OFBench: an Enterprise Application Benchmark for Cloud Resource Management Studies

Jean Moschetta
Imperial College London
Department of Computing
jean.moschetta07@imperial.ac.uk

Giuliano Casale
Imperial College London
Department of Computing
g.casale@imperial.ac.uk

***Abstract*—We introduce OFBench, a new research benchmark for enterprise applications. OFBench is a load generator for the demo e-commerce component of the Apache OFBiz enterprise resource planning (ERP) framework. ERP applications are increasingly important in the cloud market due to the growing popularity of Software-as-a-Service (SaaS) solutions, hence OFBench is useful to explore the resource consumption patterns arising in these applications. On client-side, OFBench features randomized customer surfing patterns, workload generation based on automation of real web browsers, reproducible burstiness in inter-request and inter-session think times, non-stationary workload generation. On server-side, it features JMX performance monitoring, business logic instrumentation, and synchronization with client-side monitoring via TCP sockets. We illustrate the execution of the benchmark in example runs and use them to investigate the overheads of JMX-based monitoring.**

I. Introduction

Cloud resource management studies often require the generation of representative workloads to test the effectiveness of the allocation policies being investigated. In this context, benchmarking tools for web applications are widely used to gain insights on performance issues that can arise under stress loads [17]. However, several benchmarks that are popular in the research literature, such as TPC-W and RUBiS, are dated and hence not fully representative of business applications deployed today on the cloud, such as enterprise resource planning (ERP) systems which are of growing popularity in the SaaS delivery model, e.g., in web-based order management [28, p.177]. For example, in TPC-W, the underlying software stack is composed of a database and a servlet container, whereas enterprise applications today rely on a more complex integration of web services, dynamic scripting, database connection pooling, advanced logging, management extensions, template engines, unicode, XML processing. These layers introduce additional complexity and inter-dependencies which are being investigated in the current research literature on layered software systems. Furthermore, client-side technologies such as Javascript and AJAX are not always considered in existing research benchmarks, but they can still affect the user-perceived performance and the network traffic patterns [16]. This calls for acquiring workloads from applications with a more realistic technology stack than those considered in most of today's research literature.

To deal with this requirement, we propose OFBench, a realistic research benchmark for the demo e-commerce store distributed with the Apache Open For Business (OFBiz) ERP framework. OFBiz is a framework deployed in production environments, hence our benchmark allows to experiment with a software architecture and a technology stack that are representative of real business applications. For these applications, OFBench can stress a range of performance indicators such as response times, CPU utilization, memory consumption, number of active sessions over time, multi-threading levels, data caching. JMX measurement beans and code instrumentation are used on server-side to perform monitoring. Real web browsers are used to drive user emulation, by means of the Selenium automation framework for Mozilla Firefox [10]. Compared to an HTTP replayer, this approach allows for more realistic stress testing of the presentation tier of the application, for example by including in the response times also the browser rendering time components. On the other hand, using real web browsers may require more hardware resources on client-side to emulate tens or hundreds of concurrently executing users. OFBench also defines probabilistic workload profiles to represent realistic customer behaviors. Furthermore, following the trend of benchmarking approaches that natively support reproducibility in the distribution of the number of active sessions and in the burstiness of client inter-arrival times, which were proposed only in recent years [22], [24], [21], [14], OFBench integrates the generation of request bursts using a generalization of the model used in [24]. This feature can be useful to investigate the response of cloud platforms to sudden bursts in the arrival stream of requests. Validation is performed on a case study investigating the overheads imposed by JMX monitoring as the measurement granularity increases. Our analysis suggest that sub-second resolutions for CPU and memory monitoring leads to increasingly severe overheads and, perhaps less obviously, also to measurement errors. Overheads are observed to be correlated to a large increase of CPU I/O waits. Furthermore, we illustrate example runs of OFBench on a lab testbed and on a cloud platform with similar configurations which suggests that the change in deployment environment impacts the resource demand of individual transaction types in a heterogenous way that appears difficult to predict.

The remainder of this paper is organized as follows. Sec-

tion II overviews related work and basic definitions. Section III introduces Apache OFBiz 12.04. Section IV describes the OFBench architecture and its main features. Finally, a case study related to monitoring overheads is given in Section V, followed by concluding remarks.

II. Related work

Related work includes benchmarks such as TPC-W, RUBiS, SAP-SD, and SPEC benchmarks. TPC-W [18] and RUBiS [12] are popular open source benchmarks, widely used in the research literature on multi-tier applications. The TPC-W benchmark [18] specifies an e-commerce application which simulates the activities of an on-line bookstore. Users are emulated through Remote Browser Emulators (RBEs), i.e., threads which generate a realistic HTTP traffic. RBEs send requests to the online bookstore according to probabilistic navigation patterns. These patterns are defined according to a graph with probabilistically weighted transitions between nodes, each representing a page. The long-term probabilities of visiting a page define the stationary request mix that is expected in the system during operation. TPC-W defines three such mixes: the browsing mix, the ordering mix, and the shopping mix.

RUBiS [12] is an auction site benchmark that emulates the surfing activities of three types of users: visitors, buyers, and sellers. Requests are of 27 different types and include operations such as registration, bidding, item selling, and comment rating. RUBiS was developed to compare the performance of different implementations of an auction site based on solutions such as PHP, Enterprise Java Beans (EJBs), or Java servlets, and is now an established benchmark for multi-tier application research. Client workloads offer two mixes, browsing and bidding, which are respectively composed of read-only operations and read-write operations. The software stack employed by RUBiS includes Java servlets, enterprise Java beans (EJBs), Tomcat or JBoss as servlet container, and MySQL. Recently, the RUBiS default generator has been found to lack realism in the generation of non-stationary behavior, which instead is observed in real web applications [30].

SAP SD [9] is a commercial ERP benchmark for sales and distribution operations. Requests are generated by a fixed set of clients and sent to the SAP Netweaver application. This server coordinates a set of worker threads, called work processes, which serve incoming requests. The work processes are executed as either dialog processes, which run interactive programs, or as update processes, which perform database updates. This benchmark is widely used in the SAP market for sizing ERP application deployments. Several other commercial benchmarks for enterprise applications exist and have been investigated in the research literature. For example, SPEC benchmarks for message-oriented middlewares (SPECjms, [26]) and for web servers (SPECweb, [27]).

III. Apache OFBiz

Apache Open for Business (OFBiz) [4], [29] is an open source framework for developing enterprise resource planning (ERP) applications. In this paper, we define the OFBench tool for OFBiz release 12.04. OFBiz runs within the Apache Geronimo application server [3]. The framework adopts a multi-tier architecture composed of a database back-end layer, a business logic layer, and a presentation layer that implements the standard model-view-controller (MVC) paradigm. The main features of each layer are summarized below.

Presentation layer. The presentation layer is handled by several components. We here refer to this layer as the *view engine*. The view engine main task is to generate page elements, called *views*, to be sent to the client browser. Views are composed to generate a screen, which corresponds to the display of the entire page on the browser. In OFBiz, screens are first assembled from views and then decorated with menus, footers, headers, and secondary elements by a component called the OFBiz widget toolkit. This operation is based on XML templates, which can be specified using the FreeMaker framework [6]. The view engine has also the ability to retrieve data from the data layer through actions which implement the business logic. This allows, for example, the processing of orders and the display of the results of these operations. The screen generated by the presentation layer is a combination of HTML elements, Javascript, and AJAX code. The latter two languages play a significant part of the interaction an end-user has with the OFBiz e-commerce store. This provides a motivation for the use of real web browsers for workload generation on client-side, since HTTP replayers are normally unable to handle Javascript and AJAX. Note, however, that this imposes a higher demand on the client workload generator machines, especially in terms of memory demand.

Data layer. The *entity engine* is a database abstraction layer. A view may request some data to display by sending a query to the entity engine, which will call the database to acquire data. The data layer provides a connector which allows OFBiz to accept different types of database management systems (DBMS). By default OFBiz comes with the Apache Derby database [2], but it can be easily configured to use other databases such as MySQL. Database connection pooling is managed via Apache DBCP and its control parameters (e.g., thread pool size) assigned via the OFBiz configuration files.

Business logic layer. Views can call actions which define the business logic and can be implemented in different ways, e.g., web services, Java code, Groovy scripts. In the e-commerce application considered by OFBench, the business logic is almost entirely done in Groovy. It consists of 65 scripts that invoke the Java functions of the OFBiz ERP framework library. The business logic also support the service-oriented architecture (SOA) paradigm, thus it allows to invoke remote web services via SOAP, RMI, either synchronously or asynchronous. Asynchronous interactions are not always available in existing research benchmarks.

A. Request Processing

We now describe the typical interaction pattern between the OFBiz framework and a client. The flow of an OFBiz request starts with a web browser sending a request to the servlet

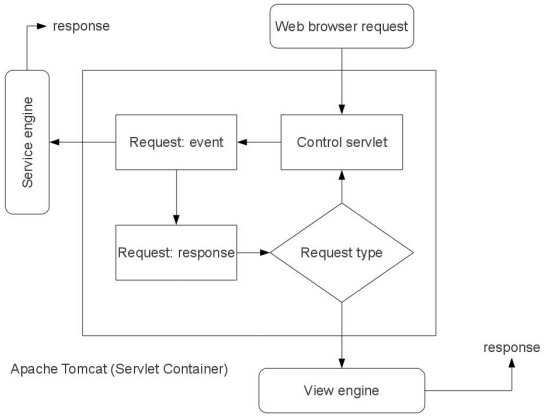

Fig. 1. OFBiz servlet architecture

container, which by default is the Tomcat container running within Apache Geronimo. At this point, the servlet container receives the request and passes it to the control servlet. At this level, events can be associated with requests. As can be seen from Figure 1, some events are processed by the service engine, which can invoke independent, possibly remote, pieces of business logic using, e.g., SOAP web services, JMS, RMI. Service responses themselves can then trigger other requests with the purpose of generating the screen returned to the user. This request-event-request loop is shown in Figure 1 at the request type diamond, which may loop to trigger other requests before ending at the view engine that returns the screen to the client browser.

B. E-Commerce Component

OFBiz components are the building blocks of the framework. Generally, a component defines an application, such as the e-commerce store component. This is shipped in OFBiz within the special-purpose components family. The e-commerce store demonstrates the implementation of a commercial website in Apache OFBiz and its interfacing with an ERP framework. The e-commerce demo implements a standard user interface that includes product search and browse by category, product listings based on multiple catalogs, shopping cart management, customer checkout, registration and login features. A demo instance[1] maintained by the OFBiz team is available online at [11]. OFBiz also comes with several small admin applications for order management and site usage analysis. In the present version, OFBench does not stress these admin components.

IV. OFBench: Architecture and Features

OFBench is organized into two software components, written in Java, running on client-side and server-side. On client-side, OFBench deploys a load generator featuring randomized customer surfing patterns, workload generation based on automation of real web browsers, reproducible burstiness in inter-request and inter-session think times. On server-side, it implements a JMX-based monitoring system that extracts performance metrics from the OFBiz Java Virtual Machine (JVM) and from the business logic code. The two software systems cooperate via TCP signals.

[1]OFBench is based on Apache OFBiz 12.04. The online demo maintained by the OFBiz is instead periodically updated to the latest version.

A. Load Generation

The client-side load generation architecture is the result of the coordination of several components, namely an experiment overseer, a workload generator, a dispatcher, and the emulated user agents. User agents are Mozilla Firefox instances running in Xvfb, an X11 server that operates in memory, not showing any graphical output on the screen. The experiment overseer is responsible for coordinating the phases of the experiment (warm-up, steady-state, cool-down). It does so by communicating via TCP sockets with the dispatcher, which is the component that instantiates and coordinates user agents. The current version of OFBench supports a single overseer-dispatcher pair, however in principle TCP signals allow a generalization of the architecture to distributed client machines. We aim to develop in future work this extension.

Steady-State, Warm-Up, and Cool-Down. Performance measures can significantly deviate from their stationary values during the initial and final stages of a benchmarking experiment. It is then common to consider a *warm-up phase*, in which the benchmark performs initial tasks (e.g., user registrations), and a *cool-down phase* in which clients are gradually taken offline, which is sometimes useful to avoid data inconsistencies. Measurement is considered significant only during the *steady-state* period, which corresponds to the observation period without the warm-up and cool-down phases. In the current implementation, the warm-up period corresponds to the time needed to register all users. Instead, the cool-down period starts after a user-predefined time since the end of the warm-up and it is carried out by simply allowing all running sessions to finish. In upcoming releases, we plan to investigate automatic methods for warm-up period detection. This may be feasible for stationary workloads, but it is a less explored domain for workloads with burstiness.

Experiment overseer. The overseer initially creates the dispatcher thread. Upon start, the dispatcher thread creates the user agent browsers. Then it enters a loop in which it feeds sessions to the user agents which have entered the dispatch queue. A session is a sequence of requests specified by a workload generator component, together with a random session inter-arrival time and a sequence of inter-request think times. Sessions feature requests of two kinds, regular requests and null requests. A regular request is an ordinary page view request. A null request is a request that does not generate load for the server, but that identifies in the logs the boundaries of a use case. This feature may be useful to reconstruct the high-level semantics of the workloads from the log files.

The user agents schedule requests using a closed-loop approach, i.e., a browser does not submit a new request until the previous one has completed. All operations are carried out

TABLE I
OFBENCH REQUEST CHARACTERIZATION – 1 CLIENT EXPERIMENTS. TIMES IN SECONDS.

	Response time	Mean		Coeff. Var.	
	Request name	Lab	Cloud	Lab	Cloud
Cart group	CartAddAll	1.02	0.65	0.36	0.86
	CartAddFirst	0.38	0.58	0.76	0.76
	CartAddRoundGizmo	0.69	0.62	1.17	0.58
	CartAddTinyChromeWidget	0.62	1.09	0.86	0.73
	CartView	0.38	0.68	0.43	0.33
	Compare	0.30	0.41	0.41	0.70
	CompareAddFirst	0.28	0.41	0.78	0.26
Products group	CategoryAccountActivation	0.14	0.14	0.38	0.09
	CategoryConfigurablesFoods	0.12	0.14	0.05	0.07
	CategoryConfigurablesPCs	0.16	0.77	0.28	2.42
	CategoryDropShipProducts	0.11	0.16	0.08	0.64
	CategoryGiftCards	0.12	0.15	0.03	0.00
	CategoryGizmos	0.10	0.15	0.06	0.47
	CategoryServices	0.14	0.17	0.07	0.34
	CategoryWidgets	0.12	0.15	0.42	0.44
	QuickAddMain	0.83	1.31	0.68	0.19
	QuickAddWidget	1.14	2.17	0.92	0.23
	SearchRand	0.59	1.25	0.19	0.32
Checkout group	CheckOut	0.43	0.73	0.34	0.49
	CheckOutAddressNext	0.58	0.67	0.23	0.46
	CheckOutPaymentNext	0.77	0.87	0.21	0.36
	CheckOutReviewSubmit	2.34	5.36	0.77	0.74
	CheckOutShippingNext	0.69	0.78	0.52	0.70
	OrderHistory	0.39	0.73	0.33	0.23
	OrderHistoryView	0.36	0.62	0.50	0.40
Customer group	ContactUs	0.28	0.63	0.45	0.14
	Home	0.39	0.93	0.26	1.00
	Login	0.26	0.43	0.25	0.40
	LoginDetails	0.69	1.13	0.21	0.27
	LoginGetPasswordHint	0.41	0.50	0.19	0.15
	Logout	0.43	0.94	0.30	0.33
	Main	0.53	0.93	0.37	0.41
	Register	0.90	1.52	N/A*	N/A*
	RegisterDetails	2.89	4.24	N/A*	N/A*
	Aggregated Mean	0.50	0.84	0.39	0.46
	Aggregated Median	0.40	0.67	0.33	0.37

$*$: *Register* and *RegisterDetails* execute only once during warm-up.

using the Selenium 2.24.1 browser automation library [10]. In case of request error, a session is immediately terminated and a newly generated session is assigned to the user agent by the dispatcher. The dispatcher also instructs user threads to terminate once it receives the cool down signal from the overseer.

Inter-Arrival Times. In OFBench, inter-arrival times are either inter-session times or inter-request times. An inter-session time is the think time of a user agent before starting a new session, either at boot or after completing the previous session. Inter-request times instead introduce think times between receiving a request and sending the following one in the session. OFBench generates both kinds of inter-arrival times according to the n-state Markovian Arrival Process, also called the MAP(n) model [15]. A MAP is a special type of hidden Markov model used to create a flow of requests that behaves significantly different from the Poisson process, e.g., with burstiness, high-variability, periodicities. A MAP(n) consists of a continuous-time Markov chain with n states where each transition $i \to j$ is associated with a probability $p_{i,j}$ of changing the state and contextually generating an arrival (*observable transition*) and with a probability $h_{i,j} = 1 - p_{i,j}$ of performing a state transition without associated arrival (also called a *hidden transition*). In state i, transitions are generated according to a Poisson process with rate v_i. It can be shown that a MAP(n) is a special case of hidden Markov model where inter-arrival times of observations are phase-type distributed. A MAP(1) model is simply a Poisson process with rate v_1. Tools for automatically fitting MAPs from empirical data are available in open source implementations, see [15] and references therein, which can then provided in input to OFBench. For example, this may be useful to study the impact of workload fluctuations at different timescales on a cloud deployment. Furthermore, given measured traces, MAPs allow to generate new traces that are statistically similar to the existing ones. This may be useful to assess the sensitivity of the experiment.

Customer Behavior Model Graphs. User agents send requests to the OFBiz instance, thus we need a systematic way of generating, ordering and dispatching these requests. Similar to other benchmarks, OFBench adopts Customer Behavior Model Graphs (CBMGs) [23]. These directed graphs describe the probability of a browser sending request j after completing request i, for any choice of i and j. Thus, CBMGs are effectively discrete-time Markov chains modelling user surfing patterns. The homepage is assumed to be the start page of all CBMGs. OFBench implements 34 browser interactions, first defined using the Selenium IDE Firefox plugin [10] and later exported to Java code. These interactions are currently organized into 5 use cases for non-logged users and 11 use cases for logged users.

User sessions. User sessions are generated by randomly traversing the reference CBMG and subject to a maximum session length defined by the user. OFBench provides a reference CBMG with a simple tree-like structure and customizable use case selection probabilities. We are currently studying the performance properties of the benchmark with the aim of defining a restricted set of default mixes to offer to the users. In addition to these probabilities, several other parameters can be manually customized in the OFBench configuration files. These include probability of login, probability of forgetting the password, probability of session abandonment, probability of continuing to browse, maximum buy quantities, random seed of the session generator, and enabling/disabling client-side browser caching. In addition, OFBench implements a novel approach to control workload stationarity. Non-stationarity requires the ability of generating CBMGs which do not lead the system to converge to a stationary regime. OFBench currently supports this by allowing the automatic generation of randomized CBMGs for each user agent. Sessions are generated from these random CBMGs, instead than using a single reference CBMG. This is makes it more difficult for the system to converge to steady state, in particular for what regards the mix of requests in operation on the server.

Workload: Requests. A list of the 34 request types defined by OFBench is given in Table I. In the table, requests are grouped by business function: *Cart* includes requests related to

cart management and product comparison; *Products* mostly involves browsing and searching operations; *CheckOut* requests deal with orders; *Customer* involves customer service related functions such as account operations and homepage access. The table provides an indication of the request response time distribution in terms of mean and coefficient of variation, i.e., the ratio of standard deviation to the mean, for an experiment with a single client. The data refers to the warmup-up phase for the *Register* and *RegisterDetails* requests, and to the steady state lasting 3 hours for all the others. Browser caching is disabled and the workload is configured to logout every user upon finishing the session. The purpose of these experiments is to show the variability in execution times in a system without congestion and compare these results between a lab testbed and a cloud deployment with a similar configuration. The cloud testbed is a 4-core virtual machine on the Flexiant cloud [5], configured with 8GB of RAM. Each virtual core has 2.6GHz and 1MB of cache for each core. OFBiz is configured to execute from a ramdisk for increased I/O performance and tuned as a production system following the guidelines in [31]. The lab testbed has similar computational power: the machine has 2 quad-core Intel Xeon 5540 2.53GHz processors, each with 2 hardware threads, and a 8MB L3 share cache. Memory for this server is 32GB, thus significantly larget than in the cloud platform, however memory is not a bottleneck for this experiment. In an effort to make the runs two comparable, we have configured the lab testbed with just 1 thread per processor and disabled 4 of the 8 available physical cores.

The results indicate that different groups behave quite differently on the lab testbed and on the cloud platform. The requests of the *Cart* group mostly decrease performance when moved to the cloud. The *Products* group shows limited sensitivity to the cloud migration. The *CheckOut* group shows instead a deterioration of performance on the cloud platform. Finally, the *Customer* group shows a mixed behavior that depends on the transaction. The coefficient of variation columns reveal a more homogenous behaviour between lab testbed and cloud platform. However, it is quite surprising to find a 40%-50% mean variability in the performance of the transactions, in particular because we are running a single client. We attribute this effect to the complexity of the enterprise application being considered (e.g., caching behaviour, data-dependence, load-dependence). This example illustrates the challenges involved in migrating enterprise applications to the cloud and provides a quantitative idea of the magnitude of such degradations even in a lightly-loaded system.

B. Monitoring system

The monitoring system of OFBench operates both at client-side and at server-side. All statistics are recorded on both sides using Log4J in a set of comma separated value (CSV) files. On client-side, the monitoring system logs response times, number of active agents over time, and generates traces that detail the workload generator activity (e.g., inter-request times, inter-session times, session information). On server-side, OFBench uses Java Management Extensions (JMX) to collect performance measures from the JVM running OFBiz [8]. JMX allows a client application, here the OFBench monitor, to create a connection with a target JVM, here the OFBiz JVM, possibly from a remote location. The monitor runs in its own JVM, separate from the OFBiz one, and retrieves performance data of the target JVM from MXBeans, which are control management objects allowing a more flexible definition of the exchanged data types compared to ordinary JMX MBeans. The polling frequency can be set in the OFBench configuration files. MXBeans are able to monitor threads, processor usage, memory usage, and several other hardware and software resources used by the JVM. Several predefined MXBeans are shipped with the Sun/Oracle JVM, including the following ones that are used by OFBench:

- The MemoryMXBean, which reports memory usage such as heap and non-heap memory as well as currently committed memory and maximum available to the JVM.
- The OperatingSystemMXBean, which reports CPU time used by the OFBiz JVM.
- The ThreadMXBean, which reports statistical information on the system threads.

Additionally, OFBench introduces a special MXBean, called BizObjGate, which is useful to study access patterns to business objects. This MXBean effectively "gates" the access to the Groovy scripts. We have manually integrated BizObjGate with all 65 Groovy scripts of the e-commerce store. For each access, BizObjGate records a timestamp and the unique identificator of the user session. In practice, this approach is similar to the one used by aspect-oriented languages for profiling, which however adopt different monitoring techniques such as bytecode instrumentation. It may be interesting in the future work to compare these two approaches on the OFBiz application.

The BizObjGate allows us to track both the inter-access times for a script and the load placed by each individual user on the business logic. This data can also be combined with the Geronimo log files, which include detail on the HTTP and HTTPS requests that have been served, and with the Derby log files, that can be configured to detail database activity up to recording individual statements.

OFBench also collects data from an external CPU monitor, Hyperic's System Information Gatherer[2] (SIGAR) [7]. The double monitoring system based on JMX and SIGAR is useful for cross-checking on measurement quality and also to assess the overhead imposed by the monitor when this is deployed on the same machine together with OFBiz.

V. Case Study: In-VM Monitoring Overhead

In this section, we present a case study that illustrates the JMX monitoring system of OFBench. The case study stems from the observation that deployments in the cloud may require the development of ad-hoc monitoring solutions to compensate the coarse-grained monitoring resolution offered

[2]SIGAR is licensed under the Apache License, Version 2.0.

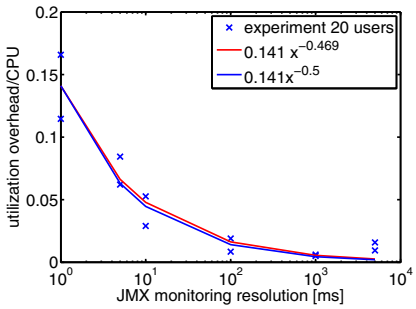

Fig. 2. OFBench JMX Monitoring Case Study

by the monitoring platform. For example, Amazon CloudWatch provides free measurements at 5-minutes frequency, whereas a subset of metrics is offered at 1-minute resolution under payment of a small fee [1]. In this case study we are interested in comparing the overheads resulting from high-resolution JMX CPU and memory utilization measures, studied with resolutions from $1ms$ to $5s$.

As before, we have collected measurements running OFBench on the FlexiScale public cloud platform [5] with a client configuration of 20 and 40 users, operating from two remote client machines in London. We used a VM instance with 4 CPUs, 8GB of RAM, and 20GB of disk. Intra-session times and inter-session think times are both exponentially distributed with a mean of $3s$. Monitoring resolutions are varied across experiments to take the values $1ms$, $5ms$, $10ms$, $100ms$, $1ms$, and $5s$. Each experiment is run for approximately 10 minutes in steady-state. Monitoring overheads are given in Figure 2, together with the best numerical fit, according the R^2 metric, obtained using MATLAB's curve fitting tool (cftool). Note that the vertical axis is an overhead *per core*. The results indicate that JMX overheads are well fitted by a function that is inversely proportional to the square root of the monitoring resolution, i.e., $f(x) = 0.141/\sqrt{x}$. A slightly better fit is obtained by the function $f(x) = 0.141x^{-0.469}$, however Figure 2 suggests that the deviation from the square root formula is small. We note that both fitted functions are unable to capture the overheads at $5s$, but the variability of this overhead from the $1s$ case is small and presumably due to background noise fluctuations. We plan to further investigate into this effect in the future.

Throughout the experiments, the average CPU utilization for the OFBiz instance with 20 users is 18%. We have experimented also at 30% utilization (40 users case), but we have found that the overheads at this resolution are similar to the ones in the 20 users case experiments. Summarizing, the results indicate that JMX overheads grow nonlinearly with the monitoring resolution. Given that production environment may typically accept monitoring overheads not larger than 5%-10% [13], resolutions in the range $100ms - 5s$ appear the safest for practical purposes. Notice that this observation is valid under the assumption of performing CPU and memory monitoring only. With additional instrumentation, e.g., using thread monitors, overheads can be significantly larger at the same resolution. Thus, a separate analysis should be performed for each monitor in isolation from the others. We plan to investigate this in future work.

In order to understand better the effects of JMX monitoring, we have also carried out a separate experimental campaign using our lab testbed described in Section IV-A. In this campaign, we consider an experiment with 50 users, having mean think times of $4s$; the testbed was configured with 2 hardware threads and all cores enabled. Monitoring resolutions are set to $5s$, $500ms$, and $50ms$ in separate experiments. Figure 3 reports experimental results. Figure 3(a) and Figure 3(b) indicate that the impact of the monitoring overheads on response times and number of active agents may be small. However, the impact on server-side performance is very visible, as shown in Figure 3(c)-(d). The mean utilization for the three experiments is 8.9% with $5s$ resolution, 11.7% with $500ms$ resolution, and 62.7% with $50ms$ resolution. High resolution measurement implies larger memory usage and higher CPU utilization for OFBiz JVM and for the monitor JVM. The latter is responsible for the difference between total CPU and OFBiz CPU. Figure 3(f) suggests that part of the reason of this difference is the increased time the CPU is blocked waiting for I/O. We have also studied the distribution of the difference between the total and the OFBiz utilization. In nearly 10% of the cases at $50s$ the difference assumed was negative, whereas this problem did not appear at $5s$ and was significantly smaller at $500ms$. This suggests that high JMX resolution measurement is not only costly in terms of resources, but may also introduce errors.

VI. Conclusion

In this paper, we have presented OFBench, a research benchmark for studying the performance offered by the OFBiz software system. OFBiz features a demo e-commerce store that leverages an enterprise resource planning (ERP) framework. ERPs are increasingly important in the cloud SaaS distribution model, thus OFBiz provides a realistic business logic and software stack for cloud resource management studies.

We have described the architecture of the OFBench tool and illustrated its execution on the cloud. We have observed complexity in predicting the performance of an application on the cloud when migrated from a lab testbed to the cloud, even in the lightly loaded case. We have also provided a case study that quantifies the overheads incurred by JMX monitoring as measurement resolution becomes finer-grained.

Future work will focus on distributed load generation and on defining standard workload mixes for the benchmark. Further information on the benchmark development status and public releases will made available at [32].

Acknowledgement

The work of Giuliano Casale is partially supported by an Imperial College Junior Research Fellowship and by the European project MODAClouds (FP7-318484). The authors wish to thank Flexiant for providing cloud access and Techavit Tantaviboonwong for his help on the JMX monitor.

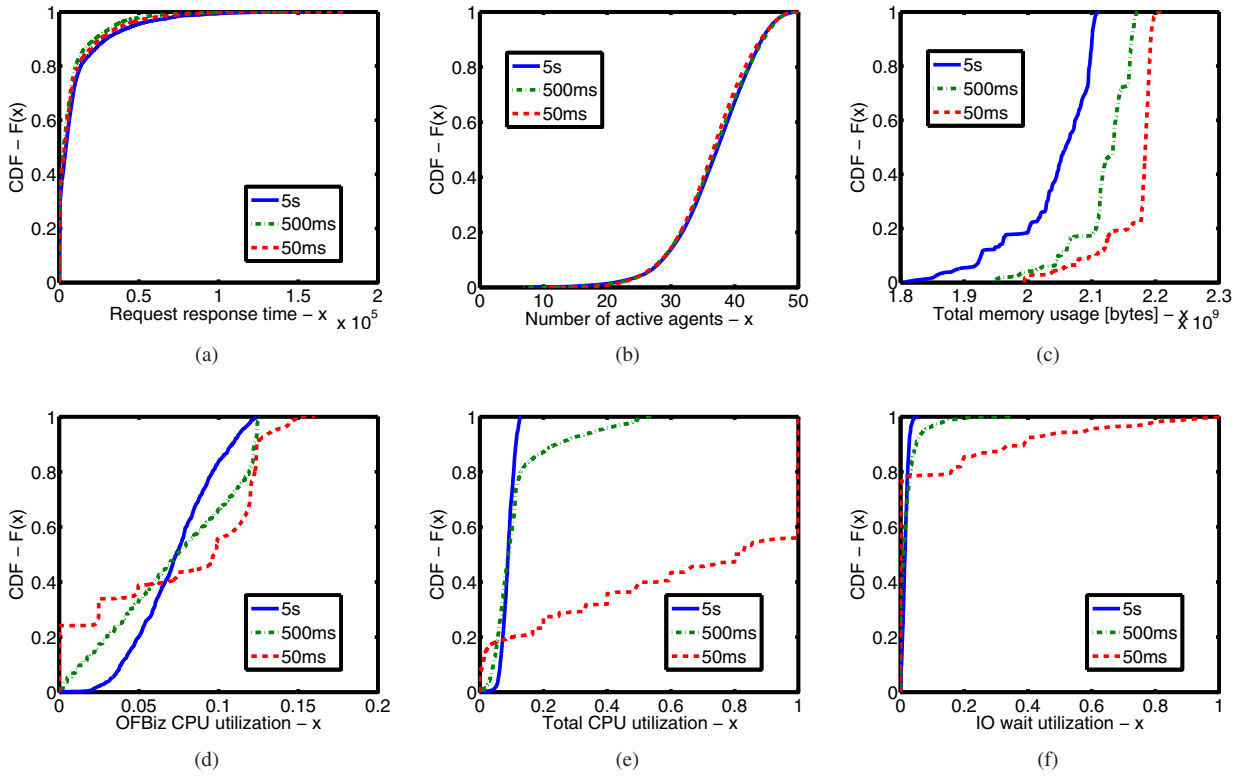

Fig. 3. JMX overhead at increasing resolution - Lab experiment (50 users, 16 cores)

REFERENCES

[1] Amazon cloudwatch. http://aws.amazon.com/cloudwatch/.

[2] Apache Derby project. http://db.apache.org.

[3] Apache Geronimo project. http://geronimo.apache.org.

[4] Apache OFBiz project. http://ofbiz.apache.org.

[5] FlexiScale public cloud. http://www.flexiscale.com/products/flexiscale/.

[6] Freemaker template engine. http://freemarker.sourceforge.net.

[7] Hyperic SIGAR. http://www.hyperic.com/products/sigar/.

[8] JMX. http://docs.oracle.com/javase/tutorial/jmx/.

[9] SAP SD. http://www.sap.com/solutions/benchmark/sd.epx.

[10] Selenium IDE Firefox plugin. http://seleniumhq.org.

[11] SPECweb 2005. http://www.spec.org/web2005/.

[12] C. Amza, A. Ch, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic Web site benchmarks. In *Proc. of IEEE 5th Annual Workshop on Workload Characterization*, Oct. 2002.

[13] M. Arnold, M. T. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol*, 21(1):2, 2011.

[14] G. Casale, A. Kalbasi, D. Krishnamurthy, and J. Rolia. Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In Proc. of *Middleware*, LNCS 5896, 393–413. Springer, 2009.

[15] G. Casale, E. Z. Zhang, and E. Smirni. KPC-toolbox: Simple yet effective trace fitting using markovian arrival processes. In *QEST*, pages 83–92. IEEE Computer Society, 2008.

[16] E. Cecchet. Performance Benchmarking in Systems Keynote at CFSE 2011 - French Conference on Operating Systems .

[17] E. Cecchet, V. Udayabhanu, T. Wood, P. Shenoy, and P. Shenoy. Benchlab: An open testbed for realistic benchmarking of web applications. 2012.

[18] D. F. García and J. García. TPC-W E-commerce benchmark evaluation. *Computer*, 36(2):42–48, Feb. 2003.

[19] A. Kalbasi, D. Krishnamurthy, J. Rolia, and S. Dawson. DEC: Service demand estimation with confidence. *IEEE Trans. Software Eng*, 38(3):561–578, 2012.

[20] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating service resource consumption from response time measurements. In Proc. of *VALUETOOLS*, ACM, 2009.

[21] D. Krishnamurthy, J. Rolia, and M. Xu. WAM - the weighted average method for predicting the performance of systems with bursts of customer sessions. *IEEE Trans. Software Eng*, 37(5):718–735, 2011.

[22] D. Krishnamurthy, J. A. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Software Eng*, 32(11):868–882, 2006.

[23] D. A. Menascé and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prenctice Hall, 2000.

[24] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In Proc. of *ICAC*, 149–158, ACM, 2009.

[25] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel. Performance impacts of autocorrelated flows in multi-tiered systems. *Perform. Eval*, 64(9-12):1082–1101, 2007.

[26] K. Sachs, S. Kounev, J. Bacon, and A. P. Buchmann. Performance evaluation of message-oriented middleware using the specjms2007 benchmark. *Perform. Eval.*, 66(8):410–434, 2009.

[27] D. Ardagna, B. Panicucci, and M. Passacantando. Generalized Nash Equilibria for the Service Provisioning Problem in Cloud Systems. *IEEE Trans. on Services Computing*, to appear.

[28] G. M. Shroff. *Enterprise Cloud Computing*. CUP, 2010.

[29] J. Wong and R. Howell. *Apache OFBiz Development: The Beginner's Tutorial*. Packt Publishing, October 2008.

[30] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. *ACM SIGOPS Operating Systems Review*, 41(3):31–44, 2007.

[31] OFBiz Technical Production Guide. https://cwiki.apache.org/OFBTECH/apache-ofbiz-technical-production-setup-guide.html .

[32] Giuliano Casale's Homepage. http://www.doc.ic.ac.uk/ gcasale/.