Performance Analysis of Containerized Applications on Local and Remote Storage

Qiumin Xu*, Manu Awasthi†, Krishna T. Malladi‡, Janki Bhimani§, Jingpei Yang‡ and Murali Annavaram*
* *Ming Hsieh Department of Electrical Engineering, University of Southern California*
{*qiumin, annavara*}*@usc.edu*
† *Department of Computer Science and Engineering, Indian Institute of Technology - Gandhinagar, manua@iitgn.ac.in*
‡ *Samsung Semiconductor Inc.* {*k.tej, jingpei.yang*}*@samsung.com*
§ *Department of Computer Engineering, Northeastern University, bhimani.j@husky.neu.edu*

***Abstract*—Docker containers are becoming the mainstay for deploying applications in cloud platforms, having many desirable features like ease of deployment, developer friendliness, and lightweight virtualization. Meanwhile, storage systems have witnessed tremendous performance boost through recent innovations in the industry such as Non-Volatile Memory Express (NVMe) and NVMe Over Fabrics (NVMf) standards. However, the performance of docker containers on these high-speed contemporary SSDs has not yet been investigated. In this paper, we first present a characterization of the performance impact among a wide variety of the available storage options for deploying Docker containers and provide the configuration options to best utilize the high performance SSDs. We then provide the first of its kind characterization results of a Dockerized NoSQL database on an NVMe-over-fabrics prototype and show that its performance matches closely to that of direct attached storage. Finally, we provide experimental results on scaling the performance of NVMf to multiple nodes and present the challenges and projections for future storage system design.**

1. Introduction

The last half-a-decade has witnessed a massive shift in the way that software is developed and deployed. With the proliferation of data centers, a number of applications have moved to new hosting models, such as Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) that forgo cumbersome software installations on individual machines. This freedom has also led to the development of micro-services and complex distributed applications. Applications are broken down into smaller silos, that interact with each other to compose an overall system that can be deployed across many environments. These environments can have different OS distributions, kernel versions, compilers, shared library versions and other dependencies that are required for program execution. Making sure all environments have the right set of dependencies is a tough challenge and is often known as the *dependency hell*. This can offset the advantages of free mobility and distributed nature of applications over a large, monolithic code base.

Containers were proposed as a solution to alleviate dependency issues [25]. Containers are operating system constructs for providing the *lightweight virtualization*. Different from full virtualization which needs to emulate an entire set of hardware components that it is configured with, containers allow multiple user space instances to share the same host OS while allowing for multiple applications to run in isolation, in parallel, on the same host machine. In this way, containers can achieve near bare metal performance by avoiding the overheads in virtualizing entire systems. Although there are multiple implementations of the container technology [6], we choose Docker, which is the most popular implementation and is heavily used in production.

One of the key components for efficient execution of any container (or even virtual machine) is the storage hierarchy. Containers rely heavily on the notion of an image. The container image holds the state of the container, including all the necessary application binaries, input files and configuration parameters to run an application within the container. Efficient representation of the image on the underlying storage is critical for performance. The advent of SSD-based storage provides new opportunities to improve the performance of a container. In particular, Non-volatile memory express (NVMe) [27] is a logical device interface specification for accessing non-volatile storage media attached via PCI Express (PCIe) bus. Many latest server platforms have already integrated NVMe interface support, including Supermicro NVMe Platform, Dell PowerEdge R920 and NVIDIA DGX-1 [32], [34], [9]. With growing number of high performance server platforms equipped with NVMe interface, NVMe SSDs are promising to play an important role in data centers in the near future. NVMe-over-fabrics (NVMf) is a new remote storage technique which allows the high performance NVMe interface to be connected to RDMA-capable networks. Coupled with the new Ethernet and InfiniBand speeds which now top out at 100Gb/s, NVMf is promising to radically change storage over the network [13].

As the world is shifting the bulk of its storage needs on to high performance SSDs, it is essential to understand the performance implications of all possible Docker usage scenarios and the related design space to fully utilize the high performance storage. However, there have been very

few studies on the performance tradeoffs of Docker on storage systems, and none of them has covered the multiple storage options exported by the Docker framework or their effect on NVMe SSDs. As far as we know, we are the first paper presenting characterization results on NVMf.

To that end, we make the following contributions:

- We provide a comprehensive characterization of container storage drivers to evaluate all possible combinations of persistent and non-persistent drivers for Docker ecosystem.
- We provide characterization results for an NVMe over fabrics implementation for various storage options and show that Dockerized applications can achieve similar performance between local and remote storage for both synthetic workloads and real world database.
- We identify the critical resource bottlenecks for scaling NoSQL databases and then optimize the resource allocation strategy for higher system throughput.

The remainder of the paper is organized as follows. Section 2 explains the benefit of using containers, NVMe and NVMf SSDs. Section 3 and Section 4 discuss a wide variety of the available storage options for deploying Docker containers. Section 5 describes our evaluation methodology. Then we present a deep-dive comparative analysis of different storage options for local and remote storage. Section 6 analysis the performance and scaling nature of Dockerized real-world NoSQL databases using the best storage option obtained from Section 5. We discuss possible system optimizations in Section 7. Section 8 describes the related work and we conclude in Section 9.

2. Benefits of Containers, NVMe and NVMf

The performance of application could be dictated by the choice of virtualization technology. Virtual machine virtualizes every resource from the CPU to memory to storage through a virtual machine manager called *hypervisor*. A hypervisor is typically run on top of a host OS or sometimes on top of the native, bare metal hardware. The virtual machine then runs on top of the hypervisor. Each guest VM will have its own guest OS, and thus isolated from other guest machines. However, this feature comes at a performance cost since there are high overheads involved in virtualizing the entire systems.

On the other hand, a Docker container comprises just the application along with other binaries and libraries. As shown in Figure 1(a), for virtual machine, each virtualized application includes an entire guest OS ($\sim 10s$ of GB). In Figure 1(b), a docker container shares the host OS with other containers but appears isolated through Docker Engine. This approach is much more portable and efficient.

To leverage Docker and containerization for scaling out hyper-scale databases on cloud hosts, a fast back-end storage is desired. Legacy hard-drives that read one block at a time are becoming a performance bottleneck for data intensive applications. Built with multiple channels and Flash chips, SSDs are inherently parallel and are able to provide orders of magnitude higher internal bandwidth compared to hard drives. Historically, SSDs conformed to legacy interfaces, like SATA, SCSI and Serial Attached SCSI (SAS), and needed a slow clock on-board platform controller hub (PCH) or Host Bus Adapter (HBA) to communicate with the host system (Figure 1(a)). Over time, those legacy interfaces have become a bandwidth bottleneck and limit the performance of SSDs. The orders of magnitude higher internal bandwidth capability has driven the transition from SATA to a scalable, high bandwidth and low-latency I/O interconnect, namely PCI Express (PCIe) as shown in Figure 1(b). NVMe standardizes PCIe SSDs and was defined to address inefficiency of legacy protocols, enable standard drivers and inter-vendor interoperability. Experimental results show that an NVMe SSD is able to deliver up to 8.5x performance for running Cassandra compared to a SATA SSD [37]. However, NVMe drives are still underutilized, in terms of both bandwidth and IOPS, when running a single instance of Cassandra without considering scaling. Scaling in NoSQL databases can be made easier with containerization. In order to deploy multiple, concurrent instances of NoSQL (or any other datacenter) application, we just need to deploy multiple instances of the containerized workload on the server. Scaling applications to incorporate multiple instances of a workload on the same machine natively is harder since we have to address multiple issues like port conflicts and other networking issues etc. Also distributing resources between applications is made easier through cgroups, which also helps to scale (and is much harder to do natively). The latter part of this work will explore best strategies for deploying and scaling *multiple* workload instances on cloud hosts using both Docker containers and high performance SSDs.

Furthermore, commodity networking has become cheaper and delivers higher bandwidths and lower latencies, which has allowed for the rise of Non-Volatile Memory Express (NVMe) over Fabrics [13]. NVMe over Fabrics shares the same base architecture and most of the code as the existing NVMe over PCIe implementation, but allows the simple implementation of additional fabrics [19]. This end-to-end NVMe semantics eliminates unnecessary protocol translations and retains NVMe efficiency and performance over network fabrics [12]. As illustrated in Figure 1(c), utilizing the NVMf protocol, the I/O requests are directed to remote storage server through a high-speed network connection. NVMe-over-Fabrics is set to deliver fast connectivity to remote NVMe devices with up to 10 microseconds of additional latency over a native NVMe device inside a server [13]. There are two types of fabric transport for NVMe currently under development: RDMA or Fibre Channel. Compared to traditional storage area network (SAN) design, NVMe-over-Fabrics achieves much lower latency and higher bandwidth [28]. NVMe-over-Fabrics enables high performance and efficient disaggregated storage, that means the high performance and low latency storage can be placed outside of the server, but still achieves the performance rivals the local attached storage. NVMe-over-Fabrics is promising to deliver high performance and scalable NoSQL solution and to improve the flexibility for deploying Docker contain-

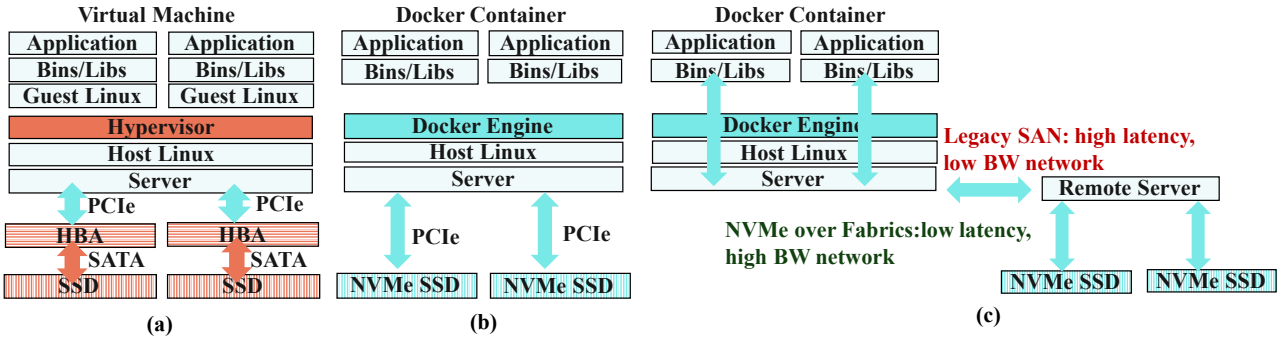


Figure 1: Comparison between virtualization and storage techniques

ers. We will explore the performance benefits and various strategies of deploying containerized NoSQL databases over NVM-over-Fabrics in the last portion of this work.

3. Container Storage Overview

Docker provides application virtualization and sandboxing using a number of known techniques that are present in the mainline Linux kernel [25]. In this work, we focus on the mechanisms that make *efficient* storage in Docker possible. The primary "executable" unit in Docker is an *image*. Figure 2a illustrates the concept of Docker images and layers. Images are immutable files that are snapshots of the container states. This is similar to VM images, except that Docker images cannot be modified. Instead, live containers have a *writable layer* created on top of the base image. Any changes to the container state during execution are made to the writable layer, which is similar to copy-on-write. Depending on the application usage scenario, sometimes the changes that are made to the image are only relevant for a given application execution run. Hence, once the application run completes, these diff layers are discarded, which we call as non-persistent storage. The fact that by default a container loses all of its data when deleted poses a problem for running applications such as databases, where persistent storage is essential [17]. Data volume is a way Docker provides persistent storage to Docker containers, as a supplement to Docker storage drivers.

3.1. Docker Storage Drivers

Docker containers use Copy-on-Write (CoW) mechanisms to handle storage within the container, namely the base image and various diff image layers shown in Figure 2a. Docker stores the data within the container using file systems that provide CoW capabilities, such as union-capable file system or device mapper thin provisioning snapshots. The granularity of difference between Docker image layers is a file for union file systems and a block for device mapper.

Union file system (UnionFS) operates many transparently overlaid file system layers and can combine those files

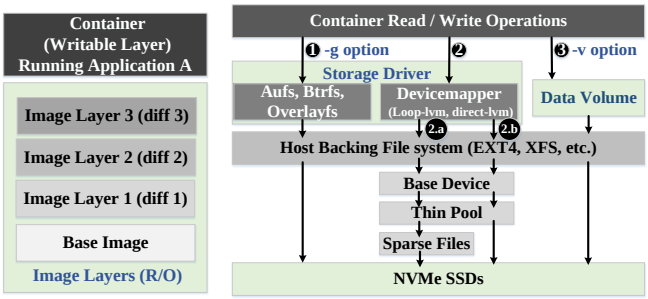


Figure 2: Illustration of (a) Docker container image layers (b) Storage paths in Docker

and directories inside these layers into a single, coherent file system. Using UnionFS, Docker images can contain many shared, read-only base image layers and a lightweight writable layer: only the files that are to be modified are copied to this additional layer. When a file is modified inside the Docker container file system, it is searched through all the file system layers, and then copied to the writable layer [15]. The write operation is then only applied to the writable layer. Docker uses this CoW mechanism to minimize the duplicate data storage, enable fast startup and reduce overall storage requirements.

Different storage drivers of path ❶ and ❷ in Figure 2b differ in the manner that they implement unionizing file system and the CoW mechanism. **Aufs** (Advanced multi-layer unification file system) is a fast and reliable unification file system with features like writable branch balancing. **Btrfs** (B-tree file system) [29] is a modern CoW file system which implements many advanced features for fault tolerance, repair and easy administration. **Overlayfs** is another union file system which has a simpler design and potentially faster than Aufs.

Devicemapper is an alternate approach to handling unionizing file system where the storage driver leverages the thin provisioning and snapshotting capabilities of the kernel-based Device Mapper framework. There are two different configuration modes for a Docker host running

devicemapper storage driver. The default configuration mode is known as **"loop-lvm"** shown in path 2.a of Figure 2b. This uses sparse files to build the thin-provisioned pools used by storing images and snapshots. However, this mode is not recommended for production, and using a block device to directly create the thin pool is preferred. The latter is known as **"direct-lvm"**, shown in path 2.b in Figure 2b. This mechanism uses block devices to create the thin pool.

In some application scenarios, it is imperative that storage modifications made by one container are visible to other containers. Docker provides a more efficient path to storage persistence for such scenarios. Apart from disk images diffs that may be stored and used across invocations, docker provides persistence through direct manipulation of host storage. This approach is particularly useful in applications such as databases where the storage persistence is a key requirement. Data stored on Docker data volume persists beyond the lifetime of the container, and can also be shared and accessed from other containers in this case. For this purpose Docker Volumes are used. Docker volumes are directories that exist as normal directories and files on the host file system. These can be declared at run-time using the $-v$ flag [26]. The path for persistent data I/O through Docker volumes is shown on the right side of Figure 2b (marked ❸).

4. Explore Container Storage Options

In this section, we explore various options for Docker data storage. Since I/O accesses to a storage device usually happen through a host file system, it is important to understand the performance implications of the Docker storage file system, such as UnionFS, and its interaction with the underlying host file systems. In particular, we are interested in understanding the performance implications when the host storage system is built on top of NVMe storage. Some previous work on the performance characterization of NVMe devices [37], [3] has highlighted the performance bottlenecks in the system software stack, particularly the file system layer preventing applications from fully exploiting the NVMe SSD performance [35]. But these studies did not consider the interaction between two different file systems that are overlaid. In order to understand the performance implications of this Docker-specific scenario, we first explore the best choice of file system, for both the host file system as well as for the container's own file system.

Host File System: The host file system refers to the file system (ext4, xfs, btrfs, etc.) used to create the Docker's storage as shown in Figure 2b. Some care has to be taken while exploring the choice of file systems are there might be compatibility issues with Docker storage driver. Not all Docker storage drivers are compatible with every file system. For example, Docker's btrfs driver works only with btrfs as the host file system. Ext4 and xfs have better compatibility with many Docker storage drivers such as overlayFS and Aufs. For devicemapper, the base device's file system can be specified as either ext4 or xfs using dm.fs option. Note that the Data volume approach of providing persistence does not have layering of two different file systems since it directly uses the underlying host's file system without any unionizing file support.

Utilizing SSDs for Container Storage: Both locally attached and remote SSDs can be used for storing either images or data volumes, or both. We describe these configurations that we studied in detail for our experiments.

Option 1: Through Docker Filesystem. All the updates to the NVMe SSDs go through the Docker storage drivers, including Aufs, Btrfs, and Overlayfs, as shown by path ❶ in Figure 2b. Note that data are not persistent and when the container is deleted, the files inside it are also removed from storage.

Option 2: Through Virtual Block Devices. The files are stored using the Docker devicemapper storage driver, which is based on Linux Device Mapper volume manager for mapping physical block devices onto higher-level virtual block devices. As previously mentioned, there are two configurations, **"direct-lvm"** and **"loop-lvm"**, depending on whether the thin pool is created using block device or sparse files. Paths **2.a** and **2.b** in Figure 2b refer to storage paths using loop-lvm and direct-lvm, respectively. This option also does not provide persistent storage beyond the lifetime of Docker container.

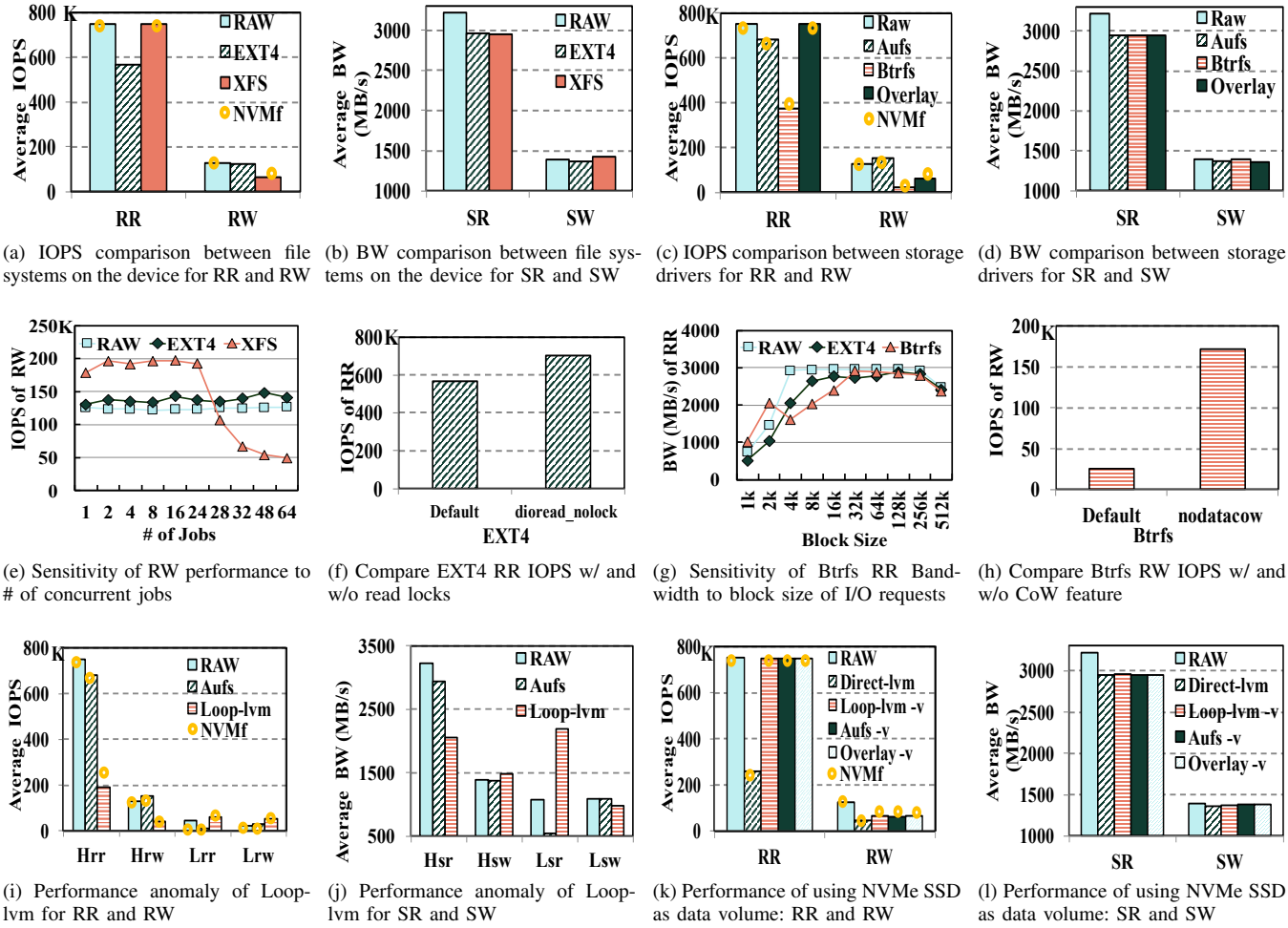
Option 3: Through Docker Data Volume. Data stored on Docker data volume persists beyond the lifetime of the container, and can also be shared and accessed from other containers [14]. This data volume provides better disk bandwidth and latency characteristics by bypassing the Docker file system, and is shown as the **-v** option in path ❸ in Figure 2b. It also does not incur the overheads of storage drivers and thin provisioning. The main reason for using data volumes is data persistence.

All these different storage drivers, host file systems and storage options create a large variety of combinations for data storage. The rest of the paper explores the performance implications of these options.

5. Container Performance Analysis

Before diving into the details of the results, we provide a brief overview of our experimental system. We used a Xeon Server that has NVMe capability. We populated that server with Samsung XS1715 NVMe SSDs. Note that the server also has a state-of-the-art 40 Gb, RDMA capable ethernet fabric which uses RoCE protocol. The RDMA capability was used for measuring remote SSD read performance which will be described in later sections. The SSD was configured as a raw block device with xfs as the host filesystem. Further details on the software and hardware setup can be found in Table 1.

In the first section of the results, we provide a characterization of multiple components of the file system stack using the flexible I/O tester (fio) tool [4]. I/O traffic is synthetically generated using fio's asynchronous I/O engine, `libaio`. Fio's O_DIRECT flag can be enabled to bypass the page cache for I/O accesses. Our experiments show that with the flag left unset, a significant amount of the I/O traffic will



(a) IOPS comparison between file systems on the device for RR and RW

(b) BW comparison between file systems on the device for SR and SW

(c) IOPS comparison between storage drivers for RR and RW

(d) BW comparison between storage drivers for SR and SW

(e) Sensitivity of RW performance to # of concurrent jobs

(f) Compare EXT4 RR IOPS w/ and w/o read locks

(g) Sensitivity of Btrfs RR Bandwidth to block size of I/O requests

(h) Compare Btrfs RW IOPS w/ and w/o CoW feature

(i) Performance anomaly of Loop-lvm for RR and RW

(j) Performance anomaly of Loop-lvm for SR and SW

(k) Performance of using NVMe SSD as data volume: RR and RW

(l) Performance of using NVMe SSD as data volume: SR and SW

Figure 3: Characterization of the throughput of Docker storage system using assorted combinations of request types (random read, random write, sequential read and sequential write), file systems (ext4, xfs, Btrfs), storage drivers (Aufs, Btrfs, Overlayfs, Loop-lvm, Direct-lvm), block sizes, # of concurrent jobs and mount options. H / L indicates High/ Low load. The yellow circles show the throughput for the corresponding configuration using NVMf.

TABLE 1: Server node configuration.

Processor	Intel Xeon E5-2670 v3, 2.3 GHz, dual socket-12 HT cores
Memory Capacity	64 GB ECC DDR3 R-DIMMs
Storage	2x Samsung XS1715, 1.6 TB [30] 2x Samsung PM1725, 1.6 TB [31]
Network between Clients and Application Server	2x 10 Gb Gigabit Ethernet NIC
Network between Application and Target Storage Server	4x 40Gb Gigabit Ethernet NIC RDMA capable, RoCE protocol
Kernel Version	Linux 4.6.0
Docker Version	1.11.2
Cassandra Version	2.1

hit in the DRAM, rather than going further down to the I/O subsystem. To ensure the measurement of disk performance (as opposed to including confounding factors like DRAM caching effects) we bypass the page cache by setting the O_DIRECT flag.

To quantify the performance impact of different file systems and Docker storage options, we first run preconditioning tests on the disk for two to four hours until it reaches the *steady state*, and then run the same set of fio experiments over a 5GB file, for ten minutes each, from within the container as well as natively on the host, and then report the steady state performance.

Unless otherwise indicated, we use RR/RW as acronyms for random reads/writes and SR/SW for sequential reads/writes under high load (32 jobs, 32 queue depth configured within the fio tool). 4 KB and 128 KB are used as the default block sizes for random and sequential workloads, respectively.

We report the IOPS for random workloads and bandwidth for sequential I/O operations, each run in isolation. Later in Section 6, we will further characterize the performance of mixed random and sequential workloads using real world containerized applications.

5.1. Choice of Host FS on Performance

We experimented with two major and popular file systems for the host file system, ext4 and xfs. Figure 3a and Figure 3b provide the comparison of the two file systems relative to the performance of the raw block device (RAW in Figure 3a and Figure 3a). We observed that the performance of file I/O using xfs closely resembles that of the raw block device for both random and sequential workloads. The only instance where xfs fares worse than the block device is for purely random writes. As these experiments are carried out with a large number of parallel jobs (32), we further vary the number of parallel jobs and repeat the experiments. We can clearly see in Figure 3e that the performance of xfs falls off the cliff when there are more than 24 jobs. Meanwhile, the IOPS of ext4 stays about the same regardless of the number of parallel jobs. Based on our analysis of the file systems we believe that xfs suffers a steep loss in performance due to lock contention in exclusive locking which is used by extent lock up and write checks. This phenomenon significantly degrades the write performance of xfs as the thread count grows, however, could be eliminated by using a shared lock in $xfs\_get\_blocks()$ function [10]. Ext4 uses exclusive locking as well but does not degrade like xfs due to its reliance on a finer grain locking with different lock types.

As a result, ext4 performs close to rated specifications for all workloads, except for random reads. In this case, it performs 25% worse than both the raw block device as well as xfs owing to the latter's support for very high throughput file I/O using large, parallel I/O requests. Xfs also allows multiple processes to read a file at once while having only one centralized resource: the transaction log. All other resources in the file system are made independent either across allocation groups or across individual inodes [33]. Therefore, reads under high thread count continue to scale well. In contrast, ext4 requires mutex locks even for read operations. As we can see in Figure 3f, if we force removing the mutex locks by enabling the $dioread\_nolock$ option for ext4, its random read performance immediately jumps up by 25%.

Therefore, if a workload is random read intensive then xfs may be a better choice for the host file system. Otherwise, if there are a large number of concurrent random write operations, it may be better to choose ext4 to avoid write lock contention in xfs. For workloads mainly consisting of sequential operations, either xfs or ext4 would work well. Since xfs performs well for the majority of our tests, we use it as the default host file system for the rest of the paper.

5.2. Performance Comparison of Different Docker Storage Drivers

Next, we compare the performance implications of the different storage drivers offered by the Docker ecosystem. Data stored though Docker storage drivers live and die with the containers. Therefore, this ephemeral storage volume is usually used in stateless applications that do not record data generated in one session for use in the next session with that user. For these experiments, we created a Docker container that is configured with a specific storage driver used in that experiment. We then did file read and write operations from within the container. In these experiments, we vary the storage driver in use while keeping the backend device the same – Samsung XS1715 NVMe SSD. The NVMe device was configured with xfs file system (except for the btrfs driver) along with the specific Docker storage driver. Figure 3c and Figure 3d present the results of aufs, btrfs and overlayfs compared to the raw block device performance. We find that aufs and overlay drivers can achieve performance similar to the raw block device for most cases under consideration. Btrfs, on the other hand, performs much worse than all the other options for random workloads.

The performance of btrfs is much lower for small block size read and write operations. Figure 3g shows a comparison among btrfs, ext4 and raw performance when changing the block size. We observed that btrfs achieves maximal random read performance when block size increased to 32KB. Some previous performance studies [36] have attributed it to the fact that btrfs operations have to read file extents before reading file data, adding another level of indirection and hence leading to performance loss.

For random writes, the performance degradation is mostly due to the CoW overheads. As shown in Figure 3h, when we disable the CoW feature of btrfs, the random write performance of btrfs increased by 5X.

Our second important observation is the fact that in order to achieve performance closer to the rated device performance, *deterministically*, it is better to use the data volume rather than the storage driver. As seen in Figure 3c and Figure 3d, the performance experienced by an application is dependent on the choice of the storage driver. However, in the case of the Docker volume, since the I/O operations are independent of the choice of the storage driver, the performance is closer to that of the raw block device. But as we stated earlier such an approach can potentially compromise repeatability and portability since the writes are persistent across container invocations.

One of the interesting observations is that using loop-lvm backend leads to anomaly performance, as shown in Figure 3i and Figure 3j. In this experiment, we compare the performance of the loop-lvm storage driver to (i) raw block device performance, and (ii) the performance of the aufs backend. Both aufs and loop-lvm experiments are done using Docker **-g** option. In certain cases, we notice that the performance of the loop-lvm storage backend is significantly higher, even higher than that of the raw block device. Typically, for random read under low load (for FIO: one job, one queue depth), the loop-lvm reports 1.4x the IOPS of a RAW block device, and 7x that of aufs. The performance anomaly under low load implies that the latency of loop-lvm experiments is even lower than that of the RAW block device and much lower than that of aufs. This is due to the fact that dm-loop does not respect the O_DIRECT flag [16], which we used for all our experiments. Hence, almost all the data is getting read from or written to DRAM, providing the illusion of better *device* performance.

5.3. Data Volume Performance

In order to test the performance implications of the data volumes, which allows for data persistence (by default), we tried a number of different experiments. The data file used as the targets for fio is stored on the NVMe drive and is accessed using the default data volume, while the images are stored on the boot drive using the specified storage backend. In these experiments, we again vary the storage backend while carrying out I/O intensive operations on the Docker volume. Using these experiments, we wanted to gauge the effects of the choice of the storage driver on the performance of operations for persistent data through the data volume. This setup is also described as *Option 3* in Section 4.

Figure 3k and Figure 3l present the results of these experiments. For the most part, the performance of the persistent storage operations through the data volume are independent of the choice of the storage driver. As expected, the performance of I/O operations matches that of xfs for the host machine in the filesystem choice experiments as represented in Figure 3a and Figure 3b. Direct-lvm, on the other hand, is based on lvm, device mapper and the dm-thinp kernel module which introduce additional code path and overheads which are not suitable for I/O intensive applications.

5.4. Choice of Storage Configuration for NVMf

Finally, we measured the performance impacts of NVMf. Recall that NVMf allows one server to read data stored on another server through RDMA. For this experiment, we used two Xeon servers and launched the container on one server while the storage was associated with the remote machine. Then the container would have to use NVMf interface to read remote data. We show that there is little performance difference between NVMe and direct attached storage for the same Docker storage configurations. The yellow circles in Figure 3 show the throughput of random requests for all the storage configurations using NVMf. The performance characteristics for random requests closely matches with directly attached SSDs, especially in high I/O request load situation. Similar to directly attached SSDs, the data volume option ($-v$) shows the best performance for NVMf as well. We used data volume option to characterize real-world applications.

5.5. Latency Breakdown

Figure 4 shows the latency breakdown of various storage configurations. *slat* denotes the time spent by the user application to transfer control to kernel level syscall. Specifically, it is the time spent from the time the application initiates an I/O request to the time the $io\_submit()$ system call is called by the asynchronous I/O library `libaio`. *slat* reflects the application-level overhead in the entire I/O path. *clat* is the time taken by the I/O request from kernel level syscall $io\_submit()$ to completion, including the time spent

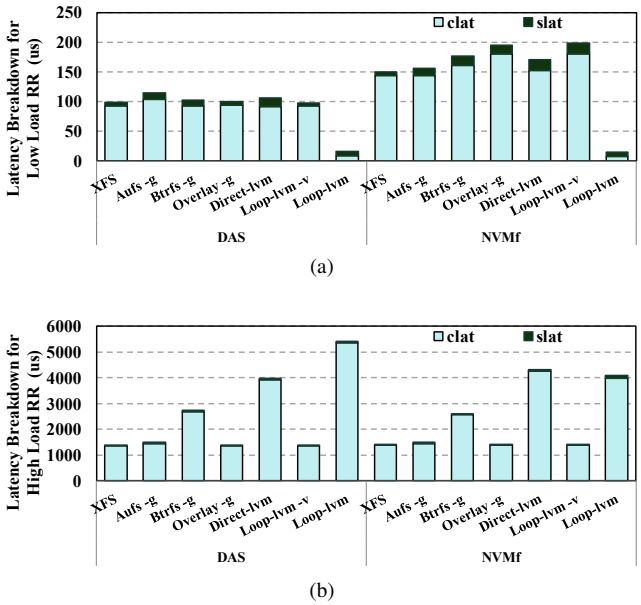


Figure 4: Latency breakdown of random read operations of directly attached NVMe SSDs (DAS) vs. NVMe-over-Fabrics (NVMf) using assorted storage configurations: (a) Low load (b) High Load

in file systems, drivers and device. *clat* takes up a substantial proportion of the total access latency of I/O requests.

Figure 4(a) shows I/O request latency when there's typically one job issuing one request at a time (low load). Since there's negligible queueing in the storage subsystem under low load, the latency is close to the raw device latency. *slat* takes up only a small portion of the total I/O latency and reflects a slight increase between DAS and NVMf - 9.7% for directly attached SSDs and 10.1% for NVMf. Compared to the similar experiment for directly attached storage (DAS) for low load cases, the extra network latency in NVMf results in a 53.3% increase in $clat$ and 54.0% of the total I/O request latency on average.

We further compared the I/O request latency of 16 parallel jobs concurrently issuing I/O requests (high load) in Figure 4(b). We observed significant queuing time spent in the storage subsystem that leads to an overall 20x longer latency for direct attached storage and 12x longer latency for NVMf compared to low load cases. The latency increase is typically longer when using Btrfs, direct-lvm and loop-lvm compared to using the other file system options, which suggests that the former filesystems are sub optimal for sustaining high bandwidth needs in NVMe SSDs. On the other hand, the extra network communication latency in NVMf becomes less than 2% compared to the much longer queueing overhead and the application overhead *slat* is less than 1%.

In low load experiments, loop-lvm option shows extremely low latency comparing to other options in both DAS and NVMf experiments. As mentioned before, loop-lvm doesn't respect the O_DIRECT flag, and as a result,

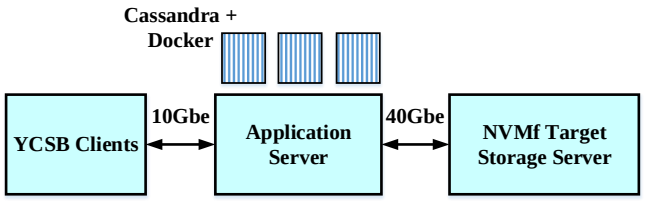


Figure 5: Experimental setup of NVMe-over-Fabrics prototype and Cassandra Experiments

most of the I/O requests hit in the DRAM. Since those requests complete much faster than the requests going to the SSDs, the average *clat* drops significantly. For understanding the benefits of different storage options, it is imperative to enable the O_DIRECT flag to avoid the case that most of the requests hit in the DRAM. Hence, we caution that loop-lvm results are not a true measurement of I/O system performance due to memory buffering interference.

6. Performance Evaluation and Optimization of Dockerized DBMS

Next, we perform detailed characterization of how much performance a containerized NoSQL database can maximally deliver over local and remote high performance SSDs. We compare and analyze the scaling nature of multiple concurrent Cassandra containers using the best Docker storage options obtained from previous FIO experiments. We conduct a deep-dive bottleneck analysis to identify the critical resources limiting the performance scaling using Linux control groups. Based on the analysis, we propose optimized resource allocation mechanisms among the servers in the same rack to deliver better performance.

We experimented with three server nodes, as is shown in Figure 5, each has a dual-socket Intel Xeon E5 server, supporting 48 hyper-threading CPU threads. We use two nodes for Cassandra client and server configuration, with the client node driving traffic to the application server over a 10Gb ethernet link. The third node is configured as an NVMf target storage server. In these experiments, we compare two main storage scenarios. In the first case, the data is stored on an NVMe SSD that is directly attached to the server. In the second case, a third node serves as an NVMf 'Target'. The target storage server exports the Samsung PM1725 NVMe SSDs on to the server over a state-of-the-art 40 Gb, RDMA capable ethernet fabric and uses the RoCE protocol. In both cases, we configure the raw block device (direct attached or remote) with xfs as the host filesystem. Further details on the software and hardware setup can be found in Table 1.

For these experiments, we use Cassandra [11], which is a popular open-source NoSQL data store that will (theoretically) scale linearly to the number of nodes in the cluster. We use YCSB [8] to drive the Cassandra database. For each container, we loaded 100 million/100GB records in the database. We have 16 client threads making 10 million queries to each Cassandra server. We measure the performance of two different workloads, with two common combinations of read and write operation mix: workload A, which contains 50% read and 50% update operations in Zipfian distribution; workload D, which contains 95% read and 5% insert operations in the uniform distribution.

6.1. Analysis the Cassandra Container Performance

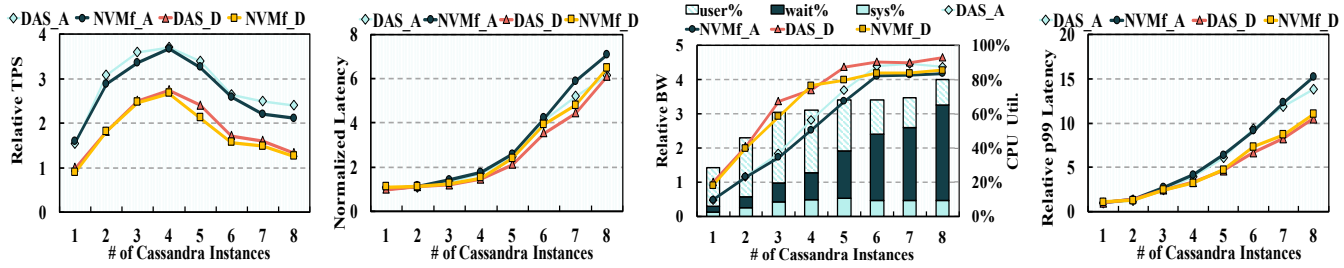
We did an extensive comparison of all system metrics, including aggregated throughput (total throughput of all the concurrent containers), average latency, tail latency, CPU utilization and disk bandwidth, to perform a systematic analysis of the performance and scalability of multiple Cassandra instances between (i) a direct attached NVMe drive, and (ii) an NVMe-over-fabrics (NVMf) prototype implementation for two workloads on the same server.

We make several important observations from these experiments. First, NVMf storage achieved throughput within 6% to 12% of direct attached SSD and incurred only 2% to 15% (75 us to 970 us) extra latency overhead, as shown in Figure 6a and Figure 6b. Figure 6c shows the average disk bandwidth for one disk when running two different workloads. We can see NVMf achieves peak disk bandwidth of $\sim 2800MB/s$, which is reasonably close to peak disk bandwidth ($\sim 3000MB/s$) of direct attached SSD. NVMf benefits from high-speed ethernet and eliminates unnecessary protocol translations, therefore it sustains the high throughput of NVMe SSDs.

Secondly, the aggregate throughput of all Cassandra instances first increases, peaks at about four instances and then decreases from that point on. This trend is observed for both DAS and NVMf cases. This also correlates with the fact that the disk bandwidth starts to saturate after four instances, as shown in Figure 6c. We also observe the time the CPU spends waiting for I/O requests (marked by $wait\%$) in Figure 6c significantly increases after the disk bandwidth saturates due to increased queueing delays.

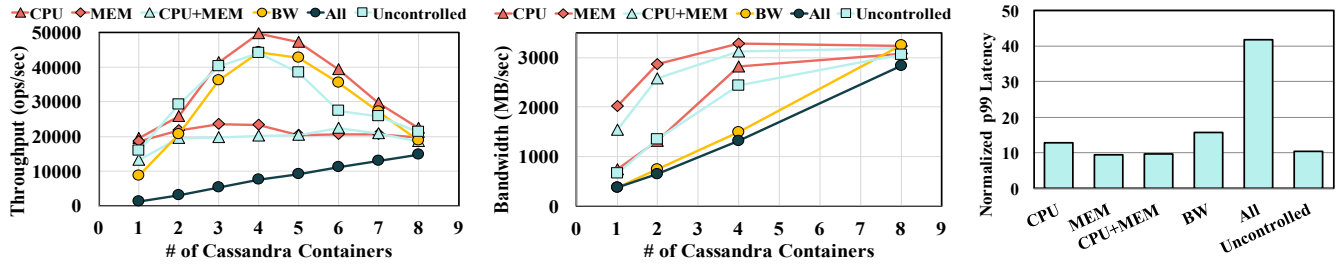
Moreover, as we scale the number of instances on the server, we observe a corresponding increase in client side latency, as shown in Figure 6b. However, the client side 99^{th} percentile latency increases much faster than the average latency. Note that, these observations remain the same, both in trends as well as absolute numbers for the NVMf implementation for both workload A and workload D.

Finally, comparing the two different workloads, workload A exhibits higher throughput as well as a higher increase in tail latency than workload D. To explain this, we need to first understand how Cassandra stores its data. Cassandra writes (inserts and updates) are stored first in a DRAM resident cache called memtable, and when the memtable size exceeds a certain threshold, the data is flushed into a disk-resident data structure called SSTables. Since Cassandra does not update data in place, the updated data will be written to a new SSTable. Reads miss in memtable often need to fetch data from multiple SSTables on the disk, which have longer data paths than writes. Therefore, the read latency is higher than write latency in Cassandra and read requests are more limited by disk bandwidth. Recall



(a) Throughput (b) Average read latency (c) Disk BW (d) p99 latency of read operations

Figure 6: Performance comparison of concurrent Cassandra containers between direct attached NVMe SSD and NVMf for workload A and workload D. P99 latencies of workload A are normalized to its own single container p99 latency, all others are normalized to the corresponding DAS workload D single container results.



(a) Throughput (b) Bandwidth (c) p99 latency

Figure 7: Performance comparison of a scaling number of concurrent Cassandra containers with various resource constraints

that workload A has 50% reads and 50% updates, while workload D has 95% reads and 5% updates. Therefore, given the same disk bandwidth, workload D achieves lower overall throughput.

To improve the read performance, Cassandra implements SSTable compaction to merge multiple SSTables into single SSTable. We observe more updates lead to frequent SSTable compactions and more java garbage collection. Those activities delay read requests intermittently, which we believe, lead to the higher read tail latency increase of workload A.

6.2. Analysis the Bottleneck for Scaling Up

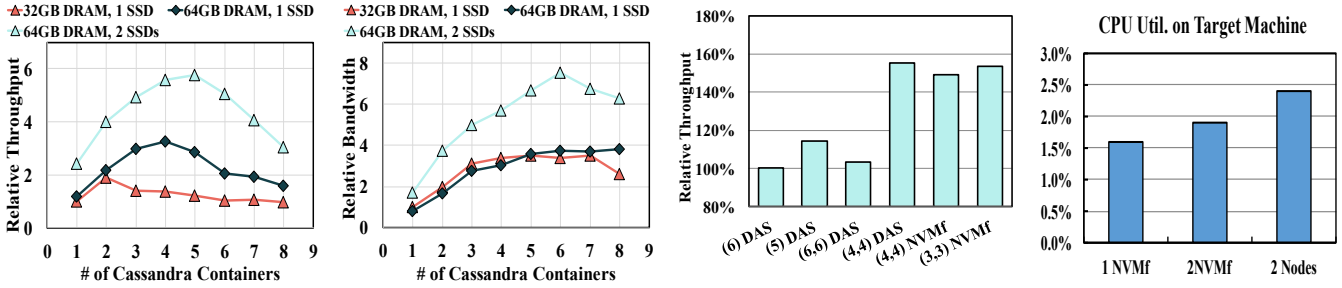
In this section, we focus on identifying the performance bottlenecks while scaling the number of concurrent Cassandra containers on high performance NVMe SSDs. For this purpose, we use Linux Control Groups, a.k.a, Cgroups [24] to limit and isolate the resource usage (CPU, memory and disk bandwidth) of each container. We then compare the performance delta between constrained resource usage and unconstrained usage and analyze the reasons that disallow the aggregate throughput of Cassandra containers from continuing to scale up.

We collect and compare the throughput between different resource control configurations: single resource control (labeled CPU, MEM, BW in Figure 7a), multiple resource control (labeled CPU+MEM, ALL) and no resource control (labeled Uncontrolled). We divided the total CPU cores by maximum number of containers (eight in this experiment) and allocate the number of CPU cores to each container. A similar process is done for memory and disk bandwidth. We allocate 6GB memory for each container. Table 2 lists the resources allocated for each container for different resource control configurations in Figure 7a: if we don't control that type of resource, and all the instances can use the up to the maximum number of total resources on demand, we leave the corresponding cell empty. In the rest of the paper, we use CPU, MEM, BW, etc. to refer to the corresponding resource control configurations for simplicity.

TABLE 2: Cgroups Configurations

Abbr.	CPU Cores	MEM	Disk Bandwidth
CPU	6	-	-
MEM	-	6GB	-
CPU+MEM	6	6GB	-
BW	-	-	400MB/s
ALL	6	6GB	400MB/s

Across all single resource control configurations that we investigated, MEM, which assigns a fixed (6GB) memory to each additional container, has the most significant impact on throughput. Different from the performance behavior of the CPU, BW or Uncontrolled, the throughput of MEM stays around 20K operations per second and doesn't increase as we increase the number of concurrently executing containers. Typically when there are four concurrent containers, the aggregate throughput of MEM is much lower than that of



(a) Throughput (b) Bandwidth (c) Optimal Concurrent Instances (d) CPU Util.

Figure 8: Performance results of various resource allocation strategies

CPU, BW or Uncontrolled: MEM decreases the aggregate throughput of Uncontrolled by 47%, while BW keeps the same with Uncontrolled; moreover, CPU even increases the aggregated throughput by 13%. Furthermore, it is surprising that controlling BW using cgroups doesn't seem to have much performance impact.

To understand these performance behaviors, we further measured the corresponding disk bandwidth. In Figure 7b, the line marked with light blue squares show the average disk bandwidth consumed having different number of concurrent containers without any resource control. In comparison, the MEM line (marked with red diamond) is much higher. This indicates that higher BW is consumed as a result of reduced memory capacity. The lower memory capacity leads to reduced memtable size, which in turn, increases the read miss rate and frequency to access the disks. Therefore, the throughput reached a bandwidth bottleneck sooner because of fewer DRAM capacity is allocated.

On the other hand, at 4 instances, we observe controlling only CPU resources increased the measured disk bandwidth only slightly. Correspondingly, the overall throughput also increases slightly. Since in the CPU control, we assign certain CPUs to particular Cassandra instances, rather than assigning them dynamically. We believe this could improve the thread affinity and reduce CPU contention, and therefore lead to higher throughput. Note that in uncontrolled case, CPUs are underutilized until 4 instances, this is possibly why the benefits start to show up only at that point.

When both CPU and MEM are controlled, the performance is similar to MEM, as disk bandwidth is the main bottleneck in both cases. If we continue to decrease bandwidth, we observe that performance is controlled by bandwidth and furthermore it linearly decreases with a linear reduction of total bandwidth (black dot line labeled ALL).

Figure 7c compares the 99^{th} percentile latency of various control options. Among all the resource control combinations, none of the mechanisms reduced the tail latency significantly. In the worst case, the 99^{th} latency could increase by more than three times when controlling all the resources. There's only slight reduction of tail latency by controlling MEM and CPU+MEM, which instead sacrifices throughput significantly. Therefore, there is not one optimal resource control strategy can both achieve higher throughput as well as reduce tail latency.

In summary, memory size and disk bandwidth are main bottlenecks for scaling Cassandra instances. Over packing Cassandra instances on the same server will severely decrease overall system throughput mainly due to the following two reasons: (i) decreased memory capacity and disk bandwidth per instance (ii) increased disk bandwidth requirement due to reduced memory capacity. The optimal number of instances assigned on one server should balance the memory and bandwidth requirement.

6.3. Optimize Container Assignment through NVMf

Based on our previous analysis, we discuss how different container assignment and resource allocation strategies can impact the aggregate throughput of Cassandra containers.

Recall in Figure 6a, the experiment results indicate that assigning four concurrent Cassandra instances to one application server achieves best aggregate throughput. However, it is not clear whether and how this number will change across multiple systems with varying DRAM capacity and storage bandwidth. To classify the effects of these parameters, we conducted multiple experiments on servers with varying DRAM capacity and disk bandwidths by using multiple disks. All the other system settings stay the same.

Figure 8a shows how optimal assignments change with DRAM capacity and disk bandwidth. For one experiment, we remove half of the DRAM on the application server, so that total DRAM capacity reduced from 64GB to 32GB. For another experiment, we add another PM1725 NVMe SSD to the system, so that the total disk bandwidth doubled.

We make three important observations from these experiments. Firstly, in all the experiments, there exists an optimal number of concurrent instances which achieved significantly higher throughput than others. Secondly, the optimal number of concurrent containers increases with more DRAM size and disk bandwidth to the server. Finally, compared to Figure 8b, we note that the container numbers that maximizes disk bandwidth may not achieve maximal system throughput. This motivates us to consider resource allocation (e.g. how many disk bandwidth to allocate to each

server) and container assignment (e.g. how many containers will assign to each server) simultaneously to achieve best system throughput.

Here we discuss some initial results using a simple optimization strategy through NVMf. We consider a scenario that there are two application servers with the same configurations listed in Table 1 and there are two PM1725 disks. In order to achieve the maximal throughput, we conducted multiple experiments to compare the following container assignment and disk allocation choices:

❶ Plug two disks to the same server and continue to increase the number of Cassandra instances until the bandwidth saturates. This results in 6 containers on one server, marked with *(6)_DAS* in Figure 8c. ❷ Plug two disks to the same server and continue to increase the number of Cassandra instances until the maximum throughput peaks and starts to decrease. This results in 5 containers in one server, marked with *(5)_DAS*. ❸ Plug one disk to each server and assign containers until saturating the throughput for each server. This results in 6 containers in each server, marked with *(6,6)_DAS*. ❹ Plug one disk to each server and assign the number of containers that maximize the throughput. This results in 4 containers in each server, marked with *(4,4)_DAS*, which improved the throughput by 51%. However, one of the challenges is that we don't know the throughput is maximal until we assign one more container to the server, and the bandwidth decreases. When this happens, without NVMf, we can't schedule the container off the server, since migrating the data from a DAS disk to the another disk on the other server have a high cost. ❺Using NVMf-over-Fabrics solves this problem. Since the storage is remote, we only need to initiate a new RDMA connection. We measured the throughput of using 4 containers on each application server and both of the application servers are connected to the same NVMf storage server, with two NVMf disks, marked with *(4,4)_NVMf*. We can see the throughput is very close to *(4,4)_DAS* and also close to the optimal throughput for 2 server NVMf system, marked with *(3,3)_NVMf*. Compared with the first strategy, which assigns two disks to the same server and then maximizes the disk bandwidth, the proposed *(4,4)_NVMf* strategy achieved 49% increase in aggregated system throughput.

7. Discussion

Distributing the computation among server nodes and connecting those server nodes to a central storage system is a promising design choice. Here we discuss the scalability of these systems. We show that the CPU utilization on the target machine as low as 2.4% for two servers connecting to two NVMf SSDs in Figure 8d. If we assume the utilization is increased linearly, we will be able to support up to 50 concurrent NVMf connections until the server CPU resources are exhausted.

We made an observation that the p99 latency increases exponentially when increasing concurrent containers. And it is a problem since it delays the average service response time in when the service needs to make multiple queries in large-scale data centers. One possible way to solve it could be to move the container with high p99 latency to another server with more resources. NVMe-over-Fabrics can significantly reduce the cost of offloading the containers.

8. Related Work

Container Performance: Lightweight virtualization mechanisms like Docker have become commonplace in the last couple of years. However, despite this popularity, there is very little work that has been done to understand the performance implications of the various elements that make up containerized applications.

Some recent studies have characterized the performance implications of containerized applications. Felter et al [18] compared the performance difference between virtualized and containerized workloads and found that Dockerized applications introduced little performance overhead as compared to virtualized ones, and were able to match native, bare-metal performance for CPU and memory intensive workload. Furthermore, for containerized applications, they also characterized the memory, CPU and network overheads of both containerized and Dockerized applications. A preliminary investigation of the storage stack overheads was also provided, but it was evaluated using *only* the default storage configurations on an HDD-based SAN device. Agarwal et al. [1] provide preliminary, experimental characterization of differences in startup times and memory footprints of VMs and containers and conclude that the latter outperforms the former by 6× - 60×, depending on the workload. Bhimani et al. [7] characterizes I/O intensive applications inside the Docker. In this work, we first present a deep-dive analysis of various Docker storage options and then explore the best practices to scale NoSQL databases using both local and remote high performance storage.

NVMe Storage: Recently, some work has been done in understanding the performance advantages of high performance NVMe devices. Xu et al. [37] provide a detailed characterization of NVMe devices as well as that of the associated system software stack. Furthermore, this work expands on the system-level characterization of data center applications carried out in [3], especially for NVMe SSDs. Although, none of these works perform comprehensive characterization of containerized applications from a high performance storage perspective. In one of the first works of its kind, Ahn et al. [2] design and implement a cgroups based mechanism to allow for proportional I/O sharing among multiple containerized applications for NVMe SSDs.

High Performance Networking: Several user space techniques to achieve high performance networking have been studied, such as IX [5], mTCP [21], Sandstorm [23] and OpenOnload [20]. Ana et al. proposed ReFlex [22], a software-based system for remote Flash access, to provide high throughput and low latency access to the remote storage system. While our studies shows that NVMe-over-Fabrics can achieve similar bare-metal performance for remote storage, we further performed competitive analysis for the containerized workloads on local and remote storage.

9. Conclusion

In this paper, we provide an extensive characterization of Docker storage drivers using state-of-art NVMe SSDs and NVMe-over-Fabrics. We showed that containerized applications can achieve similar performance between local and remote storage for both synthetic workloads and read world NoSQL database with multiple use cases. We further identified the critical resource bottlenecks for scaling containerized NoSQL databases, and showed there is not one optimal resource control strategy can both achieve higher throughput as well as reduced tail latency. Finally, we discussed container assignment and resource allocation strategy using NVMf for optimal system throughput.

Acknowledgment

We thank our paper shepherd, Thomas Schwarz, for his advice in producing the final version of this paper, as well as the anonymous reviewers for their valuable comments. We also thank Harry Li (Samsung) for setting up the NVMe-over-Fabrics experiment environment and Vijay Balakrishnan (Samsung) for his support on this work.

References

[1] K. Agarwal, B. Jain, and D. E. Porter, "Containing the Hype," in *APSys*, 2015.

[2] S. Ahn, K. La, and J. Kim, "Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems," in *Proc. of HotStorage*, 2016.

[3] M. Awasthi, T. Suri, Z. Guz, A. Shayesteh, M. Ghosh, and V. Balakrishnan, "System-level characterization of datacenter applications," in *Proc. of ICPE*, 2015.

[4] J. Axboe, "Flexible I/O Tester," https://github.com/axboe/fio, 2016.

[5] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proc. of OSDI*, 2014.

[6] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, 2014.

[7] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding performance of I/O intensive containerized applications for NVMe SSDs," in *International Performance Computing and Communications Conference (IPCCC)*, Dec 2016.

[8] C. F. Brian, S. Adam, T. Erwin, R. Raghu, and S. Russell, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of SoCC*, 2010.

[9] C. Bryant, "Dell's PE R920 Big Data Servers Get Samsung's NVMe SSDs," http://www.tomsitpro.com/articles/dell-poweredge-r920-ssd-nvme-samsung,1-1818.html, 2014.

[10] D. Chinner, "Concurrent direct IO write in xfs," http://oss.sgi.com/archives/xfs/2012-02/msg00219.html/, 2012.

[11] Datastax, "Companies using NoSQL Apache Cassandra," http://planetcassandra.org/companies/, 2010.

[12] J. M. Dave Minturn, "Under the Hood with NVMe over Fabrics," http://www.snia.org/sites/default/files/ESF/NVMe_Under_Hood_12_15_Final2.pdf, 2015.

[13] R. Davis, "NVMe Over Fabrics will radically change storage and networking," in *Storage Visions*, 2016.

[14] Docker, "Manage data in containers," https://docs.docker.com/engine/tutorials/dockervolumes/, 2016.

[15] ——, "Understand images, containers, and storage drivers," https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/, 2016.

[16] J. Eder, "Comprehensive Overview of Storage Scalability in Docker," http://developers.redhat.com/blog/2014/09/30/overview-storage-scalability-docker, 2014.

[17] C. Evans, "Docker storage: how to get persistent storage in Docker," http://www.computerweekly.com/feature/Docker-storage-how-to-get-persistent-storage-in-Docker, 2016.

[18] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. of ISPASS*, 2015.

[19] J. Green, "Disaggregated physical storage architectures and hyperconvergence," http://www.actualtech.io/disaggregated-physical-storage-architectures-and-hyperconvergence/, 2016.

[20] S. C. Inc., "OpenOnload," https://www.openonload.org/, 2013.

[21] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. of NSDI*, 2014.

[22] A. Klimovic, H. Litz, and C. Kozyrakis, "ReFlex: Remote Flash ≈ Local Flash," in *Proc. of ASPLOS*, 2017.

[23] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *Proc. of SIGCOMM*, 2014.

[24] P. Menage, "Cgroups - The Linux Kernel Archives," https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2016.

[25] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, 2014.

[26] A. Mouat, "Understanding Volumes in Docker," http://container-solutions.com/understanding-volumes-docker/, 2016.

[27] NVM Express, "NVM Express – scalable, efficient, and industry standard," www.nvmexpress.org, 2016.

[28] Z. Ori, "High availability for centralized NVMe," in *Data Storage Innovation Conference*, 2016.

[29] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," in *IBM Research Report*, 2012.

[30] Samsung, "XS1715 Ultra-fast Enterprise Class 1.6TB SSD," http://www.samsung.com/global/business/semiconductor/file/product/XS1715_ProdOverview_2014_1.pdf, 2014.

[31] ——, "PM1725 NVMe PCIe SSD," http://www.samsung.com/semiconductor/global/file/insight/2015/11/pm1725-ProdOverview-2015-0.pdf, 2015.

[32] Supermicro, "Supermicro NVMe Platforms," https://www.supermicro.com/products/nfo/NVMe.cfm, 2016.

[33] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," in *Proc. of USENIX ATC*, 1996.

[34] P. Teich, "NVIDIA's TESLA P100 Steals machine learning from CPU," https://www.supermicro.com/products/nfo/NVMe.cfm, 2016.

[35] V. Tkachenko, "ext4 vs xfs on SSD," https://www.percona.com/blog/2012/03/15/ext4-vs-xfs-on-ssd/, 2012.

[36] M. Xie and L. Zefan, "Performance Improvement of Btrfs," in *Proc. of Linux Con*, 2011.

[37] 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 *Proc. of SYSTOR*, 2015.