Let Latency Guide You: Towards Characterization of Cloud Application Performance

Hamed Saljooghinejad, Felix Cuadrado, Steve Uhlig
School of Electronic Engineering and Computer Science
Queen Mary University of London

***Abstract*—Public cloud infrastructures provide flexible hosting for web application providers, but the rented virtual machines (VMs) often offer unpredictable performance to the deployed applications. Understanding cloud performance is challenging for application providers, as clouds provide limited information that would help them have expectations about their application performance. In this paper we present a technique to measure the performance of cloud applications, based on observations of the application latency. We treat the cloud application as a black box, making no assumption about the underlying platform. From our measurements, we can observe the varying performance provided by the different VM profiles across well-known commercial cloud platforms. We also identify a trade-off between the responsiveness and the load of the measured servers, which can help application providers in their deployment and provisioning.**

I. Introduction

Emerging new technologies such as augmented-reality devices (e.g., Google Glass prototype) demand a higher degree of responsiveness from applications. This phenomenon is not new; response times are proven to have a direct effect on business revenue. Amazon found that every 100ms of latency costs 1% in sales [1], and eBay and Google have reported similar findings about the impact of user latency in revenue [2], [3], [4]. These companies deploy their services on their own infrastructure. Software is adapted to the execution platform, to compensate the effect of the long tail of latency [5] in their services.

Many start-ups and medium size application providers deploy their applications on a public cloud infrastructure. Cloud Service-Level Agreements (SLAs) provide guarantees on infrastructure reliability and availability. However, they contain no performance-related clauses [6], and VM instance types hide the underlying hardware details. As a result, providers don't know the expected performance of their application when deployed on the cloud infrastructure.

This leads to two practical questions about Cloud application performance in terms of throughput and latency: 1) how well will an application perform once it runs in production-mode on a specific Cloud infrastructure? 2) how will the application perform under different workloads?

To tackle the above questions, we have designed a black-box approach with no assumption about VM underlying. It provides a way to examine and estimate the application performance in terms throughput. Therefore, Our goal is to develop a framework that is used to explore the performance space under certain latency SLA. Being black-box adds simplicity as there is no need to deploy extra software and manage configuration at server side.

We propose a methodology that estimates the maximum load an application server can sustain at which the latency of the responses increase to an undesirable level. Our methodology samples the latency values at various workloads and observe the trade-off between application performance and server responsiveness. We evaluate our methodology and the corresponding tool using a data-store application deployed in three well-known public cloud providers, namely Amazon EC2 [7], Microsoft Azure [8] and Google Compute Engine [9].

Our contributions are as follows:

- We use the statistical properties of latency to detect the fine-grained behavior of an application server under load.
- We present a black-box methodology that estimates the workload a Cloud application can sustain for a given latency SLA.
- We identify trade-off between the throughput and latency of application servers and also evaluate performance for different types of workloads, which can help application providers in their deployment and provisioning decisions.

II. Methodology

The main goal of our methodology is to observe the trade-off between application performance (in terms of operations per second) and server responsiveness (as observed through request/response latency). We observe the *server response time* across various workloads by remotely measuring the end-to-end latency.

A. Approach

We rely on the measured end-to-end latency of individual request/response pairs. This end-to-end latency includes the network latency, as well as the client and server side processing latencies. We generate controlled amounts of requests/responses at a constant rate, while monitoring the corresponding end-to-end latency values. The rationale is the following: Application servers are typically designed to absorb a given workload of requests. Hence, when the workload is below a given threshold, we expect the server latency to be low and relatively constant over time. When the workload increases beyond what the server is able to absorb on the other hand, we expect the end-to-end latency to increase, at least statistically, i.e., some responses will take more time to be processed by the server. Our methodology is designed to

probe an application server in order to identify the workloads at which the corresponding latency increases statistically, which we call *transition state*[1].

Our server probing algorithm sends requests to the server, at varying request rates (called ***test throughput*** and denoted by T_{test}). At the same time, we timestamp requests and responses to obtain the end-to-end ***observed latency***. As only statistically significant variations in the observed latency are relevant, each value of the T_{test} is probed for a few seconds. This period of time during which the T_{test} is constant is called a ***sample window***. The reason for selecting a time window is to avoid wrongly inferring that the server is overloaded as the result of artefacts in the measurements, e.g., if the network path latency increases for a very short period of time without the server being actually overloaded.

Roughly speaking, our strategy is to go through increasing workloads, until we reach one for which the server shows signs of being overloaded, observed through increases in end-to-end latency.

Our design is separated into two phases: 1)***Fast ramp-up***: increasing exponentially the T_{test} until the observed latency shows signs that the server is getting overloaded; 2) ***Fine-tuning***: roll back and converge slowly towards a value of the T_{test} in the transition state where the observed latency meets the user defined latency threshold.

B. Design

In both phases a given T_{test} is generated, and the observed latency is measured. A pseudocode overview of our strategy is depicted in Algorithm 1. During the ***Fast ramp-up*** phase, the algorithm searches for a value of T_{test} at which the latency starts increasing. This phase starts with an initial T_{test}, with subsequent sample windows exponentially increasing the T_{test} as long as the current window is marked as "accepted".

In this paper, we are interested in the server-side latency only. However, the network delay component of the end-to-end latency might inflate the end-to-end latency and mislead us to believe that it is a server-side problem. To mitigate such occurrences, we also measure the network latency (RTT) using tcpping [10]. If the measured network latency during a sample window increases significantly compared to the baseline, we discard the measurements from the corresponding sample window and start a new measurements of the sample window.

If the median end-to-end latency is close enough to the measured RTT, meaning that the server-side latency is very small, then we assume that the server is able to handle the current T_{test}. For each second of the sample window, we check that the median end-to-end latency is within a fraction of the measured RTT, in which case this second will be tagged as "Pass". If more than half the seconds of the window are flagged as "Pass", the sample window is marked as "accepted" and the next T_{test} to be probed within the next sample window will be increased.

On the other hand, if a sample window is not marked as "accepted", it means that we have entered the transition state, ***Fine-tuning*** phase will be started. *Fine-tuning* phase of the algorithm tries to refine the value of the T_{test} to get close to the true value of T_{test} with which the server gets overloaded. In this phase, we increase the T_{test} linearly instead of exponentially, starting from the highest T_{test} that did not show signs of latency increase, i.e., the T_{test} from the previously accepted sample window. In this phase, the increase in T_{test} between consecutive sample windows is a fixed rate of the gap between the last two T_{test} during the ramp-up phase. The exact rate used will define the degree of fine-tuning of this phase of the algorithm.

[1]The rationale for this terminology is that the application server behavior changes quantitatively during the transition state.

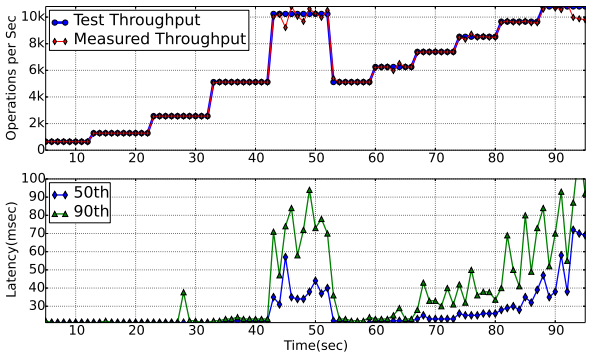

Fig. 1: Time series of a run of the methodology on a Cassandra server deployed in the GCE.

At the end of the fine-tuning phase, we will have sampled different T_{test} values, the last one being the one that our algorithm stops at, which we call $T_{estimate}$. As stopping criterion for the fine-tuning phase, we decided to rely on a specific percentile and latency threshold of the response time. The reason for this lies in the common use of response time percentiles for SLA's [11].

If the nth percentile response time of a T_{test} crosses the user defined *SLA latency threshold* during the fine-tuning phase, the algorithm terminates and the T_{test} from the previously accepted sample window is returned as $T_{estimate}$. We use $T_{estimate}$ in Section III to evaluate application performance on various cloud platforms. The specific SLA settings (percentile and latency threshold) are configurable parameters of our technique, as different applications require various levels of responsiveness. In our experiments, we use as SLA latency the 90th percentile and threshold of 150ms.

C. Example

To illustrate our methodology, we present in Figure 1 the results from a single run of the algorithm using a server located in the Google Compute Engine cloud platform [9]. The top plot of Figure 1 shows the values of the T_{test} (in operations per second), as well as the measured throughput (from the client side).

We observe that the fast ramp-up phase ends at second 52, followed by the fine-tuning phase. We can also see on the top plot of Figure 1 the exponential increase in the values of the T_{test} during the fast ramp-up phase, and the linear increase during the fine-tuning phase.

Algorithm 1: Pseudo-code of the methodology.

```
Phase = FastRampUp;
while TRUE do
    SampleWindow = probeServer(T_test);
    if RTTChanges(SampleWindow) then
        continue;
    else
        switch Phase do
            case (FastRampUp)
                if accept?(SampleWindow) then
                    T_test=T_test * 2;
                else  /* Overloaded_server */
                    Phase = FineTuning
                    T_test = T_test/2;
                    inc = rate * (T_test/2);
            case (FineTuning)
                if accept?(SampleWindow, SLA) then
                    T_test = T_test + inc;
                else  /* Found value of T_test */
                    return T_test − inc;
```

The lower plot of Figure 1 shows the 50th and 90th percentiles of the end-to-end latency over time, from which the RTT was subtracted. We observe that the latency percentile values are in the same range as the RTT during the fast ramp-up phase, until the last sample window of the phase (between seconds 42 and 52). The second part of the algorithm (fine-tuning phase), between seconds 52 and 95, shows that the transition happens at throughput values between 5k and 10k operations per second. We also observe, as expected, that the lower percentiles of the latency are less sensitive to the changes of the T_{test}.

D. Discussion

Because our goal was initially benchmarking, we designed our methodology toward estimating the throughput that a server can absorb without significantly increasing the end-to-end latency. Therefore, the final look of our algorithm is strongly dependent on our initial goal. During the design of the algorithm, we had to choose the ramp-up policies in each phase, and these policies affects the granularity of the T_{test}. We have selected those policies with the goal of going fast enough through the T_{test} values, to find the one at which the latency starts increasing. The exponentially increasing fast ramp-up phase intends to find a rough approximation of the throughput range of the phase transition, while the linearly increasing *fine-tuning* phase narrows down to a finer throughput region around the phase transition. While other ways are possible, we believe that our current design leads to a reasonable compromise between speed and accuracy of the T_{test} estimation.
As already explained, we measure the RTTs and compare them with the end-to-end latency to decide about the increase of the next T_{test}. Moreover, Tcpping values that are captured at the end of each sample window are monitored and verified that RTTs are not affected with low and high volume of load. Figure 2 shows cumulative distribution of tcpping values when

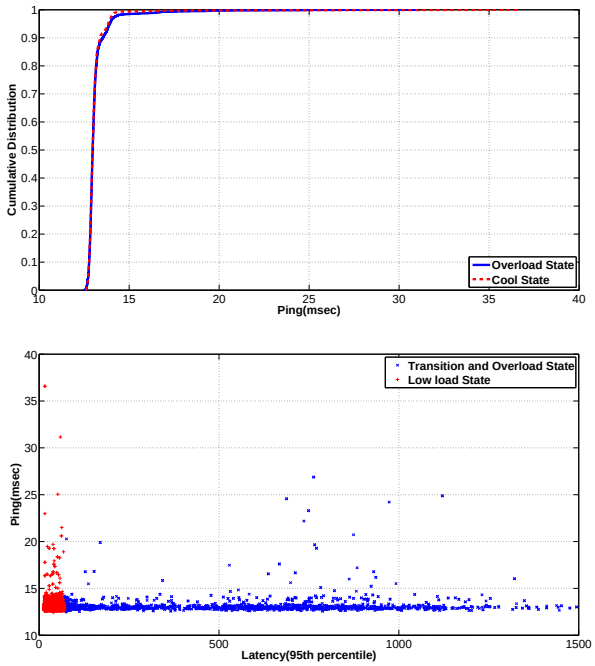

Fig. 2: Top)CDF of tcpping values at two state of high and low load bottom)Scatter plot of tcpping values Vs 90th percentile end-to-end latency repeated every hour for about 24 hours in medium size instance in EC2.

we ran our benchmark of about 24 hours at both low and high load traffics. It shows that in both cases the variation is low and RTTs has not been affected by high load. We saw same results on a longer experiment for about one week.

It is possible that the RTTs spike during the measurements, e.g., due to routing instabilities or congestion on the path. Figure 2 shows the scatter plot of the tcpping values of same previous measurements. Very few (1.45% out of total) high values of RTT spikes (more than 15msec) are observed but our methodology has been designed to be robust not to be affected by this issue. Indeed, if a statistically significant change of the measured RTTs across consecutive sample windows is observed, we discard the measurements from the suspicious window and redo the same T_{test}. Shorter and more bursty network latency changes are discarded implicitly thanks to the sample window. Smarter approaches are possible, e.g., by trying to identify the exact time when the change in RTT happened and removing the corresponding bias in the end-to-end latency. However, given that many routing events are transient in nature and likely affect the RTTs for relatively short periods of time [12], we believe that for our current purpose, the added complexity is not worth the effort.

In the next sections we pick a data-store application and first, we benchmark that when it is deployed on various cloud platforms and multiple types of VMs. Second, we show the observed trade-off between server responsivity and workload.

III. Exposing VM Performance

We use our methodology and test a data-store application using different VMs and estimate its performance in terms of throughput. The methodology that is described in Section II is implemented as a plugin[2] for Apache Jmeter [13] (tested with v2.11), a tool for load test and performance testing of distributed applications. In our experiments, we tested Apache Cassandra deployments [14] using the CassJmeter [15] 0.2 plug-in for JMeter integration. [16] was also another option to implement our methodology on top of it but due to load instability we preferred Jmeter. Apache Cassandra is a distributed key-value storage system for managing large amounts of data potentially partitioned and replicated across multiple servers. This type of application provides on-line read/write access to data in web. Usually when a web user is waiting for a web page to load, reads and writes to the database are carried out as part of the page construction and delivery. Testing more types of applications is subject of our future work. In all our experiments we consider the simplest operation to be run on each node. So, we perform read operation for a specific row in database and test it on different commercial and non-commercial platform ranging from PlanetLab to well known public Cloud players.

A. PlanetLab

Planetlab [17] has been used by researchers for more than a decade for network and distributed services experimentation. With PlanetLab, each user receives a slice equivalent to a virtual machine using which, experiments can be run, without control on the underlying hardware and network infrastructure. Therefore, the performance of applications deployed in this platform is highly variable. We expected concurrent experiments, and hardware heterogeneity to drive the observed performance to figures below current commercial platforms.

Fig. 3 shows the distribution of $T_{estimate}$ values for a worldwide sample of 193 nodes. We have split the nodes across continents to ease the comparison. The CDFs illustrate the broad spectrum of nodes with different performance ranges across the platform. Surprisingly, some PlanetLab nodes have a performance equivalent to a high-end node in a public cloud platform, e.g., an "large" node in EC2 or a "standard2" in GCE. We observe an overall better performance for European nodes, most likely because the blades have been installed more recently.

B. Public Cloud platforms

Public cloud providers offer various on-demand virtual machines each with different types of resources. (e.g. AmazonEC2 offers about 25 different types of instances at the moment). Having such a variety of options makes the task of selecting nodes for deployment very difficult.

Public Cloud platforms typically span multiple geographic regions around the world. Each region contains several availability zones (AZs) that are physically isolated and have independent failure probabilities. One AZ is roughly equivalent to one data center. A VM in such platform is called an instance. Different types of instances come with different performance

[2]We plan to release the tool as open source software soon

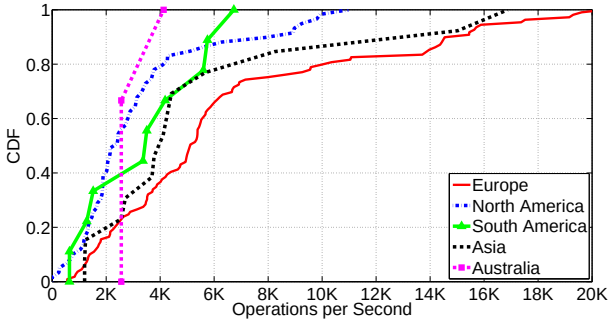

Fig. 3: $T_{estimate}$ measured from 193 Planetlab nodes: 109, 59, 9, 13, 3 nodes in Europe, North and South America, Asia and Australia, respectively

characteristics and price tags in each platform. It is challenging for users of the platforms to know how the application will perform on each type of instance. We pick from each platform a few instances with different cpu/memory specifications and test our technique. We rely on our methodology as a building block, and obtain for each run a $T_{estimate}$ (highest throughput a node can sustain). This will be used as a metric to evaluate the performance of Cassandra on various platforms under our SLA latency constrains.

Microsoft Azure. We chose three types of instances among the options offered by Azure (small, medium and large with 1, 2 and 4 cores and 1.75GB, 3.5GB and 7GB memory, respectively). For each instance type, we deployed a Cassandra node in 6 different AZs (2 in US, 2 in Europe, 2 in Asia). Figure 4 shows the box-plot diagram of the returned throughput for each combination of location and instance type. Each box shows the result of experiments ran for a 24 hour period during which we performed a run every 15 minutes. Figure 4 shows a clear differentiation of the $T_{estimate}$ for different types of instances. Within the same instance type, different locations showcase varying levels of performance. We expect that one of the main factors explaining the measured performance of different instances is the type of hardware used, as well as the level of user multiplexing on the sampled blades. We also observed performance variations on equivalent instances at the same location. In particular, the figure shows a temporary performance degradation observed in two different instance types, namely the "Large" instance in Asia East (ASIA-E inside LARGE (A3) in Figure 4) and the "Medium" instance in US East (US-E inside MEDIUM (A2)). Our observations show that for a short period of time, a sudden dip in performance happens, followed by a gradual recovery. We ran the same type of measurements on two other well known public providers, Amazon EC2 and Google Compute engine as well as 193 nodes using non-commercial platform (PlanetLab) but due to space limitation we will not present them here.

Overall, our results illustrate that our methodology is useful for cloud users to observe performance of application on different types of VMs with different specifications. We were able to observe performance under a very simple workload. Our measurements show variation in performance and the capability of our proposed methodology to detect that. We

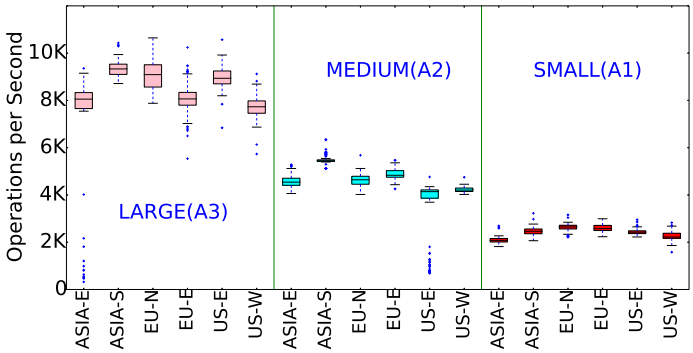

Fig. 4: Benchmarking three types of instances across six different datacenters in Microsoft Azure.

believe that our methodology is useful to expose nodes with poor performance, triggering their redeployment to a fresh VM that exhibits the expected performance.

C. Sampling the Latency/Throughput Trade-Off

We showed in the previous section multiple estimations of the maximum load an application server can sustain, under a predefined latency SLA. While valuable, this metric only reflects a partial view of reality. Application providers might also want to know how a server will behave in the presence of different ranges of load. We can also use our methodology to understand this aspect of application-level performance. Indeed, by design, our methodology samples multiple values of the T_{test}, and for each value it records the end-to-end latency percentiles.

Figure 5 depicts the observed latency measurements for large and medium instances in Microsoft Azure public cloud platform. The surface plane interpolates the median end-to-end latencies that are observed for the various values of T_{test}. Note that the more the throughput increases the more the corresponding median end-to-end latency values increase (in particular higher latency percentiles are more sensitive). The surface shows that the transition state is abrupt beyond certain T_{test} in each case, i.e., as soon as the T_{test} reaches values close to the overloaded state of the server, latency increases significantly for the high percentiles (e.g., typically the 75th and higher). Note that similar node instances tend to have similar latency/throughput surfaces, with larger instances (with more RAM memory and computing power) displaying less abrupt changes at the transition state compared to smaller instances. Moreover, the impact of the design of our ramp up policies are also reflected in Figure 5. The fast ramp up tends to sparsely sample low throughput values, while the linear ramp up during the fine-tuning phase will more extensively cover the higher loads. Despite our attempt at finely sampling around the transition state, we observe on the figure that the abrupt changes in latency around the transition state make it challenging to sample in a controlled manner around the transition phase. This might be an interesting aspect for future work.

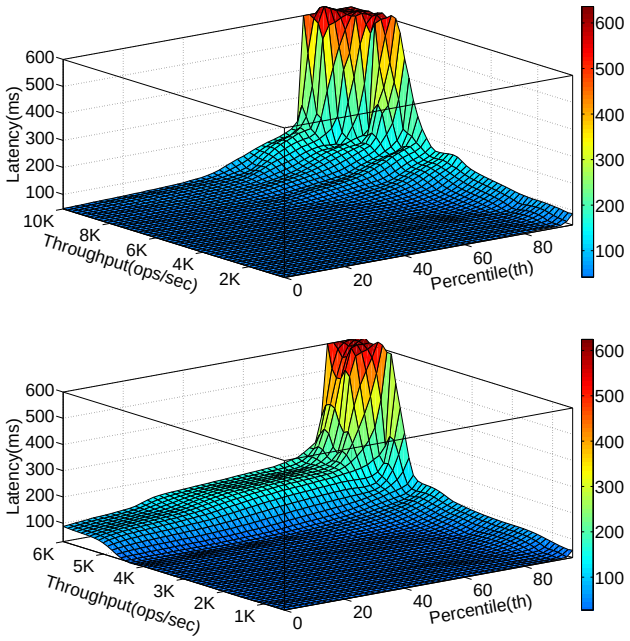

Fig. 5: Surface plot for different throughputs and the corresponding percentiles and the result of median latency for a large (A3) (top) and medium (A2) (bottom) size instances.

IV. More Complex Workload and Throughput Estimation

While the assumption about the workload in the above sections was simple, i.e. one type of operation (read) over one record from disk, often application providers intend to test more complex workloads and observe the performance of application in those scenarios. Based on those observations they would be able to plan for deployment and provisioning VMs. In this section, we generate different workloads using both preliminary operations (read and write). We choose a few workloads mentioned in [16] which is claimed as representative examples of realistic workload. The records are requested according to uniform and zipfian distribution (e.g. the distributions of article access in Wikipedia is zipfian [18]). In case of zipfian when choosing records, some records will be extremely popular while most records will be unpopular. We utilized the implementation of zipfian used in [16] and integrate it into our tool.

Workload	Operations	Distribution	Application Example
W1-only Write	Write:100%	Uniform	e.g. A back-end to an Internet of Things (IOT) application keep writing the incoming data where data is fed into a BI system for further analysis
W2-only Read	Read:100%	Zipfian	e.g. User profile cache, where proles are constructed elsewhere
W3-Read/Write	Read:95% write:5%	Zipfian	e.g. Photo tagging; add a tag or update it, but most of the operations are read
W3-Read/Write	Read:50% write:50%	Zipfian	e.g. Session store recording recent actions in a user session

TABLE I: List of Workloads

Table. I lists three different workloads that are considered as use-cases. We estimate the throughput for each case. The

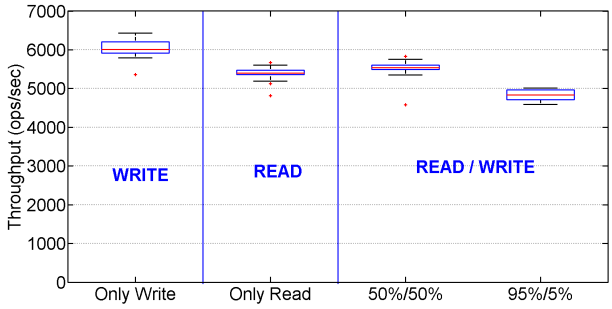

Fig. 6: Throughput estimation of different workloads on a m.medium VM in EC2

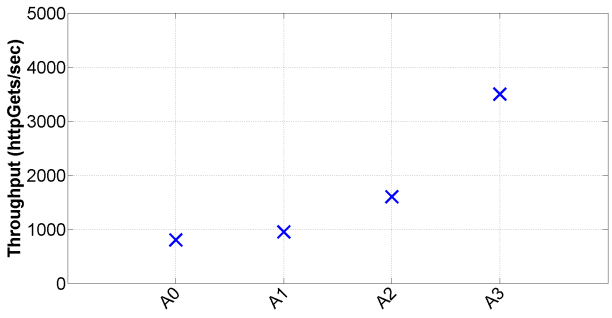

Fig. 7: Estimated throughput (in terms of httpGet requests per seconds) of an Apache webserver deployed on four types of instance in Microsoft Azure

application (Cassandra) that we use, involves both memory and disk to perform read and write operations.

Figure 6 depicts the estimations of throughput for each workload in table I on a medium size node in EC2 platform. It shows that only write has the highest performance. Moreover, in W3 involving more write operations results in higher throughput.

V. Throughput Estimation for Another Application

In this section we show that our methodology is valid using different types of application. Apache web server is deployed on different types of instances in Microsoft Azure cloud platform. Our tool is set to generate "Get" requests for a simple web page (default Apache home page). We chose 4 types of instances (A0, A1, A2, A3 with share, 1, 2 and 4 cores and 768, 1.75, 3.5 and 7GB memory respectively) from this cloud provider using Ubuntu v14.04LTS. Figure 7 shows median $T_{estimate}$ values (httpGet requests per seconds) returned from each instance based on SLA latency of 100msec for 90th percentile. All the instances belong to Euroup north datacenter.

VI. Related Work

In this part we review different types of tools and studies related to performance evaluation of infrastructure and the deployed application. Benchmarking and monitoring tools are used for evaluating and tracking the performance of both infrastructure and application. Different benchmarking tools exist at various levels of abstraction, from the low level system part like CPU, to the whole application, e.g., database system. We review some of the benchmarking and monitoring tools and mention the difference between those tools and our work.

Low level Benchmarking. With the advent of Cloud Computing, more and more providers offer Infrastructure-as-a-Service (IaaS) platforms. IaaS allows the customers to launch different types of virtual machines (VMs) with varying combinations of CPU, memory, storage and networking capacity to choose the appropriate mix of resources for the applications. To measure the performance of a VM often low level benchmarking tools (or micro-benchmarking) are used. Micro-benchmarks, e.g., [19], [20], [21] are mostly designed to stress each of the main compute resources individually (e.g., CPU, disk or network) and generate one score to be used for comparison. Li et al. [22] compared multiple cloud providers using different types of micro-benchmarks. [23] showed the high performance variation for most of their metrics related to CPU, disk I/O and network in EC2. Wang et al. [24] showed the variability of network performance in EC2 compared to non-virtualized clusters. Barker et al. [25] quantified the jitter of CPU, disk, and network performance in EC2 and its impact on latency-sensitive applications. The purpose of low level benchmarking tools is different from our work. They are designed to benchmark different components of a VMs. The results can be used to compare performance of individual components of VMs in different cloud providers. But benchmarking without the ability to interpret the results at application level cannot be useful for application providers. Therefore, our goal is to characterize the application that is deployed on a given VM.

Application level benchmarking. There exist different tools for benchmarking different types of applications. GridMix [26], HiBench [27], and Berkeley WL Suite [28] are benchmarks that are designed for evaluating the Hadoop framework. MineBench [29] benchmarks data mining algorithms. CloudRank-D [30] offers a benchmark suite for various machine learning and data mining algorithms. CloudSuite [31] has collected individual benchmarking tools for different types of applications e.g. data analytic, web search, media streaming and etc. The above data analytic and data mining tools do not deal with user requests. However, our focus is on user requests and dealing with corresponding response time of that requests. Cloudstone [32] offered a benchmarking tool that contains one web 2.0 application using one front-end and support for one back-end application that serve the users. However, they support one type of front-end and back-end and focus on answering the question of how many concurrent users can be logged-in and supported for a fixed dollar amount. Previously the on-line transaction processing (OLTP) benchmarks are used to evaluate the performance of the databases at back-end. For example, TPC-C [33] simulates a few types of transactions against a database and generates the results in two ways, performance (transaction per minutes) and performance/price. Their main goal is to deliver a single performance number to compare different database systems and answer the question of "which is the best database system for OLTP?". However, a single estimation is not representative for application. moreover, changing the workloads might end up to different

performance values which is not considered in above tools. The above benchmarking tools provide a single number representing performance of a particular application, without involving the latency into their evaluation. However, our estimation of the application performance is based on defined latency value. Moreover, our target is to provide performance evaluation based on different latency values. We sample latency values at various loads and observe the trade-off between application performance and server responsiveness.

Monitoring tools. Performance monitoring tools usually provide monitoring and analysis of specific parameters of system and notification about critical changes in their status. Nagios [34] monitors system metrics, network protocols and services. Processor load, disk usage, system logs, interactions and connectivity can be monitored. Zabbix [35] is another monitoring tool that tracks the status of CPU, memory, network, disk I/O, disk space and log files. The purpose of monitoring tools is to monitor and alert users about the changes in server performance elements. Moreover, monitoring tools are by nature white-box, i.e., they have to access and track the status of internal elements of system. The results of monitoring tools are difficult to translate into global application level-performance. However, the focus of our work is to provide a fine grain behaviour of performance at application level. Our approach is black-box, i.e., we treat the whole application server as a black-box with no assumption about internal of application and we employ the external values of latency from a remote node and explore the performance space of application.

VII. Conclusion

In this paper, we present a black-box technique that measures the performance of cloud applications. We probe the application remotely, iteratively adjusting the generated load based on the measured latency from previous steps. Using our technique, we have estimated the maximum capacity of an application for a given SLA over multiple cloud platforms. Our results show that not only we can detect the performance differences between instance types and platforms, but we can also pinpoint individual VMs that exhibit unusually poor performance. Moreover, our methodology samples the server behaviour for a range of loads by recording the observed latencies for each load. From sampling results, we identify trade-off between performance and latency. We have also involved more complex workloads and showed performance estimation in different scenarios.

References

[1] G. Linden, "Make data useful," *Presentation, Amazon, November*, 2006.

[2] P. Dixon, "Shopzilla's site redo - you get what you measure, velocity conference talk. http://velocityconf.com/velocity2009/public/schedule/detail/7709," 2009.

[3] J. Hamilton, "The cost of latency," *Perspectives Blog*, 2009.

[4] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and http chunking in web search," in *Velocity Web Performance and Operations Conference*, 2009.

[5] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[6] Amazon ec2 service level agreement(2014). [Online]. Available: http://aws.amazon.com/ec2-sla/.

[7]

[8] Microsoft azure(2014). [Online]. Available: http://azure.microsoft.com/en-us/.

[9] Google compute engine(2014). [Online]. Available: https://cloud.google.com/products/compute-engine/.

[10] R. van den Berg. tcpping. [Online]. Available: http://www.vdberg.org/~richard/tcpping.html

[11] A. Croll and S. Power, *Complete Web Monitoring: Watching your visitors, performance, communities, and competitors*. " O'Reilly Media, Inc.", 2009.

[12] H. Pucha, Y. Zhang, Z. M. Mao, and Y. C. Hu, "Understanding network delay changes caused by routing events," in *Prof. of ACM SIGMETRICS*, 2007, pp. 73–84.

[13] Apache jmeter. [Online]. Available: https://jmeter.apache.org/

[14] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[15] Cassjmeter.cassandra jmeter driver. [Online]. Available: https://github.com/Netflix/CassJMeter/.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[17] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[18] G. Urdaneta, G. Pierre, and M. Van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[19] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[20] Dbench. [Online]. Available: https://www.samba.org/ftp/tridge/dbench

[21] Unixbench. [Online]. Available: http://freecode.com/projects/unixbench

[22] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 1–14.

[23] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[24] G. Wang and T. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[25] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. ACM, 2010, pp. 35–46.

[26] gridmax. [Online]. Available: http://hadoop.apache.org/mapreduce/docs/current/gridmix.html/

[27] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[28] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 2011, pp. 390–399.

[29] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 182–188.

[30] C. Luo, J. Zhan, Z. Jia, L. Wang, G. Lu, L. Zhang, C.-Z. Xu, and N. Sun, "Cloudrank-d: benchmarking and ranking cloud computing systems for data processing applications," *Frontiers of Computer Science*, vol. 6, no. 4, pp. 347–362, 2012.

[31] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 37–48, 2012.

[32] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proc. of CCA*, vol. 8, 2008.

[33] Tpc-c. [Online]. Available: http://www.tpc.org/tpcc/.

[34] Nagios, infrastructure monitoring(2014),. [Online]. Available: http://www.nagios.org/

[35] Zabbix, enterprise-class monitoring solution(2014),. [Online]. Available: http://www.zabbix.com/