jBM - a CPU benchmarking tool for cloud environments

Pietro Piazzolla
Politecnico di Milano
Dip. di Elettronica e Informazione
Via Ponzio 34/5,
20133 Milano, Italy
piazzolla@elet.polimi.it

Marco Gribaudo
Politecnico di Milano
Dip. di Elettronica e Informazione
Via Ponzio 34/5,
20133 Milano, Italy
gribaudo@elet.polimi.it

Giuseppe Serazzi
Politecnico di Milano
Dip. di Elettronica e Informazione
Via Ponzio 34/5,
20133 Milano, Italy
serazzi@elet.polimi.it

ABSTRACT

Cloud computing is an ever growing reality nowadays. The possibility of leasing and provisioning resources is an important alternative to the construction of dedicated data centers. At the same time, multi-core and many-core architectures are becoming more and more popular and widely available. However, such architectures present several challenges due to their high complexity. In particular it is essential to be able to study, test, model and simulate various aspects of these new environments, with suitable tools. In this paper we present jBM, a simple tool for the benchmarking the CPU of new architectures like multi-core or cloud systems. The tool consists of a multi-class workload generator and service centers that can be distributed along different nodes. The applicability of the tool is proven through several applications to evaluate the performance of virtual machines on Amazon EC2.

1. INTRODUCTION

Evaluation techniques have improved with the continuous evolution of knowledge and of the complexity of computer systems architectures. Among the various techniques applied since the early days of computers, *benchmarking* is one of the most widely used. It is primarily used in design and procurement studies since it applies a direct measurement approach and thus, it requires that the system under evaluation is built and running. A *known* workload is input to the system to be evaluated and the values of various performance indexes are measured during the run. In spite of the simplicity of the operation involved, i.e., execution and measurements, benchmarking is a complex technique. The main reason for this is that the measured values are influenced by a very large number of variables; some of them are related to the architecture of the system under observation while other are connected to the ways in which are run the experiments and to the workload characteristics.
The characteristics of the benchmarking tools evolved in parallel with the evolution of system architectures. Thus, the initial tools suitable for mainframes were followed by those for distributed systems, networks, client server systems, intranets, and web servers.
Among the tools appeared on the market for the various architectures are those to measure the performance of web servers (see e.g., httperf [3]), those oriented to the transactional web e-commerce servers (see, e.g., TPC benchmarks [10], SPECs [8], or RUBiS [5]), and those for the evaluation of websites that use dynamic content (see, e.g., Apache Jmeter [9]). Some of them are commercially available, while others are open source. The focus of this paper is to present the jBM tool package in its main features. A more in-depth discussion about related works deserves a full paper itself.
With the recent advent of multi-core architectures and cloud infrastructures, a new set of problems appeared whose solution resulted in deep changes to the existing tools with the introduction of new sophisticated features.
The jBM tool tries to solve one of these new challenges, cosidering first issues related to CPUss. In multi-core systems the behavior of a process is a function of several parameters: power of the cores, bandwidth and size of the memories, hw and sw multi threads level, characteristics of the algorithm to be implemented, expertise of the software developers on parallel processing, and others. As we will see, jBM allows the collection of data that can be used to study the degree of parallelism of an architecture, by allowing the comparison with very simple analytical models.
In cloud environments, users have a very limited, or null, knowledge about several important operational conditions like the processing capacity allocated to their VMs or the workload shared by the same physical system where their VMs are allocated. However, in several problems this knowledge is fundamental, e.g., performance forecast of an application, reliability assessment of a particular configuration, or flexible acquisition and release of resources. jBM can be used to identify the power of a CPU, either single or multicore, allocated to a user request.
It allows also the generation of a workload consisting of several types (classes) of applications and the control of the mixes of different types of applications in concurrent execution (mix of programs).
The paper is organized as follows. Sections 2 and 3 are devoted to the description of the architecture of the tool and of its modules. Section 4 describes some examples of its applications to the identification of the operational conditions of a VM and of the computational power allocated to a VM. An experiment with the generation of a controlled workload with two classes of applications is also shown in the same

Simutools 2013, March 05-07, Cannes, France

DOI 10.4108/icst.simutools.2013.251751

section. Section 5 concludes the paper.

2. THE JAVA APPLICATION

In Figure 1 a graphical representation of the jBM architecture is presented. The core module of the jBM, the Java Application, is composed by three elements interacting together. They are called *Master*, *Client* and *Resource* (M, C and R respectively in Figure 1). Since the main goal of the application is the reproduction of multi-tier systems, these components are meant to be distributed along several different machines connected via a *network*.

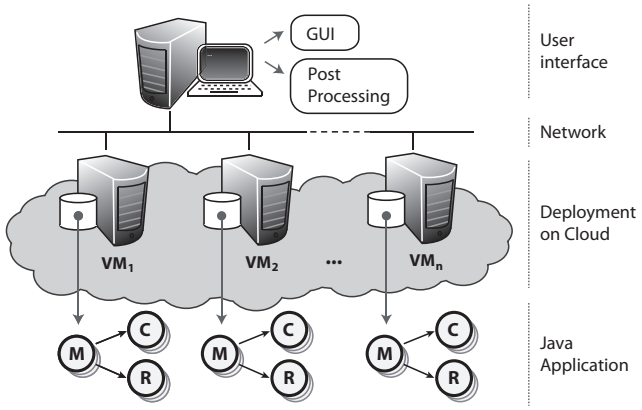

Figure 1: Architecture of the jBM benchmark tool. On each machine VM_n the user can instantiate one or more Masters M to control Clients C and/or Resources R.

The application is conceived to operate in a Cloud Computing environment, hence in this paper we call *Virtual Machines* (VM_n in Figure 1) the nodes where the jBM core module is deployed. This however does not preclude other deployment solutions, because the same components can be equally deployed over physical machines or other types of mixed environments.

All the VMs that compose the experiments run the main jBM component, the Master M. It is loaded at startup as the only user-defined process on the OS of the VMs composing the testing environment. Since no other processes external to jBM are started, all the CPU resources are at the application's disposal. The Master M component is a control service that listens to incoming commands on a given port. Its main feature is the capability to start one or more instances of either Client C or Resource R components. These components are instanced as new processes running over the same VM as the Master M, and they compose the core Java application. Obviously, this means that each new instance of either Client C or Resource R is going to use some of the CPU power provided by the VM where they are executed.

In order to avoid unwanted interference between processes or an overload of some CPU, it is usually better to design a single VM to be either a Client C or Resource R. In a multi-core or multi-processor environment, if the operating system allows it, the user may also set the processor affinity and assign a component to a specific core[1]. The benchmark could also run on a single machine in which more than one Client C or Resource R share the same VM. This configuration however is less preferable since the contention of resources (even other than the CPU) might interfere with the benchmark execution.

The Clients C component is a load generator that sends requests to a user-specified address according to a Poisson process with a specified rate λ. Resources R are servers that receive requests from Clients and process them with an exponentially distributed service time S. Let $\mu = 1/S$ be the rate of the corresponding exponential distribution. Both components can be controlled via a web interface.

The jBM tool can support several workload classes and multiple resources. In case of multiple Resources R, each job must visit all the stations according to a user-specified order before being considered completed. If there are more than one Client C, the requests started from each of them are considered belonging to a different class. Each class c is defined by its own arrival rate λ_c and by a vector S_{cr} specifying the service time requirement of the class at each resource, and by an array of IP addresses that identifies the target resources. Every request generated by a Client C is controlled by a new thread that contacts all the resources defined in the associated sequence. The mean service time for that specific resource is embedded in the request generated by the client. Resources R are thus passive devices, that are controlled exclusively by the requests incoming from the Clients C. Each request taken in charge by a Resource R is also served by a new dedicated thread and it is executed by running a random number of basic iterations, each performing a CPU intensive activity, corresponding to the computation of the Discrete Cosine Transform (DCT) of a large number of random data. The choice of the DCT allows to tune the length of each basic iteration for the particular underlying architecture: in particular the number of coefficients that must be transformed can be increased to produce longer executions on faster machines, or it can be reduced to fit the data entirely in the processor cache to avoid unwanted memory locks or bus interference. Moreover the execution of the transform involves both integer and floating point operations, creating thus a good basic block for a CPU bound application. The actual number of iterations performed during a request is generated according to an exponential distribution[2], and can be configured by the user in two way: a) to obtain a specific mean service time, or b) to execute a mean number of iterations.

In order to ensure the same mean service time on each VM instance, independently from the underlying processor power, a *calibration value* CV is introduced. The value of the CV represents the mean number of iterations per millisecond that the considered CPU is able to perform. In particular, if the target service time that the resource must spend to perform the service is Sms, the total number of iterations is chosen according to an exponential distribution of parameter $EXP(S \cdot CV)$. Each Master M, at its startup, determines its CV_{CPU} with a calibration procedure. This CV is then used to scale the number of iterations required to obtain a given mean service time. In particular, the cali-

[1]In Linux environments this can be usually achieved by using the `taskset` command of the `util-linux` package.

[2]The actual number of iterations is chosen as the nearest integer to a sample of an exponential distribution. Other arrival distributions can be implemented easily by modifying the Client C component only.

bration procedure measures the time τ (in ms) required to perform a large number of iteration N_{it} and sets the calibration value to their ratio: $CV_{CPU} = N_{it}/\tau$.
For the second mode of operation, where the goal is to obtain the same mean number of iterations regardless of the particular architecture, the calibration procedure is ignored, and the calibration value is set to a constant value CV_{fixed} selected by the user. In this way, the number iteration required to serve a request is chosen according to the same exponential distribution with parameter $EXP(S \cdot CV_{fixed})$ on all the CPUs. For this particular configuration, the parameter S that defines the service time of a server, does not correspond to an actual mean time, but to a measure of its complexity. After the service, the corresponding thread is terminated and a message is sent back to the Client C who generated the request. The Client C, as soon as it receives this message, either sends the request to the next Resource R in the sequence, or, if no more Resources R are present, considers the request completed. Timestamps of threads start and end at each Resource R are logged by the Masters M and stored into files for post processing. Various performances indexes, such as utilization and mean response time for a request, are computed by from the log files using the other components of the package.

The user can interface the core Java Application components (Masters M, Clients C, and Resources R) by mean of a web-based GUI, that can contact the VM on which each of them reside through the network. The user interface is also provided with tools for processing data obtained from the benchmarking operations. Both the GUI and the Post Processing tools modules are detailed in Section 3.

3. TOOL PACKAGE

A general description of a framework for measurement-based performance evaluation is described in [11]. The jBM tool, beside the Java Application module described in Section 2, includes other modules that will be described in this section. In particular, a web-based GUI to control the execution of the benchmarks, and a set of tools for Post Processing the data obtained and compute the relative performance metrics are provided. The latter tools are distributed as VBA Macros in Excel Spreadsheets. The complete package can be downloaded from [7], and its use is free for academic and non commercial purposes. In Figure 2 the modules of the tool package are presented in order of use.

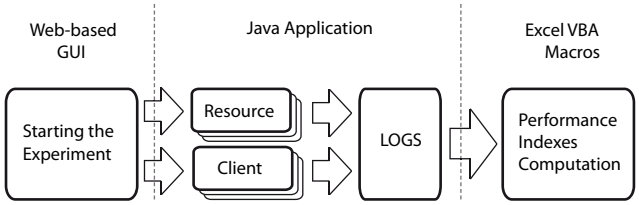

Figure 2: The jBM benchmark tool usage pipeline.

As briefly introduced, both the Master M, the Client C, and the Resource R components are controlled from a GUI. The tool interface is accessed via a standard internet browser. and it is written in Javascript and HTML. The main goal of the GUI is setting up the test bed environment. The jBM core elements are controlled by means of `HttpRequest`s containing the parameters required to issue a specific task. The actual parameter string may be complex to compose manually, especially when several resources are considered or multiple experiments must be performed. The GUI automatize the string composition by allowing the use of a textual description of the experiment. In particular, the user has to set two kind of parameters: *connection* and *experiment*. *Connection* parameters specify how many *Master*s are listening as well as their IP addresses and TCP ports. Moreover they are also used to define how many *Client*s (classes) and *Resource*s are required from a given *Master*. Because instanced *Client*s and *Resource*s can be contacted independently from their *Master*, the user is required to specify a listening port for each of them[3]. *Experiment* parameters on the other hand specify all the S_{cr} vectors and the λ_c for each of the *Client*s, as well as the duration of the benchmarking process. It is also possible to have more benchmarking processes run in a sequence. All these successive runs share the same *connection* parameters but can have different *experiment* parameters. These runs are automatically started and stopped by the GUI. The other commands that can be issued using the GUI are those that retrieve informations about the CPU type (`cpuinfo`) of the *Master*s machines, that start or stops the current benchmarking run, that save or display on screen the timestamps from the logs.

```
# number of classes
1
# for each class: IP masterPort clientPort
127.0.0.1 8080 8090
# number of resources
2
# for each resource: IP masterPort serverPort
127.0.0.1 8081 8091
127.0.0.1 8082 8092
# Trial Time in seconds
1500
# Number of Trials:
2
# Trial 1
# Lambda vector (c1, c2, c3 ... cn)
0.001
# Demand matrix (row: classes, cols: resources)
800 500
# Trial 2
# Lambda vector (c1, c2, c3 ... cn)
0.0012
# Demand matrix (row: classes, cols: resources)
800 500
```

Listing 1: Example of textual parameters used in the GUI

In Listing 1 an example of the textual parameters used to set up the experiments with the GUI is presented. Here the GUI is used to set up a system composed by one client and two resources[4]. As said, for each client and resource to be instanced, the user is asked to specify the IP address:port of their Master, as well as their own. In this particular example, two Masters are running locally, albeit listening at different ports. These connection parameters are going to be shared among the different experiments, called *Trials*. For each trial the experiment parameters specified are the λ vector and the Demands matrix D. The example shows

[3] *Client*s and *Resource*s share the IP address of their *Master*.
[4] Note that lines starting with the # symbol are comments.

the settings for two trials, each with a different arrival rate λ but identical demand matrix D.

As said, the jBM was originally conceived to operate in a Cloud Computing environment, hence an integration with the Amazon EC2 web service API is also provided in the package of the tools. This module allows to launch and stop EC2 VM instances and it can be used as deployment layer for the $Master$s. As introduced, a set of Microsoft Excel `*.xslm` macro is also included in the package: it contains VBA procedures useful for two type of tasks. First, it is possible to analytically compute expected performance values from the parameters used to tune the testbed environment. Second, it is possible to compute utilization, response time, mean queue length and other performance indexes from the log files data, with the possibility to exclude outliers by performing a simple transient elimination procedure to reduce the effects of the initial state of the system. The two kind of results obtained using the macros can be compared to obtain a deeper understanding of the analyzed system. In Figure 3 some screenshots from the jBM package are shown.

4. EXAMPLES

In this section we show some examples of the use of the jBM tool. All the following experiments have the $Master$ components deployed on several Amazon EC2 virtual machines and use a standard laptop as the client from where the user interface is operated. One of the peculiarities of Amazon EC2 VMs is that, although providing similar performances, each VM instance may be executed on a different hardware characterized by a different CPU. This feature can produce large fluctuations in the perceived performance (see [2]). The OS installed on the VM where the masters are executed is the default Linux release provided by Amazon.

4.1 Interference measurement

We first run a group of experiments to check the level of interference of the operational conditions in the cloud, e.g., we test whether the number of VMs on the physical host of the benchmark has some influence on the performance experienced by the benchmark. To make the system more similar to an M/M/1 queue, we chose a VM with a single-core CPU, and we set the Operating System scheduling to the FIFO policy (as opposed to the *Completely Fair Scheduler* (CFS) [6] that is the default on the considered VMs). The service times S are exponentially distributed with mean value $S = 0.8s$, and the arriving flow of requests is generated by a Poisson process with a parametric arrival rate λ job/sec. Since Amazon randomly provides VMs with different CPUs types [5], we perform the calibration to obtain the same mean service time on all the CPUs. If the interference of other VMs running on the same physical hardware is negligible, we expect that the provisioned VM instance behaves as a ture M/M/1 queueing station. In this case we can compute the expected utilization and mean response time using the well known expressions (see, e.g., [4]): $U = \lambda S$ and $R = S/(1-U)$. The comparison of the analytical results for the response time (solid lines) and the measured values (points corresponding to the different CPUs) is shown in Figure 4. For each value of arrival rate λ we

[5]In our test, the provided VMs were equipped with one of the following CPUs: Intel Xeon E5507, Intel Xeon E5645, Intel Xeon E5430, AMD Opteron A2218

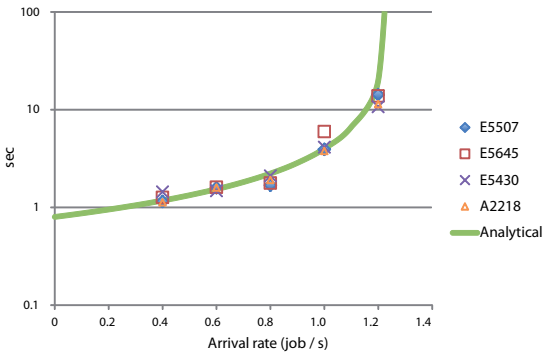

Figure 4: Response time of different Amazon EC2 `m1.large` instances, single-core, calibrated and with FIFO scheduling

ran one hundred benchmark executions at different times of different days. The confidence intervals at 95.0% have been computed, but they are not shown in the figure to improve its readability. The expected analytical result falls inside the confidence interval: for example, with $\lambda = 0.4$ $jobs/s$ the theoretical value of R is 1.176 s and for `E5507` the interval is 1.052-1.248. As it can be seen from the figure, the measurements closely match the values derived analytically for all the CPUs. This conclusion is important to proove the validity of the jBM tool.

We then show how jBM can be used by fixing the mean number of iterations to study performance of a given CPU. In this case, since the calibration value CV is not computed, the time required to perform a given mean number of iterations depends on the CPU type. From the measurement collected by the tool, we can compute the actual service time of the corresponding CPU using a fitting procedure. The estimated service time S for the different CPUs are show in Table 1. Figure 5 shows the response time of the various CPUs in single-core mode, and of four M/M/1 models with the service time matching the estimated result. As it can be seen, the measurement closely match the analytical values.

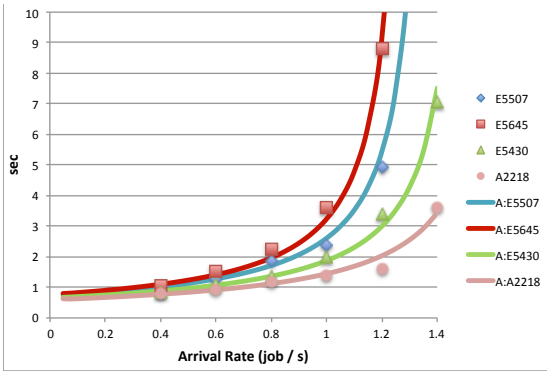

Figure 5: Response time of different Amazon EC2 `m1.large` istances, single-core

We continue the study the architectural differences among the CPUs provided by Amazon, by measuring the response times of the different CPUs when serving the incoming work-

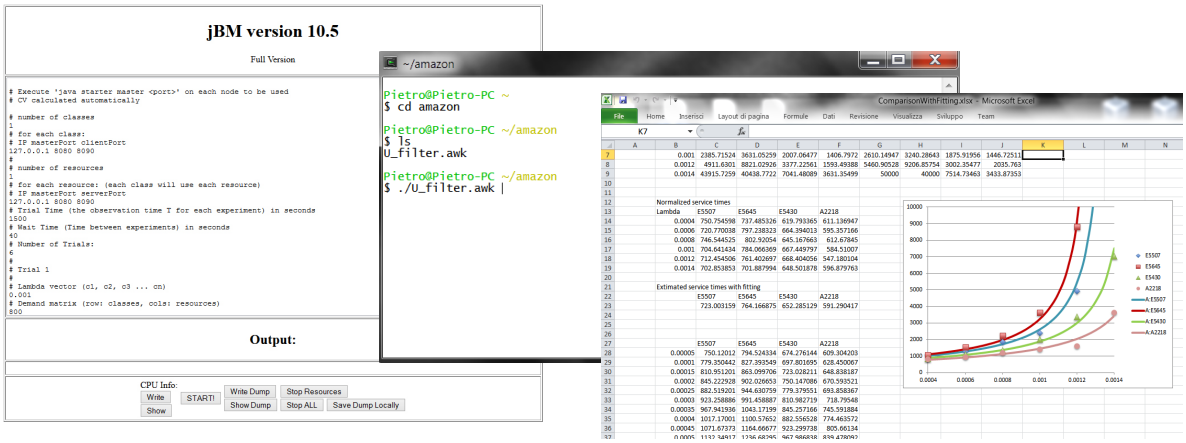

Figure 3: A screenshot of the jBM benchmark tool package.

CPU	E5507	E5645	E5430	A2218
Estimated S	0.723	0.764	0.652	0.591

Table 1: Estimated service demands for the different CPUs.

load in dual-core mode. Since the jBM application uses a different thread for each request, software parallelization is possible and we expect the system to serve an higher workload before reaching saturation. We perform the measurements of our benchmark application with the calibration activated to obtain the same mean service time on all the CPUs. Ideally, a dual-core machine with exponential service time, FIFO policy, handling an incoming Poisson workload, should behave like a M/M/2 queue. In practice this does not occur because during simultaneous operations of both cores in a single CPU, memory locks, limited size of the caches and reduced bandwidth decrease the actual parallelization between the two cores. The experimental response times for different CPUs and the one of an ideal M/M/2 are compared in Figure 6. As it can be seen, the similarity between a dual-core and an M/M/2 server strongly depends on the CPU type. Only the E5645 processor performs very close to an M/M/2 queue, meaning that its capability to exploit the parallelism of the cores is higher with respect to the other CPUs. The results collected with the jBM tool can then be used to measure the degree of parallelism that can be achieved by a particular multi-core architecture.

4.2 Multi-tier systems

To show the use of the tool in a more complex setting, we consider a system with two resources involving two successive CPU-bound operations for two different classes of jobs (e.g. a two-tier web service involving an application server and a data-base, that has to serve two different classes of requests). The requests for the two classes are produced by two different load generators, and they are characterized by the demand matrix **D** shown in Table 2. To reduce the interferences among the classes, the complete system has been implemented by setting up four EC2 instances: two corresponding to the server resources, plus two for load generators

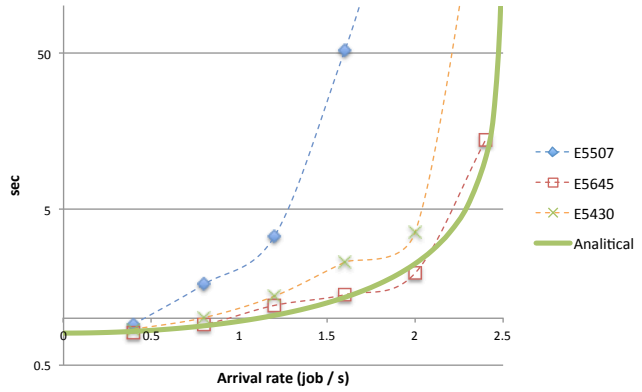

Figure 6: Response time of different Amazon EC2 `m1.large` instances, dual-core, calibrated

(one for each class). All the instances are single-core, and uses a FIFO scheduling discipline. The considered queueing network is product form and its behavior can be studied analytically using for example the Mean Value Analysis technique, which is implemented in several existing tools such as the JMVA component of the JMT [1] tool. We set the total

D	Class 1 (*sec.*)	Class 2 (*sec.*)
Resource 1	0.8	0.25
Resource 2	0.45	0.7

Table 2: Service demands of the two classes of the test workload.

arrival rate to $\Lambda = 1.3 req./sec.$, and we perform several experiments where we divide the requests along the two classes with a different proportion β, that we address as *population mix*. In particular we set the arrival rate for class one jobs $\lambda_1 = \Lambda * (1 - \beta)$, and for class two jobs to $\lambda_2 = \Lambda * \beta$. Figure 7 shows the response times and Figure 8 the utilization of the system, computed both with the analytical solution and with the data collected by our benchmark. As it can be seen, there is a close match between the two. Note that for

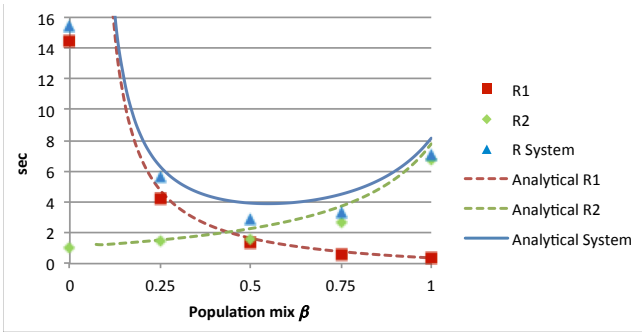

Figure 7: Response Time Results using EC2 AMIs compared with expected analytical values.

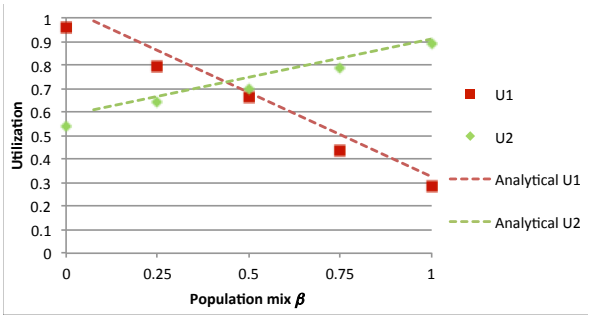

Figure 8: Utilizations Results using EC2 AMIs compared with expected analytical values.

$\beta = 0$ (i.e., all class 1 jobs), the real system is unstable, since in this case the inter-arrival time is shorter than the service time. In jBM we measure the instability with a response time that increases with the duration of experiment. The saturation of the system can be better appreciated from the utilization graph (Figure 8), where the utilization of the first resource is 1 for $\beta = 0$.

5. CONCLUSIONS

In this paper we have presented the jBM CPU benchmarking tool, and proved its validity by matching the experimental data with the corresponding analytical results. Although the workloads handled by the tool are very simple, it allows to determine how far the system under study differs from its ideal behavior. The tool has already been used in previous works to study both multi-core systems and cloud based solutions (e.g. see [2]).
On going extensions focus on improving its user interface and its deployment. Moreover, the possibility to benchmark not only CPU, but also I/O activity is currently being developed, together with the support of more complex arrival patterns and non-exponential service time distributions.

5.1 Acknowledgments

This work has been partially supported by the "AWS in Education research grant" from Amazon, and by the "ForgeSDK" project sponsored by Reply S.R.L.

6. REFERENCES

[1] M. Bertoli, G. Casale, and G. Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[2] D. Cerotti, M. Gribaudo, P. Piazzolla, and G. Serazzi. Flexible cpu provisioning in clouds: A new source of performance unpredictability. In *Quantitative Evaluation of Systems (QEST), 2012 Ninth International Conference on*, pages 230 –237, sept. 2012.

[3] HP Labs. httperf. http://www.hpl.hp.com/research/linux/httperf/, Nov. 2012.

[4] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice-Hall, 1984.

[5] OW2 Consortium. Rubis. http://rubis.ow2.org, Nov. 2012.

[6] C. S. Pabla. Completely fair scheduler. *Linux J.*, 2009, Aug. 2009.

[7] Polimi performance group website. http://www.perflib.net, Nov. 2012.

[8] Standard Performanc Evaluation Corporation. The spec benchmarks. http://www.spec.org, Nov. 2012.

[9] The Apache Software Foundation. Apache jmeter. http://jmeter.apache.org, Nov. 2012.

[10] Transaction Processing Performance Council. the tpc benchmarks. http://www.tpc.org/information/benchmarks.asp, Nov. 2012.

[11] D. Westermann, J. Happe, M. Hauck, and C. Heupel. The performance cockpit approach: A framework for systematic performance evaluations. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pages 31 –38, sept. 2010.