distributed to more resources, therefore decreasing the queue wait times and also decreasing the network communications bottleneck. Hence, we can say that the cloud environment is *horizontally scalable*, which means that we can increase the performance of the system by acquiring more resources. From Figure 4(b) it can be seen that acquiring a more powerful resource causes the queue wait time, bounded slowdown and response time to decrease. We conclude that the cloud environment is also *vertically scalable*, which means that we can increase the performance of the system by acquiring more powerful resources.

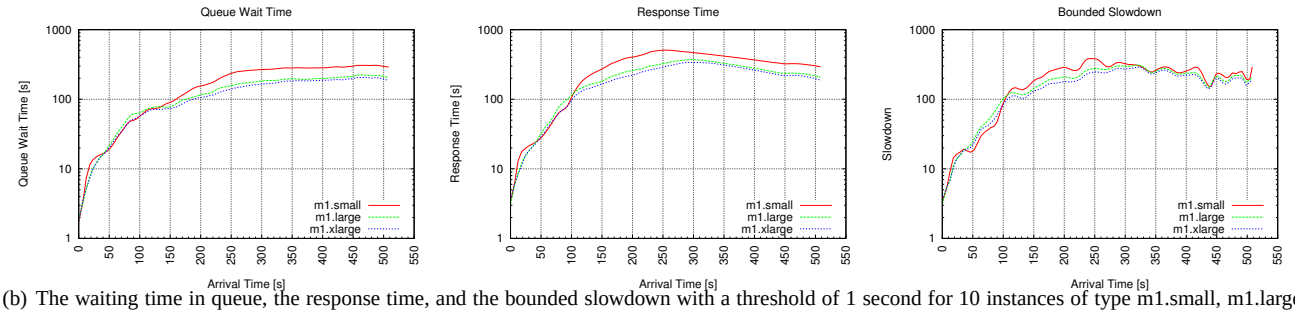
TABLE IV
Summary of the results for the experiment with 10 instances of type m1.small, m1.large, and m1.xlarge. (Figure 4(b))

Metric	m1.small	m1.large	m1.xlarge
Avg. Response Time [s]	313.92	228.12	207.40
Avg. Bounded Slowdown	257.15	203.10	184.85
Avg. Wait Time in Queue [s]	189.10	137.69	126.37

Another important issue is the cost of our experiments. Each instance type has a different cost and the user is charged per



(a) The waiting time in queue, the response time, and the bounded slowdown with a threshold of 1 second for m1.small instance types of 5,10 and 20 instances



(b) The waiting time in queue, the response time, and the bounded slowdown with a threshold of 1 second for 10 instances of type m1.small, m1.large, and m1.xlarge

Fig. 4. The waiting time in queue, the response time, and the bounded slowdown with a threshold of 1 second for m1.small instance types of 5,10, and 20 instances (a), and 10 instances of type m1.small, m1.large, and m1.xlarge (b). (Arrival times are relative to the first job submission and the vertical axis has a logarithmic scale.)

instance-hour consumed for each instance type, and fractional instance-hours consumed are rounded up and billed as full hours. The costs of a single experiment with a specific instance type and specific number of instances are shown in Table V.

TABLE V
COSTS OF A SINGLE EXPERIMENT IN US $.

Instance Type	5 instances	10 instances	20 instances
m1.small	0.50 $	1 $	2 $
m1.large	N/A	4 $	N/A
m1.xlarge	N/A	8 $	N/A

C. Performance Evaluation of Different Scheduling Algorithms

In C-Meter we have already implemented two different scheduling algorithms. The first scheduling algorithm is the round robin scheduling algorithm. The second algorithm is a simple heuristic that selects the resource with the minimum predicted response time. The response time is predicted by a simple time-series prediction method which uses the average of the last two response times of that resource as the prediction. By using C-Meter, users can implement different scheduling algorithms and evaluate their performance.

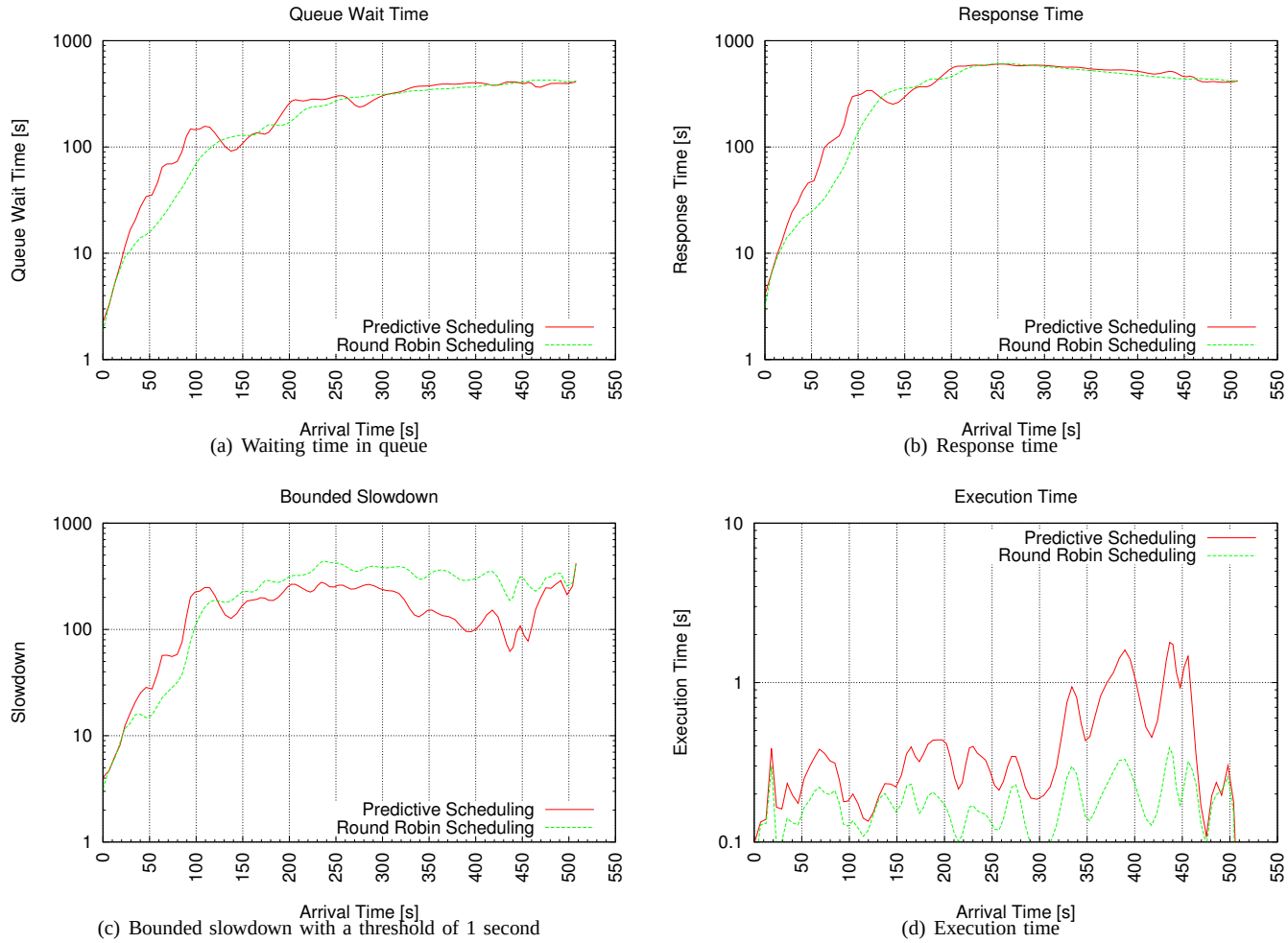
In this experiment we have used 5 instances of m1.small instance type and submitted the workload with two different resource scheduling algorithms. The results for this experiment are presented in Figure 5 together with a summary in Table VI. Although the average response time with the predictive scheduling algorithm is higher, the average slowdown is lower than with the round robin algorithm. We attribute this situation to increased job execution times. This increase is due to the fact that the predictive algorithm has scheduled many jobs to the same instance causing an uneven load distribution which resulted in higher loads and network latencies for that instance. The number of jobs scheduled per instance is given in Table VII. The average accuracy of the prediction method was quite good which is 80% by using the definition in [7]. This is an important result as it shows that better accuracy does not imply better performance.

TABLE VI
SUMMARY OF THE WAITING TIME IN THE QUEUE, THE RESPONSE TIME , THE BOUNDED SLOWDOWN AND THE EXECUTION TIME FOR 5 INSTANCES OF TYPE M1.SMALL WITH TWO DIFFERENT SCHEDULING ALGORITHMS.

Metric	Predictive	Round Robin
Avg. Response Time [s]	427.19	396.40
Avg. Bounded Slowdown	283.98	321.87
Avg. Wait Time in Queue [s]	150.02	145.29
Avg. Job Execution Time [s]	13.30	2.77

TABLE VII
DISTRIBUTION OF JOBS TO INSTANCES WITH DIFFERENT SCHEDULING ALGORITHMS.

Instance	Predictive Scheduling	Round Robin Scheduling
I1	404	200
I2	247	200
I3	89	200
I4	146	200
I5	114	200



(a) Waiting time in queue
(b) Response time
(c) Bounded slowdown with a threshold of 1 second
(d) Execution time

Fig. 5. Waiting time in the queue (a), the response time (b), the bounded slowdown with a threshold of 1 second (c) and the execution time (d) for 5 instances of type m1.small with two different scheduling algorithms.(Arrival times are relative to the first job submission and the vertical axis has a logarithmic scale.)

V. Related Work

Garfinkel [8], [9] has explained his experiences with Amazon EC2, S3 and SQS (Simple Queue Service) infrastructures. He has also made an analysis of the ease of use of the APIs, EC2 security and management facilities, availability and also the end-to-end performance analysis of S3 throughput and latency by making use of 'probes' from EC2. Walker [10] focused on analyzing the performance of EC2 for high-performance scientific applications by using micro and macro benchmarks. In his analysis he uses High-CPU instance types and compares EC2 performance with a cluster consisting of equivalent processors; he has also uses the NAS Parallel Benchmarks and the *mpptest* micro benchmark for demonstrating the performance of message passing applications. Wolski et al. [11] introduce their open source cloud computing software framework *Eucalyptus* and present the results of their experiments, which compare the instance throughput and network performance against Amazon EC2. They conclude that the performance of Eucalyptus is quite good and that it outperforms EC2 in an environment with significantly fewer resources. Palankar et al. [12] evaluate Amazon S3 in terms of availability, throughput and usage costs. They also discuss the security features of S3 and the necessary support of a storage service like S3 to satisfy the requirements of the scientific community. They conclude that S3 is not targeting the scientific community and they recommend using S3 when the costs of providing high data availability and durability are driven up by specialized hardware and needs nontrivial engineering effort.

VI. Conclusion and Future Work

In this paper we have presented C-Meter, which is a portable, extensible and easy-to-use framework for performance analysis of cloud computing environments. It is portable in the sense that it is implemented in Python which is a platform-independent programming language, and it is extensible in the sense that it can be extended to interact with many cloud computing environments and it can also be extended with different resource scheduling algorithms. We have also presented our early experiences with C-Meter on Amazon EC2 and performed various experiments and analyzed

the resulting performance in terms of response time, wait time in queue, bounded slowdown with a threshold of 1 second and job execution time with the standard instance types. We have also analyzed the costs of our experiments.

As future work we plan to perform more extensive experiments with Amazon EC2, and to extend C-Meter with different resource scheduling algorithms. We also plan to incorporate C-Meter into a complete resource management framework for computing clouds. For such a resource management framework many other problems should be addressed, such as job monitoring and control, and also important issues like fault tolerance.

REFERENCES

[1] T. Economist, "A special report on corporate it," October 2008, "http://www.economist.com/specialReports/showsurvey.cfm?issue=20081025".

[2] T. Killalea, "Meet the virts," *Queue*, vol. 6, no. 1, pp. 14–18, 2008.

[3] A. Iosup and D. Epema, "Grenchmark: A framework for analyzing, testing, and comparing grids," in *Proc. of the 6th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGrid06)*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 313–320, 13-17 May 2006, Singapore. An extended version can be found as Technical Report TU Delft/PDS/2005-002, ISBN 1387-2109.

[4] Amazon, "http://aws.amazon.com/ec2/," 2008.

[5] O. G. Forum, "Job submission description language (jsdl) specification, version 1.0," 2005.

[6] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, vol. LNCS 1291, 1997, pp. 1–34.

[7] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Trans. Parallel & Distributed Syst.*, vol. 18, no. 6, pp. 789–803, Jun 2007.

[8] S. L. Garfinkel, "An evaluation of amazons grid computing services: Ec2, s3 and sqs," Center for Research on Computation and Society School for Engineering and Applied Sciences, Harvard University, Tech. Rep., 2007.

[9] S. Garfinkel, "Commodity grid computing with amazon's s3 and ec2," *;login*, vol. 32, no. 1, pp. 7–13, 2007.

[10] E. Walker, "Benchmarking amazon ec2 for high-performance scientific computing," *;login*, vol. 33, no. 5, pp. 18–23, 2008.

[11] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "Eucalyptus : A technical report on an elastic utility computing architecture linking your programs to useful systems," Department of Computer Science, University of California, Santa Barbara, Tech. Rep., 2008.

[12] M. Palankar, A. Onibokun, A. Iamnitchi, and M. Ripeanu, "Amazon s3 for science grids: a viable solution?" Computer Science and Engineering, University of South Florida, Tech. Rep., 2007.