Insights in Duet Benchmarking on Cloud Servers - Measuring Runtime Interference

Niklas Fomin
Technische Universität Berlin
niklas.fomin@campus.tu-berlin.de

***Abstract*—This report examines a novel area of cloud service benchmarking, with a focus on understanding how concurrent processes influence performance metrics in cloud servers during *Duet Benchmarks.* Utilizing *PostgreSQL*, benchmarked with *HammerDB* on *Google Cloud Platform*, this study sets a foundation for a scalable and automated benchmarking environment for research purposes. Central to this exploration is the application of *Docker* and *Linux Containers (LXC)* to measure potential impacts of additional workloads on SUT Servers statistically. By simulating realistic operational conditions, the study provides key insights into how isolated container environments respond to external stress, utilizing statistical and time series analysis methods to dissect and understand these interactions. The goal is to offer insights in understanding of cloud server dynamics, particularly how modern container platforms like *Docker* and *LXC* maintain resource efficiency and isolation against interfering processes thus enhancing the fidelity of cloud benchmarking methodologies.**

I. Introduction

Cloud service benchmarking plays a crucial role in the optimization and validation of cloud-based resources, addressing several significant challenges inherent to the field. Research has demonstrated the value of integrating advanced suites within cloud-based CI/CD pipelines to detect crucial performance variations[6]. To mimic real-world conditions, studies often employ virtual machines (VMs) or container technologies that are configured anew for each experiment, mitigating inconsistencies from cloud-specific anomalies or variance between instances. One innovative approach in this domain is *Duet Benchmarking*, which seeks to address these issues by orchestrating a setup where multiple Systems Under Test (SUTs) are operated concurrently on a single host machine, while the benchmark server is positioned on a distinct cloud server to secure consistent and dependable outcomes[4][3]. Furthermore this methodology aims at accurately gauges the influence of a cloud server that concurrently hosts multiple applications or SUTs, shedding light on the reproducibility of benchmark outcomes. Concurrently, change point detection techniques specialize in identifying precise moments of significant performance shifts, offering an analytical edge in understanding software behavior over time. These approaches together empower a more accurate and reliable performance evaluation in cloud environments, vital for continuous software improvement and ensuring optimal operation[3][4]. While VMs have been the conventional choice, the emergence of container technology offers a lighter, more resource-efficient alternative[8][9]. This project work serves as an initial exploration of current potentials in the cloud service benchmarking domain, where the goal lies in *(1) Setting up a prototype for a highly automated benchmarking environment* and *(2) exploring the field of statistical tools that allow for appropriate analysis of benchmarking data.* For the SUT the choice is the relational database management system (RDBMS) *PostgreSQL* which is benchmarked by the *HammerDB* engine, hosted on two distinct servers on the *Google Cloud Platform (GCP).* During the benchmark execution, an additional workload is deployed on the SUT server to simulate realistic operational conditions on cloud servers. The experiment serves as a fundamental framework with the potential for scaling, aiming to contribute to the domain of application Duet Benchmarking in cloud ecosystems. It intends to leverage advanced virtualization technologies, notably *Docker* and *Linux Containers (LXC)*, to assess how interference affects their isolated resources and efficiency.

II. Related Work

The foundation of this study builds on related work in the realm of cloud service benchmarking. Although the field itself is well-established, the concrete focus extends toward novel methodologies. The study of methods in microbenchmarking, *Duet Benchmarking*, and the use of statistical measures provide not just a context but also an inspiration to further develop and expand upon current research. The aim is to experiment with an analytical toolbox, thereby contributing to a better understanding of new frontiers in cloud service benchmarking. Grambow et al.investigates the viability of using optimized microbenchmark suites for detecting application performance changes using a *Duet Benchmarking* strategy. Their study delves into the comparative effectiveness of microbenchmarks and traditional application benchmarks within CI/CD pipelines, particularly under the constraints of frequent code updates and the need for swift feedback. Their approach is backed up by the research outlined in "Duet Benchmarking: Improving Measurement Accuracy in the Cloud" by Bulej et al. They suggest that duet measurement is not only feasible but also advantageous for performance regression testing in cloud infrastructures, prompting further exploration into its applicability within CI/CD pipelines, especially in environments without dedicated virtual instances. In the realm of software performance analysis, the significance of change point detection as a method for identifying performance variations is underscored in the study of *Hunter* by Fleming et al., an open-source tool designed to detect both performance regressions and improvements within time series data effectively. By comparing *Hunter* against established algorithms like *PELT* and *Dynp* using real time series data, the study demonstrates *Hunter's* capabilities in identifying performance shifts. Next to methodological approaches Avula and Zou research provides an insightful comparison of the performance of the TPC-C benchmark — a well-regarded industry standard for Online Transaction Processing (OLTP) systems — across three major cloud service providers: *Amazon Web Services (AWS)*, *Microsoft Azure* and *GCP*. Their study leverages this benchmark to understand how these platforms handle transaction-intensive workloads by evaluating different aspects of cloud performance.

III. Contribution

This project work aims to provide insights into performance interference of *Duet Benchmarks* at runtime for containerized SUTs. The experimental framework offers a highly automated code base, which is not only adaptable to the chosen computational infrastructure, tool stack, and analysis methodology but also designed for customization and scalability leading to further research possibilities. Key to this setup is the ability to observe the particular effects that a resource-intensive application could impose on benchmarking accuracy by incorporating a fixed-timed execution of a resource exhaustive *Go* program. This feature simulates operational stress and can be investigated by monitoring capabilities fetching time series data from the virtualization hypervisor. Incorporating time series analysis, particularly change point detection methods suggested by Daly et al. research, aligns with the recognized utility of these analytical means in identifying performance inflections.

IV. Experiment Setup

The technical specification of the experimental framework can be seen in I. The local computer being the orchestrative machine, *Terraform* is used to provision the compute infrastructure. Within the *Terraform* script, a local executor is integrated to facilitate the invocation of an *Ansible* playbook, which automates the deployment of software on both the benchmark and the SUT server.

The design of the prototype setup is meant to furnish a reproducible, automated, and comparable environment for conducting *Duet Benchmarking* experiments. Upon establishing the infrastructure, one is advised to establish ssh-connections to the benchmark server — and ideally to the SUT as well — to examine logs for diagnosing any potential issues related to connectivity or otherwise. As an entry point the benchmark server introduces an *Experiment Wizard*, which acts as the primary interface for initiating and concluding the experiment. This tool allows users to select and configure two possible SUTs - one deployment on *Docker* and on *LXC*. During the experiment, the benchmark server records and logs data from *HammerDB* operations. Once the benchmarking process concludes successfully, users are redirected back to the *Experiment Wizards* interface. Picking up on Amazon Web Services, Inc. and Avula and Zou, *HammerDB* is the choice as the benchmarking tool to generate database loads. *HammerDB* implements the TPC-C benchmark, published by the TPC for OLTP. The TPC-C specification on which TPROC-C is based, implements a computer system to fulfill orders from customers to supply products from a company. The benchmarks mock a company selling items and keepin its stock in warehouses. The test schema can be as small or large as you wish with a larger schema meaning an increased level of transactions. The system's workload comprises a diverse mix of five transaction types. These transactions are randomly selected in alignment with their respective percentage distributions. The following list shows the *HammerDB* configuration with regards to target metrics for post-benchmark time series analysis, while for detailed information *HammerDB's* documentation should be compiled[1]:

- Benchmark: TPC-C
- Database: *PostgreSQL*
- Transactional Iterations: 10.000.000
- Virtual Users: Set to the number of CPUs on the system
- Warehouses: Is multiplied by 5 with the amount of virtual users
- Benchmark Duration: 20 Minutes
- Timeprofile: etprof[2]
- Metrics: Transaction Response Times in P50% [3] (in milliseconds), Transaction Counts (#)

[1] *HammerDB* Documentation

[2] *HammerDB* offers xtprof as the standard time profile to calculate MIN, MAX, AVG & the Percentiles P50, P99, P95, ratio and the standard deviation. However only cumulative values over the benchmark run are available and no time series is outputted. Therefore the older etprofile was modified to print the Percentiles at 2 second intervals.

[3] For the analysis only the Neword Transaction Type is chosen due to having the highest occurrence in the distribution with 45%.

Configuration	Benchmark-Server	SUT-Server	
		Docker	LXC
Compute Platform	GCP us-central1-c	GCP us-central1-b	GCP us-central1-b
Instance Type	n1-standard-2	n2-standard-8	n2-standard-8
Instance Architecture	x86/64 Intel Haswell	x86/64 Intel Cascade Lake	x86/64 Intel Cascade Lake
Instance Spec.	2 Cores - 2,3GHz - 7,5GB RAM	8 cores - 2,8Ghz - 32GB RAM	8 cores - 2,8Ghz - 32GB RAM
SUT Version	HammerDB 4.9	PostgreSQL 12	PostgreSQL 16
OS Platform	ubuntu-20.04-focal	ubuntu-20.04-focal	ubuntu-20.04-focal
Container Platform Version	-	24.0.5	4.0.9

TABLE I
Configuration of Experiment Setup

V. Implementation

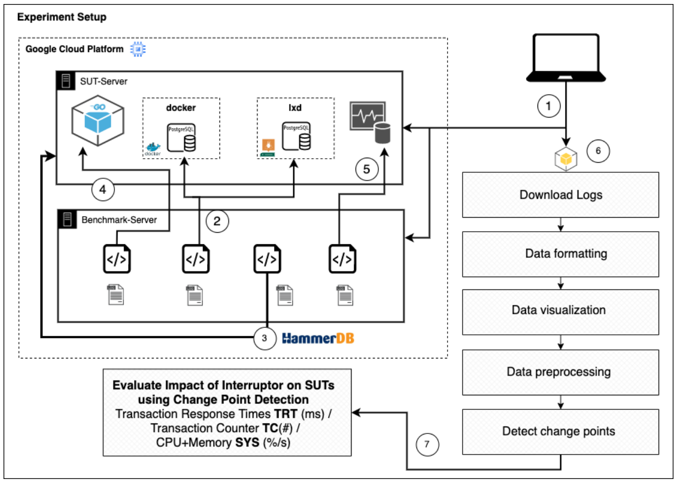

Fig. 1. Experiment Setup

1) Provision Infrastructure: *Terraform* is employed for setting up and configuring the infrastructure necessary for benchmarking and the SUT servers aiming to establish an environment adaptable to development. This step lays the groundwork for all subsequent actions.
2) Setup of the SUTs: Using a *Ansible* playbook the setup and deployment of necessary software on the benchmark server is streamlined by installing *HammerDB* alongside shell scripts written in Bash for coordinating the benchmark execution. On the SUT server its specific dependencies, software, SUT-setup scripts and the *Interruptor* Application are installed and ready to be launched.
3) Run *HammerDB* benchmark: The *Experiment Wizard* initiates the benchmarking run, engaging with the SUT via ssh to execute a shell script that incorporates connection verification within the distributed setup. The *Wizard* then oversees the execution of the benchmark through four main stages: *(1) Schema Build (2) Run Benchmark (3) Calculate results (4) Delete schema.*
4) Run *Interruptor* App: Following the initial benchmark stage, the experiment proceeds with the activation of the *Interruptor* application. The *Interruptor* is scheduled to start five minutes after its invocation and will run for five minutes before it self-deactivates. This introduces an additional layer of realism to the testing environment bringing up the assumption of causing visible change points in the time series data. The *Go* program that serves as a stress test is designed to intensely utilize CPU and Memory through concurrent processes. The application leverages the *Go* runtime's ability to optimize CPU core usage, thus exploiting system parallelism. Its tasks, such as calculating π, matrix multiplication, and memory copying, showcase varying algorithmic complexities and computational intensities, from CPU-bound operations to memory bandwidth demands. The implementation of concurrency utilizing goroutines and synchronization demonstrates practical parallel processing and thus puts the system properly under load.
5) Monitor resource consumption: Concurrently to the invocation of the *Interruptor*, a monitoring script written in *Bash* for *Docker Engine* and *LXC* detects the current SUT after the schema is constructed providing measurements of the SUTs CPU and Memory consumption at a time-frequency adapted to the calculation of the transactional metrics (except Transaction Counts).
6) Local Analysis: Post-benchmark, *HammerDB's* output along with the collected resource usage data undergoes local analysis. The *Experiment Wizard* enhances this process by compensating for *HammerDB's* limited logging capabilities, supplying functionality for filtering the relevant metrics results ready for refinement. To conclude the experiment the SUT server undergoes a cleanup process.
7) Interpretation of results: Finally, the outcomes of the benchmark are processed for time series analysis. The goal is to detect the impact of the *Interruptor* application at the known Start and End Time of the app. For this sake the local computer offers a prepared *Jupyter* notebook offering simple preprocessing for

each of the three recorded metrics per SUT leading to visualization change point detection on the time series data[4].

VI. Analysis

The following analysis is concerned with a basic exploration of the broad spectrum of statistical methods applied to time series data. The main intention lies in taking the raw data followed by "trying out" and comparing the performance of different detection algorithms with different hyperparameters. While letting much room for further preprocessing of the data, as well as precise parameter tuning, the result should much more open an initial window on how change point detection algorithms perform on raw database benchmark data overall. Therefore the desired result of the analysis is to observe if the detected change points the actual timestamps of the *Interruptor* application start and end time.

A. Evaluation approach

The analysis is done using *Python 3.12.2* and makes use of the features implemented in the following libraries.

- *Matplotlib*
- *Pandas*
- *Numpy*
- *Ruptures*
- *Seaborn*

B. Detection Algorithms

With a focus on univariate data, this analysis uses both the *PELT* algorithm and *Dynp* technique offered by ruptures, chosen for their alignment with the analytical requirements[4]. *PELT* is an efficient algorithm designed to identify change points in a signal while maintaining a linear computational cost relative to the number of data points. It employs a pruning rule to discard unlikely indices, thereby optimizing the search process without compromising the ability to locate the optimal segmentation. The *Dynp* algorithm accommodates various cost functions. It precisely computes the minimum sum of costs across all potential segmentations of the signal, therefore organizing the search sequence efficiently.

C. Results

The application of both *PELT* and *Dynp* to the benchmark time series data opens up significant possibilities for evaluation, comparison and analysis. Due to this project's limited scope a generic evaluation approach classifies a detected change point as the change point caused by the *Interruptor* Application when the timestamps have a maximum difference of 10 seconds. This method is formalized in the following section.

Let $P = \{p_1, p_2, \ldots, p_n\}$ represent the set of detected points, where n is the total number of detected points. Let k_1 and k_2 represent the known points, knownPoint1 and knownPoint2, respectively.

Define two indicator functions for the detection status of k_1 and k_2:

$$I_1 = \begin{cases} 1, & \text{if } \exists\, p \in P \text{ such that } |p - k_1| \leq 10 \\ 0, & \text{otherwise} \end{cases}$$

$$I_2 = \begin{cases} 1, & \text{if } \exists\, p \in P \text{ such that } |p - k_2| \leq 10 \\ 0, & \text{otherwise} \end{cases}$$

The output, detectionStatus, is then defined as:

$$\text{detectionStatus} = I_1 \wedge I_2$$

Where:

- $I_1 \wedge I_2 = 1$ (true) if both $I_1 = 1$ and $I_2 = 1$.
- $I_1 \wedge I_2 = 0$ (false) otherwise.

Algorithm 1 Evaluate Change Point Detection

```
Initialize adjusted indices $P \leftarrow [i - 1 \text{ for } i \text{ in } R]$
Initialize changepoint timestamps list $C \leftarrow []$
for each index $p$ in $P$ do
    if $p$ is a valid index within $D$ then
        Append timestamp $D[p]$ to $C$
    end if
end for
Initialize detection flags $I_1 \leftarrow 0, I_2 \leftarrow 0$
for each point $p$ in $C$ do
    Calculate differences $d_1 \leftarrow |p - k_1|, d_2 \leftarrow |p - k_2|$
    if $d_1 \leq 10$ then
        Set $I_1 \leftarrow 1$
    end if
    if $d_2 \leq 10$ then
        Set $I_2 \leftarrow 1$
    end if
end for
Determine final status detectionStatus $\leftarrow I_1 \wedge I_2$
return detectionStatus
```

D. Key Observations

Target Metric	Detection Algorithm	SUT1 Docker	SUT2 LXC
TRT	PELT	True	False
	Dynp	False	False
TC	PELT	True	True
	Dynp	False	False
CPU Usage	PELT	False	False
	Dynp	False	False
Memory Usage	PELT	False	True
	Dynp	False	False

TABLE II
Overall Detection Comparison

[4] *Github* Source code and Experiments are accessible.

This study reveals that the anticipation of visually and statistically identifying change points following planned interventions is not fully met as shown in II. While *PELT* manages to detect 50% of the actual change points, the *Dynp* approach fails to detect any. It's worth noting that the identified change points often manifest as time ranges within the time series data which does not allow for a precise and exclusive interference detection of the *Interruptor* Application. Furthermore, *PELT* offers configurable parameters such as the *penalty* value to minimize the detection of false positives. This significantly influences the detection ranges and could be further enhanced by determining optimal values for *min size* and *jump* that influence the minimum segment length and subsample point in the time series. For example a penalty value chosen too low results in a vast amount of change points not allowing for specific clarification. While this study didn't delve into determining these parameters accurately, further experimentation shows promising for precise hyperparameter tuning. Considering *Dynp's* expected performance on data with known change points beforehand, its failure to detect any in this study is notable. An interesting observation pertains to the TRT metric for both SUTs. Slicing the time series data into two pieces to calculate mean, variance, and standard deviation for comparison reveals unexpected results. Values observed in the second slice, where the *Interruptor Application* was running, exhibit greater deviation from the mean compared to the first slice, potentially indicating that additional resource consumption on the host VM, at least for database workloads in containerized environments, does not necessarily lead to measurable interference impacts. This could be explained by the virtualized runtime environment of the SUT. *Docker* achieves its robust isolation through the use of namespaces and cgroups. With namespaces, Docker assigns each container its isolated workspace, effectively cloaking it from the processes of other containers or the host system. On the other hand, *LXC* offers a similar level of isolation but leans more towards a conventional virtualization model, utilizing namespaces and cgroups to fashion containers that act more like lightweight virtual machines. This model not only preserves the high degree of isolation seen in *Docker* but also provides an environment that closely mimics a full operating system[7]. In the context of benchmarking, this means that *LXC's* isolation measures keep the container's operations insulated, ensuring that the performance data accurately reflects the container's internal processes without being distorted by external activities.

VII. Conclusion and Future Work

The conclusion of this study opens up various pathways for future research. In terms of the experimental setup, there is potential to fully leverage the capabilities of *HammerDB* or explore other benchmark engines. It's essential to delve deeper into how benchmarking affects not only databases but also other applications, particularly in contexts where workloads are assessed in tandem. Expanding the scope of SUTs or employing different virtualization or isolation technologies could provide a broader perspective on system behavior under stress. On the statistical analysis front, there is an opportunity to broaden the research base and adopt a more sophisticated toolset of established analytical methods and preprocessing techniques to decode the time series data. The field of detection algorithms, particularly, is far from being exhausted. By applying a variety of algorithms and focusing on precise hyperparamater-tuning the adaption of *Duet Benchmarking* in the cloud could be further understood and potentially underlined.

References

[1] Amazon Web Services, Inc. *SQL Server Performance on AWS.* Online. Available: Amazon Web Services, Inc. website. Amazon Web Services, Inc., Oct. 2018.

[2] Raghu Nandan Avula and Cliff Zou. "Performance Evaluation of TPC-C Benchmark on Various Cloud Providers". In: *2020 11th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON).* 2020, pp. 0226–0233. DOI: 10.1109/UEMCON51285.2020.9298047.

[3] Lubomír Bulej et al. "Duet Benchmarking: Improving Measurement Accuracy in the Cloud". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering.* ICPE '20. ACM, Apr. 2020. DOI: 10.1145/3358960.3379132. URL: http://dx.doi.org/10.1145/3358960.3379132.

[4] David Daly et al. "The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System". In: Apr. 2020, pp. 67–75. DOI: 10.1145/3358960.3375791.

[5] Matt Fleming et al. *Hunter: Using Change Point Detection to Hunt for Performance Regressions.* 2023. arXiv: 2301.03034 [cs.DB].

[6] Martin Grambow et al. "Using Microbenchmark Suites to Detect Application Performance Changes". In: *IEEE Transactions on Cloud Computing* (2022), pp. 1–18. ISSN: 2372-0018. DOI: 10.1109/tcc.2022.3217947. URL: http://dx.doi.org/10.1109/TCC.2022.3217947.

[7] Marek Moravcik et al. "Comparison of LXC and Docker Technologies". In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA).* 2020, pp. 481–486. DOI: 10.1109/ICETA51985.2020.9379212.

[8] Vasily A. Sartakov et al. *CAP-VMs: Capability-Based Isolation and Sharing for Microservices.* 2022. arXiv: 2202.05732 [cs.OS].

[9] Syed Shah et al. "Benchmarking and Performance Evaluations on Various Configurations of Virtual Machine and Containers for Cloud-Based Scientific

Workloads". In: *Applied Sciences* 11 (Jan. 2021), p. 993. DOI: 10.3390/app11030993.