Docker Container Scheduler for I/O Intensive Applications Running on NVMe SSDs

Janki Bhimani, Zhengyu Yang, Ningfang Mi, Jingpei Yang, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan

Abstract—By using fast back-end storage, performance benefits of a lightweight container platform can be leveraged with quick I/O response. Nevertheless, the performance of simultaneously executing multiple instances of same or different applications may vary significantly with the number of containers. The performance may also vary with the nature of applications because different applications can exhibit different nature on SSDs in terms of I/O types (read/write), I/O access pattern (random/sequential), I/O size, etc. Therefore, this paper aims to investigate and analyze the performance characterization of both homogeneous and heterogeneous mixtures of I/O intensive containerized applications, operating with high performance NVMe SSDs and derive novel design guidelines for achieving an optimal and fair operation of the both homogeneous and heterogeneous mixtures. By leveraging these design guidelines, we further develop a new docker controller for scheduling workload containers of different types of applications. Our controller decides the optimal batches of simultaneously operating containers in order to minimize total execution time and maximize resource utilization. Meanwhile, our controller also strives to balance the throughput among all simultaneously running applications. We develop this new docker controller by solving an optimization problem using five different optimization solvers. We conduct our experiments in a platform of multiple docker containers operating on an array of three enterprise NVMe drives. We further evaluate our controller using different applications of diverse I/O behaviors and compare it with simultaneous operation of containers without the controller. Our evaluation results show that our new docker workload controller helps speed-up the overall execution of multiple applications on SSDs.

Index Terms—Docker containers, flash-memory, SSDs, NVMe, MySQL, cassandra, FIO, database, scheduling, controller

1 Introduction

Docker containers are gaining traction due to their simple and efficient operation characteristics [1]. Container technology is projected to be the backbone on which software development cycle can be shortened [2], [3], [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 virtual machines [5], [6]. Among different container technologies (e.g., docker, LxC, runC), docker has become one of the major technologies due to its ease of deployment and scalability.

Containers versus Virtual Machines. The two virtualization technologies of virtual machine (VM) and docker container each have their own specializations and benefits. Understanding their features and limitations can help us improve the performance of applications running in these virtualization environments. The recent advancement of docker is different from traditional VMs. First, the VM hypervisor manages resources among different virtual machines in a distributed mode, where each VM is assigned the maximum limit of resources it can use. Unlike VM, containerized virtualization performs centralized resource management among different containers. Resources like processing unit, memory and page cache are shared among all containers. No prior resource allocations are done and all containers compete for the shared resources at runtime. As a result, resource contention and performance interference become more severe when using Docker containers than using VMs. Thus, it is more urgent and necessary to have an effective scheduling algorithm to launch Docker containers in order to mitigate such performance interference and improve resource utilization in the Docker container environment. Second, each VM has their own VM image that consists of guest OS and libraries. This type of virtualization results in better security and isolation, but causes higher startup overhead. Thus, a casual practice is to launch all VMs at the beginning. On the other side, as Docker containers are lightweight without guest OS, they can be launched when needed. This thus gives us a good opportunity to dynamically schedule and launch Docker containers at runtime.

Containerized Database Using NVMe SSDs. While characterizing performance of applications on a bare metal or virtual machine is not new [7], [8], I/O intensive dockerized applications deserve a special attention. First, the lightweight

- *J. Bhimani, Z. Yang, and N. Mi are with Northeastern University, Boston, MA 02115. E-mail: {bhimanijanki, yangzy1988}@gmail.com, ningfang@ece.neu.edu.*
- *J. Yang, Q. Xu, R. Pandurangan, and V. Balakrishnan are with Samsung Semiconductor, Inc, San Jose, CA 95134-1713. E-mail: {jingpei.yang, rajini.pandu, vijay.bala}@samsung.com, qiumin@usc.edu.*
- *M. Awasthi is with the IIT - Gandhinagar, Gandhinagar, Gujarat 382355, India. E-mail: manu.awasthi@gmail.com.*

Manuscript received 14 July 2017; revised 29 Jan. 2018; accepted 30 Jan. 2018. Date of publication 2 Feb. 2018; date of current version 14 Sept. 2018. (Corresponding author: Janki Bhimani.) Recommended for acceptance by A. Shrivastava. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TMSCS.2018.2801281

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 the increasing number of containerized instances, resulting in performance variation of each instance. However, the behavior of simultaneous instances of different applications has not been thoroughly investigated. Third, with the world looking forward towards high performance SSDs, such as Non-Volatile Memory express (NVMe) devices, for their massive storage needs, more performance benefits could be gained on I/O intensive dockerized applications by using these NVMe SSDs as the backend. Given the state of the art, it is highly demanded to better understand the performance of different types of workloads and applications for NVMe high end SSDs.

Performance Characterization. We segregate the application layer setup into *homogeneous* and *heterogeneous*. The container instances of a single 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 different database applications with different workloads are called heterogeneous. For example, 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. We remark that the above examples are to project the idea of a generic class of homogeneous and heterogeneous mix of an application. In this paper, we first characterize the operation of docker containers on NVMe SSDs considering the number of simultaneously operating containers and the application. We observe that for a homogeneous environment with the increase in number of containers, the performance can initially get better but eventually may saturate and even worse degrade due to limitation of hardware resources. For a heterogeneous environment, we observe that the throughput of some applications degrades significantly when they run simultaneously with some other applications.

Docker Workload Controller. Docker containers are a group of processes that belong to a combination of Linux namespaces and control groups (cgroups). Using cgroups [10] to manage the resources between the containers helps to prioritize instances and achieve isolation. Resource sharing is the main feature provided by the containerized technology. By default, a container has no resource usage constraints and thus, if needed, can use (and share) resource as much as the hosts kernel scheduler allows. Cgroups provides ways to control the fraction of each resource (e.g., memory, CPU, or block IO) that a container can use. However, setting such fixed limits of resource usage actually restricts online resource sharing and overall resource utilization. Thus, in this work, we propose a docker container scheduler that is able to take advantage of resource sharing to maximize resource utilization and avoid application interference by determining which containers should be operated simultaneously in the same batch.

In order to improve the performance based on our derived design guidelines from the characterization, we build a docker workload controller to schedule the execution of the batches of simultaneously operating containers. We model this optimization problem into a mathematical objective function and its constraints. We evaluate our controller with five different optimization solvers and illustrate the performance results when compared to simultaneous operation of containers without the controller. Our experimental results show that our new docker workload controller helps speed-up the overall execution of multiple applications on SSDs.

In this paper, we aim to explore the behavior of different real containerized database applications with high performance NVMe SSDs and develop a controller to automatically schedule the operation of different containers. The summary of major contributions of this paper is as follows:

- Understanding the performance of write and read intensive containerized workloads for a 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 dockerized applications on high performance NVMe SSDs.
- Developing a docker workload controller based on derived guidelines to minimize the overall throughput, maximize resource utilization and minimize application interference such that the overall operation of simultaneously operating application containers can be sped up.

The remainder of this paper is organized as follows. In Section 2, we describe the related work. In Section 3, we introduce docker container's data storage and our experimental setup. In Section 4, we explore homogeneous docker containers and heterogeneous docker containers. In Section 5, we summarize our results as guidelines and in Section 6, we develop a new controller to packetize workload containers based on derived guidelines. Finally, we draw our conclusion and discuss the future work in Section 7.

2 Related Work

The docker containers of most of the database applications like Redis, MySQL etc. are available for download [1]. The reported increase in performance with the use of docker containers over the virtual machines have attracted many users in a very short time. Charles Anderson introduced docker [2] as container virtualization technology, which is very light-weight compared to virtual machines. Docker containers are becoming the current mainstay mechanism [11], [12], [13] for deploying applications in public and private clouds.

On the other hand, in order to support parallel data intensive applications at application layer with multiple database 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 [14], [15], [16], [17], [18]. But as the $/GB of flash drives kept decreasing the use of SSDs as primary storage became prominent [19], [20]. Now-a-days, the use of SSDs in

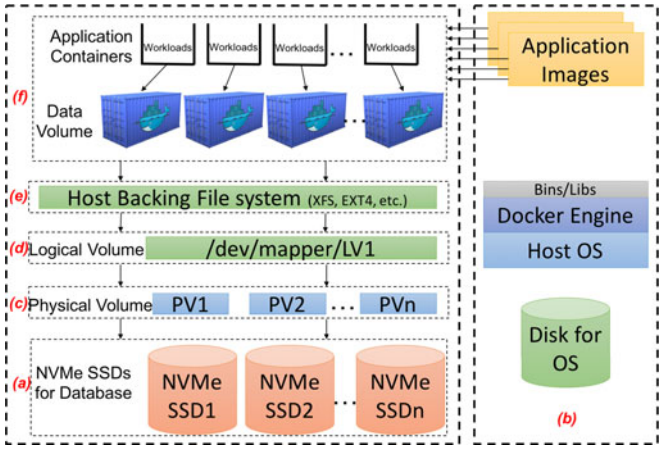


Fig. 1. Containerized system on flash volume of SSDs.

enterprise server and data center is increasing. In [21] and [22], authors explore the SSDs as main storage for database applications. Extensive research has also been performed on enhancement of energy efficiency of database applications using SSDs [23], [24], [25], [26]. 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 [27].

The design of traditional clusters like Mapreduce Hadoop mainly focuses on running massive jobs, but container clusters are often used to run dozens of small instances that need to be organized to optimize data storage and computational power. Thus, the traditional VM schedulers such as Apache YARN [28] and ZooKeeper [29] are not efficient if applied directly to docker containers. On the other hand, the existing container scheduling techniques such as Docker Swarm [30] and Mesos [31] do not control the starting time of different containers on a single host. Moreover, the current approaches of container scheduling aim to maximize resource utilization. However, we found that these approaches actually suffer from application performance interference due to the shared OS resources among multiple containers.

We studied existing literature on well known optimization problems like different variations of bin packing [32], [33], [34], [35], knapsack [36] and job scheduling [37], [38], [39], but did not find a good solution that can be directly applied to our problem. Thus, in this work, we develop a solution to a variation of the classic bin-packing problem with different problem constraints and interference penalties. To the best of our knowledge, this is the first attempt to investigate the performance of such a controller in a system with dockerized applications. In this paper, we aim to explore the behavior of different real containerized database applications with high performance NVMe SSDs and develop a controller to automatically schedule the operation of different containers.

3 Hardware Architecture and Application Layout

3.1 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 (CoW) technique. The fundamental idea of CoW is that if multiple callers ask for resources which are initially indistinguishable, then the pointers to the same resource can be returned. This function can be maintained until a caller tries to modify its "copy" of the resource, at which point a true copy is created. For applications that generate heavy write workloads CoW is not useful. Thus, for write intensive application it is advisable to maintain persistent copy of data. 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. Fig. 1 shows the stacked I/O path of underlying hardware. As shown in Fig. 1f, I/O requests are generated by containerized workloads. Data volume is a specially designated directory within one or more containers that bypasses the docker file system. I/Os on the host can be managed by the host backing file system such as XFS or EXT4 (see Fig. 1e). 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. These logical and physical volumes are constructed over an array of NVMe SSDs, see Figs. 1a, 1c, 1d.

3.2 Experimental Setup

We built a platform consisting of multiple docker containers operating on an array of three enterprise NVMe drives (3TB) in order to provide higher disk bandwidth as shown in Fig. 1a The stack of host OS, docker engine and docker application images is maintained on a separate disk that is different from the one used for storing containerized application files and databases (see Fig. 1b).

Each physical volume that maps the full (i.e., 100 percent) capacity of one SSD, is created through Logical Volume Manager (LVM) (see Fig. 1c). These multiple physical volumes are combined to form a single stripped logical volume using `lvcreate` [40], [41]. The data written to this logical volume is laid out in a stripped fashion across all the disks by the file system. The size of this logical volume is equal to the total capacity of all SSDs. Table 1 gives the detailed hardware configuration of our platform.

We analyze the performance of different workloads of relational database and NoSQL database applications like MySQL, MongoDB, RocksDB, Canssandra, ForestBD etc. We observe that operation of relational database applications

TABLE 1
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	20,480 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

varied significantly from that of NoSQL database applications. The relational database applications performs comparatively higher proportion of random input/output operations (as shown in Fig. 8) resulting in higher memory usage than NoSQL database applications [42], [43], [44], [45]. Thus here, we choose to show and discuss our results using different workloads of MySQL [46] (relation database) and Cassandra [47] (NoSQL database) for our docker performance analysis. These two database applications are not only the most popular relational database and NoSQL database applications, but also widely adopted by companies using docker for production. Respectively, we run the workloads of *TPC-C benchmark* [48] in MySQL container and Cassandras built-in benchmark tool, *cassandra-stress* [49] for our experiments.

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.

4 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 [50] 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 persistent storage devices. Note that everywhere we mention "write workload", we mean update operation. We observe and analyze interesting characteristics while running database workloads in containers. We notice that 8 to 16 simultaneous containers are sufficient to discuss all our interesting observations. Furthermore, we did consider more (> 25) docker containers in our evaluation. But, our experimental results show that the resources (e.g., disk or CPU) have already been saturated when we have $8 \sim 16$ containers simultaneously running in the system, see Figs. 2, 3, 4, 5, 6, 7, 8, 9, 10, and 11. Therefore we discuss our observations by using sufficient number of containers.

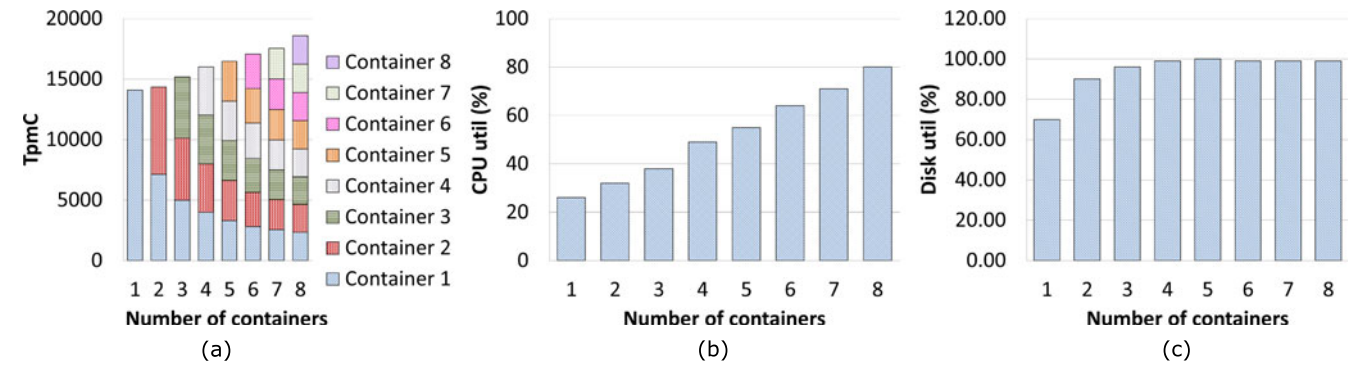


Fig. 2. Homogeneous with MySQL (TPC-C workload). (a) MySQL throughput with evaluation metric as the number of transactions completed per minute (TpmC), (b) CPU utilization, and (c) Disk bandwidth utilization.

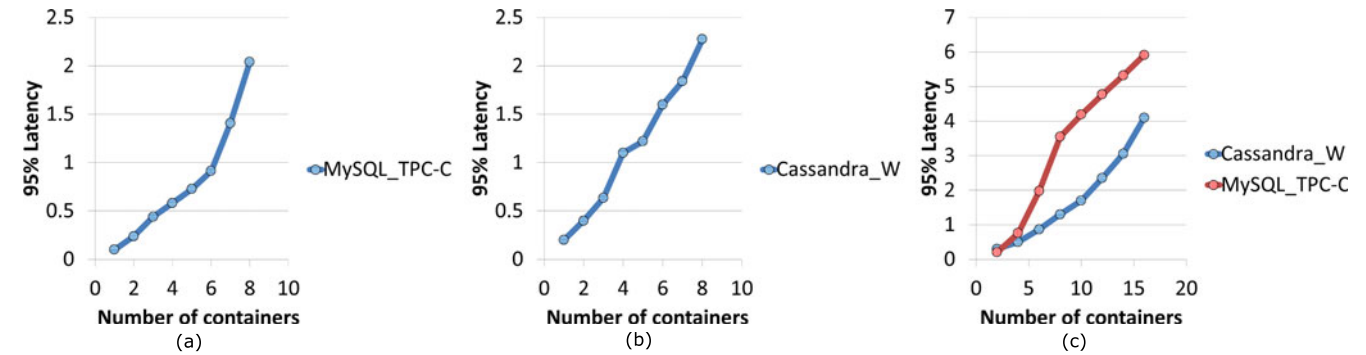


Fig. 3. Average latency for all running containers. (a) Homogeneous MySQL (TPC-C workload), (b) homogeneous Cassandra_W, and (c) heterogeneous MySQL (TPC-C workload) + Cassandra (Cassandra-stress Cassandra_W workload).

TABLE 2
Homogeneous Workload Configuration

	Workload		
MySQL	TPC-C	# Warehouses - 1,250	# Connections - 100
Cassandra_W	Cassandra-stress 100% Writes (i.e., Updates)	# Records - 50 million	Record size - 1 KB
Cassandra_R	Cassandra-stress 100% Reads	# Records - 50 million	Record size - 1 KB

4.1 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 experiment with Cassandra 100 percent 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 percent read workload, where we note an interesting observation of throughput valley. Then, we investigate the reasons 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.

Fig. 2 shows the results of standalone containerized instances of MySQL. The workload configuration of MySQL is given in Table 2. Fig. 2a shows that containerized instances of MySQL (TPC-C workload) scale well with increasing number of containers. We observe that in-spite of the decreasing throughput of each individual containerized instance, the cumulative throughput of MySQL containers increases with increasing number of simultaneously executing containers. The cumulative throughput is the sum of throughput of all simultaneously operating containers. Figs. 2b and 2c shows that disk bandwidth utilization gets saturated at four simultaneous containers, but cumulative throughput keeps increasing with higher CPU utilization on increasing number of simultaneously executing containers. Fig. 3a further presents the average 95 percentile latency across all containers as a function of number of containers under the homogeneous MySQL TPC-C workload. We observe that average 95th percentile latency of simultaneously executing containers increases with increasing number of simultaneous 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 percent writes (Cassandra_W) and 100 percent reads (Cassandra_R). The workload configuration of Cassandra_W and Cassandra_R are given in Table 2. Fig. 4a, shows that containerized instances of Cassandra 100 percent writes scales with increasing number of containers till six simultaneous containers. From Fig. 4b, 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 Figs. 4a and 3b).

Thus, from the above two experiments (i.e., MySQL TPC-C workload and Cassandra_W workload of Cassandra), 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 across all containers would be the number of containers at which CPU gets saturated.

Next, we investigate the performance under the 100 percent read workload using Cassandra_R, see Table 2 for workload details. Fig. 5a 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 from memory and page cache. The combined 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 of a large amount of disk I/O operations. The throughput then becomes limited by disk bandwidth. Fig. 5b shows that the maximum CPU utilization for read-only operations is lower (i.e., 65 percent)

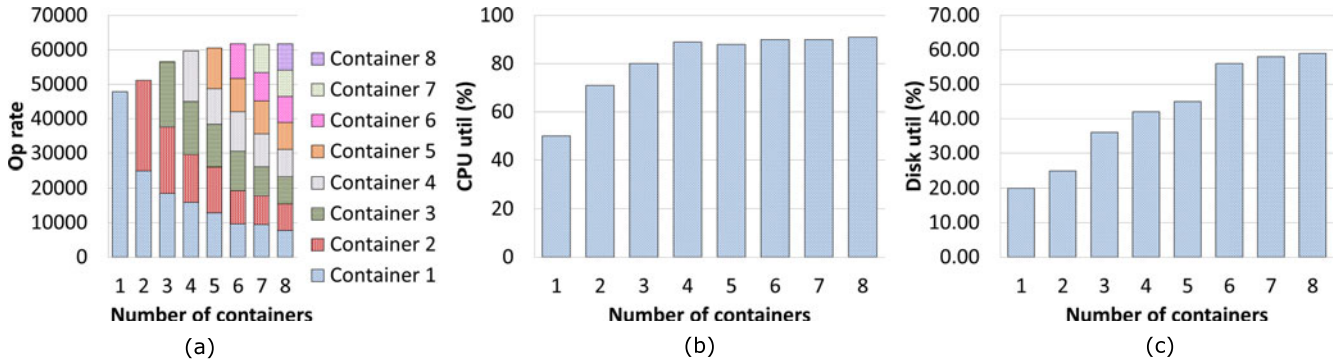


Fig. 4. Homogeneous with Cassandra (Cassandra_W workload). (a) Cassandra throughput, (b) CPU utilization, and (c) Disk bandwidth utilization.

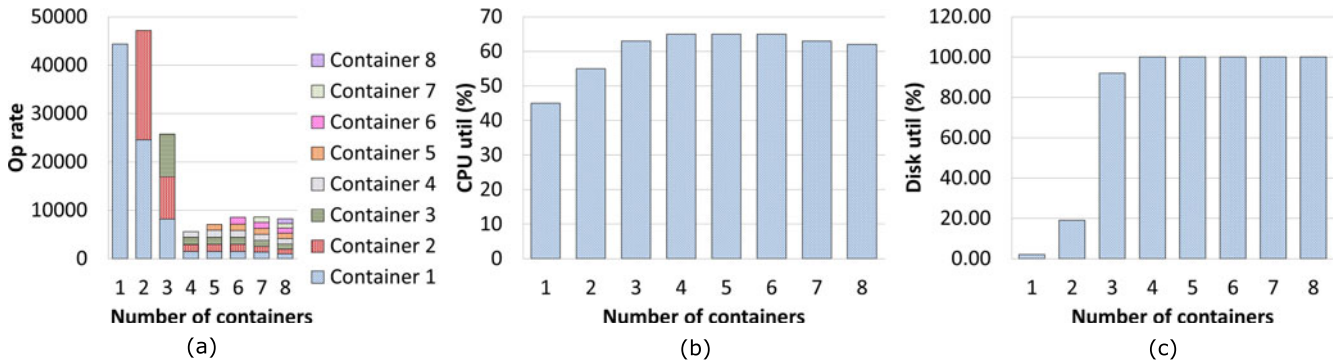


Fig. 5. Homogeneous for Cassandra (Cassandra_R workload). (a) Cassandra throughput, (b) CPU utilization, and (c) Disk bandwidth utilization.

when compared to that of the write-only operations (i.e., 90 percent in Fig. 4b). Fig. 5c further shows that initially for a small number of simultaneous containerized instances most read operations are performed from memory keeping the disk bandwidth utilization low. But, when the throughput valley is observed at four simultaneous containers, most 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 use I/O generating benchmark called FIO (Flexible I/O) [50]. We develop FIO workload to generate random reads that replicates the I/O behavior of Cassandra_R workload. 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.

Fig. 6 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 of size 50 GB. The FIO workload is random read of size 4K, job size of 32 and IO depth of 32. Fig. 6 also shows the throughput valley similar as observed in Fig. 5. Fig. 6 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 the other hand, for write intensive applications, once CPU becomes saturated then increasing the number of containers only increases the latency without any improvement in throughput. Finally, if an application is read intensive and the size of the database that is being accessed by the application executing within the container is small, then the majority of its operations can be performed from page cache and memory. For such cases, it is advisable to limit the number of containers before falling into the throughput valley.

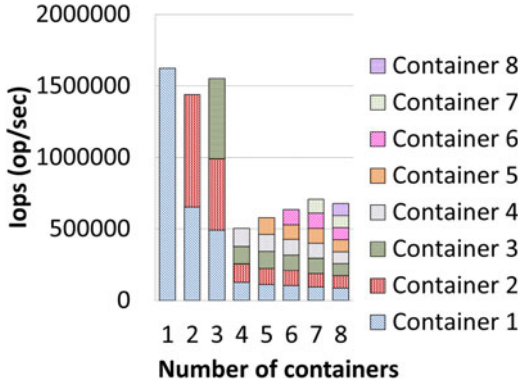


Fig. 6. Homogeneous for FIO (4 KB random read buffered IO).

4.2 Heterogeneous Docker Containers

We explore heterogeneous docker containers by running MySQL and Cassandra applications simultaneously. We make an interesting observation that while operating simultaneously, the throughput of MySQL degrades to more than 50 percent of its standalone throughput observed in homogeneous experiments. But, the throughput of Cassandra degrades only around 16 percent of its standalone throughput observed in homogeneous experiments. Thus, we say that an unfairness behavior is observed in this heterogeneous container mix. Here, fairness means that when N application containers are co-scheduled to run simultaneously, none of the application containers degrade their throughput more than $1/N\%$ of their standalone throughput. 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. 7, shows the results of simultaneously operating containerized instances of Cassandra and MySQL with

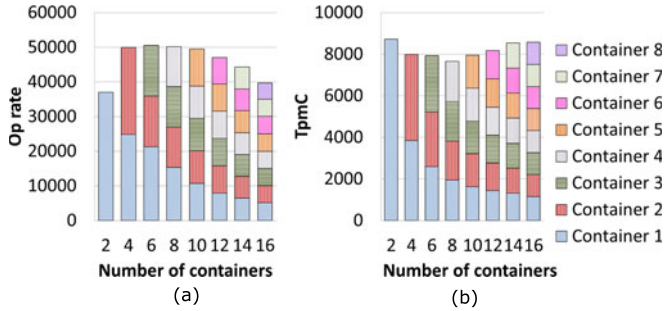


Fig. 7. Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and MySQL (TPC-C workload). (a) Cassandra throughput and (b) MySQL throughput.

Fig. 8. Percentage of sequential and random writes by: (a) Cassandra application and (b) MySQL application.

workload configurations of Cassandra_W and TPC-C as given in Table 2. Figs. 7a and 7b, shows the application throughput of Cassandra and MySQL, respectively. For example, the first bar of Fig. 7a, represents the throughput of Cassandra, when in total two instances, each one of Cassandra and MySQL are running simultaneously. Thus we show results up to 16 containers that includes 8 Cassandra and 8 MySQL containers running simultaneously. We observe the unfair throughput throttle between Cassandra and MySQL. The best throughput of standalone homogeneous Cassandra instances from Fig. 4 is 60K op rate. The best throughput with homogeneous MySQL instances from Fig. 2 is 18K TpmC for 8 concurrent TPC-C containers. But, for heterogeneous experiments of Cassandra and MySQL containers running simultaneously, the best throughput observed for Cassandra and MySQL from Figs. 7a and 7b 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 percent (i.e., from 60K to 50K op rate). But throughput degradation of MySQL containers is around 50 percent (i.e., 18K to 9K TpmC). Thus, MySQL containers sacrifice higher than Cassandra containers when both the application containers are operated simultaneously in the heterogeneous setup. From Fig. 3c, 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, because in such a combination the throughput and performance of some applications were sacrificed drastically.

We anticipate that the reason for such unfair throughput distribution might be the DRAM memory controller, which

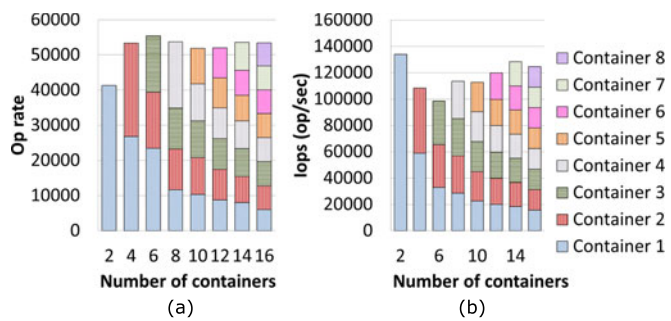


Fig. 9. Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and FIO (4 KB random write). (a) Cassandra throughput and (b) FIO throughput.

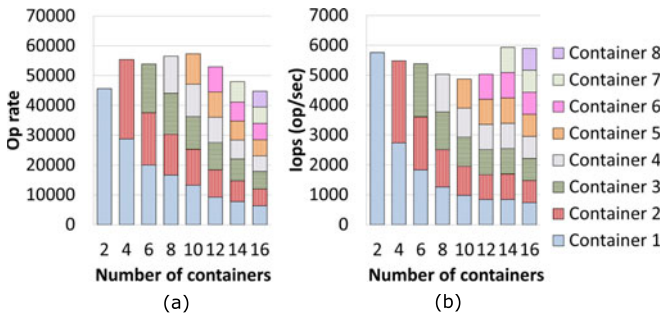


Fig. 10. Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and FIO (128 KB sequential write). (a) Cassandra throughput and (b) FIO throughput.

favors sequential writes more than random writes [51], [52], [53]. Cassandra, which is based on Log Structured Merge (LSM) trees, will always do a lot more sequential writes than MySQL [42], [43], [44], [45]. Figs. 8a and 8b show the proportion of sequential and random writes among total write I/Os in Cassandra and MySQL for Cassandra_W and TPC-C workloads. From Fig. 8a we see that, for any number of simultaneously operating Cassandra containers, the proportion of sequential writes is above 80 percent. On contrary, Fig. 8b shows that, MySQL containers perform a higher proportion of random writes when compared to sequential writes. We also observe that increasing the number of simultaneously operating containers increases proportion of random writes for both these applications (i.e., Cassandra and MySQL). This can be attributed to increased fragmentation during data placement.

To validate unfair throughput throttle due to the nature of application, we conduct following three experiments. First, we run simultaneous instances 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, Fig. 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 Fig. 9b is 134k IOPS. This verifies the initial hypothesis that if containers of an application having a 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 sacrificed terribly with respect to their standalone homogeneous operation throughput.

Second, we present the results of operating Cassandra containers simultaneously with FIO sequential write containers in Fig. 10. We see almost fair throughput throttle under these two applications because containers of both these applications are performing sequential writes. Fig. 10, shows the result of simultaneously operating Cassandra with FIO sequential writes. The best throughput with standalone homogeneous FIO sequential write instances is 6,500 IOPS. From Fig. 10, we observe that the throughput of each application individually degrades only by 10 to 15 percent. Thus as expected, we observe fair throughput throttle.

Third, in order to investigate the behavior of simultaneously operating write intensive and read intensive application containers, we show the results of Cassandra_W

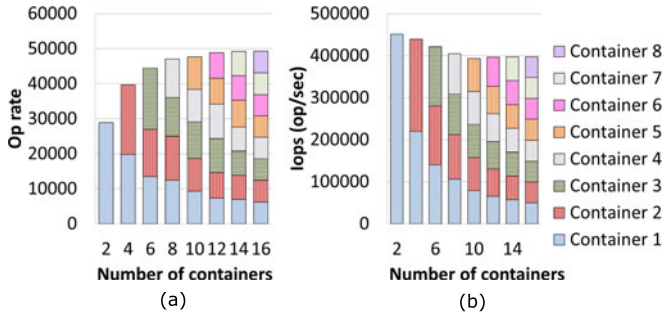


Fig. 11. Heterogeneous: Simultaneous Cassandra (Cassandra_W workload) and FIO (4 KB random read workload). (a) Cassandra throughput and (b) FIO throughput.

containers operating simultaneously with FIO random read containers in Fig. 11. The best throughput with standalone homogeneous FIO random read instances is 450k IOPS. From Fig. 11, we observe that the throughput of both applications individually degrades only by 10 to 15 percent. 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.

5 Implications

In this section we provide the high level design guidelines for homogeneous and heterogeneous containerized docker platform. These are guidelines on how to choose which type of applications to run simultaneously. 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 the working set size of all the application containers is small such that majority of its operations can be performed from page cache, then it is advisable to limit number of containers before falling into throughput valley.

Heterogeneous mix of workloads has exaggerated impact while using Docker when compared to that of VM, due to higher possibility of resource contention in shared resources management performed by Docker. 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 across containers as observed is unfair. In such a mix the throughput of an application performing random writes (or reads) may degrade drastically.
- Co-scheduling containers that execute applications with similar characteristics (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 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 percent when compared to the standalone implementation of respective applications.

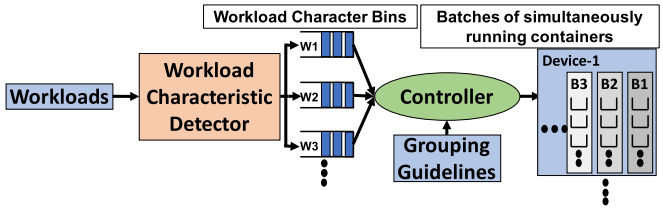


Fig. 12. Controller architecture.

6 Container Controller

Docker containers are lightweight without guest OS, they can be launched when needed. This gives us a good opportunity to dynamically schedule and launch Docker containers at runtime. As shown in the guidelines above, simultaneously operating more containers increases system utilization. But, it is not a good idea to run all containers of different applications simultaneously as it cannot guarantee the minimum make-span (i.e., the overall execution length of all these applications). Launching all workload containers at the same time may increase transaction failures due to resource contention, and thus incur extra time to re-perform those failed transactions. Unfair throughput throttle among containers of different applications might also happen as observed in Section 4. Therefore, in this section we present a new container controller to determine which containers should be operated simultaneously in the same batch under the goal of fully utilizing resources and avoiding severe resource contention. In our design of controller, we assume that the user is aware of all the workload containers which are desired to be scheduled prior to scheduling.

6.1 Architecture

To evaluate the feasibility of our controller, we implemented it as a loadable module in the Linux kernel. Our controller module is plugged on top of the Docker engine to control the Docker container handling functionalities like Docker_start, Docker_wait, Docker_run, Docker_exec, Docker_pause, etc. As shown in Fig. 12, the schematic diagram illustrates the peripheral architecture of the components of our controller. Each application workload, is first fed into the Workload Characteristic Detector (WCD) which is responsible to label each workload based on their characteristics. To obtain the

	A 1	A 2	A 3	. . .	A j	. . .	A (n*m)
A 1	0	1	2		S 1j		3
A 2	1	0	4		S 2j		1
A 3	2	4	0		S 3j		5
:				. . .	:	. . .	
A i	S i1	S i2	S i3		S ij		S i (n*m)
:					:		
A (n*m)	3	1	5		S (n*m) j		0

Fig. 13. Conflict penalty matrix (S).

characteristics of a particular workload, WCD needs to sample this workload by running it for a short period of time (e.g., 5 minutes) on the local host and collect the I/O block trace at the block layer using some trace collection tool (e.g., *blktrace*). The collected I/O block trace is then parsed by using the parser tool (like *blkparse*) for analyzing the workload characteristics. WCD then places this workload into a workload character bin that contains workloads with same characteristics, see Fig. 12. The total number of workload character bins depends on the types of workloads. By leveraging the grouping guidelines that were derived in Section 5, the controller analyzes the penalty of grouping different workloads and then packetizes workloads into optimal batches/bins for each available device. Finally, these independent workload batches are scheduled to run one by one at the application layer.

We give an example of how penalty values are determined by the controller. From design implications, we know that combining read intensive and write intensive workloads is beneficial. Thus, the resultant penalty is set to null. On the other hand, our guidelines show that it is not a good idea to group workloads with a higher fraction of random writes and sequential writes. Thus, positive penalty will be set up for this combination.

The objective of the controller is to packetize workloads from workload character bins into different batches, such that the operation penalty and the number of batches can both be minimized. Here, each batch should contain a group of containerized workloads that will be run at the same time. Furthermore, if more than one logical volume of storage drives are available, then the controller should consider multiple sets of batches for these logical volumes, see Fig. 1.

6.2 Problem Formulation

Now, we present a mathematical formulation of this optimization problem. Suppose there are *n* applications and *m* containers of each application. Thus, there are a total of $n * m$ containers. Each of these containers can only belong to one of the workload character bins. Without loss of generality, we assume that *n* and *m* are both positive integers. Running workloads A_i and A_j each from different character bins simultaneously may have some penalty depending upon their characteristics. We use S_{ij} to denote the conflict penalty between A_i and A_j, if they are run simultaneously in the same running batch.

Essentially, we build a penalty matrix which consists of different relative penalty weights (i.e., S_{ij}) to differentiate performance interference in the order of severity when running two containers simultaneously. In more detail, we configured the relative weights of our penalty matrix based on the experimental implications that we derived in Section 5 and used different values of S_{ij} (e.g., 0, 1, 2) to indicate the relative penalty when running container i and container j at the same time. For example, the penalty weight to run a read intensive workload with another write intensive workload may be set to 0 because no performance interference occurs and this implies a good decision for better resource utilization. In contrast, the penalty to run a sequential write workload (like 100 percent write Cassandra workload) and a random write workload (like 100 percent write MySQL workload) can be relatively high (e.g., with the weight value of 2) because we can observe performance degradation for that random write workload is dramatically significant. Thus, we can build a matrix (see an example in Fig. 13) to list possible conflict penalties of all given containers. The size of the penalty matrix (S) is $n * m$ x $n * m$.

Each of the given containers only runs once, thus each belongs to one and only one running batch. A naive case is to run a single container in each running batch such that we have maximum number of running batches which is equal to the total number of containers (i.e., $n * m$). Clearly in this case, the system resources might not be fully utilized and the execution time of these applications cannot be reduced. Another naive case is to run all the containers in the same batch. In this case all the applications will be competing for available resources, which might lead to contention and throttling. Therefore, controlling the concurrency of application containers is important. Our controller has mainly two objectives, i.e., both the number of batches and the conflict penalties (e.g., unfair throughput throttle) should be minimized. Eq. (1) gives the total conflict penalty (T_p) and Eq. (2) shows the total number of batches (N_z), where S_{ij} is penalty cost between items A_i and A_j when grouping them in same batch, P_{ik} is a binary variable which equals 1 if item A_i is packed in batch k, otherwise 0, and z_k is a binary variable which equals 1 if batch k is used, otherwise 0

$$T_p = \sum_{k=1}^{n*m} \sum_{j=1}^{n*m} \sum_{i=j}^{n*m} S_{ij} P_{ik} P_{jk} \tag{1}$$

$$N_z = \sum_{k=1}^{m} z_k. \tag{2}$$

We transform this problem into an objective function derived to minimize both T_p and N_z. The goal of our container controller is to minimize the interference penalty cost (T_p) and the makespan (N_z), i.e., the overall execution time of all I/O workloads. We defined these two performance metrics in Eqs. (1) and (2) and aimed to minimize them (i.e., T_p and N_z) in an objective function. However, as we know, the unit and value ranges of T_p and N_z are different. To solve the problem, we further used the MIN-MAX normalization method to scale the values of T_p and N_z both in the [0, 1] range and considered the same weight for both metrics by summing two normalized items, see Eq. (3). We remark that different weights can also be adopted in the objective function. Since our goal is to minimize T_p and N_z, we built the objective function as the maximization of the function framed by addition of these two normalized items. The variables that are used to describe the mathematical formulation of our optimization problem are mentioned in Table 3.

TABLE 3
Objective Function Variables (Para.- Parameters)

Para.	Description
n	Number of applications
m	Number of containers of each application
A_j	Application container j
z_k	A binary variable which equals 1 if batch k is used, otherwise 0
P_{jk}	A binary variable which equals 1 if item A_j is packed in batch k, otherwise 0
S_{ij}	Penalty cost between items A_i and A_j when grouping them in same time batch
φ	Maximum number of application containers that are allowed to run in same batch
Max_p	Maximum possible penalty for a given S by running all workloads in a single batch
Min_p	Minimum possible penalty for a given S by running only one workload in each batch (Min_p = 0)
Max_z	Maximum possible number of batches (Min_z = n*m)
Min_z	Minimum possible number of batches (Min_z = 1)

The constraints for solving the objective function are listed in Eqs. (4), (5), and (6). Constraint C1 indicates if workload A_i runs in batch k. Constraint C2 ensures that each workload can be run in one and only one batch. Lastly, depending on the platform specification, a user might want to limit the maximum number of simultaneous containers, which is then enforced by constraint C3. Thus, the goal of the controller is to determine the best matrix P such that we can obtain the maximum value of the optimization function f. P then gives the scheduling information, i.e., workload A_i can be run in batch k if $P_{ik} = 1$ and vice-verse. For example, as shown in Fig. 14, item P_{31} is 1, representing that application container A_3 is run in the first time batch.

Objective:

Find the best P such that

$$f = Max\left\{\left[\frac{Max_p - T_p}{Max_p - Min_p}\right] + \left[\frac{Max_z - N_z}{Max_z - Min_z}\right]\right\}. \quad (3)$$

Constraints:

$$C1 : P_{ik} \in \{0, 1\} \; \forall \; workload \; A_i \; and \; batch \; k \quad (4)$$

$$C2 : \sum_{k=1}^{n*m} P_{ik} = 1 \; \forall \; workload \; A_i \quad (5)$$

$$C3 : \sum_{i=1}^{n*m} P_{ik} \leq \varphi \; \forall \; batch \; k. \quad (6)$$

6.3 Techniques for Solving Optimization Problem of Controller

The simplest approach towards solving this convex optimization problem is *brute force* evaluation of the objective function by generating all possible P. This method definitely gives us the best possible batches of workloads, which ensures the minimum penalty and minimum number of batches. But, due to the high complexity of the problem, such a brute force approach is intractable for solving large-scale instances that often arise in practice. Moreover, it is high time consuming. So, we solve our optimization function using two linear programming algorithms by implementing constrained matrix optimization in Matlab. We use *Interior Point (IP)* and *Standard Quadratic Programming (SQP)* algorithms provided by Matlab. All other optimization algorithms provided by Matlab (version R2017a - academic use) are not applicable to solve our optimization problem. Using each of the above algorithms, we further adapt two possible options of local and global search which results in local[1] and global maxima.[2] In our solution, we consider maximum number of supported simultaneous number of application containers (φ) as $n * m$.

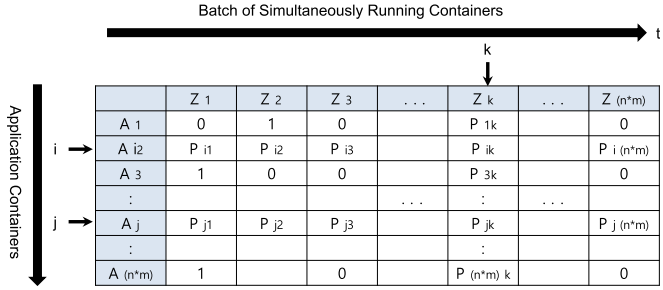


Fig. 14. Bin packing matrix (P): Batches of simultaneously running containers.

TABLE 4
Controller Workload

Application	Number of Containers
MySQL	5
Cassandra_W	5
FIO Random Read 4 KB	5
FIO Random Write 4 KB	5
FIO Sequential Write 128 KB	5

6.4 Results

Next, we present the results of the controller, which uses the optimization objective function to schedule workload containers. Here, we compare the performance results as well as the overhead of using different optimization algorithms. Our platform has a single logical volume striped over three SSDs to provide 3 TB storage space. We refer the readers to Section 3.2 for detailed platform configurations. As shown in Table 4, we use five different applications as representative of diverse I/O behaviors and then configure five workloads with different parameter settings (e.g., number of updates, number of threads) for each application. More detailed information of two database applications (i.e., MySQL and Cassandra) can also be found in Table 2. For each workload, we have the fixed record size (e.g., 1 KB)

1. A real-valued function f defined on a domain X, is said to have a local (or relative) maximum point at the point x^* if there exists some $\epsilon > 0$ such that $f(x^*) \geq f(x)$ for all x in X within distance ϵ of x^*.

2. f has a global (or absolute) maximum point at x^* if $f(x^*) \geq f(x)$ for all x in X.

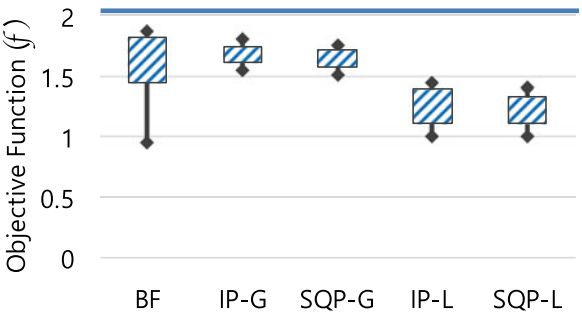


Fig. 15. Objective function evaluation with different optimization algorithms. (BF - Brute Force, IP-G - Interior Point using Global search, SQP-G - Standard Quadratic Programming using Global search, IP-L - Interior Point using Local search, and SQP-L - Standard Quadratic Programming using Local search).

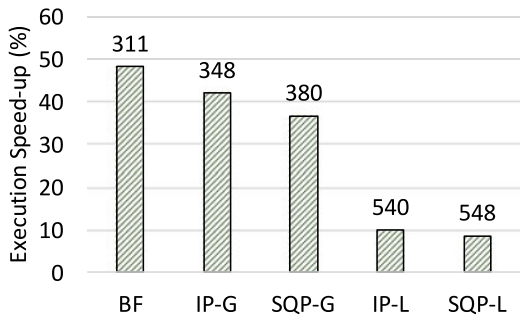


Fig. 17. Speed-up in execution of containers by using controller with baseline as a case without controller, where all application containers are launched at once.

but change the number of records. As a result, we can have different record sizes for each workload (even from the same application) ranging from 50 to 100 GB.

Thus, we have five containers for running each application and totally 25 containers in this experiment to evaluate our controller. The goal of our controller is to determine optimal running batches of these 25 containers. Actually, we did consider to use more containers in our experiment. But, our experimental results in Section 4 show that the resources (e.g., disk or CPU) have already been saturated when we have $8 \sim 16$ simultaneously running containers in the system, see Figs. 2, 4 and 5. Performance improvement can be obtained by the controller until all the resources are utilized to their maximum. After saturation, further increasing the number of containers does not give better performance. Given that, we consider to use up to 25 different containers (or I/O intensive database workloads) in this experiment. Another reason to limit our experiment with 25 container is due to the high overhead for data loading and warming up for the related workload database. Specifically, the average database size of each workload in a container is approximately $50 \sim 100$ GB. It took us about 1 week to setup the databases for total 25 containers.

Fig. 15 shows the results of the objective function (i.e., Eq. (3)) in terms of maximum, minimum and one standard deviation from the mean under different optimization techniques. As the goal of our objective function is to maximize the values, the higher values are better. The objective function value on the y axis of Fig. 15 is the result of evaluating the optimization function mentioned in Eq. (3) considering the constraints given in Eqs. (4), (5), and (6). The x axis of Fig. 15 shows the results using different methods of optimization discussed in Section 6.3. We can see that the Brute Force [3] (*BF*) method obtains both the best maxima and the least maxima because it tries all possible P matrices. Where each P matrix would represent a possible combination of the workload batches. Both, the interior point and the standard quadratic programming approaches use approximated derivative solver. Thus, the result is slightly different for each run, even for the same input parameters. Therefore, for the remaining four methods, we get their results each over 50 independent runs and over 50 iterations in each run.

We further notice that although the brute force method gives us the best maxima for our objective function, the operation of BF is very time consuming as the brute force algorithm evaluates all possible combinations. The complexity of this brute force method is $O((nm)!)$ that is only fine when the values of both n and m are small.

We observe that both global search methods, i.e., *IP-G* and *SQP-G*, also achieve a good maxima of our objective function. The maximum values by these algorithms are very close to the maximum one from the brute force evaluation, see Fig. 15. More importantly, these global evaluation techniques consume much less time than the brute force method. Each optimization method consumes some time to solve the problem known as overhead of that method. As the brute force method evaluates all possible test cases, so it has the maximum possible overhead. Fig. 16, shows the normalized overhead of four optimization algorithms compared with the brute force one. We can see that two global search algorithms use less than 4 percent of time used by the brute force method. The two local search algorithms consume even less overhead. However, they cannot provide the maxima as well as the global search approaches, see *IP-L* and *SQP-L* in Figs. 15 and 16. Note that the controller overhead shown in Fig. 16 includes the overhead by all the components of the controller shown in Fig. 12, like labelling time by *WCD*, time consumed to optimize objective function, and management delays.

To further investigate the effectiveness on actual performance, we conduct real experiments by running the 25 containers of five different applications as shown in Table 4 under different container batches results obtained by those five optimization algorithms. Fig. 17 shows the speed-up in

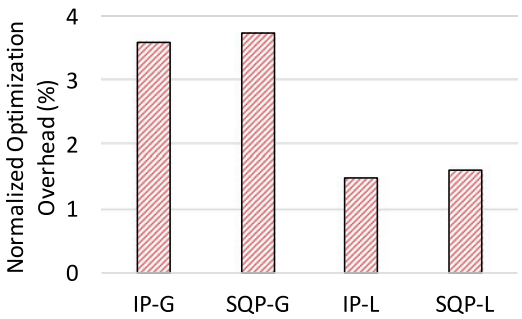


Fig. 16. Normalized total controller overhead with baseline as time required by brute force optimization.

3. The detailed algorithm we developed to process the brute force method by hashing can be found in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TMSCS.2018.2801281.

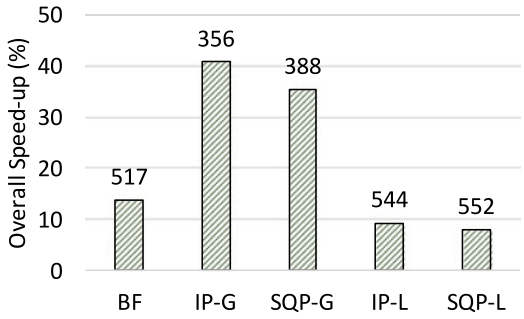


Fig. 18. Overall speed-up including optimization by using controller, with baseline as a case without controller, where all application containers are launched at once.

execution time of all the containers by using results from the controller w.r.t. baseline where all application containers are launched at once without any controller. In our evaluation, the execution time under the baseline case is about 600 hours. We also gave the actual execution time (hours) under each controller on top of the corresponding bar in Fig. 17. With a proper scheduling of containers, the overall execution of all workloads could be faster than when compared to the case of launching all application containers at once without any scheduling efforts. We observe that BF gives the best quality of optimized batches, showing the highest execution speed-up. The global search methods (i.e., *IP-G* and *SQP-G*) achieve very comparable speed-up results to BF. The local searches using IP-L and SQP-L give only 10 percent of speed-up in execution of workloads.

Finally, we evaluate the overall effectiveness by considering both execution speed-up and time overhead as operations required by controller may also consume some additional time. Fig. 18 shows the speed-up in overall runtime (i.e., application execution time plus the time consumed by controller to solve objective function) under five optimization algorithms. The baseline is still the case without the controller. Again, the values on top of each bar indicate the actual overall runtime in hours for each algorithm. Obviously, the BF approach loses its superiority due to its high overhead. In contrast, the two global search approaches become the best with $35 \sim 40$ percent speed-up in overall.

7 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 explore important phenomenon like throughput valley in a containerized docker environment. Furthermore, our work showed that it is not advisable to simultaneously run the containers of applications that perform sequential writes (or reads) and random writes (or reads). We presented some design guidelines that are implicated from performance engineering carried out in this paper and then developed a workload controller to effectively schedule the launching time of all application containers. We showed that our controller can decide the batches of docker containers that can run simultaneously such that the overall execution time can be reduced and workload interference (like unfair throughput throttle) can be avoided. In the future, we plan to embed our docker controller with open source docker engine installation such that user can easily use it just by selecting an auto-scheduling option.

Acknowledgments

This work was partially completed during Janki Bhimani's internship at Samsung Semiconductor Inc., and was partially supported by US 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]. Available: https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=728586136. Accessed on: Jul. 12, 2016.

[2] C. Anderson, "Docker software engineering," *IEEE Softw.*, vol. 32, no. 3, p. 102–c3, 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, 2015, Art. no. e1273.

[4] J. Fink, "Docker: A software as a service, operating system-level virtualization framework," *Code4Lib J.*, vol. 25, p. 29, 2014.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 171–172.

[6] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization versus containerization to support PaaS," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2014, pp. 610–614.

[7] A. Olbert, et al., "Managing multiple virtual machines," U.S. Patent 10 413 440, 2003.

[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," *Adv. Sci. Technol. Lett.*, vol. 66, pp. 105–111, 2014.

[10] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," *Haifux*, vol. 186, May 2013.

[11] Q. Xu, M. Awasthi, K. Malladi, J. Bhimani, J. Yang, and M. Annavaram, "Performance analysis of containerized applications on local and remote storage," in *Proc. Int. Conf. Massive Storage Syst. Technol.*, 2017.

[12] Q. Xu, M. Awasthi, K. Malladi, J. Bhimani, J. Yang, and M. Annavaram, "Docker characterization on high performance SSDs," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2017, pp. 133–134.

[13] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud," in *Proc. 21st IEEE High Perform. Extreme Comput. Conf.*, 2017, pp. 1–7.

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

[15] L.-P. Chang, "Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2008, pp. 428–433.

[16] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," in *Proc. USENIX Conf. File Storage Technol.*, 2010, pp. 101–114.

[17] H. Jo, Y. Kwon, H. Kim, E. Seo, J. Lee, and S. Maeng, "SSD-HDD-hybrid virtual disk in consolidated environments," in *Proc. Eur. Conf. Parallel Process.*, 2009, pp. 375–384.

[18] 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," *Proc. VLDB Endowment*, vol. 5, no. 10, pp. 1076–1087, 2012.

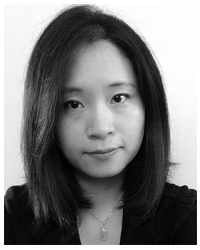
[19] R. Chin and G. Wu, "Non-volatile memory data storage system with reliability management," U.S. Patent 12 471 430, May 25, 2009.

[20] A. Leventhal, "Flash storage memory," *Commun. ACM*, vol. 51, no. 7, pp. 47–51, 2008.
[21] Y. Wang, K. Goda, M. Nakano, and M. Kitsuregawa, "Early experience and evaluation of file systems on SSD with database applications," in *Proc. IEEE 5th Int. Conf. Netw. Archit. Storage*, 2010, pp. 467–476.
[22] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: Analysis of tradeoffs," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 145–158.
[23] D. Schall, V. Hudlet, and T. Härder, "Enhancing energy efficiency of database applications using SSDs," in *Proc. 3rd C* Conf. Comput. Sci. Softw. Eng.*, 2010, pp. 1–9.
[24] S. Park and K. Shen, "A performance evaluation of scientific I/O workloads on flash-based SSDs," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, 2009, pp. 1–5.
[25] S. Boboila and P. Desnoyers, "Performance models of flash-based solid-state drives for real workloads," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–6.
[26] 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 *Proc. Symp. VLSI Circuits*, 2012, pp. 134–135.
[27] 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 *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–11.
[28] V. K. Vavilapalli, et al., "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 5.
[29] A. Takefusa, H. Nakada, T. Ikegami, and Y. Tanaka, "A highly available distributed self-scheduler for exascale computing," in *Proc. 9th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2015, Art. no. 56.
[30] Docker Swarm. Accessed on: Jan. 20, 2018.
[31] B. Hindman, et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.
[32] K. Li, H. Liu, Y. Wu, and X. Xu, "A two-dimensional bin-packing problem with conflict penalties," *Int. J. Prod. Res.*, vol. 52, no. 24, pp. 7223–7238, 2014.
[33] N. Karmarkar and R. M. Karp, "An efficient approximation scheme for the one-dimensional bin-packing problem," in *Proc. 23rd Annu. Symp. Found. Comput. Sci.*, 1982, pp. 312–320.
[34] A. Scholl, R. Klein, and C. Jürgens, "BISON: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem," *Comput. Operations Res.*, vol. 24, no. 7, pp. 627–645, 1997.
[35] R. Sridhar, M. Chandrasekaran, C. Sriramya, and T. Page, "Optimization of heterogeneous bin packing using adaptive genetic algorithm," in *Proc. IOP Conf. Series: Mater. Sci. Eng.*, 2017, Art. no. 012026.
[36] B. Schulze, L. Paquete, K. Klamroth, and J. R. Figueira, "Bi-dimensional knapsack problems with one soft constraint," *Comput. Operations Res.*, vol. 78, pp. 15–26, 2017.
[37] T. K. Ghosh, S. Das, S. Barman, and R. Goswami, "A comparison between genetic algorithm and cuckoo search algorithm to minimize the makespan for grid job scheduling," in *Proc. Int. Conf. Comput. Intell.*, 2017, pp. 141–147.
[38] M. Paul, R. Sridharan, and T. R. Ramanan, "A multi-objective decision-making framework using preference selection index for assembly job shop scheduling problem," *Int. J. Manage. Concepts Philosophy*, vol. 9, no. 4, pp. 362–387, 2016.
[39] H. Afsar, P. Lacomme, L. Ren, C. Prodhon, and D. Vigo, "Resolution of a Job-Shop problem with transportation constraints: A master/slave approach," *IFAC-PapersOnLine*, vol. 49, no. 12, pp. 898–903, 2016.
[40] M. Hasenstein, "The logical volume manager (LVM)," White paper, 2001.
[41] G. Banga, I. Pratt, S. Crosby, V. Kapoor, K. Bondalapati, and V. Dmitriev, "Approaches for efficient physical to virtual disk conversion," U.S. Patent 13 302 123, 2013.
[42] How to fix MySQL high memory usage?. Accessed on: Sep. 04, 2017.
[43] J. Schad, J. Dittrich, and J.-A. Quian é-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 460–471, 2010.
[44] How MySQL Uses Memory. Accessed on: Sep. 04, 2017.
[45] D. McCreary and A. Kelly, *Making Sense of NoSQL*. Shelter Island, NY, USA: Manning, 2014, pp. 19–20.
[46] A. MySQL, "MySQL database server," Internet WWW page, 2004. [Online]. Available: http://www. mysql. com
[47] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[48] Francois, W. Raab, A. Kohler, and Shah, *MySQL TPC-C Benchmark*. [Online]. Available: http://www.tpc.org/tpcc/detail.asp. Accessed on: Sep. 6, 2016.
[49] *Cassandra-Stress Benchmark*. [Online]. Available: https://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsCStress_t.html. Accessed on: Sep. 6, 2016.
[50] *FIO - Flexible I/O Benchmark*. [Online]. Available: http://linux.die.net/man/1/fio. Accessed on: Sep. 7, 2016.
[51] L. R. Mote Jr, "Memory controller which executes read and write commands out of order," U.S. Patent 5 638 534, Jun. 10, 1997.
[52] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *ACM SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 128–138, 2000.
[53] R. J. Bowater, S. P. Larky, J. C. S. Clair, and P. G. Sidoli, "Flexible dynamic memory controller," U.S. Patent 5 301 278, Apr. 5, 1994.

Janki Bhimani received the BTech degree in electrical and electronics engineering from Gitam University, India, in 2013 and the MS degree in computer engineering from Northeastern University, in 2014. She is working toward the PhD degree with Prof. Ningfang Mi at Northeastern University, Boston. Her current research focuses on performance enhancement of parallel computing heterogeneous platforms, containerized cloud, and multi-stream flash storage.

Zhengyu Yang received the BSc degree in communication engineering from Tongji University, Shanghai, China, and the MSc degree in telecommunications from the Hong Kong University of Science and Technology. He is working toward the PhD degree at Northeastern University, under the supervision of Prof. Ningfang Mi. His research interests include cache algorithm, deduplication, cloud computing, datacenter storage and scheduler, and spark optimization.

Ningfang Mi received the BS degree in computer science from Nanjing University, China, the MS degree in computer science from the University of Texas at Dallas, Texas, and the PhD degree in computer science from the College of William and Mary, Virginia. She is an associate professor at Northeastern University, Boston. Her current research interests include capacity planning, MapReduce/Hadoop scheduling, cloud computing, resource management, performance evaluation, workload characterization, simulation, and virtualization.



Jingpei Yang received the PhD degree in computer science from the University of California, Santa Cruz. She is a staff engineer in the Samsung Memory Solutions Lab focusing on optimizing data center solutions for the cutting edge storage features and developing forward looking platforms for enterprise flash products. Prior to Samsung, she worked at Fusion-IO building efficient systems for emerging Storage Class Memories technologies.

Qiumin Xu received the bachelor's degrees in microelectronics and statistics from Peking University, in 2011 and the master's degrees in electrical engineering and computer science from the University of Southern California. She is currently working toward the PhD degree in the Ming Hsieh Department of Electrical Engineering, University of Southern California. Her research interests includes computer architecture and storage systems.

Manu Awasthi received the BTech degree from the Indian Institute of Technology, Varanasi, India, and the PhD degree in computer science from the University of Utah. He is an assistant professor at the Indian Institute of Technology Gandhinagar. His research interests include computer architecture with a focus on memory and storage systems.

Rajinikanth Pandurangan received the master's degree from Annamalai University India. He is a senior staff engineer at the Samsung Memory Solutions Lab R&D. He is focusing on enabling next generation features for Samsung's enterprise and consumer class flash SSD products. Prior to Samsung, he worked at PMC-Sierra and Adaptec Inc., designing and building storage virtualization and RAID products.

Vijay Balakrishnan received the bachelor's degree in engineering from the Birla Institute of Technology, Mesra, India, and the MS degree from the University of Cincinnati. He is the director of the Datacenter Performance and Ecosystem Team, Samsung Memory Solutions Lab. building efficient and high-performance solutions for the datacenter. He works on solutions that leverage Samsung's industry leading flash, SSD, and DRAM technologies. Prior to Samsung, he worked at Sun Microsystems and Microsoft designing and building high-performance systems.