Multi-Tenancy Performance Benchmark for Web Application Platforms

Rouven Krebs, Alexander Wert, and Samuel Kounev

SAP AG, Applied Research,
Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
{rouven.krebs}@sap.com,
Karlsruhe Institute of Technology, IPD,
Kaiserstrasse 12, 76131 Karlsruhe, Germany
{alexander.wert,kounev}@kit.edu

Abstract. Cloud environments reduce data center operating costs through resource sharing and economies of scale. Infrastructure-as-a-Service is one example that leverages virtualization to share infrastructure resources. However, virtualization is often insufficient to provide Software-as-a-Service applications due to the need to replicate the operating system, middleware and application components for each customer. To overcome this problem, multi-tenancy has emerged as an architectural style that allows to share a single Web application instance among multiple independent customers, thereby significantly improving the efficiency of Software-as-a-Service offerings. A number of platforms are available today that support the development and hosting of multi-tenant applications by encapsulating multi-tenancy specific functionality. Although a lack of performance guarantees is one of the major obstacles to the adoption of cloud computing, in general, and multi-tenant applications, in particular, these kinds of applications and platforms have so far not been in the focus of the performance and benchmarking community. In this paper, we present an extended version of an existing and widely accepted application benchmark adding support for multi-tenant platform features. The benchmark is focused on evaluating the maximum throughput and the amount of tenants that can be served by a platform. We present a case study comparing virtualization and multi-tenancy. The results demonstrate the practical usability of the proposed benchmark in evaluating multi-tenant platforms and gives insights that help to decide for one sharing approach.

Keywords: Platform, SaaS, Multi-tenancy, Benchmark

1 Introduction

Cloud Computing enables ubiquitous and convenient on demand access to computing resources over network [2]. Cloud users benefit from the lower costs and increased flexibility, in an efficient and scalable manner, the elimination of an upfront commitment, and payment on a short-term pay per use basis [2]. The

National Institute of Standards and Technology [19] defines three service models for cloud computing. The Infrastructure-as-a-Service (IaaS) model allows to provide and share hardware resources using virtualization technology. The Platform-as-a-Service (PaaS) model allows to deploy and develop applications of different customers within a shared cloud middleware environment. Finally, the Software-as-a-Service (SaaS) model provides hosted applications accessed remotely via the Internet.

Multi-tenancy is an architectural style in SaaS scenarios that enables the sharing a single application instance among multiple independent customers. This style increases efficiency by sharing not only the hardware but also the operating system, the middleware and the application components themselves. The term tenant refers to a group of users sharing the same view on an application. This view includes the data they access, the application configuration, the user management, particular functionalities and related non-functional properties [18]. According to the Gartner's hype cycle from 2011 [24] [21], multi-tenancy is estimated to become mainstream in 2-5 years.

Implementing the functionality to share a single application instance among several tenants is a complex task [11][20] that has to be performed for every developed application. Therefore, the best approach for realizing multi-tenancy is to employ a middleware platform or a PaaS environment that natively supports the development by encapsulating basic functionality such as for example the management and identification of tenants. Google App Engine [10], SAP NetWeaver Application Server and force.com [28] support the developer with predefined interfaces and implicit functionality reducing the development effort for creating a multi-tenant application.

While Cloud Computing provides many advantages as described above, it still fails to provide high availability and response time guarantees required for running mission-critical applications. Various reports [22] [4] indicate that performance is still one of the major obstacles for the adoption of the cloud paradigm. To gain insight into the performance provided by cloud platforms, representative application benchmarks and metrics are needed. Various benchmarks and metrics with focus on cloud environments were developed in the last view years. However, such benchmarks are usually focused on specific aspects of cloud services like persistence or features like infrastructure elasticity.

To the best of our knowledge, no benchmark that explicitly supports the evaluation of multi-tenant platforms exists so far. To fill this gap, in this paper we propose an extended version of an existing benchmark to support multi-tenancy. This benchmark can be used to evaluate the performance of an on premise middleware system or a PaaS environment supporting multi-tenant applications.

The selected case study to evaluate the usability of the benchmark is motivated by our former publication [20]. In this paper, we present an estimation approach to balance the increasing development costs for developing a multi-tenant application (MTA) with the decreasing operating costs resulting from the improvements in resource efficiency. Furthermore, given that, running multiple copies of an application in separate virtual machines (VM), each customized

for a given tenant, is often considered as an alternative to adopting a multi-tenant architecture, we decided to evaluate the two approaches in terms of the performance they provide. Our approach allows to find the point at which multi-tenancy is more efficient with respect to resource utilization in a given application scenario.

In summary, the contribution of this paper is twofold: We present an extended version of an established benchmark to support multi-tenancy. Furthermore, we present a comparison of virtualization and multi-tenancy which helps to estimate the efficiency of the approaches.

The remainder of the paper is structured as follows. In Section 2, we outline important and common design aspects of multi-tenant systems. Section 3 presents our extensions of the TPC-W benchmark based on the outcomes of the previous section. Furthermore, we give an insight into our implementation. Section 4 presents our case study investigating the efficiency of multi-tenant systems compared to virtualization. Section 5, surveys related work and Section 6 concludes the paper.

2 General Design Concerns in Multi-tenant Architectures

To ensure isolation in a multi-tenant application (MTA) one has to make a number of architectural decisions. Furthermore, PaaS scenarios raise some additional requirements concerning the actual implementation. In this section, we give a short overview of the most important architectural aspects related to our work and the impact of multi-tenancy on potential benchmarks and metrics.

2.1 Tenant Identification

When a request arrives at a MTA not only the specific user has to be identified, but also the tenant it belongs to. Various approaches exist to identify the tenant. One solution is to attach the tenant specific information to the user identification. However, this approach requires an authentication of the user and duplicate user names in different tenants are not possible, thus, violating isolation. Another widely used approach (e.g., Google App Engine [10]) is to use the host name as a basis for the tenant identification. In this scenario, various host aliases point to the same application instance/IP address. Thus, a tenant identification is possible without requiring a login and duplicate user names are supported. A common approach to transfer the tenant's identifier along the execution path leverages the thread context to which the relevant information is attached.

2.2 Database

In general, we distinguish three major approaches to separate a tenants data from the data persisted by other tenants (Wang et al. [27] and Chong et al. [6]). The dedicated database system provides a separate dedicated database for each

tenant and has the best isolation at the cost of the highest overhead. In a dedicated table/schema approach, every tenant uses the same database management system, but separate tables or schemas. This scenario enables at least a partial sharing. However, some mutual performance influences between tenants are now possible. The highest degree of sharing, respectively efficiency, is established by sharing the same tables and schemas. To differentiate the data, a column with the tenant id is added to each table. This approach also has the largest consequences on the application or platform. An application has to take care of the tenant id in every database statement. If the platform provides an abstraction of the database, it might handle the additional tenant id in a transparent way (e.g., EclipseLink [8]).

2.3 Tenant Meta-Data

Koziolek [17] presents a high level architecture for MTAs based on observations he made about existing offerings. In general, this architecture reflects a Web Application Architecture with an additional meta data storage for the tenant specific information (e.g., customization, database id, tenant name, SLAs). Another element is the meta-data manager which enables access to the meta-data and adjusts the application according to the information stored in the meta-data. The variability of information stored in the meta data is high. However, we can assume that at least an id for the tenant, a display name for the tenant and a database identifier is available. Depending on the employed data management approach, the latter may refer to an tenant specific id or database connection. Platforms with multi-tenancy support usually provide access to the tenant meta-data.

2.4 Security

Normally either the implemented persistency APIs of the platforms or the application developer has to ensure the separation of data by using SQL statements with tenant id as aforementioned. In addition, tenant specific caches might be required. The identification of a tenant might base on an identity management system as part of the meta-data manager. However, the access to the application might allow to run attacks against other tenants with extended privileges. This has to be reflected by special measures like SQL encoding and stack overflow prevention.

2.5 Metrics for a Multi-tenant Benchmark

In traditional benchmarks, usually one or several performance metrics are observed in relation to the amount of simulated users, the request rate, and sometimes price (e.g., [1]). Based on this information, a quality metric of the system is derived.

In a multi-tenant system, we can also incorporate the amount of tenants, for example, considering the throughput and response time in relation to the amount

of tenants. This metric might be of interest when the per tenant overhead and the total amount of tenants a platform could serve is relevant. Furthermore, it might answer the question about the optimal amount of tenants for one application server. Another metric might define a fixed number of tenants by observing the QoS based on the amount of users for each.

It is worth to mention that real applications serve tenants with different amounts of users and various database sizes and consequently various resource demands. If these factors are known for the scenario under investigation the benchmark might reflect this.

2.6 PaaS Persistence

We consider traditional middleware and PaaS environments with multi-tenancy support. Existing PaaS environments provide an application runtime container and various embedded services accessed via an API (e.g., Google App Engine, SAP NetWeaver Cloud). Persistence services are of major importance. Existing offerings provide SQL or key value stores. However, in the majority of cases, the access to the storage is only permitted within the application runtime container. Even in cases where a user interface to manipulate individual data records exists, it is normally impossible to directly load high amounts of data.

3 Multi-tenant Benchmark

In this section, we present our extensions of the TPC-W benchmark [1] based on the implementation provided in [5] and focus on our modifications for cloud environments with multi-tenancy support. TPC-W was already used successfully in the field of multi-tenancy [26] and already satisfies some of the requirements for a cloud benchmark [3] .

3.1 TPC-W

The Transaction Processing Performance Council (TPC) developed a transactional Web e-commerce benchmark (TPC-W) [1]. Its focus is on business oriented transactional Web servers. The workload models an Internet commerce environment emulating an online bookshop. The benchmark emulates multiple on-line browser sessions by accessing dynamically generated Web pages. The benchmark provides three workload profiles that differ in their the browse-to-buy request ratio resulting in different proportions of database reads or inserts/updates: primarily shopping, browsing and Web-based ordering. The load can be varied by the amount of emulated browsers (EB) sending requests to the system. One EB corresponds to one user calling various Web transactions in a closed workload. To ensure portability, TPC does not require the use of a specific implementation. Instead a detailed specification of the functionality that must be provided by an implementation is published.

3.2 Multi-tenant TPC-W Specification

We extended the specification of TPC-W in several points to cover the relevant conceptual aspects of multi-tenant systems described in Section 2. The PaaS persistence related concerns (cf. Section 2.6) do not directly relate to the specification of the benchmark and will be discussed in Section 3.3.

The Tenant Meta Data Manager (cf. Section 2.3) provided by a platform is used to render the tenant's display name as part of various Web pages (Home Page, Customer Registration Page, Buy Confirm Page). In one the pages (Buy Confirm) the tenant's identifier is also rendered.

For environments with a native connection to one schema on one database server, a `tenantId` column is added to every table (cf. Section 2.2). Consequently, the primary key has to be a combination of the `tenantId` and the entity specific id field. In addition to the TPC-W standard the `tenantId`, retrived from the meta-data manager, is added to every SQL request from the application to ensure data isolation and thus privacy of the data. In addition, we recommend to encode all SQL parameters for security reasons. TPC-W does not specify application internal caches; thus, we do not have to provide a tenant specific access mechanism.

Several database management systems do not support the auto generation of combined primary keys. Thus, an application based key generation mechanism is applied to generate the primary keys. To ensure portability, we specify the usage of a key-value table with segment support to reduce overhead. This solution consists of a database table which provides a key counter for each table and each tenant. To avoid overhead, the key-value table is accessed via an application local cache. This cache increases the counter by a count of 1000 and thus it could return 1000 ids before the next update of the key table. It has to be ensured that increasing the database key counter by several application instances does not result in unresolved conflicts. This key counter mechanism is used to generate the primary keys. It is worth mentioning, that the `tenantId` part of the key must not be generated, as this is a value derived by the request that triggered the database update.

For environments with a native connection to various SQL servers or schemas for each tenant the auto generation for the keys can be reused and the additional column for the tenant id becomes obsolete. In such situations we assume that the database connection/schema is either provided in a transparent way by the platform or is stored in an application specific configuration where it is mapped to the tenant. In the latter case, for every SQL request, the appropriate connection must be selected based on the tenants id returned by the tenant meta-data manager.

For environments with an API based access to the persistence layer, where the data isolation aspect is transparent to the application the above methods might be irrelevant. However, if the data isolation aspect is not transparent the described solutions have to be considered.

The load driver has to support the platform specific tenant identification mechanism (cf. Section2.1). As every tenant accesses the same application, we assume similar workload profiles.

The relevant metrics and the exact setup concerning the number of users for each tenant depends on the goals of the benchmarking scenario. For our case study, we defined static workload profiles for each tenant with an increasing amount of tenants.

3.3 Implementation

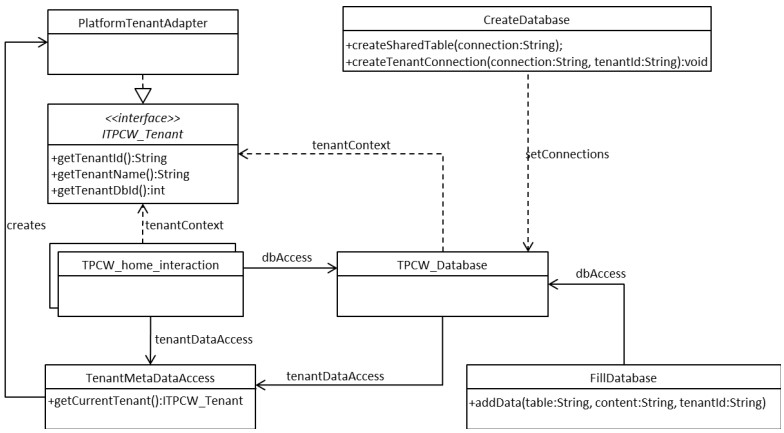


Fig. 1: Overview of the Multi-Tenant TPC-W Benchmark.

The basis of our version [5] provides a Java Servlet based application that accesses the database with the help of one central class using a JDBC connection. Figure 1 shows an overview of the elements used in our version of the TPC-W benchmark. In the following, we briefly describe the functionality of the various elements and how they are related to each other.

TPCW_home_interaction is one example of 14 servlets used in our implementation. The servlets render the html pages and implement the expected work flow. Every servlet has a reference to the *TenantMetaDataAccess* and uses an implementation of the interface *ITPCW_Tenant* to access the meta-data for the tenant that owns the current thread.

TPCW_Database implements the JDBC-based communication with the database, which is implemented as described in Section 3.1. It also encapsulates the key generator.

The *TenantMetaDataAccess* class implements the access to the platform's tenant meta-data. It hides the platform specific implementation for accessing information about the tenants. Thus, it is possible to port the implementation to

another platform by changing the implementation of this class. The *TenantMetaDataAccess* provides a platform specific implementation of the *ITPCW_Tenant* interface.

ITPCW_Tenant defines the interface that represents a concrete tenant encapsulating the communication with the meta-data manager to provide tenant specific information.

CreateDatabase extends *HttpServlet* and is a proxy to create the required schema in the platform environment when no direct access is available. The method *createSharedTable* creates a shared schema in the database. Method *createTenantConnection* creates a schema without tenant id for each tenant. The corresponding connection and type of database multi-tenancy is then set in the *TPCW_Database*. Thus, using *createTenantConnection* enables separate schema and separate databases to be used. If the platform provides the tenant specific connections in a transparent way, one has to modify *TPCW_Database*.

FillDatabase is a proxy extending *HttpServlet* to initialize the databases data for the benchmark run using *TPCW_Database*.

The *Load Driver* is provided in [5]. The target platform in our case differentiates tenants by the host name. Therefore, we created one instance of the load driver for each tenant with a tenant specific hostname as a target.

4 Case Study

In this section, we apply our extended version of the TPC-W benchmark in a case study demonstrating its use for performance evaluation. In addition, we present a comparison of virtualization and multi-tenancy which helps to estimate the efficiency of the approaches.

4.1 Goals

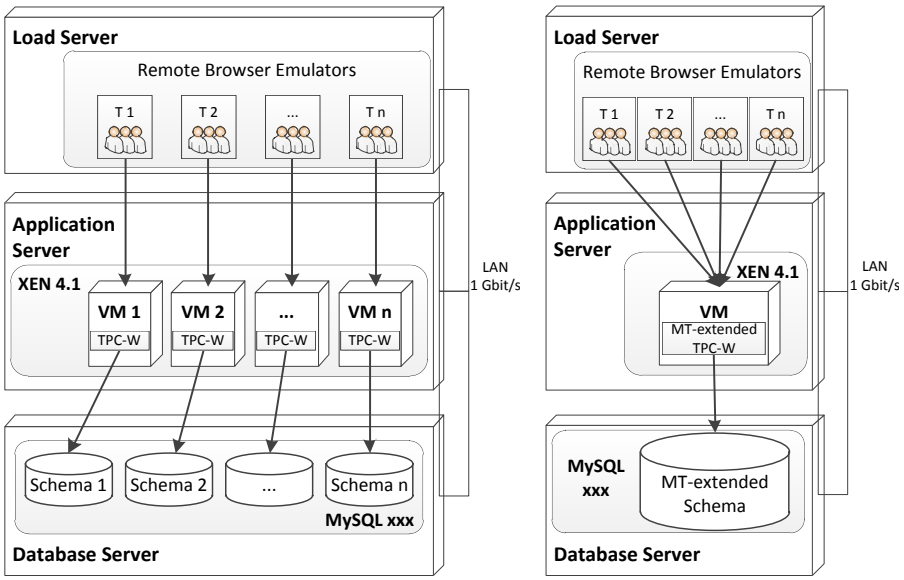
The main goal of the presented case study is to compare an application-based multi-tenancy approach with a pure virtualization-based approach in terms of performance. In particular, we investigate the following main question: *Given a certain setup, under which conditions is an application-based multi-tenancy approach more efficient than a virtualization-based approach, and vice versa?*

In order to address this question, we investigate the following research questions for each of the two scenarios:

- **RQ1:** What is the maximum throughput that can be achieved with the corresponding sharing approach depending on the tenants-size.
- **RQ2:** Under which relationship between the tenant size and number of tenants is a multi-tenant architecture more efficient?

4.2 Experimental Setup

In order to address the research questions mentioned above, we perform a series of experiments.



(a) Setup for Virtualization-Based Sharing Approach

(b) Setup for Application-Based Multi-Tenancy Approach

Fig. 2: Experimental Setup

Figure 2 shows the experimental setups for the virtualization-based scenario (Figure 2a) and the multi-tenancy scenario (Figure 2a). In both cases, the experimental setup comprises three physical servers: The *Load Server* is used for user emulation, the *Application Server* hosts the application logic part of the benchmark and the *Database Server* serves as the persistence layer. These three physical machines have the same characteristics. In particular, each of them has a processing power of 16 x 2,13 GHz, a memory capacity of 16 GB, and SUSE Enterprise 11 as operating system. The machines are connected by a 1 Gbit/s LAN. For our experiments, we assume there are `n` equal-sized tenants `T1...Tn` each comprising `m` users (cf. Section 3.2) emulated by means of Remote Browser Emulators (RBE) (cf. [1]) running on the Load Server. The browsing workload mix defined by [1] is used to generate load.

In the following, we explain the differences in the two scenarios.

Scenario I: Virtualization-Based Approach In this scenario, the customer contexts are separated by means of separate VMs and separate database schemata (cf. Figure 2a). Thus, for each customer context the Application Server hosts a VM on top of a common XEN 4.1 hypervisor. Each VM is running a separate application instance of TPC-W within an SAP-specific customized version of Apache Tomcat. Given that TPC-W is an I/O-intensive application, compared to the Database Server, the CPU consumption on the Application Server is rela-

tively small. Thus, given that the focus of our comparison is on the Application Server tier, it is reasonable to pin the cores of all VM to one physical core to avoid the database from being the bottleneck. The available memory capacity is equally distributed among the VM and the host operating system. Similarly to the application layer, the separation on the persistence layer is realized by means of a separate database schemata. Thus, each TPC-W instance uses its own, dedicated database schema. However, all database schemata are hosted within a common MySQL 5.1 process executed on the Database Server.

Scenario II: Multi-Tenancy Approach In the multi-tenant scenario, the tenants are separated by the notion of separate tenant contexts at the application layer and an extended database schema which allows for accessing tenant-specific data. Correspondingly, the experimental setup for Scenario II comprises only one VM and only one database schema (cf. Figure 2b). The single VM hosts the multi-tenant version of TPC-W (cf. Section 3.2) deployed on the extended Apache Tomcat. The Tomcat instance provides a Tenant Meta Data Manager (cf. Section 2.1). Based on the tenant-specific meta-data, the benchmark accesses the extended database schema. Similarly to the setup of Scenario I, the virtual processing unit of the single VM is pinned to a single physical CPU core.

Testing Methodology We performed 10 experiment series in total, five for each scenario. For every series, the size of each tenant was fixed to 250, 500, 750, 1000 or 1500 users. We are interested in the maximum throughput of the system. Thus, we started each experiment series with one active tenant and increased the amount of tenants stepwise until the application started to throw time out exceptions. To ensure equal conditions the databases were newly created, and filled with data before every run. Afterwards, the database management system and the VMs were restarted prior to starting the load driver. In the multi-tenancy scenario we restarted the VM as well. The warm-up phase was set to 10 minutes and the measurement period was 30 minutes.

4.3 Results

In this section we present the results of our measurements. Figure 3 presents a general overview of the most important data gathered, whereas the specific research questions **RQ1** and **RQ2** are addressed by Figure 4a and Figure 4b. The confidence intervals in all measurements were negligibly small and are thus omitted for compactness.

In Figure 3, the number of tenants is shown on the x-axis and the throughput in transactions/second on the y-axis. The various curves represent measurements with the multi-tenancy and virtualization-based sharing approach for various amounts of users per tenant. In general the CPU utilization became a bottleneck and prevented the system to achieve higher throughputs. In the virtualization scenario with a tenant size of 250 users, the amount of guest domains was limited to 12 due to a lack of memory which resulted in memory exceptions when the

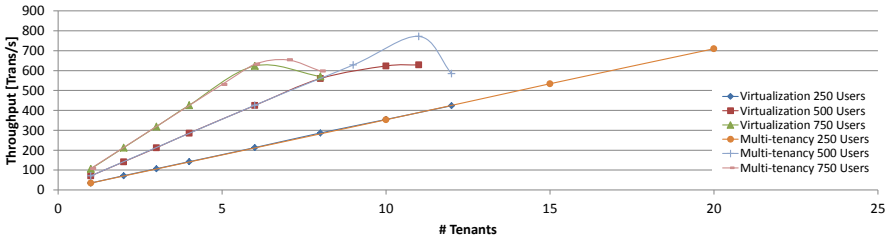
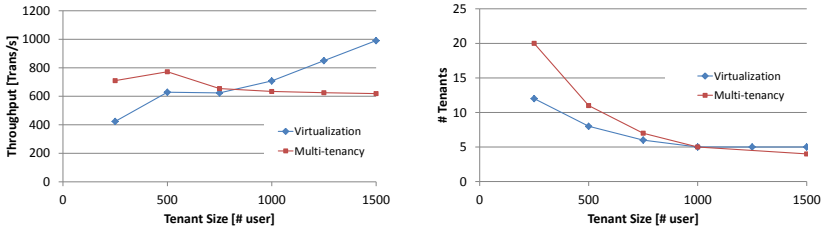


Fig. 3: Throughput Dependent on the Amount of Tenants.



(a) Maximum Achievable Throughput (b) Paredo Optimal Configurations

Fig. 4: Maximum Throughput and Paredo Based Decision Support.

server was lunched. Although the systems CPU was underutilized with only 12 domains. The measurement with 250 users and multi-tenancy were stopped at 20 tenants due to time limitations for further experiments and the already existing data to answer our research questions. We assume unused potential concerning the amount of tenants because of very low response times and a CPU utilization of around 70%. The figure also shows that the advantage of multi-tenancy is less for the 500 users scenario and even lower for 750 users. In similar measurements for 1000 and 1500, users we observed lower maximum throughputs in the multi-tenancy case.

Figure 4a focuses on **RQ1**. The maximum overall throughput of all tenants is shown on the y-axis. The x-axis presents the number of users for each tenant. By increasing the number of tenants for each tenant size, the maximum throughput was determined. The maximum throughput decreases with lower values for the tenant size in the case of virtualization whereas in the case of multi-tenancy throughput remains stable.

Figure 4b shows the tenant size on the x-axis and the amount of tenants at which the maximum throughput was achieved on the y-axis. Thus it presents paredo optimal configurations in terms of the maximum throughput for the virtualization and multi-tenancy scenario. It shows, that multi-tenancy is less efficient in situations with more than 1000 users as there the amount of served tenants and the throughput is below the capabilities of virtualization. In the range between 250 and 1000 users per tenant, virtualization is a usable model,

for the given hardware configuration, if the amount of tenants to be served is below the curve for virtualization. Nevertheless, multi-tenancy is able to serve more tenants with a higher total throughput by using the same hardware in these boundaries. The benefits of multi-tenancy becomes more significant in scenarios with 250 users or less. At these, the total throughput for virtualization was not longer limited by the CPU, instead the memory become the bottleneck. Multi-tenancy uses memory resources very efficiently as it avoids to allocate static memory for the application, application server and OS. Consequently multi-tenancy can still achieve a high throughput and good utilization of the CPU in these cases. Thus, this figure addresses **RQ2**.

Especially for stateless web applications with low memory demands the CPU is the primary bottleneck, beside I/O which is not subject of this discussion. Based on our results we can conclude that virtualization produces additional overhead on the CPU with an increasing amount of VMs hosted on one server, thus the throughput was limited. Nevertheless, these drawbacks are widely negligibly. The most important observation is the inefficient usage of the memory when virtualization is used to serve one application for several customers. Consequently, the primary factor for selecting one of the solutions should be the memory. If an application requests a high amount of memory the overheads for the OS and application server may also become less important. Especially in stateless applications with small memory demands multi-tenancy outperforms virtualization for small tenants as here the static memory allocations of the runtime environment become the limiting factor. Furthermore, multi-tenancy allows to over commit memory, which is not possible using Xen.

5 Related Work

Performance is of major interest in cloud computing [2] [4]. Conventional middleware benchmarks for classical platforms (e.g., SPECjEnterprise [25]) do not support essential cloud features like multi-tenancy. Therefore, several new benchmarks have emerged in the last years to support the performance evaluation of cloud platforms. Most activities focus IaaS and cloud-specific features like elasticity [14]. Others focus on cloud specific services like persistence [7] [13].

Virtualization enables sharing at the infrastructure level. Thus, it is a key enabler for IaaS clouds and it has been widely used over the past years in data centers. A number of benchmarks have been developed in the past years for evaluating virtualization platforms.

One example is VMmark [12], a benchmark developed by VMware. VMmark defines a tile as a set of VMs serving different predefined applications (e.g., SPECweb2005). The benchmark score is based on a normalized overall throughput of the applications as a function of the amount of deployed tiles. The total throughput increases as long as the system is not saturated. As part of the benchmark results VMware publishes the maximum throughput and the number of tiles. This approach is similar to our approach, where we consider the overall system throughput depending on the amount of tenants.

Binnig et al. [3] discuss characteristics of cloud services and derive a list of requirements for a cloud benchmark. Afterwards, they analyze the existing TPC-W benchmark, discuss why the TPC-W benchmark satisfies requirements for cloud benchmarking and discuss some initial ideas for a new benchmark that overcomes some shortcomings of the TPC-W benchmark. Major shortcomings reported are the requirement of ACID properties for data operations and invalid metrics for adaptable and scalable systems in terms of elasticity. However, we observe a trend in PaaS environments to support the ACID properties (e.g., SAP NetWeaver Cloud[23]) for complex Web applications. Furthermore, a PaaS provider or customer usually has the opportunity to control the elasticity mechanisms as required for a performance test. Finally, our focus is on multi-tenancy features that were not considered in [3].

The authors of [26] present a method for resource demand estimation on a per tenants base. Furthermore, they provide a mechanism to ensure performance isolation. For the evaluation, they used an implementation of TPC-W. However, they did not report any extensions for data isolation nor any usage of platform provided multi-tenancy services.

MulTe [15] is a framework that helps building and running existing database benchmarks to evaluate various performance metrics of multi-tenant database management systems. However, our definition of multi-tenancy assumes a shared application instance, as opposed to merely a shared DBMS used by several applications. Therefore, MulTe goals defer from our own.

Regarding the tradeoff decisions several papers present approaches to increase the efficiency of multi-tenant systems (e.g., [29], [9]). However, they do not help to come to a tradeoff decision for various resource sharing approaches.

In [27], various sharing options for implementing multi-tenant persistence are discussed. The authors evaluate their non-functional behavior including performance aspects. Given that our focus is on the application tier the database was not a bottleneck in our scenario.

6 Conclusion

Performance concerns are one of the major obstacles for potential cloud customers. We analyzed the most important concepts of multi-tenant applications and identified features provided by platforms to support multi-tenancy. To support the performance engineering process this paper proposes an extension of the TPC-W benchmark for platforms that support the identified multi-tenancy features. This includes various multi-tenant persistence models, tenant identification mechanisms and access to tenant specific meta-data. We evaluated the usability of the proposed benchmark in a case study where the maximum throughput of a multi-tenancy supporting platform based on the amount of tenants and users per tenant was evaluated. Furthermore, we leveraged the benchmark to compare a virtualization based with a multi-tenancy based sharing approach.

Multi-tenancy shows only a moderate benefit as long as additional virtual machines can be started to handle new tenants. Once the lack of memory start

limiting the capability to lunch further virtual machines, multi-tenancy shows significant advantages as it still serves and increasing amount of tenants with good performance. Overall, multi-tenancy exhibits significantly higher efficiency for a high amount of tenants with low usage, because it avoids a high static memory allocation. In our case study, we observed that memory was the primary limitation of virtualization. As long as CPU is the bottleneck the advantages of MTAs are less.

In our future research, we will leverage this benchmark for the evaluation of performance isolation between different tenants. Furthermore, we are interested in the efficiency of mutual utilized resources when load profiles underlie fluctuations and the impact of various load profiles for different tenants.

7 Acknowledgements

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement № 258862 and was supported by the German Research Foundation (DFG), grant RE 1674/6-1 (Transfer project KIT-SAP).

References

1. TPC BENCHMARK W, 2002. Transaction Processing Performance Council.
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.
3. Binnig, C., Kossmann, D., Kraska, T., and Loesing, S. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems* (2009).
4. bitcurrent. Bitcurrent cloud computing survey 2011. Tech. rep., bitcurrent, 2011.
5. Cain, H. W., Rajwar, R., Marden, M., and Lipasti, M. H. An architectural evaluation of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture* (2001).
6. Chong, F., Carraro, G., and Wolter, R. Multi-tenant data architecture. website, June 2006.
7. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10.
8. Eclipse Foundation. Eclipselink/development/indigo/multi-tenancy. website, Oct 2012.
9. Fehling, C., Leymann, F., and Mietzner, R. A framework for optimized distribution of tenants in cloud applications. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (2010).
10. google. Google Cloud Platform, Nov 2012. https://cloud.google.com/index.

11. Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. A framework for native multi-tenancy application development and management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007* (2007).
12. Herndon, B., Smith, P., Roderick, L., Zamost, E., Anderson, J., Makhija, V., Herndon, B., Smith, P., Zamost, E., and Anderson, J. Vmmark: A scalable benchmark for virtualized systems. Tech. rep., VMware, 2006.
13. Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshops* (2010).
14. Islam, S., Lee, K., Fekete, A., and Liu, A. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering* (New York, NY, USA, 2012).
15. Kiefer, T., Schlegel, B., and Lehner, W. Multe: A multi-tenancy database benchmark framework. In *TPC Technology Conference, TPCTC 2012* (2012).
16. Koziolek, H. Towards an architectural style for multi-tenant software applications. In *Proc. Software Engineering (SE'10)* (February 2010), vol. 159 of *LNI*.
17. Koziolek, H. The sposad architectural style for multi-tenant software applications. In *Proc. 9th Working IEEE/IFIP Conf. on Software Architecture (WICSA'11), Workshop on Architecting Cloud Computing Applications and Systems* (July 2011).
18. Krebs, R., Momm, C., and Kounev, S. Architectural Concerns in Multi-Tenant SaaS Applications. In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science, CLOSER 2012)*.
19. Mell, P., and Grance, T. The NIST definition of cloud computing. digital, 2011.
20. Momm, C., and Krebs, R. A Qualitative Discussion of Different Approaches for Implementing Multi-Tenant SaaS Offerings. In *Proceedings of Software Engineering 2011 (SE2011), Workshop(ESoSyM-2011)* (2011).
21. Natis, Y. Gartner reference model for elasticity and multitenancy. Gartner report, Gartner, June 2012.
22. Packman, E., Taylor, P., Rachitsky, L., Rejali, S., Power, S., Rae, I., and Koffler, D. Bitcurrent: Cloud comuting performance. Tech. rep., bitcurrent, bitcurrent, June 2010.
23. SAP AG. SAP NetWeaver Cloud, Nov 2012. https://netweaver.ondemand.com.
24. Smith, D. Hype cycle for cloud computing, 2011. Tech. rep., Gartner, July 2011. ID Number: G00214915.
25. spec. Specjenterprise2010, Nov 2012. http://www.spec.org/jEnterprise2010/.
26. Wang, W., Huang, X., Qin, X., Zhang, W., Wei, J., and Zhong, H. Application-level cpu consumption estimation: Towards performance isolation of multi-tenancy web applications. In *IEEE CLOUD* (2012).
27. Wang, Z. H., Guo, C. J., Gao, B., Sun, W., Zhang, Z., and An, W. H. A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *e-Business Engineering, 2008. ICEBE '08. IEEE International Conference on*.
28. Weissman, C. D., and Bobrowski, S. The design of the force.com multitenant Internet application development platform. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, ACM.
29. Zhang, Y., Wang, Z., Gao, B., Guo, C., Sun, W., and Li, X. An effective heuristic for on-line tenant placement problem in saas. *Web Services, IEEE International Conference on 0* (2010).