

DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation

Miryeong Kwon, Donghyun Gouk, and Changrim Lee, *KAIST;*
Byounggeun Kim and Jooyoung Hwang, *Samsung;* Myoungsoo Jung, *KAIST*

https://www.usenix.org/conference/fast20/presentation/kwon

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Open access to the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
is sponsored by



DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation

Miryeong Kwon[1], *Donghyun Gouk*[1], *Changrim Lee*[1], *Byounggeun Kim*[2],
Jooyoung Hwang[2], *Myoungsoo Jung*[1]
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)[1], *Samsung*[2]
http://camelab.org

Abstract

We propose DC-store, a storage framework that offers deterministic I/O performance for a multi-container execution environment. DC-store's hardware-level design implements multiple NVM sets on a shared storage pool, each providing a deterministic SSD access time by removing internal resource conflicts. In parallel, software support of DC-Store is aware of the NVM sets and enlightens Linux kernel to isolate noisy neighbor containers, performing page frame reclaiming, from peers. We prototype both hardware and software counterparts of DC-Store and evaluate them in a real system. The evaluation results demonstrate that containerized data-intensive applications on DC-Store exhibit 31% shorter average execution time, on average, compared to those on a baseline system.

1 Introduction

Docker container technology is applied to diverse computing domains such as datacenter, serverless, cloud, and high-performance computing thanks to its predictable development and efficient resource isolation [1–7]. Docker can isolate the execution of specific applications from running other applications as Sandbox [8]. Specifically, users can explicitly set a resource limit for each service by using *control groups* (cgroups [9]) and namespace [10]. Therefore, different services can be free from conflicting dependencies and resource contention. Since this consistent environment supports completely isolated tenants, it is used to improve computing utilization and portability [11, 12]. For example, LEMP (Linux, Nginx, MySQL, & PHP) in the Google cloud platform can create multiple tenants per node, each running a separate web server (Nginx), fast CGI (PHP-FPM), and database (MySQL) [13].

While Docker is one of the best options to increase the computing utilization by sharing and/or isolating its resources, it does not yet well manage underlying storage [14–16]. Docker's virtualized environment enables multiple container executions atop the host-side kernel directly, which is different from hardware stack virtualization of the conventional virtual machines (VMs) [17]. Because of this OS-level virtualization, launching a container is much faster and lighter than executing a VM [14, 18, 19]. However, the multiple containers and host kernel often share the persistent states on a solid-state drive (SSD), which can unfortunately interfere with their executions and I/O services of any peers at a device-level. As modern SSDs in practice exploit internal parallelism to enhance performance [20–26], such interference introduces many storage resource conflicts, thereby degrading the performance. To manage the storage resource, ones may utilize proportional bandwidth of the blkio cgroup adopted by Docker [27] and assign different proportions to each of multiple containers, individually. However, this approach cannot address the interference issue for multi-container execution due to two root causes. First, even though the blkio cgroup throttles the target I/O queue with different proportions [28], the flash-level resource conflicts cannot be resolved as the host is unaware of the physical layout of the underlying SSD [29, 30]. Second, metadata I/O services, including page frame reclaiming, are not taken by I/O throttling of the blkio cgroup.

In this paper, we introduce *DC-Store*, a storage framework that supports deterministic I/O performance and resource isolation for the multi-container execution environment. DC-Store consists of two major components, called *Divided SSD* and *I/O Tacker*. Generally speaking, Divided SSD addresses I/O interference at the hardware-level, thereby offering deterministic performance per volume. At the same time, I/O Tacker isolates I/O requests of noisy neighbors from the execution of other peers by considering each container's volume ownership at the software-level. We design and implement Divided SSD, which separates physical flash channels and internal resources to support multiple NVM sets, which were very recently included by NVMe workgroup (NVMe 1.4 [31]). In specific, we statically partition the computing and hardware resources per NVM set such that different I/O requests

heading across different NVM sets can have no or minor flash-level resource conflicts. The proposed Divided SSD can provide an independent environment, which allows multiple containers to operate without the device-level I/O interference. It also enables us to use a large size of shared storage with better utilization per node, which is in typical required by modern data centers [32, 33].

Nevertheless, when multiple containers mount each of different NVM sets (provided by our Divided SSD), we still observe that the user experience of some containerized applications degrades from an execution time angle. Even though Linux cgroups and namespaces can limit/split the CPU, memory, and blkio subsystems for different containers, if there is a noisy container that unintentionally issues page-out I/O requests imposed by a low on virtual memory, the performance of other peers is severely interfered and degraded. Since Docker is not involved in metadata I/O management (and Linux treats containers as just like other processes), it cannot fully support the consistent environment for completely isolated tenants.

To address this shortcoming, the proposed I/O Tacker modifies Linux kernel in order to enable per-container page frame reclaiming by considering the container ownership and exposed NVM set organization. The current version of Linux applies the same page-in/out procedures to all processes and containers, which can introduce container-level performance interference. In contrast, our I/O Tacker assigns a per-container swap area and informs its the swap location to the kernel by modifying the kernel memory controller. Therefore, the noisy neighbor performs page in/out only from/to its own NVM sets. To this end, we also revise the Docker stack for users to pass through the swap-pinning information from the Docker client to the container engine. The I/O Tacker's swap pinning mechanism being aware of the container ownership can isolate noisy neighbors from other containers, thereby addressing the performance degradation of all containers running on the shared storage.

To the best of our knowledge, this paper is the first work that implements multiple NVM sets in real hardware as well as isolates the low on virtual memory impact from the execution environment of containers. Our results show that, even though the average latency of our Divided SSD prototype is worse than a baseline NVMe SSD for a single volume accesses (with a specific pattern), it provides better and deterministic I/O performance under concurrent storage accesses. When we co-run containerized data-intensive applications with memory-hungry containers (e.g., LEMP), the proposed DC-Store improves user-level experiences of the applications by 31%, on average, without degrading the performance of such memory hungry noisy neighbors.



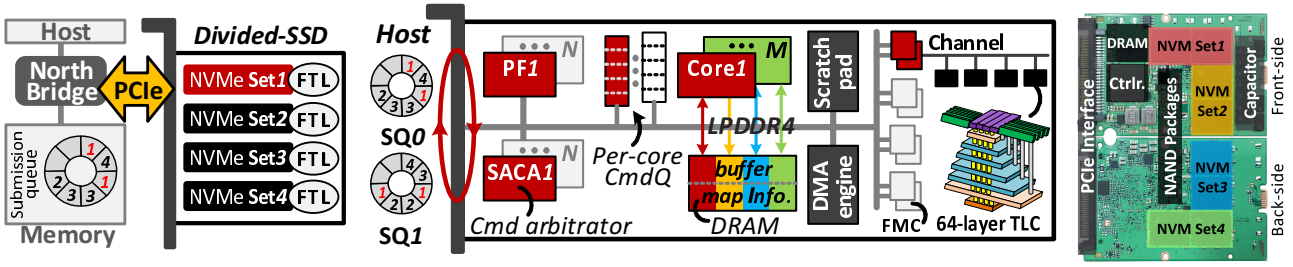
Figure 1: Interference in storage sharing.

2 Divided SSD: Hardware Level Design

Challenges in storage sharing. For better storage utilization and parallel accesses, a simple solution to allocate multiple containers to a shared SSD is creating multiple disk partitions on the storage; each partition can be mounted to a specific container for its I/O services. However, concurrent SSD accesses for the logically partitioned storage volumes can introduce many flash-level resource conflicts, thereby degrading performance per partition. To be precise, we perform I/O interference tests on a high-performance NVMe SSD, and the results are shown in Figure 1. In this evaluation, we split a 2TB SSD (whose device configuration is explained in Table 1) into four disk partitions. We execute a Facebook-like workload (interspersing bursty writes into sustained reads [33]) as a representative I/O access pattern of cloud environment by executing four containerized FIO applications [34] on the different disk partitions. We measure the read latency on a partition while all peer partitions are in serving bursty writes; the writes on each peer partition are started with every 30 secs (to understand the I/O interference impact). One can observe from the figure is that the random read latency is around 280 us and very sustainable at the first 30 secs period. However, once the bursty writes on another partition begin, the latency increases as high as 400 us (1.42 times longer). When all other partitions are dedicated to handling the bursty writes, the latency on the read service is 4.5 times higher than the normal operations.

Overview. To address the performance degradation in storage sharing, our Divided SSD physically partitions all SSD resources in the datapath. Figure 2a illustrates a high-level organization of the proposed Divided-SSD. Even though there exists a single PCIe SSD device connected to the host, our Divided SSD exposes multiple storage volumes; each has its own flash firmware such as *flash translation layer* (FTL) and block address space. These multiple volumes (of our Divided SSD) are visible from the host as "physically" separated NVM sets. Thus, the host need to employ completely different storage stack instances across multiple NVM sets, which can be mounted to each of the different containers.

Hardware design. Figure 2b shows the internal design of an ideal Divided SSD. Specifically, the proposed Divided SSD employs multiple low-power cores, each having instruction/data memory (I-cache/D-cache). These cores have their



(a) High-level view of Divided SSD. (b) SSD internal architecture. (c) HW prototype.

Figure 2: NVM set design and implementation (Divided SSD).

own scratchpad memory for latency-sensitive and/or shared data accesses. The cores are all connected to a global DRAM through LPDDR4, which are also accessed by a direct memory access (DMA). The DRAM can buffer the host requests and manage the mapping information of FTLs. Each core can have one or more flash memory controllers (FMCs), each being also connected to flash buses, called *channels*. Each channel employs multiple flash dies, which operate in parallel as well, called *ways* [35].

To offer multiple and physically separated storage volumes (which can address the resource conflicts), we statically split the all SSD internal resources in the datapath per NVM set. Specifically, as shown in Figure 2b, the global DRAM's memory address space is partitioned and independently allocated to each core at design time. As we disable the shared last-level cache, flash firmware instances, operating across different cores do not interfere with each other. In addition, each core exposes the independent flash address space of its own underlying channel(s) to the corresponding NVMe controller. The design of our static partitions makes multi-core management easier and enables flash firmware to operate on different DRAM and storage regions in a separate way as well as work entirely in parallel. Thus, in our design, each FTL (assigned to each of statically partitioned DRAMs and cores) independently performs the address translation between logical to physical flash pages per NVM set.

NVM set control. To connect the underlying cores and firmware to the host processor(s), the device requires employing multiple PCIe *physical functions* (PFs). Note that, while the ideal design of Divided SSD can employ as many NVMe PFs as the number of NVM sets, the number of PFs that an SSD exploits in a real-world scenario would be limited; it can even be smaller than the number of cores. To address this practical challenge, we introduce an *NVM set-aware command arbitrator* (SACA) for each PF. SACA employs an NVMe command queue per NVM set (and core). SACA distributes all the incoming NVMe commands, associated with different NVM sets, to the appropriate NVMe command queue by fetching the commands from a specific NVMe *submission queue* (SQ) in a round-robin fashion. This design can make the firmware on each core (statically assigned to different NVM sets) separately perform I/O processing on the designated NVM set (even with a limited number of PFs). Even though different host CPUs/cores of the host can in parallel issue the NVMe commands for the different NVM sets per SQ, SACA can appropriately serve them atop physically partitioned different flash channels (per NVM set). To manage the I/O completion, SACA collects the I/O completion messages (coming from different cores) and puts them into the target NVMe *completion queue* (CQ), which is associated with the submission queue. Thus, SACA can handle I/O requests, targeting to different NVM sets but being enqueued in the same SQ/CQ, with a design of limited PFs.

Prototype. Figure 2c shows the current Divided SSD that we prototyped. The prototype hardware has a front-side and back-end PCBs, each containing four flash channels and eight ways. In total, our prototype hardware employs 64 flash dies, each using 64 layered triple-level cell (TLC) flash technology. We connect them to four cores that run four different flash firmware instances. All these set-related cores are interconnected to 3GB LPDDR global memory (physically partitioned to each core). Thus, the prototype exposes four independent NVM sets (2 channels per set) over one PF, which is capable of storing 2TB data into the single storage pool of the device.

3 I/O Tacker: System Level Support

Page-frame reclaiming interference. While our Divided SSD physically separates its shared storage space into multiple NVM sets, each being mounted to different containers, the I/O interference, caused by noisy neighbors, can degrade the performance of peer containers. To be precise, we issue random reads to the first NVM set using FIO configured with the Facebook-like workload used in Section 2, while executing a memory hungry application on the other NVM sets. The configuration of such application, LEMP, is described in Section 4. Figures 3a and 3b respectively show the amount of storage accesses (footprint) and latency of the first NVM set when we increase the number of LEMP instances, each mounting different NVM sets. Even though there is no instance working on the first NVM set except for FIO, the storage footpoint

increases as the number of LEMP increases by 15.8% at most (Figure 3a). As shown in Figure 3b, this unfortunately makes the average and worst-case latency of random reads as high as 22% and 931%, respectively. The reason why we can observe the performance degradation on Divided SSD is that the memory subsystem of LEMP containers introduces extra I/O requests, targeting to the storage area, mounted to the first NVM set. In Linux, containers are treated just as processes, the I/O requests, caused by demand paging, are all together heading to the swap disk or file that the host OS designates, which cannot be solely addressed by device-level performance isolation.

Overview of ownership-based I/O isolation. To remove the impact of the memory-hungry container(s) from other peers, our I/O Tacker pins the swap area to the target NVM set based on container ownership. Note that, while cgroups' memory subsystem is configured by users (when the target container is created), page frame reclaiming is performed by the host kernel. Thus, we modify several data structures of kernel and cgroup to link the kernel and cgroup by being aware of the container ownership. In addition, we modify the Docker stack to inform our per-container swap area information to the Docker container engine. Putting it all together, noisy neighbors running page-out and page-in operations are dedicated their own NVM sets, which do not interfere with other peer containers.

Cgroup and kernel modification. To redesign the swap operations (related demand paging), we modify the kernel memory controller (`linux/mm/memcontrol.c`) and define the read and write actions of memory subsystem's cgroup. The actions are declared by our new function that modifies the memory cgroup structure (`mem_cgroup`). As shown in Figure 3c, we add two swap information to the `mem_cgroup`, one for swap type (`swap_type`) and another for swap destination (`swap_dst`). While the swap type explains whether the current cgroup uses a default page frame reclaiming method or our per-container demand paging technique, the swap destination includes the index of swap information list (`swap_info`). An entry of the swap information list (`swap_info_struct`) contains the actual swap disk or swap file information. Thus, when the host kernel evicts a page frame, it will only page in/out from the designated location, if the swap type indicates the per-container demand paging. We also modify the eviction-related kernel function, `get_swap_pages()`, to appropriately refer the target index of swap information list by referring our modified `mem_cgroup`.

Note that all page frames (`page_struct`) contain a pointer indicating `mem_cgroup`, associated with the owner processes (container as well). Thus, when the host kernel selects a victim page frame by looking up the page table at `get_swap_pages()`, the function can retrieve the index of swap information of `mem_cgroup`. In `get_swap_pages()`, we forward the retrieved index to `scan_swap_map()` such that it will update the corresponding page table entry (PTE) with `swp_entry_t` that consists of the retrieved index and the specific offset within the swap area target. Once the kernel puts the victim page to its swap cache, which will page out to the backend storage, the kernel can check a specific swap destination location from `swap_info_struct` using the list index and perform per-container demand paging.

SSD	**Capacity**	2TB	**DRAM**	LPDDR4
	# channel	8	**# die**	64
	# core for sets	4	**NAND**	64 Layered TLC
Testbed	**CPU**	Xeon E5-2690 v3 (48 cores)		
	Nginx	1.17.1	**DRAM**	DDR4 256GB
	MySQL	8.0.16	**Linux**	5.0.7+
	PHP	7.2	**Docker**	18.09.7+

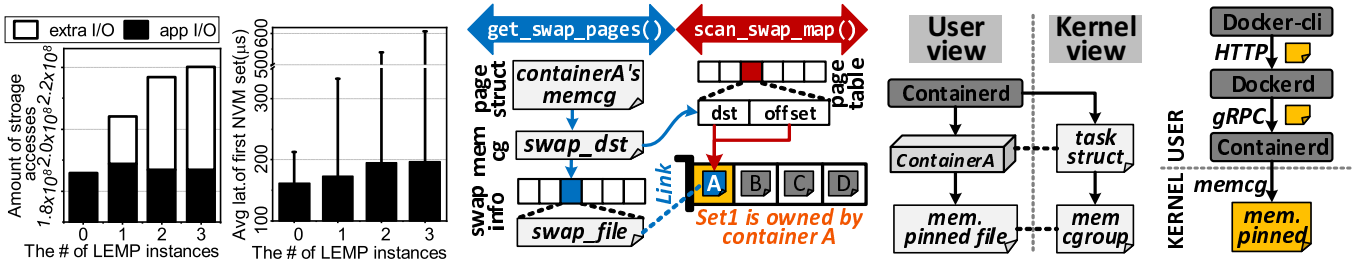
Table 1: Important parameters of our device and system.

Docker stack modification. In Docker stack, we modify the container engine (`containerd`) to write our pinned swap information parameter (indicated by `mem.pinned`) to `mem_cgroup`. Specifically, the modified `containerd` will update the swap type and destination fields of `mem_cgroup` by collaborating with our kernel memory controller. As shown in Figure 3d, when users create a container, `containerd` generates the file of `mem.pinned` under the corresponding container's memory subsystem directory (`sys/fs/cgroup/memory/docker/<container-ID>/mem.pinned`). In the file creation process, it will jump to our function and update `mem_cgroup` (associated with the host kernel) in indicating a specific NVM set, explained by `swap_info_struct`. Once this cgroup update is completed, `containerd` changes the processor control block (`task_struct`) to link the `mem.pinned` with the structure of `cgroups`. Thus, the host kernel can be aware of the swap information that the users designate.

Since the proposed I/O Tacker takes `mem.pinned` information when creating a container, we also modify Docker stack to deliver such new parameter from Docker client (`docker-cli`), Docker daemon (`dockerd`), to `containerd`, as shown in Figure 3e. As Docker modules in the stack communicate with `docker-cli` through HTTP, the modified `docker-cli` delivers the `mem.pinned` information by packing it into `containerConfig` structure using a JSON format [36]. Docker engine (`dockerd`) atop `containerd` then parses such parameter information and forwards it to `containerd` using a Google remote procedure call (gRPC). Therefore, `containerd` can access the `mem.pinned` information and create the container that uses per-container page frame reclaiming.

4 Evaluation

We prepare two real NVMe SSD hardware devices, one for a high-performance NVMe SSD and another for our Divided SSD prototype; the hardware configuration of those two is



(a) Storage footprint. (b) Degradation. (c) Per-container demand paging. (d) Container engine modif. (e) Docker stack modif.

Figure 3: Overview of system-level support (I/O Tacker).

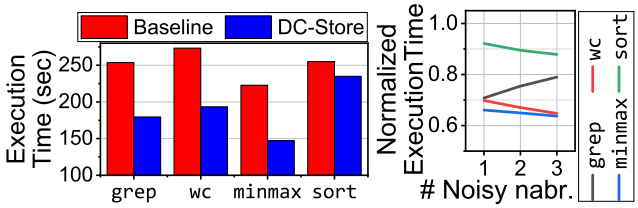
exactly same (e.g., the same number of cores, controllers, channels and flash devices). We connect each device into our testbeds; each of them employs two processors (total 48 cores), 256 GB DDR4 DRAM and uses PCIe 3.0 4 lanes for the SSD prototype connections. In the *Baseline* testbed, we connect the NVMe SSD and logically partition it into four (each offering 480 GB), while the *DC-Store* testbed uses four NVM sets of Divided SSD with I/O Tacker (modified by the same version of Linux kernel 5.0.7 that Baseline installs). All our testbeds use EXT4 as their file system. We use grep, wordcount (wc), minmax for containerized data-intensive applications, while sort is evaluated for a compute-intensive application. All the containerized applications generate their outputs by examining a set of randomly generated data vectors, stored to 630,000 files on EXT4. For noisy neighbors, we prepare LEMP, consisting of one Nginx, two PHP-FPM, and one MySQL per container. The important device information is summarized in Table 1.

Overall performance. Figure 4a shows the execution time of Baseline and DC-Store by co-running containerized applications with a LEMP stack. For the scenario of co-running data-intensive containers with LEMP, DC-Store finishes the applications in 174 secs, on average, which is 31% shorter than Baseline. Since LEMP performs demand paging, the corresponding I/O requests are sporadically issued to the NVMe volume that the data-intensive containers operate, which in turn degrades the performance of Baseline. In contrast, since Divided SSD completely splits the NVMe volumes through different channels, and I/O Tacker isolates the page frame reclaiming from the peer NVMe volumes in processing the data, the performance degradation is not observed with DC-Store. Specifically, grep, wc and minmax on DC-Store shorten the container-level execution times (compared to those of Baseline) by 29.2%, 29.2%, and 33.9%, on average, respectively. On the other hand, sort (compute-intensive) exhibits similar performance between Baseline and DC-Store (7.8% better than Baseline). Since most of the sort executions are involved in CPU processing (for data comparisons for sorting), the I/O requests from the containerized sort occasionally occur. We will dig deeper the performance of Divided SSD and I/O Tacker in Sections 4.2 and 4.3, respectively.

Note that LEMP itself shows negligible performance difference (under 0.16%, on average) for all the co-running scenarios. Even though the degree of internal parallelism per NVM set would be lower than that of Baseline's NVMe SSD, LEMP's writes related to demand paging can be enqueued and buffered by the host-side memory. Thus, the LEMP execution can be tolerable at some extent. Specifically, we observe that, compared to Baseline, the target NVM set's I/O depth increases by 56%, on average. However, since the swap cache issues such I/O requests with long intervals (60 ms in average), all the requests are served within the long intervals thereby being invisible to the users.

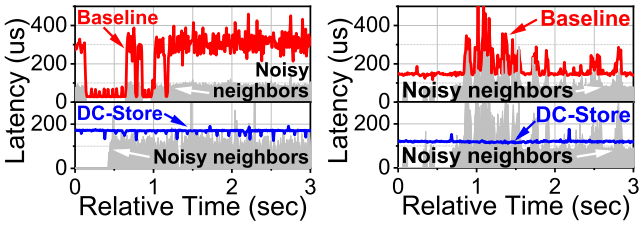
Different levels of interference. We increase the number of LEMP containers from one to three to analyze the different levels of interference, imposed by noisy neighbors; each LEMP mounts different NVM sets, and all remaining sets are used for the containerized applications. Figure 4b shows the application execution time of DC-Store, normalized to that of Baseline. We observe that DC-Store exhibits all similar execution time behaviors, which have no impact with the varying numbers of noisy neighbors. However, as Baseline performance gets worse (except for grep) due to extra reads and writes brought by demand paging, the performance of DC-Store looks get better in this normalization comparison. For Baseline, we observe that grep is a bit more sensitive on the access pattern that other grep instances generate. As increasing the number of LEMP (and relatively decreasing the number of running grep instances), Baseline shortens grep's completion time by 20 secs. Since DC-Store provides deterministic performance, which does not have impact on other peer NVM sets, the benefits on grep are only around by 21% compare to Baseline, on average (the right most of Figure 4b).

Time series analysis. Figure 5 shows a time series analysis of grep and sort as the representative of data- and compute-intensive applications, respectively. One can observe from Figure 5a, that the latency of containerized grep fluctuates as much as LEMP generates page frame reclaiming related I/O services. In contrast, DC-Store shows highly predictable and sustainable latency, irrespective of LEMP's page frame re-



(a) Execution time comparison. (b) Sensitivity.

Figure 4: DC-Store performance analysis.



(a) Data-intensive (grep). (b) Compute-intensive (sort).

Figure 5: Timeline analysis.

claiming activities. This is because paging in/out of LEMP's memory subsystem occurs only to the NVM set that LEMP owns. In contrast, such I/O services of DC-Store (at the device-level) are completely isolated from other NVM sets. The storage latency of Baseline is shorter than DC-store when there is no interference from LEMP. This is because, unlike Divided SSD, Baseline NVMe SSD can utilize the internal DRAM optimized for sequential accesses. Note that the reason why there is no I/O from LEMP between 0 and 1 sec in Figure 5a is that all I/Os during this period were absorbed by the DRAM. We will further analyze this behavior in Section 4.2. For the containerized sort, it exhibits a different performance characteristic. Since the performance of sort is dominated by the host computing power, sort introduces I/O requests sparsely (Figure 5b). Nevertheless, its latency trend of Baseline still fluctuates, which is similar to that of LEMP's page frame reclaiming activities. This, in turn, makes the containers on Baseline in overall show 35% performance degradation, compared to the execution time with no I/O interference.

4.1 Eliminating Noisy Neighbors

4.2 Device-level Analysis

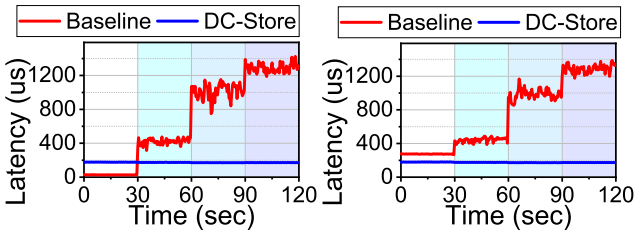
Figures 6a and 6b respectively show the level of I/O determinism for sequential and random access patterns in terms of latency. The evaluation is performed by the same scenario of flash workloads from Facebook [33] (read-intensive with bursty writes); the latency values are the read performance of the first NVM set while writing data into other NVM sets for every 30 secs. There are two insights observed in this evaluation. At the first test epoch (∼30 secs), Divided SSD is, on average, 5.8 times slower than Baseline NVMe for the sequential accesses (Figure 6a) while it exhibits better performance on random accesses (157% faster, on average). As Divided SSD physically partitions the underlying flash channels, it cannot take the full advantage of internal parallelism; Baseline's NVMe SSD reads data across all eight different channels in parallel, whereas Divided SSD only uses two channels. More importantly, this parallelism can increase if DRAM buffer is sufficiently large as the firmware can perform a read-ahead by enabling all the underlying channels. Divided SSD splits the internal DRAM region and assigns to each core, it exhibits the performance degradation per NVM set on the sequential accesses. In contrast, the latency on random accesses is difficult to take a full benefit of the read-ahead and parallelism; for the random, the resource conflict can be higher, and there is a low spatial locality, which makes the latency of Divided SSD shorter than Baseline's NVMe SSD (Figure 6b).

Note that the key benefit of hardware-level NVM set implementation is to remove I/O interference and provide deterministic performance. As shown in the figure 6a, Baseline's NVMe SSD increases the latency for the remaining epochs and shows as high as 14.8 times longer latency, compared to the first epoch test. In contrast, Divided SSD exhibits 55%, 83%, and 86% shorter latency than Baseline's NVMe SSD for 2nd, 3rd and 4th epochs, on average, respectively. The main reason why Divided SSD offers deterministic I/O performance is to statically split all SSD resources per set such that the requests on other NVM sets cannot interfere the I/O services on the first NVM set.

4.3 I/O Tacker Performance Impact

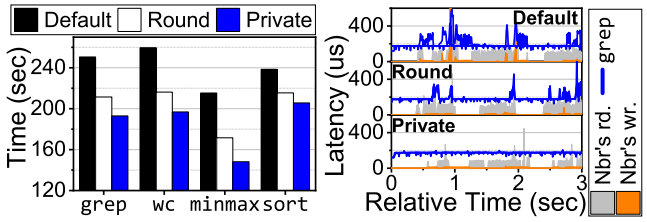
We evaluate different methods of demand paging on multiple NMVe sets to see the performance benefits solely brought by I/O Tacker. *Default* uses the default demand paging configuration of kernel and *Round* pages in/out across different NVM sets in a round-robin fashion. In contrast, *Private* employs the per-container page frame reclaiming technique of I/O Tacker, which pins the swap area to a specific NVM set that LEMP owns. In this test, only a single containerized LEMP stack is executed, but it generates demand paging related I/O requests more than the test performed in Section 4.1 by double. The user-level execution time of all containers is shown in Figure 7a. Default requires 251, 260, 215 and 239 secs to execute grep, wc, minmax and sort (by co-running with LEMP), respectively. In contrast, Private requires 193, 197, 148 and 206 secs to complete their task, on average; Private for the data-intensive applications is faster than those on Default by 23% on average. Round also shows some benefits, compared to Default, but it is still slower than Private by 10%, on average.

Figure 7b shows the results of time series analysis for the containerized grep (as the representative of data-intensive applications). The latency of Default and Round fluctuates when the reads and writes related to demand paging occur



(a) Seqential reads. (b) Random reads.

Figure 6: Read-intensive with bursty writes.



(a) Execution time. (b) Time series analysis (grep).

Figure 7: I/O Tacker performance analysis.

even with Divided SSD. This is because even though Divided SSD implements hardware-level NVM sets with static resource partitioning, the I/O requests for demand paging are managed without the consideration of container ownership. In contrast, Private keeps the predictable and sustainable latency while serving such I/O requests by pinning the swap area to the NVM set that LEMP operates on. Importantly, we observe that the execution time of LEMP with Private is 60% and 22% shorter than those with Default and Round, respectively as well. This is because Private removes the serialized I/O requests on a specific NVM set for the page in/out of LEMP.

5 Related Work

There exist several studies that virtualize an SSD by isolating its channels or flash dies [37–39]. [37] proposed an over-provisioning space isolation to reduce garbage collection overheads, and similarly, [40] proposed an SSD-aware host-level I/O scheduler to improve proportionality on a shared storage system. [38] partitioned the flash resources of Open-Channel SSD (OCSSD) [41, 42] with a kernel-level modification to enhance the storage lifetime. In similar, [39] changed kernel driver(s) to make OCSSD support multi-tenants with artificial intermixed block-level I/O traces. While the prior work virtualizes the SSD with software support or performs simulation-based studies, we split all internal resources, including flash, buffer, and core, as well as memory subsystem (demand paging) in real hardware. Our prototype of DC-Store successfully achieves deterministic I/O performance and resource isolation.

On the other hand, some prior works [40, 43, 44] that tried to make the underlying SSD be aware of different degrees of QoS are unfortunately agnostic to container-level information and Docker storage stack. There are a few OS-level virtualization studies that attempt to satisfy different levels of QoS requirements of multiple applications in a shared storage system. Specifically, [28] and [45] modify a block layer and page cache in an attempt to utilize proportional bandwidth by being aware of per-container I/O weight. Unfortunately, these prior researches cannot address the storage-level interference issues for multi-container execution environment because the block layer and page cache that throttle the bandwidth (based on per-container I/O weight) are not aware of the physical layout of SSDs and cannot split metadata I/O services from their I/O throttling.

6 Discussion

The number of NVMe set per device can be greater or smaller than the number of user applications running on Docker as the current prototype of our Divided SSD statically partitions all the internal resources at design time. While we consider designing different hardware platforms to dynamically allocate the different number of NVMe sets as future work, there are still several alternatives to map between different numbers of user applications and NVMe sets. In cases where users want to fully utilize all the bandwidth of Divided SSD with a small number of containers, it is possible to group multiple NVMe sets into a single shared storage partition using a conventional RAID scheme. In contrast, if there are containerized applications greater than the NVMe sets offered by Divided SSD, we can classify the applications based on the read and write ratios and then allocate them in homogenous groups to different NVMe sets. Even though this approach may not be optimal, it can provide a significant benefit by protecting read-intensive applications from the interference of heavy writes (requested by other containerized applications).

7 Conclusion

In this paper, we proposed DC-Store to offer deterministic I/O performance for a multi-container execution environment. We prototyped both hardware (Divided SSD) and software (I/O Tacker) modules of our DC-Store, and evaluate them in a real system, which shortens the data-intensive applications by 31%, on average, compared to those on a baseline system.

Acknowledgements

This research is mainly supported by NRF 2016R1C182015312, KAIST Start-up package (G01190015) and Samsung grant (G01190271). The authors thank John Bent of Seagate Technology for shepherding their paper. Myoungsoo Jung is the corresponding author.

References

[1] A. Eivy, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[2] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[3] J. Higgins, V. Holmes, and C. Venters, "Orchestrating docker containers in the hpc environment," in *International Conference on High Performance Computing*, pp. 506–513, Springer, 2015.

[4] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, 2015.

[5] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pp. 52–57, IEEE, 2016.

[6] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

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

[8] J. Shukla, "Application sandbox to detect, remove, and prevent malware," Jan. 17 2008. US Patent App. 11/769,297.

[9] "cgroup-v1." https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[10] "Linux namespace." http://man7.org/linux/man-pages/man7/namespaces.7.html.

[11] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 74–81, IEEE, 2017.

[12] B. P. Rimal and M. Maier, "Workflow scheduling in multi-tenant cloud computing environments," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 1, pp. 290–304, 2016.

[13] S. Sakr, "Cloud-hosted databases: technologies, challenges and opportunities," *Cluster Computing*, vol. 17, no. 2, pp. 487–502, 2014.

[14] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference (Middleware)*, p. 1, ACM, 2016.

[15] S. McDaniel, S. Herbein, and M. Taufer, "A two-tiered approach to i/o quality of service in docker containers," in *2015 IEEE International Conference on Cluster Computing (Cluster)*, pp. 490–491, IEEE, 2015.

[16] J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Docker container scheduler for i/o intensive applications running on nvme ssds," *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, vol. 4, no. 3, pp. 313–326, 2018.

[17] "VirtualBox." https://www.virtualbox.org/.

[18] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 386–393, IEEE, 2015.

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

[20] S. Park, E. Seo, J. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based ssds," *IEEE Computer Architecture Letters (CAL)*, vol. 9, no. 1, pp. 9–12, 2010.

[21] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (o3): An out-of-order flash memory controller architecture," *IEEE Transactions on Computers (TC)*, vol. 60, no. 5, pp. 653–666, 2010.

[22] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance," *IEEE Transactions on Computers (TC)*, vol. 62, no. 6, pp. 1141–1155, 2012.

[23] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the International Conference on Supercomputing (ICS)*, pp. 96–107, ACM, 2011.

[24] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 524–535, IEEE, 2014.

[25] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "Hios: A host interface i/o scheduler for solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 289–300, 2014.

[26] M. Jung, E. H. Wilson III, and M. Kandemir, "Physically addressed queueing (paq) improving parallelism in solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 404–415, 2012.

[27] "cgroup-v1 blkio." https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.

[28] S. Ahn, K. La, and J. Kim, "Improving i/o resource sharing of linux cgroup for nvme ssds on multi-core systems," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[29] M. Jung, "Exploring parallel data access methods in emerging non-volatile memory systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 3, pp. 746–759, 2016.

[30] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of ultra-low latency solid state drives," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[31] "NVM Express Base Specification Revision 1.4." https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[32] C. Petersen and A. Huffman, "Solving latency challenges with nvm express ssds at scale," *Flash Memory Summit*, 2017. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_SIT6_Petersen.pdf.

[33] C. Petersen, W. Zhang, and A. Naberezhnov, "Enabling nvme i/o determinism at scale," *Flash Memory Summit*, 2018. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-102A-1_Petersen.pdf.

[34] J. Axboe, "Flexible i/o tester." https://github.com/axboe/fio.

[35] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–481, IEEE, 2018.

[36] "Docker engine." https://docs.docker.com/engine/api/v1.24/.

[37] J. Kim, D. Lee, and S. H. Noh, "Towards slo complying ssds through ops isolation," in *13th USENIX Conference on File and Storage Technologies (FAST)*, pp. 183–189, 2015.

[38] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds," in *15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 375–390, 2017.

[39] J. González and M. Bjørling, "Multi-tenant i/o isolation with open-channel ssds," in *Nonvolatile Memory Workshop (NVMW)*, 2017.

[40] J. Kim, E. Lee, and S. H. Noh, "I/o scheduling schemes for better i/o proportionality on flash-based ssds," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 221–230, IEEE, 2016.

[41] M. Bjørling, J. González, and P. Bonnet, "Lightnvm: The linux open-channel ssd subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 359–374, 2017.

[42] "Open-Channel Solid State Drives Specification Revision 2.0." http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf.

[43] W. Shin, M. Kim, K. Kim, and H. Y. Yeom, "Providing qos through host controlled flash ssd garbage collection and multiple ssds," in *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, pp. 111–117, IEEE, 2015.

[44] H. Park, S. Yoo, C.-H. Hong, and C. Yoo, "Storage sla guarantee with novel ssd i/o scheduler in virtualized data centers," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 8, pp. 2422–2434, 2015.

[45] K. Oh, J. Park, and Y. I. Eom, "Weight-based page cache management scheme for enhancing i/o proportionality of cgroups," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–3, IEEE, 2019.