Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs

Janki Bhimani*, Jingpei Yang†, Zhengyu Yang*, Ningfang Mi*,
Qiumin Xu†, Manu Awasthi†, Rajinikanth Pandurangan† and Vijay Balakrishnan†
Northeastern University - Boston*, Samsung Semiconductor, Inc - San Jose†
Email: *bhimani@ece.neu.edu, †jingpei.yang@samsung.com, *yang.zhe@husky.neu.edu, *ningfang@ece.neu.edu,
†q.xu@samsung.com, †manu.awasthi@samsung.com, †rajini.pandu@samsung.com and †vijay.bala@samsung.com

***Abstract*—Our cloud-based IT world is founded on hypervisors and containers. Containers are becoming an important cornerstone, which is increasingly used day-by-day. Among different available frameworks, docker has become one of the major adoptees to use containerized platform in data centers and enterprise servers, due to its ease of deploying and scaling. Further more, the performance benefits of a lightweight container platform can be leveraged even more with a fast back-end storage like high performance SSDs. However, increase in number of simultaneously operating docker containers may not guarantee an aggregated performance improvement due to saturation. Thus, understanding performance bottleneck in a multi-tenancy docker environment is critically important to maintain application level fairness and perform better resource management.**

In this paper, we characterize the performance of *persistent* storage option (through data volume) for I/O intensive, dockerized applications. Our work investigates the impact on performance with increasing number of simultaneous docker containers in different workload environments. We provide, first of its kind study of I/O intensive containerized applications operating with NVMe SSDs. We show that 1) a six times better application throughput can be obtained, just by wise selection of number of containerized instances compared to single instance; and 2) for multiple application containers running simultaneously, an application throughput may degrade upto 50% compared to a stand-alone applications throughput, if good choice of application and workload is not made. We then propose novel design guidelines for an optimal and fair operation of both homogeneous and heterogeneous environments mixed with different applications and workloads.

***Keywords*—Docker containers, Flash-Memory, SSDs, NVMe, MySQL, Cassandra, FIO, Database**

I. Introduction

Docker containers are gaining more users due to their simple and efficient operation characters [1]. Container technology is projected to be the backbone on which software development cycle can be shortened [2]–[4]. Containers and virtual machines have similar resource isolation and allocation benefits but different architectural approaches, which allows containers to be more portable and efficient compared to bare metal and virtual machines [5], [6]. Containers are also proposed as a solution to alleviate dependency issues. Among different container technologies (e.g., docker, LxC, runC), docker has become one of the major adoptees for its ease of deploying and scaling.

While characterizing performance for multiple instances on a bare metal or virtual machine is not new [7], [8], I/O intensive dockerized applications deserve a special attention. First, the lightweight characteristic of docker containers promotes the simultaneous use of multiple containers to deploy multiple instances of same or different applications [9]. Second, the resource contention increases with increasing number of containerized instances, resulting in performance variation of each instance. However, the behavior of each instance is not thoroughly investigated given limited hardware resources. Third, there is always a requirement for data persistency in container for data accessibility which could be achieved through docker data volume. With the world looking forward towards high performance SSDs for their massive storage needs [10], more performance benefits could be gained on I/O intensive dockerized applications by using these devices as a backend. Given the state of art, understanding the performance of different types of workloads and applications for these NVMe high end SSDs is highly demanded.

In this paper, we focus on docker's persistent storage option called docker data volume, supported by an array of high-end enterprise SSDs. At the first glance, we observe that with the increase in number of containers, initially the performance can get better but eventually may saturate or even degrade due to limitation of hardware resources. We segregate application layer setup into *homogeneous* and *heterogeneous*. The container instances of same database application with same workload characteristics are called homogeneous as all such containers would compete for similar resources. For example, the setup is called homogeneous if all containers are of MySQL running TPC-C. While, the container instances of either different database applications or different workloads are called heterogeneous. The setup is called heterogeneous if some containers are of MySQL running TPC-C and simultaneously some other containers are of Cassandra running Cassandra-stress.

The major contributions of this paper are:

- Understanding the performance of write and read intensive workloads for homogeneous application setup.
- Analyzing and improving resource utilization and fair sharing of resources among different application containers.
- Investigating application throughput throttle for simultaneous operations of different application containers.
- Proposing novel design guidelines for optimal and fair operations of mixed dockeraized applications on high

978-1-5090-5252-3/16/$31.00 ©2016 IEEE

performance NVMe SSDs.

We provide, first of its kind work to show that, 1) for write intensive workloads, application throughput increases with increasing number of containers; 2) for read intensive workloads, application throughput may experience a throughput valley with increasing number of containers due to memory limitation; 3) for simultaneous operation of application containers performing sequential writes (or reads) and random writes (or reads), throughput of application performing random writes (or reads) is scarified terribly when compared to their respective standalone throughput; and 4) simultaneous operation of write intensive and read intensive applications is beneficial with better scope of increasing resource utilization and fair resource sharing.

The remainder of this paper is organized as follows. In Section II, we describe the related work. In Section III-B, we explain docker container data storage and our experimental setup. In Section IV, we explore homogeneous docker containers and heterogeneous docker containers. Finally, in Section VI we summarize our results as guidelines.

II. Related Work

The docker containers of most of the database applications like Redis, MySQL etc. are available to download [1]. The reported betterment in performance with the use of docker containers over the bare metal and virtual machines have attracted many users in a very short time. Charles Anderson introduced docker in [2] as container visualization technology, which is very light-weight compared to virtual machines. Docker containers help to address couple of problems, first, overcome the challenges of speed, performance and additional latency introduced by traditional VMs. Second, to reduce the development cycle of software, as sharing becomes easier with application dockers. Third, portability of application increases by making them compatible to run on different platforms. Finally, the recent research work [11] was emphasized on the use of docker for increasing computational reproducibility of research. With all these attractive features, docker containers are becoming the current mainstay mechanism for deploying applications in cloud platform.

Tracking the rapid growth of docker containers, it becomes important to evaluate its performance. Few attempts have been made to compare the performance of docker containers with virtual machines. In [5], authors explore the performance of two real applications (MySQL and Redis) individually and show that better throughput (transactions/s) can be obtained by using docker container compared to virtual machine. [9] and [6] explore the performance of Linux containers for building cloud and PaaS. Thus, most of the performance evaluation on docker container environment are concerned about exploring the hardware dependency of containers in terms of how name-space is used. In addition few research works have explored different container file system storage drivers (e.g. AUFS, Btrfs) [5], different copy-on-write strategy and data volume. Although containerized environment provide good scope of application level parallelism, but no study has explored the performance with increasing number of simultaneously executing containers. With more and more companies tending to run multiple containers simultaneously on each host, analyzing the performance of such an environment becomes highly important. In this work, we explore homogeneous and heterogeneous container environment to excavate different effects on application performance.

On the other hand, in order to support such a parallel application layer with multiple application containers operating simultaneously, a very fast storage is required. SSDs were initially used as a bufferpool to cache data between RAM and hard disk [12]–[16]. But as the $/GB of flash drives kept decreasing then the use of SSDs as a main storage became prominent [17], [18]. Now-a-days, the use of SSDs in enterprise server and data center is increasing. In [19] and [20], authors explore the performance of SSDs as main storage for database applications. Extensive research has also been performed on enhancement of energy efficiency of database applications using SSDs [21]–[24]. Furthermore, with the world looking forward towards high performance SSDs for their massive storage needs, NVMe is emerging as the protocol for communication with high performance SSDs over PCIe [25].

With the emergence of containerization techniques it becomes important to characterize the performance of high performance NVMe SSDs, with containerized application. To the best of our knowledge, this is the first attempt of performance evaluation of such a system. In this paper we aim to explore behavior of different real database containerized applications with high performance NVMe SSDs.

III. Hardware Architecture and Application Layout

A. Container Data Storage

Docker provides application virtualization using a containerized environment. Docker image is an inert, immutable, file that is essentially a snapshot of a container. Multiple docker containers can be instantiated with an application image. In order to maintain lightweight characteristics, it is advisable to keep the installation stack within the container as small as possible for better performance. The data management of containers is superintend either by docker storage drivers (e.g. OverlayFS, AUFS, Btrfs, etc.) or by docker data volumes.

Docker daemon can only run one storage driver, and all containers created by that daemon instance use the same storage driver. Storage drivers operate with copy-on-write technique, which thus provide more advantage for read intensive applications. For applications that generate heavy write workloads, it is advisable to maintain data persistence. Docker volume is a mechanism to automatically provide data persistence for containers. A volume is a directory or a file that can be mounted directly inside the container. The biggest benefit of this feature is that I/O operations through this path are independent of the choice of the storage driver, and should be able to operate at the I/O capabilities of the host.

In this paper, we characterize the performance of I/O intensive applications using the persistent storage option of docker data volumes. Figure 1, shows the stacked I/O path of underlying hardware. I/Os are generated by containerized workloads. Data volume is a directory or a file on the host system that can be mounted inside the container, and should be able to operate at the I/O capabilities of the host (see Figure 1

(f)). I/Os on the host are managed by the host backing file system such as XFS or EXT4 (see Figure 1 *(e)*). In all our experiments, we use XFS as the backing file system. This backing file system relays on a stack of logical and physical volumes, formed over an array of NVMe SSDs.

B. Experimental Setup

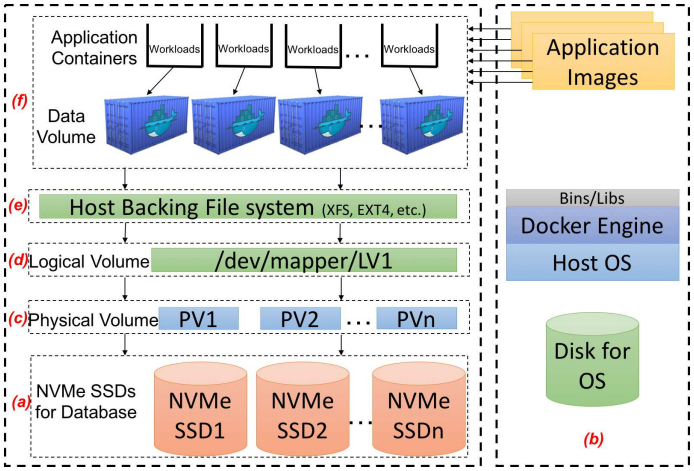

Fig. 1: Containerized system on flash volume of SSDs

TABLE I: Hardware Configuration

CPU type	Intel(R) Xeon(R) CPU E5-2640 v3
CPU speed	2.60 GHz
CPU #cores	32 hyper-threaded
CPU cache size	20480 KB
CPU Memory	128 GB
OSType	linux
Kernel Version	4.2.0-37-generic
Operating System	Ubuntu 16.04 LTS
Backup storage	Samsung PM953 960 GB
Docker version	1.11
MySQL version	5.7
Cassandra version	3.0
FIO version	2.2

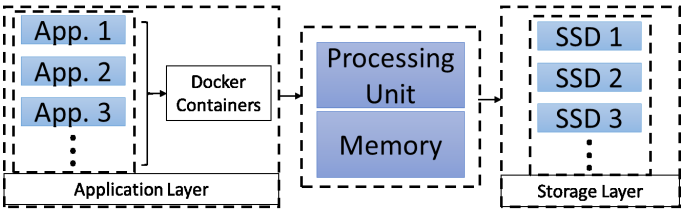

Fig. 2: System Stack

We built a platform consisting of multiple docker containers operating on an array of three enterprise NVMe drives in order to provide higher disk bandwidth as shown in Figure 1 *(a)*. Thus, an array of multiple SSDs is used to persist data of database applications running in the docker. The stack of host OS, docker engine and docker application images is maintained on a separate disk than that used for storing containerized application files and databases (see Figure 1 *(b)*).

The physical volume mapping 100% capacity of each SSD is created through LVM (Logical Volume Manager) (see Figure 1 *(c)*). These multiple physical volumes are combined to form single stripped logical volume using `lvcreate` [26], [27]. The data written to this logical volume is laid out in a stripped fashion across all the disks by the file system. The sum of the sizes of all SSDs maps to the size of logical volume (see Figure 1 *(d)*). Table I gives the detailed hardware configuration of our platform.

We chose MySQL [28] and Cassandra [29] for our docker performance analysis as these two are not only popular in relational database or NoSQL database applications, but also widely adopted by companies using docker for production. Respectively, we run the *TPC-C benchmark* [30] in MySQL container and Cassandras built-in benchmark tool, *cassandra-stress* [31] for our experiments.

In summary, Figure 2 shows the system stack of our platform. At the application layer, we have multiple simultaneously operating containers. Each container works in its own separate workspace in terms of file system and database. We analyze the performance of two database applications (MySQL 5.7 and Cassandra 3.0), for increasing number of docker containers. We evaluate two different scenarios of homogeneous and heterogeneous container traffic. The homogeneous container traffic is caused by containers running the same application under the same workload resources (e.g., client threads, data set size, read/write ratio, etc.). The heterogeneous container traffic is caused by containers running different workloads and different applications. The processing layer consists of a single processing unit for which all the docker containers compete. Memory and page cache resource is shared among all containers. No prior resource allocations in terms of processing unit and memory are done and all the containers compete on these shared resources at run time. The storage layer is managed in two different stacks: a hard drive that is responsible to store OS, docker engine, container images etc; and an array of three SSD drives dedicated to store database of all containerized applications managed by LVM.

IV. Experimental Results

In this section, we present the results to show the scaling of containerized docker instances on SSDs. We experiment with increasing number of simultaneously operating containerized instances. We evaluate two different types of containerized setup: 1)*Homogeneous* and 2)*Heterogeneous*. For each, we use the FIO benchmark [32] to cross verify the observations which we obtain from I/O intensive applications. We will use NVMe SSDs interchangeably with disks for the rest of the paper to refer to the persistent storage devices.

A. Homogeneous Docker Containers

We explore homogeneous docker containers by using MySQL and Cassandra applications. We first experiment with increasing number of MySQL containers to observe that application throughput scales due to increasing CPU utilization although disk bandwidth utilization saturates. Second, we

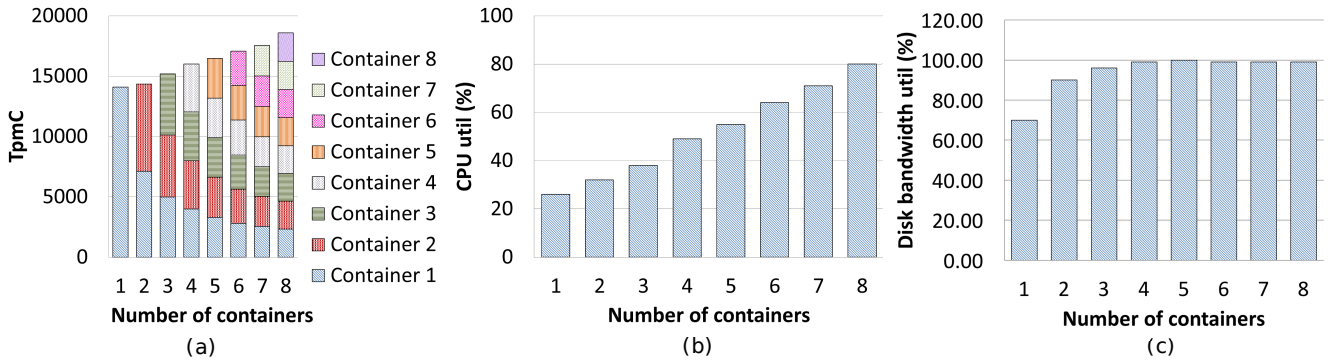

Fig. 3: Homogeneous with MySQL (TPC-C workload). (a) MySQL throughput with evaluation metric as the number of transactions completed per minute (TpmC), (b) CPU utilization, and (c) Disk bandwidth utilization

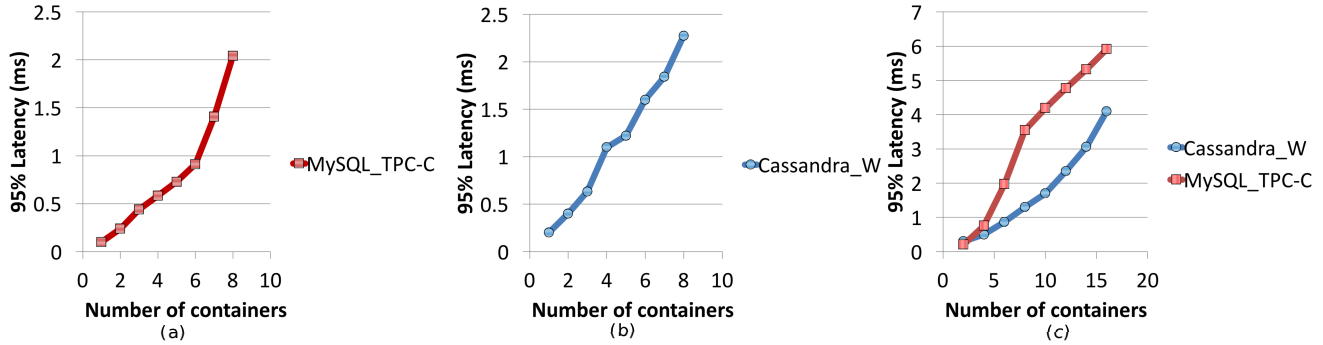

Fig. 4: Average latency for all running containers, (a) homogeneous MySQL, (b) homogeneous Cassandra_W, (c) heterogeneous MySQL + Cassandra_W

experiment with Cassandra 100% write (i.e. update) workload, which also scales with increasing number of containers but is limited by saturation of CPU utilization. Third, we experiment with Cassandra 100% read workload, where we note an interesting observation of throughput valley. Then, we investigate the reason behind this throughput valley to be page caching and memory. Lastly, we cross verify the throughput valley observation by constructing a similar FIO benchmark workload setup. Note that everywhere we mention write workload, we mean update operation.

Figure 3 shows the results of standalone containerized instances of MySQL. The workload configuration of MySQL is given in Table II. Figure 3 (a) shows that containerized instances of MySQL scale well with increasing number of containers. We observe that inspite of the decreasing throughput of each individual containerized instance, the cumulative throughput of MySQL containers increases with increasing number of simultaneous containers. The cumulative throughput is the sum of throughput of all simultaneously operating containers. Figure 3 (b) and (c) shows that disk bandwidth utilization gets saturated at four simultaneous containers, but cumulative throughput keeps increasing with higher CPU utilization on increasing number of simultaneous containers. Figure 4 (a) further presents the average 95 percentile latency as a function of number of containers under the homogeneous MySQL TPC-C workload. We observe that 95 percentile latency increases with increasing number of containers. Thus, we notice that there exists a trade off between cumulative throughput and I/O latency when we add more containers.

The similar experiments were conducted using a Cassandra application for 100% writes and 100% reads. The workload configuration of Cassandra_W is given in Table II. Figure 5 (a), shows that containerized instances of Cassandra 100% writes scales with increasing number of containers till six simultaneous containers. From Figure 5 (b), we observe that due to saturation of CPU utilization, the throughput saturates for further increase in number of containers. Even after throughput saturates at six containers, note that the latency keeps increasing with increasing number of simultaneous containers (see Figure 5 (a) and Figure 4 (b)).

Thus, from the above two experiments (i.e., MySQL and Cassandra_W), we notice that effects of CPU as bottleneck are more drastic on overall performance when compared to disk as bottleneck. So, an optimal operating number of simultaneous containers for achieving maximum throughput and minimum possible latency would be the number of containers at which CPU gets saturated.

Next, we investigate the performance under the 100% read workload using Cassandra_R, see Table II for workload details. Figure 6 (a) shows the jagged behavior of containerized instances. The exceptionally high performance can be observed till the number of containerized instances is increased upto three. This is because, after fetching data once from disk into main memory, the read operations are performed mainly

TABLE II: Homogeneous Workload Configuration

	Workload		
MySQL	TPC-C	# Warehouses - 1250	# Connections - 100
Cassandra_W	Cassandra-stress 100% Writes (i.e. Updates)	# Records - 50 million	Record size - 1KB
Cassandra_R	Cassandra-stress 100% Reads	# Records - 50 million	Record size - 1KB

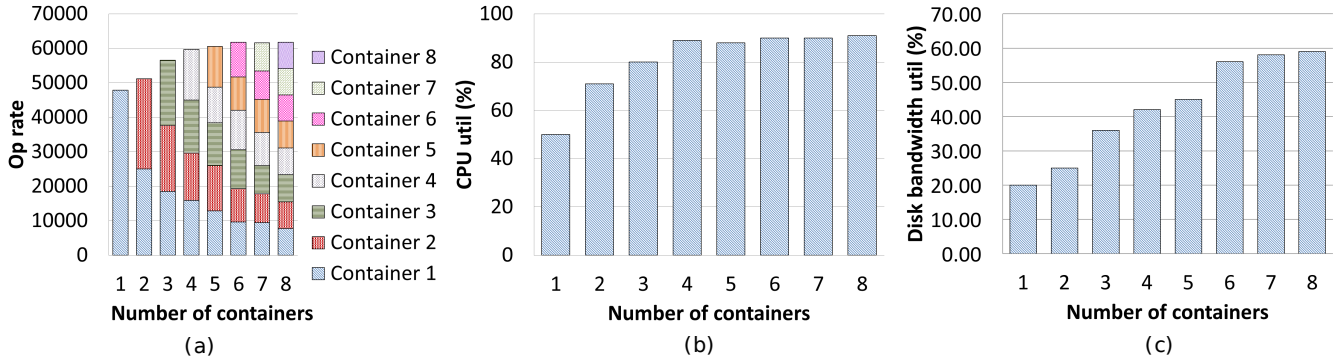

Fig. 5: Homogeneous with Cassandra (Cassandra_W workload). (a) Cassandra throughput, (b) CPU utilization, (c) Disk bandwidth utilization

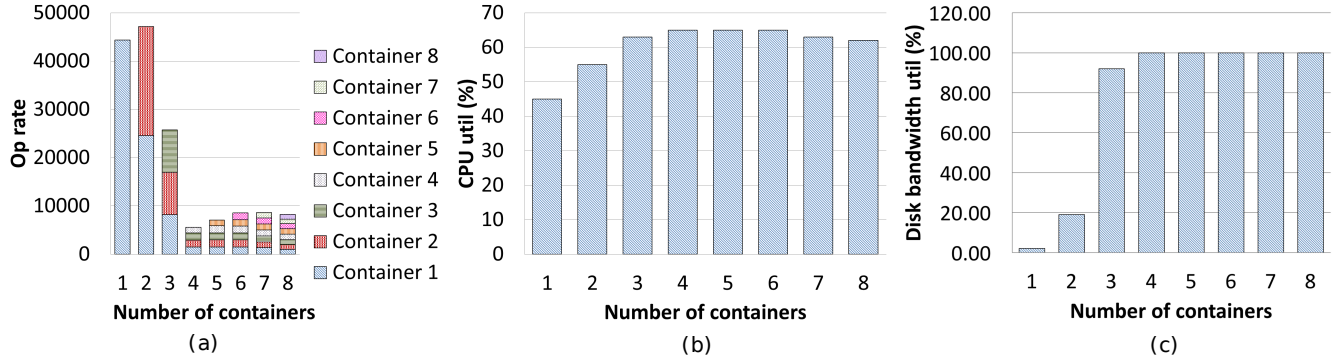

Fig. 6: Homogeneous for Cassandra (Cassandra_R workload). (a) Cassandra throughput, (b) CPU utilization, (c) Disk bandwidth utilization

from memory and page cache. The united size of four and more containers is not sufficient to fit in page cache and memory. Thus, the data is paged in and out leading to higher number of disk accesses. As disk access latency is much higher than the page cache and memory access latency, so when the number of simultaneous containers is more than four, throughput drops because a large amount of I/Os hit the disk. The throughput then becomes saturated by disk bandwidth. Figure 6 (b) shows that the maximum CPU utilization for read-only operations is lower (i.e., 65%) when compared to that of the write-only operations (i.e., 90% in Figure 5 (b)). Figure 6 (c) further shows that initially for a small number of simultaneous containerized instances most read operations are performed from memory and thus disk bandwidth utilization is very low. But, when the throughput valley is observed at four simultaneous containers, most of the operations are performed from disk. This leads to the increase and the saturation of disk bandwidth utilization.

In order to cross verify the above observed anomalous phenomenon of throughput valley, we perform similar FIO (Flexible I/O) benchmark [32] setup experiment. The size of each containerized FIO instance is set similar to the data set size of Cassandra container running the Cassandra-stress read workload. In order to observe the effect of operations from memory, page cache is not bypassed in this FIO experiment.

Figure 7 shows the results of containerized instances of the FIO benchmark. In order to obtain the similar setup as that of the Cassandra_R experiments, each FIO container operates on a file with the size of 50GB. The FIO workload is random read of size 4K, job size of 32 and IO depth of 32. Figure 7 also shows the throughput valley similar as observed in Figure 6. Figure 7 further shows that the cumulative throughput of read operations observed for 6, 7 and 8 simultaneous instances is very close to the rated throughput of disk. Thus the above observations cross verifies the throughput valley effect.

In summary, for a write intensive application, if CPU is not the bottleneck, then increasing number of homogeneous containers increases throughput till CPU gets saturated. On

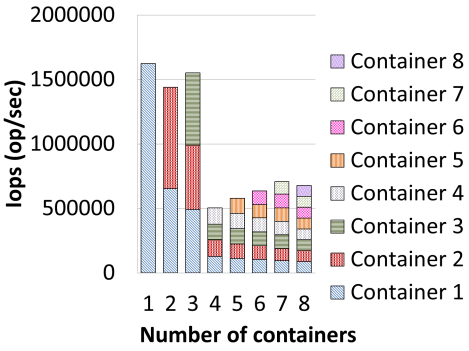

Fig. 7: Homogeneous for FIO (4KB random read buffered IO)

the other hand, for write intensive applications, once CPU becomes saturated then increasing the number of container only increases the latency without any improvement in throughput. Finally, if an application is read intensive and the container size is small, then the majority of its operations can be performed from page cache and memory. For such a case, it is advisable to limit the number of containers before falling into throughput valley.

B. Heterogeneous Docker Containers

We explore heterogeneous docker containers by running MySQL and Cassandra applications simultaneously. We observe an interesting observation that while operating simultaneously, the throughput of MySQL degrades to more than 50% of its standalone throughput observed in homogeneous experiments. But, the throughput of Cassandra degrades only around 16% of its standalone throughput observed in homogeneous experiments. In order to investigate the reason behind the observed unfair throughput scarifies, we experiment with different types of heterogeneous mixes such as, 1) Cassandra with FIO random write workload; 2) Cassandra with FIO sequential write workload; and 3) Cassandra with FIO random read workload. For all the heterogeneous experiments, we report the results for equal number of operating containers of both applications (i.e., total 16 containers would pertain to 8 containers of each application).

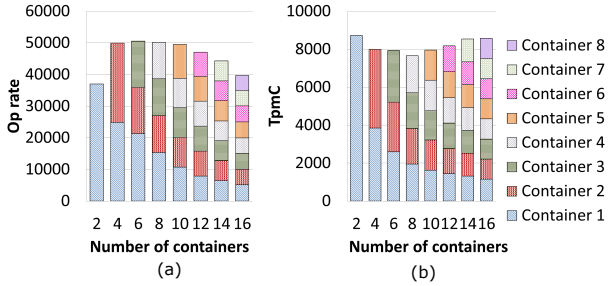

Fig. 8: Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and MySQL (TPC-C workload). (a) Cassandra throughput, (b) MySQL throughput

Figure 8, shows the results of simultaneously operating containerized instances of Cassandra and MySQL with workload configurations of Cassandra_W and TPC-C as given in Table II. Figure 8 (a) and (b), shows the application throughput of Cassandra and MySQL, respectively. For example, the first bar of Figure 8 (a), represents the throughput of Cassandra, when in total two instances, each one of Cassandra and MySQL are running simultaneously. We observe the unfair throughput throttle between Cassandra and MySQL. The best throughput of standalone homogeneous Cassandra instances from Figure 5 is 60K op rate. The best throughput with homogeneous MySQL instances from Figure 3 is 18K TpmC. But, for heterogeneous experiments of Cassandra and MySQL containers running simultaneously, the best throughput observed for Cassandra and MySQL from Figure 8 (a) and (b) is 50K op rate and 9K TpmC, respectively. While comparing the throughput of applications in standalone homogeneous deployment and heterogeneous deployment, we observe that the average throughput degradation of Cassandra containers is around 16% (i.e., from 60K to 50K op rate). But throughput degradation of MySQL containers is around 50% (i.e., 18K to 9K TpmC). Thus, MySQL containers scarify higher than Cassandra containers when both the application containers are operated simultaneously in the heterogeneous setup. From Figure 4 (c), we also observe that for this heterogeneous setup, latency of MySQL increases at higher rate when compared to Cassandra.

This is an interesting observation and we believe the nature of applications plays an important role. While operating simultaneous containers of different applications in the heterogeneous setup, we observed better resource utilization like CPU and disk. However, we would not like to group such application containers together, where unfair throughput sacrifices may compensate performance of one of the application drastically. We anticipate the reason behind such unfair throughput distribution to be the memory controller, which favors sequential writes more than random writes. Cassandra application performs comparatively higher proportion of sequential writes when compared to MySQL. To validate it, we conduct following three experiments.

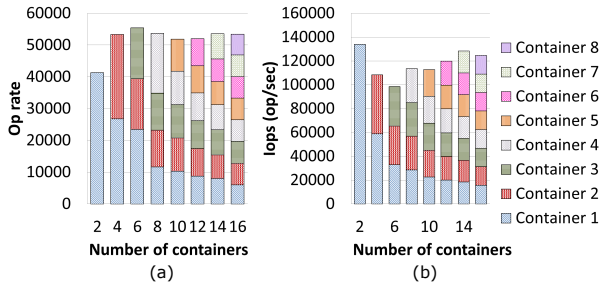

Fig. 9: Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and FIO (4KB random write). (a) Cassandra throughput, (b) FIO throughput

First, we run simultaneous operations of Cassandra_W with FIO random writes. We expect to see similar unfair throttle of throughput with containers of FIO random writers sacrificing higher than Cassandra, when compared to their respective standalone homogeneous operation. As expected, Figure 9 shows the unfair throughput throttle. The best throughput for standalone homogeneous FIO random write containers is 250k IOPS, but the maximum throughput we observe in Figure 9

(b) is 134k IOPS. This thus verifies that if containers of application having higher proportion of sequential writes is operated simultaneously with containers of another application performing random writes. Then the throughput of the application performing random writes is scarified terribly with respect to their standalone homogeneous operation throughput.

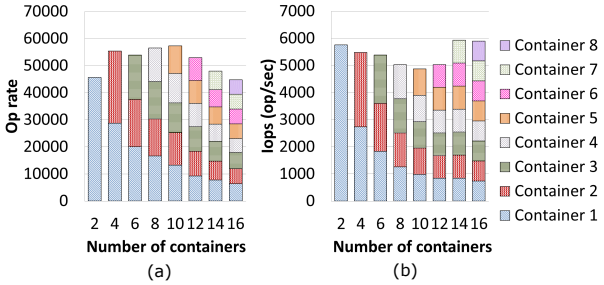

Fig. 10: Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and FIO (128KB sequential write). (a) Cassandra throughput, (b) FIO throughput

Second, we present the results of operating the Cassandra containers simultaneously with FIO sequential write containers in Figure 10. We see almost fair throughput throttle under these two applications because containers of both these applications are performing sequential writes. Figure 10, shows the result of simultaneously operating Cassandra with FIO sequential writes. The best throughput with standalone homogeneous FIO sequential write instances is 6500 IOPS. From Figure 10, we observe that the throughput of each application individually degrades only by 10% to 15%. Thus as expected, we observe fair throughput throttle.

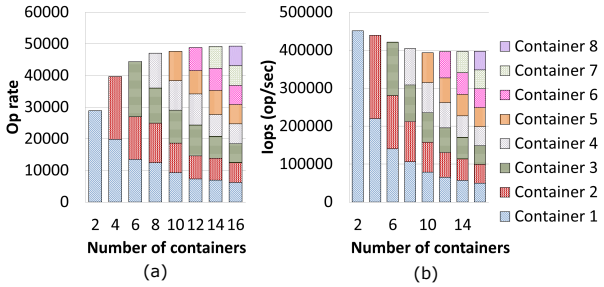

Fig. 11: Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and FIO (4KB random read workload). (a) Cassandra throughput, (b) FIO throughput

Third, in order to investigate the behavior of simultaneously operating write intensive and read intensive application containers, we show the results of Cassandra_W containers operating simultaneously with FIO random read containers in Figure 11. The best throughput with standalone homogeneous FIO random read instances is 450k IOPS. From Figure 11, we observe that the throughput of both applications individually degrades only by 10% to 15%. So, combining containers of write-intensive and read-intensive applications can achieve much better resource utilization and fair throughput distribution.

Thus, we summarize that the heterogeneous mix of containers of different applications leads to better utilization of overall resources like CPU and disk. However, it is not advisable to mix containers of applications that perform sequential writes (or reads) and random writes (or reads), because the throughput distribution as observed is unfair.

V. Design Implications

In this session we provide the high level design guidelines for homogeneous and heterogeneous containerized docker platform. For homogeneous case, we particularly analyze the behavior of write-intensive and read-intensive homogeneous grouping of containers. We propose the following guidelines to decide optimal number of simultaneously operating homogeneous containers.

- If application is write intensive then increasing number of homogeneous containers till CPU gets saturated, increases throughput.
- If application is read intensive and container size is small such that majority of its operations can be performed from page cache and memory, then it is advisable to limit number of containers before falling into throughput valley.

For heterogeneous case, we conclude that an effective heterogeneous mix of containers of different applications leads to better resource utilization and application throughput. Regarding the choice of good heterogeneous mix, we propose the following guidelines.

- It is not advisable to mix containers of applications that perform sequential writes (or reads) and random writes (or reads), because the throughput distribution as observed is unfair. In such a mix the throughput of an application performing random writes (or reads) may degrade drastically.
- Combining the containers of applications performing similar operations (e.g., both have majority of sequential writes) leads to the fair throughput throttle between application containers compared to their respective standalone homogeneous operation throughput.
- Combining the containers of write intensive and read intensive applications then we can achieve much better resource utilization and attain fair throughput degradation. In this case, despite of doubling the number of simultaneous containers, the cumulative throughput of each application only degrades around 10% when compared to the standalone implementation of respective applications.

VI. Conclusion

In this paper, we investigated the performance effect of increasing number of simultaneously operating docker containers that are supported by docker data volume on a stripped logical volume of multiple SSDs. We analyzed the throughput of applications in homogeneous and heterogeneous environments. We excavated the reason behind our interesting observations and further verified them by using the FIO benchmark workloads. To best of our knowledge, this is the first research work to characterize I/O intensive applications and explore important phenomenon like throughput valley in a containerized docker

environment. Further more, our work shows that it is not advisable to run simultaneously the containers of applications that perform sequential writes (or reads) and random writes (or reads). We finally presented some design guidelines that are implicated from performance engineering carried out in this paper. In the future, we plan to develop an automatic scheduler for docker engine to effectively distribute and group the simultaneous containerized instances of applications.

VII. Acknowledgements

This work was completed during Janki Bhimani's internship at Samsung Semiconductor Inc., and was partially supported by National Science Foundation Career Award CNS-1452751 and AFOSR grant FA9550-14-1-0160.

References

[1] Wikipedia, "Docker (software) — wikipedia - the free encyclopedia," 2016, [Online; accessed 12-July-2016]. [Online]. Available: {https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=728586136}

[2] C. Anderson, "Docker." *IEEE Software*, vol. 32, no. 3, 2015.

[3] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame, "The impact of docker containers on the performance of genomic pipelines," *PeerJ*, vol. 3, p. e1273, 2015.

[4] J. Fink, "Docker: a software as a service, operating system-level virtualization framework," *Code4Lib Journal*, vol. 25, 2014.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[6] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 610–614.

[7] A. Olbert, D. O'Neill, C. Neufeld *et al.*, "Managing multiple virtual machines," 2003, uS Patent App. 10/413,440.

[8] M. Ronstrom and L. Thalmann, "MySQL cluster architecture overview," *MySQL Technical White Paper*, 2004.

[9] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, pp. 105–111, 2014.

[10] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Analysis of NVMe SSDs and their Implication on Real World Databases," in *Proceedings of SYSTOR*, 2015.

[11] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[12] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD bufferpool extensions for database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[13] L.-P. Chang, "Hybrid solid-state disks: combining heterogeneous NAND flash in large SSDs," in *2008 Asia and South Pacific Design Automation Conference*. IEEE, 2008, pp. 428–433.

[14] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches." in *FAST*, vol. 10, 2010, pp. 101–114.

[15] H. Jo, Y. Kwon, H. Kim, E. Seo, J. Lee, and S. Maeng, "SSD-HDD-hybrid virtual disk in consolidated environments," in *European Conference on Parallel Processing*. Springer, 2009, pp. 375–384.

[16] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang, "hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1076–1087, 2012.

[17] R. Chin and G. Wu, "Non-volatile memory data storage system with reliability management," May 25 2009, uS Patent App. 12/471,430.

[18] B. aDam LeVenthaL, "Flash storage memory," *Communications of the ACM*, vol. 51, no. 7, pp. 47–51, 2008.

[19] Y. Wang, K. Goda, M. Nakano, and M. Kitsuregawa, "Early experience and evaluation of file systems on SSD with database applications," in *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*. IEEE, 2010, pp. 467–476.

[20] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 145–158.

[21] D. Schall, V. Hudlet, and T. Härder, "Enhancing energy efficiency of database applications using SSDs," in *Proceedings of the Third C* Conference on Computer Science and Software Engineering*. ACM, 2010, pp. 1–9.

[22] S. Park and K. Shen, "A performance evaluation of scientific I/O workloads on flash-based SSDs," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–5.

[23] S. Boboila and P. Desnoyers, "Performance models of flash-based solid-state drives for real workloads," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2011, pp. 1–6.

[24] H. Fujii, K. Miyaji, K. Johguchi, K. Higuchi, C. Sun, and K. Takeuchi, "x11 performance increase, x6. 9 endurance enhancement, 93% energy reduction of 3D TSV-integrated hybrid ReRAM/MLC NAND SSDs by data fragmentation suppression," in *2012 symposium on VLSI circuits (VLSIC)*. IEEE, 2012, pp. 134–135.

[25] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue SSD and I/O virtualization framework," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–11.

[26] M. Hasenstein, "The logical volume manager (LVM)," *White paper*, 2001.

[27] G. Banga, I. Pratt, S. Crosby, V. Kapoor, K. Bondalapati, and V. Dmitriev, "Approaches for efficient physical to virtual disk conversion," 2013, uS Patent App. 13/302,123.

[28] A. MySQL, "MySQL database server," *Internet WWW page, at URL: http://www. mysql. com*, 2004.

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

[30] Francois, W. Raab, A. Kohler, and Shah, *MySQL TPC-C benchmark*, (accessed Septenber 6, 2016). [Online]. Available: http://www.tpc.org/tpcc/detail.asp

[31] *Cassandra-stress benchmark*, (accessed Septenber 6, 2016). [Online]. Available: https://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsCStress_t.html

[32] *FIO - flexible I/O benchmark*, (accessed Septenber 7, 2016). [Online]. Available: http://linux.die.net/man/1/fio