*International Journal of Internet, Broadcasting and Communication Vol.13 No.1 180-186 (2021)*
*http://dx.doi.org/10.7236/IJIBC.2021.13.1.180*

**IJIBC 21-1-21**

# Design and Implementation of I/O Performance Benchmarking Framework for Linux Container

Gijun Oh, Suho Son, Junseok Yang and Sungyong Ahn

*School of Computer Science and Engineering, Pusan National University, Korea*
*{kijunking, suho.son, junseokyang, sungyong.ahn}@pusan.ac.kr*

***Abstract***

*In cloud computing service it is important to share the system resource among multiple instances according to user requirements. In particular, the issue of efficiently distributing I/O resources across multiple instances is paid attention due to the rise of emerging data-centric technologies such as big data and deep learning. However, it is difficult to evaluate the I/O resource distribution of a Linux container, which is one of the core technologies of cloud computing, since conventional I/O benchmarks does not support features related to container management. In this paper, we propose a new I/O performance benchmarking framework that can easily evaluate the resource distribution of Linux containers using existing I/O benchmarks by supporting container-related features and integrated user interface. According to the performance evaluation result with trace-replay benchmark, the proposed benchmark framework has induced negligible performance overhead while providing convenience in evaluating the I/O performance of multiple Linux containers*

***Keywords:*** *Benchmark, Cloud Computing, Linux container, Virtualization, Resource Monitoring*

## 1. Introduction

In a cloud computing environment, it is very important to meet the requirements of multiple users for system resources through QoS (Quality of Service) control[1,2]. The virtualization is a key technology to share the pool of computing resources among multiple isolated instances (applications or users)[3]. In particular, the container-based virtualization is paid attention because it can allocate system resources more efficiently than the traditional hypervisor-based virtualization. As shown in Figure 1(a), the hypervisor-based virtualization must create a separate guest OS for each instance, resulting in storage and memory space overhead due to redundant installation of the guest OS[4]. Moreover, performance degradation occurs because requests for system resources must go through both the guest OS and the host OS. On the other hand, as can be seen in Figure 1(b), container-based virtualization does not require a separate guest OS and has much less performance degradation due to virtualization because it supports system resource distribution and isolation at the operating system level[5,6].

Linux container[7] is most popular container-based virtualization technology, using Linux Cgroups(Control

Manuscript Received: January. 3, 2021 / Revised: January. 8, 2021 / Accepted: January. 12, 2021
Corresponding Author: sungyong.ahn@pusan.ac.kr(Sungyong Ahn)
Tel: +82-51-510-2422, Fax: +82-51-517-2431
Assistant Professor, School of Computer Science and Engineering, Pusan National University, Korea

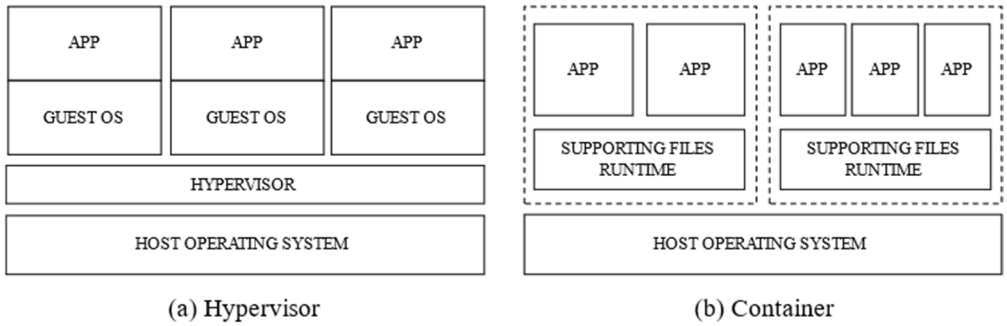


**Figure 1. Virtualization Architectures**

groups)[8] to allocates isolated system resources such as CPU, memory, I/O bandwidth to multiple Linux containers. Recently, as processing of large amounts of data becomes important due to the rise of emerging data-centric technologies such as big data and deep learning, research is being conducted to efficiently distribute I/O resources for high-performance storage such as NVMe SSDs in Linux cgroups[9-11].

However, the existing widely used I/O benchmark programs are not suitable for evaluating the resource distribution methods of Linux containers due to the following problems. At first, it is difficult to measure and track the I/O performance of multiple containers concurrently because most of the I/O benchmark programs do not support container related features such as container creation and allocation. The second is that an integrated interface is needed to use various I/O benchmarks because I/O performance should be evaluated with a variety of workloads such as synthetic workload, trace-driven workload, and specific application workload. Therefore, in this paper, we propose a novel I/O performance benchmarking framework for Linux container that can simultaneously measure the I/O performance of multiple containers and monitor the resource distribution status in real time using various kinds of benchmark programs. The proposed framework is implemented using JSON and Python, and provides web-based interface supporting various I/O benchmarks. The performance evaluation is conducted by applying trace-replay, a block I/O trace driven benchmark. According to the evaluation results, it is possible to evaluate the distribution of I/O resources of multiple containers with less than 1% performance overhead using the proposed framework.

The remainder of this paper organizes as follows. First, Section 2 describes the overall architecture and detailed implementation of the proposed I/O benchmarking framework as well as its design goal. The experiment results are presented to verify the efficiency and scalability of the proposed framework in Section 3. At last, Section 4 gives the conclusion of this paper.

## 2. Design and Implementation

### 2.1 Design goals

As mentioned above, existing I/O benchmark programs lack the support of evaluating the resource distribution method of Linux container. Table 1 describes whether the popular I/O benchmark programs

**Table 1. Linux Container Support in I/O benchmarks**

| Benchmark | Support or Not |
|---|---|
| Fio[12] | Support only cgroup allocation (not I/O weight, namespace) |
| Filebench[13] | Does not support |
| YCSB[14] | Does not support |

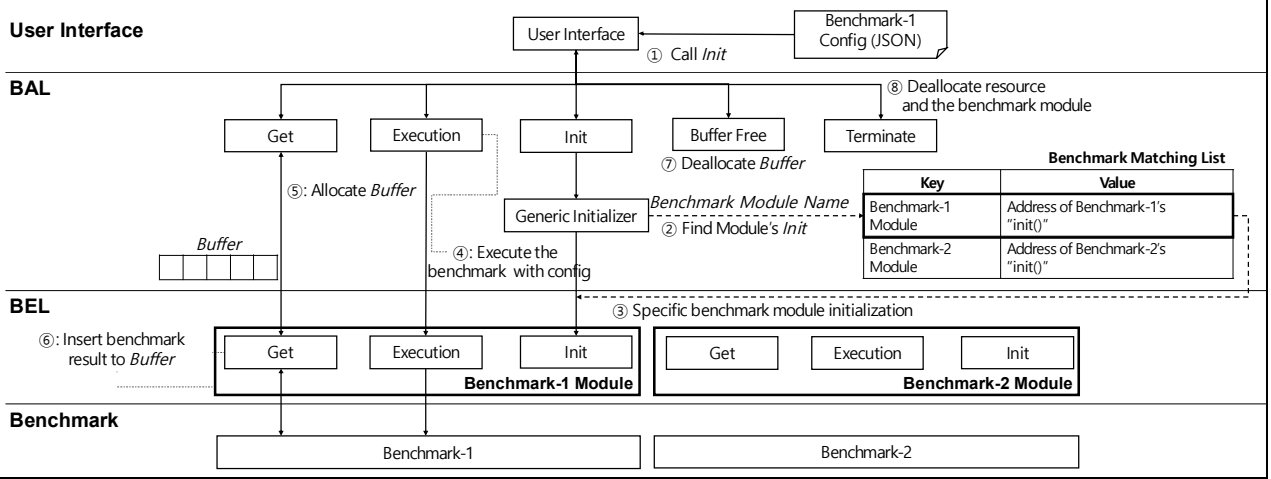


**Figure 2. The Architecture of the Proposed Benchmarking Framework for Linux container**

support Linux containers. According to Table 1, I/O benchmark programs other than Fio do not support for Linux container at all. Fio is also not sufficient to evaluate the distribution of I/O resource for Linux container because it cannot allocate I/O weights and namespaces for each cgroup.

Therefore, we have several design goals for implementing real-time I/O benchmarking framework for Linux container as follows: (1) It should support plentiful features related to Linux container such as cgroup creation / assignment and configuration of I/O weight of each container. (2) I/O resource used by each container should be able to be tracked and monitored concurrently in real time. (3) Various I/O benchmark programs should be able to be simply merged to our proposed benchmarking framework and operated with web-based unified interface. According to these requirements, we propose a novel I/O benchmarking framework for the Linux container that makes the best use of the existing benchmark programs, automatically creates containers and assigns benchmark to each of them.

### 2.2 Implementation

As you can see in Figure 2, the proposed system consists of the four layers contain the user interface, BAL(Benchmark Abstraction Layer), BEL(Benchmark Execution Layer), and benchmark programs. Because each layer independent from the other layers, they do not affect each other. To ensure independence between layers, each layer communicates with the other layer with structured data written in JSON format, as well as BAL and BEL are provided in the form of Linux shared library.

At first, as you can see in Figure 3, the user interface layer provides web-based integrated interface that receives parameters needed to configure benchmark programs and containers such as the number of cgroups, I/O weights, target benchmark, and target device. The submitted parameters by a user are turned into a structured data format by using JSON that is transferred to the below layer, BAL. Here, the name of benchmark program is used as a key to select the appropriate module in the below layer, BEL. By using this, BEL's module can assign and execute benchmarks in each container. After completion of the benchmark program user interface layer would get evaluation result of benchmark program for each container from the below layer as structured data form. In addition, since the BAL and BEL are provided in the form of a Linux library, various user interfaces such as web, desktop application, or terminal can be used if they support Linux shared library.

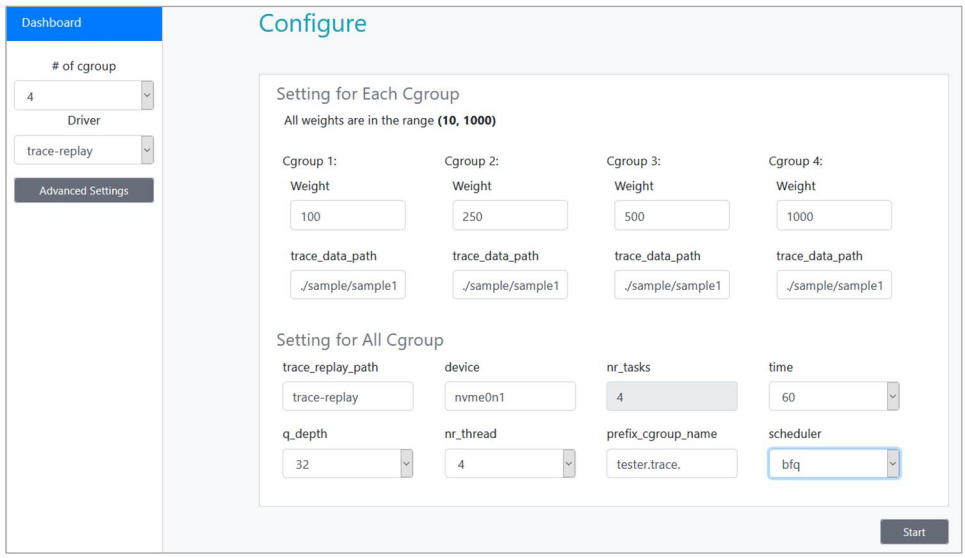


**Figure 3. The Web-based User-Interface of the Proposed Benchmarking Framework**

BAL is the abstraction layer that decodes various parameters received from the user interface layer and call appropriate methods of BEL to perform benchmarking. BAL exposes five interfaces (*Init, Execution, Get, Buffer Free, and Terminate*) to the user interface. The "*Init*" interface performs initialization based on the configuration data delivered from user interface layer. It selects an appropriate module in the BEL via a key which is in configuration data. After an appropriate module is selected, the initialization method is called from the module in the BEL. The "*Execution*" interface that can be called after initialization sends a signal about executing the benchmark program to the module. After that, the "*Get*" interface takes the I/O performance status like I/O bandwidth and IOPS in real time from a benchmark running on each container. The collected I/O performance data is stored at the buffer allocated by "*Get*" interface. The data is delivered to the user interface and displays as seen as in Figure 4. To manage the buffer storing I/O performance data, the BAL has "*Buffer Free*" interface that deallocates the buffer, so that user interface layer does not have to care about the management of the buffer. Finally, the "*Terminate*" interface is called when we want to stop or finish benchmarking. It decouples the BAL and the benchmark module of BEL and resets the configurations of the benchmarking system.

BEL is an implementation layer that provides the methods for the interface in the abstraction layer, BAL. The BEL has modules per benchmark, each of which has three methods for "*Init*", "*Execution*" and "*Get*". The selected module's "*Init*" method initializes the configurations of the system consisted BAL and BEL. It generates containers according to configuration delivered from BAL and allocates each benchmark process to them. Then, the "*Execution*" method of BEL starts to run designated benchmark program in response to the execution signal delivered by "*Execution*" interface in the BAL. The "*Get*" method is executed by BAL's "*Get*" interface. It first requests a measured I/O performance data to each container's benchmark. So, each benchmark process sends its own measured I/O performance data to the module in BEL, either through pipes, message queues, or shared memory, depending on the kind of benchmark. For example, the Fio generates its real-time benchmarking output via standards out. In this case, we can get the information by using pipes that allow redirecting standard I/O to the other files. At last, the module in BEL converts performance data collected from benchmark processes into structured data form by using JSON, then transfers it to BAL through the buffer dynamically allocated by BAL's "*Get*" interface.

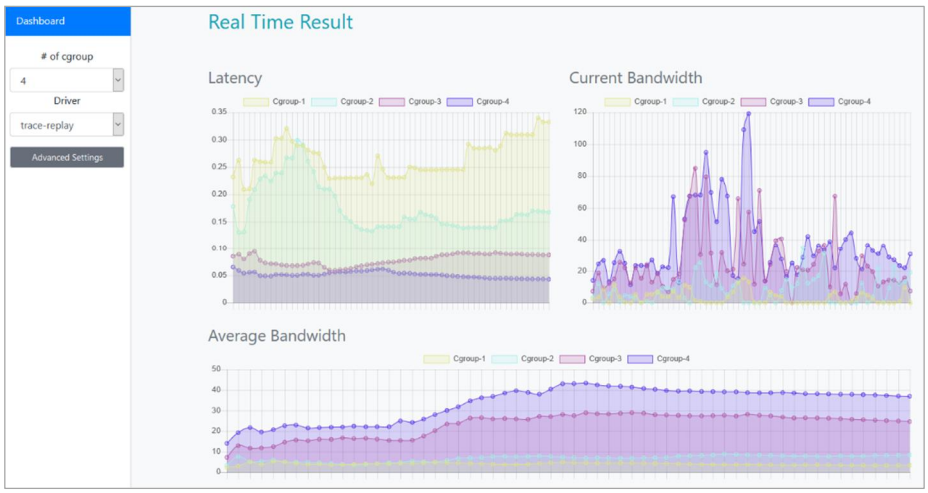


**Figure 4. Monitoring Benchmark Results of Multiple containers in real time**

As Figure 2 describes, the sequence of the proposed framework can be organized as follows: (1) A user inputs the configuration values in the user interface and submits them as structured data form to BAL's "*Init*" interface. (2) The "*Init*" interface finds and executes the initialization method of the appropriate benchmark module in BEL based on the configuration data delivered from BAL. (3) The benchmark module initializes the configuration of the specified benchmark and Linux containers, and allocates benchmark programs to each container. (4) The benchmark program in each container is started by an execution signal send by "*Execution*" interface of BAL. (5) To get the I/O performance data measured by benchmark programs in real-time, the "*Get*" interface allocates the buffer for receiving the benchmarking result data and call the "*Get*" method of the specified module in the BEL. (6) The benchmark module retrieves the result data from the benchmark in the container, and inserts it into the buffer. The user interface layer can access the result data through this buffer, then print it through the web-based interface. (7) After the received data is used, BAL's "*Buffer Free*" interface deallocates the buffer. (8) When all sequences are complete, the BAL's "*Terminate*" interface resets the system configuration and deallocates the benchmark module in BEL.

**Table 2. Hardware and Software Setup**

| Component | Specification |
|---|---|
| CPU | Intel Xeon CPU E5-2620 v4 @ 2.10 GHz |
| Memory | 32GB |
| Storage | Intel SSD DC P4500 1.0TB |
| OS (Kernel) | Ubuntu 18.04.2 LTS (Linux Kernel 5.3) |

**Table 3. Trace-Replay Workloads Characteristic**

| Component | Specification |
|---|---|
| Size of Total I/O | 30GB |
| Size of Average I/O | 14KB |
| Read/Write Ratio (R:W) | 3.64:1 |

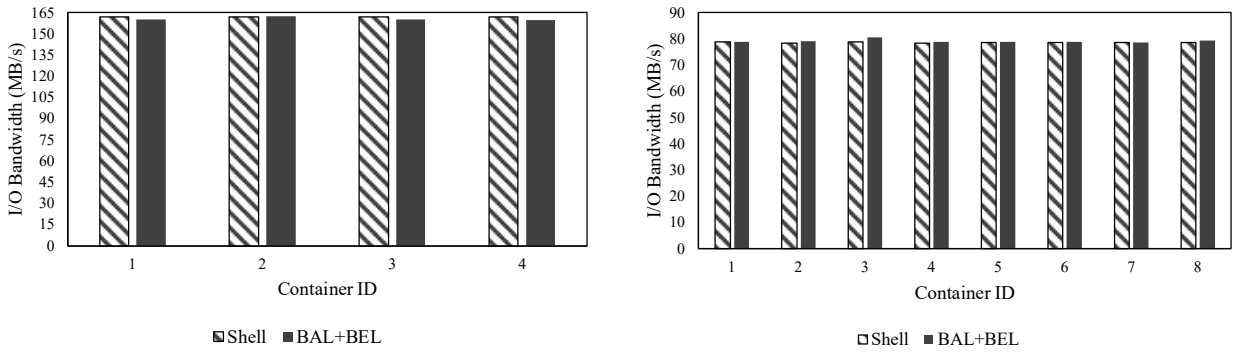


**Figure 5. I/O Performance Evaluation Results with Multiple containers**

## 3. Experiment Results

In this section, we evaluate the performance overhead and scalability of the proposed benchmark framework. The proposed benchmark framework is implemented by using JSON and Python and supports web-based integrated interface and trace-replay benchmark[15] which replays block I/O traces. The Figure 3 and 4 shows the configuration page and real-time I/O performance monitoring view, respectively. In the Figure 3, you can see the several input boxes on the figure which are related to the initial configurations for benchmark and Linux container as well as "*Start*" button. When the button is pressed, both BAL's "*Init*" and "*Execution*" interface are called sequentially. The Figure 5 shows the changes in I/O performance of each container in 1-second intervals, including I/O bandwidth, I/O latency and average I/O bandwidth. The experimental environment and workload characteristics used for trace-replay are described in Table 2 and 3, respectively. Note that sample block I/O trace(sample1.dat) provided by the trace-replay benchmark is used as the workload.

To verify the performance overhead of our benchmarking framework, we measured I/O bandwidth of multiple containers using trace-replay through the proposed benchmarking framework and compared it to the case where the benchmark was driven using a shell script. The Figure 5 shows that the proposed benchmarking framework (BAL+BEL) can evaluate the I/O performance of multiple containers with negligible performance overhead compared to using shell script (Shell). Moreover, even if the number of containers increases, the performance overhead does not increase. It means that the proposed framework has good scalability as well as low performance overhead. In addition, the proposed benchmarking framework is more convenient to evaluate the distribution of I/O resources for Linux container than using shell command because it provides graphical information on real-time benchmarking results.

## 4. Conclusion

A cloud computing service has a responsible for satisfying the service requirements of different users through QoS(Quality of Service) control. In particular, with the rise of emerging data-centric technologies such as big data and deep learning, the issue of efficiently distributing I/O resources across multiple instances has become important. However, the existing popular I/O benchmarks are not suitable for evaluating the resource distribution method of Linux containers due to the lack of support for containers. In this paper, we have proposed the novel I/O performance benchmarking framework for Linux container that supports container related features and an integrated user interface for existing I/O benchmarks. As a result, the proposed benchmarking framework can simultaneously measure the I/O performance of multiple containers and monitor

changes in resource distribution status by using various kinds of benchmark programs. The proposed benchmarking framework is implemented by using JSON and Python, and supports web-based integrated interface to configure benchmark programs and containers. According to the performance evaluation result with trace-replay benchmark, the proposed benchmarking framework has induced negligible performance overhead while providing convenience in evaluating the I/O performance of multiple Linux containers.

## Acknowledgement

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2018R1D1A3B07050034).

## References

[1] K. Jang. S. Shin, and J. Jung, "A Study on Recognization for Quality Importance of Cloud Services," *The Journal of the Institute of Internet, Broadcasting and Communication(JIIBC)*, Vol. 15, No. 2, pp. 39-44, April 2015. DOI: https://doi.org/10.7236/JIIBC.2015.15.2.39

[2] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad, "Cloud Scale Resource Management: Challenges and Techniques," in *Proc. 3rd USENIX conference on Hot topics in cloud computing*, pp. 3-3, June 14-17, 2011. DOI: https://dl.acm.org/doi/abs/10.5555/2170444.2170447

[3] B. Jennings and R. Stadler, "Resource Management in Clouds: Survey and Research Challenges," *Journal of Network and Systems Management*, Vol. 23, No. 3, pp. 567-619, Mar. 2015. DOI: https://doi.org/10.1007/s10922-014-9307-7

[4] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues," in *Proc. 2nd International Conference on Computer and Network Technology*, pp. 222-226, April 23-25, 2010. DOI: https://doi.org/10.1109/ICCNT.2010.49

[5] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, " Performance Overhead Comparison between Hypervisor and Container Based Virtualization," in *Proc. IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pp. 955-962, March 27-29, 2017. DOI: https://doi.org/10.1109/AINA.2017.79

[6] A. M. Joy, "Performance comparison between Linux containers and virtual machines," in *Proc. 2015 International Conference on Advances in Computer Engineering and Applications*, pp. 342-346, March 19-20, 2015 DOI: https://doi.org/10.1109/ICACEA.2015.7164727

[7] LXC. *https://linuxcontainers.org/lxc/introduction/*

[8] Linux kernel cgroups document. *https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt*

[9] P. Valente and A. Avanzini, "Evolution of the BFQ Storage I/O Scheduler," in *Proc. 2015 Mobile Systems Technologies Workshop*, pp.15-20, May 22-22, 2015. DOI: https://doi.org/10.1109/MST.2015.9

[10] J. Kim, D. Lee, S. H. Noh. "Towards SLO Complying SSDs Through OPS Isolation," in *Proc. 13th USENIX Conference on File and Storage Technologies (FAST 2015)*, pp. 183-189. Feb. 16-19, 2015. DOI: https://dl.acm.org/doi/10.5555/2750482.2750496

[11] P. Kwon and S. Ahn, "Dynamic Bandwidth Distribution Method for High Performance Non-volatile Memory in Cloud Computing Environment," *The Journal of the Institute of Internet, Broadcasting and Communication(JIIBC)*, Vol. 20, No. 3, pp. 97-103, Jun. 2020. DOI: https://doi.org/10.7236/JIIBC.2020.20.3.97

[12] Fio: Flexible I/O tester. *https://github.com/axboe/fio*

[13] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A Flexible Framework for File System Benchmarking," *USENIX ;login*, Vol. 41, No. 1, pp. 6–12, 2016.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sear, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. 1st ACM symposium on Cloud computing*, pp. 143-154, June 10-11, 2010 DOI: https://doi.org/10.1145/1807128.1807152

[15] Trace-replay. https://github.com/yongseokoh/trace-replay.