Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems

Sungyong Ahn, Kwanghyun La
Memory Division, Samsung Electronics Co.
Hwasung, Korea
{sungyong.ahn, nala.la}@samsung.com

Jihong Kim
Dept. of CSE, Seoul National University
Seoul, Korea
jihong@davinci.snu.ac.kr

Abstract

In container-based virtualization where multiple isolated containers share I/O resources on top of a single operating system, efficient and proportional I/O resource sharing is an important system requirement. Motivated by a lack of adequate support for I/O resource sharing in Linux Cgroup for high-performance NVMe SSDs, we developed a new weight-based dynamic throttling technique which can provide proportional I/O sharing for container-based virtualization solutions running on NUMA multi-core systems with NVMe SSDs. By intelligently predicting the future I/O bandwidth requirement of containers based on past I/O service rates of I/O-active containers, and modifying the current Linux Cgroup implementation for better NUMA-scalable performance, our scheme achieves highly accurate I/O resource sharing while reducing wasted I/O bandwidth. Based on a Linux kernel 4.0.4 implementation running on a 4-node NUMA multi-core systems with NVMe SSDs, our experimental results show that the proposed technique can efficiently share the I/O bandwidth of NVMe SSDs among multiple containers according to given I/O weights.

1 Introduction

Container-based virtualization is emerging as a key cloud computing platform for serving various cloud services because it allows multiple isolated instances (called containers) to share system resources more efficiently over hypervisor-based virtualization. In container-based virtualization, since multiple containers run independently on top of a single common operating system, it is important for a kernel-level resource manager to support resource isolation and sharing in an efficient and proportional fashion among multiple containers with different service requirements.

Linux Cgroup [1] is such a resource control framework in Linux which supports many container-based virtualization solutions such as Linux container (LXC), Docker and libcontainer [2, 3]. Linux Cgroup manages, for example, the I/O bandwidth of a storage system in a proportional way so that the total I/O bandwidth of the storage system can be properly shared among multiple containers.

Although Linux Cgroup efficiently supports proportional I/O sharing for SATA-based HDDs/SSDs inside the CFQ I/O scheduler at the single-queue block layer, the current Cgroup implementation does not adequately support I/O resource sharing for recent high-performance SSDs (such as NVMe SSDs). For example, since these high-performance SSDs, which can achieve more than 1 million IOPS, need to work with the newly proposed multi-queue block layer [4] for realizing its performance potential, the existing proportional I/O sharing scheme, which was implemented at the single-queue block layer, cannot be used. In this paper, we propose a weight-based dynamic throttling scheme for NVMe SSDs which can provide efficient and proportional I/O sharing. Our proposed throttling scheme is implemented as an extension to the existing I/O throttling layer of Linux Cgroup.

While implementing the proposed scheme, we also discovered that the current Linux Cgroup is not scalable on NUMA multicore systems when it works with high-performance NVMe SSDs. Since these NVMe SSDs are expected to be shared in practice by a large number of containers (because of their high bandwidth as well as their high capacity), it is an important requirement for Linux Cgroup to work in a scalable way as the number of containers increases. Furthermore, since a host system for these SSDs are likely to be based on NUMA multi-core systems, Linux Cgroup should support NUMA-aware scalable I/O sharing as well. In order to make the proposed scheme to be NUMA-scalable, we modified Linux Cgroup to employ per-container locks instead of sharing a single request-queue lock among multiple containers.

In order to understand the effectiveness of our proposed improvements to the current Cgroup implementation, we implemented the proposed scheme on Linux kernel 4.0.4 running on a 4-node NUMA multi-core system and evaluated it using Samsung XS1715 NVMe SSDs [5]. The experimental results show that our scheme can efficiently share the I/O bandwidth of NVMe SSDs among multiple containers in proportion to their I/O weights with scalable performance.

The remainder of this paper is organized as follows. Sec. 2 explains the limitations of the current Linux Cgroup when NVMe SSDs are shared among containers. Sec. 3 describes the proposed I/O resource sharing scheme. Experimental results are presented in Sec. 4. Sec. 5 summarizes related work. Sec. 6 concludes with a summary and future work.

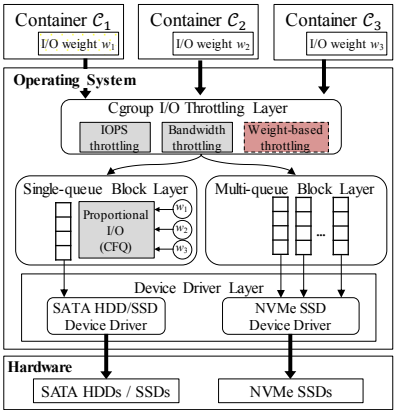

Fig. 1: An overview of the I/O resource control in Linux Cgroup.

2 Limitations of Linux Cgroup for NVMe SSDs

In this section, we evaluate how the existing I/O resource control mechanisms of Linux Cgroup work with NVMe SSDs in sharing I/O resource among multiple containers. As shown in Fig. 1, in Linux Cgroup, I/O resource sharing can be supported at two layers, the Cgroup I/O throttling layer and single-queue block layer.

For SATA HDDs and SSDs, proportional I/O resource sharing has been supported inside the CFQ I/O scheduler of the single-queue block layer [6]. However, since Linux kernel 3.13, NVMe SSDs have been supported under the multi-queue block layer because the single-queue block layer cannot achieve a high performance potential of NVMe SSDs [4]. Therefore, the existing CFQ-based proportional I/O policy cannot be reused for NVMe SSDs.

Linux Cgroup also provides I/O throttling at the Cgroup I/O throttling layer which can be used for I/O resource sharing by limiting the maximum I/O bandwidth or maximum IOPS available for each container. As a simple proportional I/O sharing solution at the Cgroup I/O throttling layer, we developed a static throttling scheme, *ST*, which assigns different upper limits on the read bandwidth and write bandwidth to containers according to their I/O weights.

In order to quantitatively evaluate the limitation of the existing Cgroup resource sharing mechanisms (including *ST*) for NVMe SSDs, we performed simple experiments using four containers, C_1, C_2, C_3, and C_4, where the I/O weight ratios among four containers are given as 10:5:2.5:1. For the experiments, a Dell R920 with 4 Samsung XS1715 NVMe SSDs was used. R920 has 4 NUMA nodes where each NUMA node supports 12 CPU cores. We created four containers using LXC [2]. Each container ran the I/O workloads summarized in Table 1[1]. As shown in Fig. 2, the default Cgroup policy, *BASELINE*, has no support for proportional I/O sharing for NVMe SSDs, thus producing meaningless resource sharing result. Although *ST*, which assigns the maximum bandwidth allowed for each container based on the I/O weight of the container, works much better than *BASELINE*, it still performs poorly for proportional I/O sharing. For example, the required I/O weight ratio of C_1 to C_2 is 2:1, but *ST* achieves the ratio of 9.9 to 2.2.

Table 1: Characteristics of I/O workloads in four containers.

Container	Workload	Request size (total / average)	R : W
C_1	Exchange	126.7GB / 9.3KB	0.34 : 1
C_2	MSNMeta	71.5GB / 5.0KB	1.94 : 1
C_3	MSNFS	56.0GB / 5.1KB	1.67 : 1
C_4	Finance	28.2GB / 2.7KB	0.87 : 1

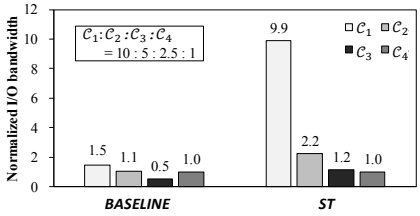

Fig. 2: Evaluation results of proportional I/O sharing in Linux Cgroup.

The poor performance of *ST* can be attributed to two main factors. First, although the static throttling approach used in *ST* is effective in guaranteeing that no container is allocated with the I/O bandwidth over the specified maximum bandwidth, it is not useful to meet required I/O weights of containers. Furthermore, *ST* is likely to waste the I/O bandwidth allocated for a container if the container is not I/O-intensive. For example, Fig. 3 shows C_1 wastes a significant amount of the allocated read bandwidth because its read request are not intensive enough to fully consume the allocated read bandwidth.

Second, *ST* separately manages read bandwidth and write bandwidth (following the basic throttling mechanism of the Cgroup I/O throttling layer), making it difficult to manage the I/O bandwidth in an integrated fashion. For example, Fig. 4 shows that C_2 consumed most of the allocated read bandwidth but it significantly under-utilized the allocated write bandwidth. Since this asymmetric I/O consumption pattern between reads and writes is application specific (e.g., MSNMeta is read-intensive of C_2), *ST* cannot easily estimate the required read bandwidth and write bandwidth in advance. Therefore *ST* may waste a significant amount of the allocated read / write bandwidth.

Our proposed scheme improves these two weaknesses of *ST* by dynamically adjusting each container's maximum I/O bandwidth by predicting future I/O demands and managing both the read bandwidth and write bandwidth in a combined fashion.

3 Weight-based Dynamic Throttling Scheme

In this section, we describe our proposed weight-based dynamic throttling scheme, *WDT*, for NVMe SSDs.

[1] We used the block I/O trace replay tool [7] to generate I/O requests from the workloads in Table 1. (These traces are from UMass [8] and SNIA [9]). In our experiments, these workloads were executed by 12 concurrent threads with a queue depth of 32.

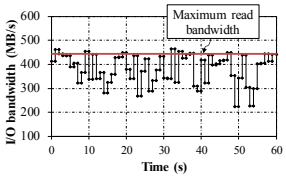

Fig. 3: Under-utilized read bandwidth in $\mathcal{C}_1$.

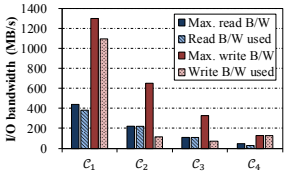

Fig. 4: Asymmetric I/O bandwidth consumption between reads and writes.

3.1 Key Design Decisions

As the first design decision, we decided to implement *WDT* at the Cgroup I/O throttling layer instead of at the multi-queue block layer as shown in Fig. 1. Our decision is affected by three factors: 1) adding a new policy at the I/O throttling layer is easier by reusing most of the existing throttling layer code, 2) implementing a CFQ-like I/O scheduler for the multi-queue block layer can be quite expensive (because per-process I/O scheduling queues necessary in the CFQ I/O scheduler incurs a large overhead in the multi-queue block layer), and 3) employing an I/O scheduler is not recommended for high-performance SSDs.

Another important decision we made in designing the current *WDT* scheme was how to define *I/O proportionality*. An *ideal* proportional I/O sharing technique must satisfy the required I/O weight ratios among containers both *locally* and *globally*. By locally-proportional I/O sharing, we mean that the I/O weight ratios are satisfied among *I/O-active* containers for a given short time interval. On the other hand, in a globally-proportional I/O sharing technique, the total I/O resource usage of multiple containers (over entire execution times) should be proportionally maintained. Since even formally defining the requirements of an ideal proportional I/O sharing technique is challenging, in the current version of *WDT*, we focus on locally-proportional I/O sharing only.

3.2 Overview of *WDT*

In order to support locally-proportional I/O sharing in *WDT* using dynamic throttling, we employ an interval-based approach. A fixed-length interval I_{tw}, called as the throttling window, is used as a basic unit of I/O resource control in *WDT*. (We denote the size of the throttling window as $size_{tw}$.) For the j-th throttling window I_{tw}^j, we associate the following three parameters for a container $\mathcal{C}_k$: $\mathcal{B}_k^j$, $\mathcal{U}_k^j$, $\mathcal{R}_k^j$. The credit budget $\mathcal{B}_k^j$ of the container $\mathcal{C}_k$ for I_{tw}^j indicates the total number of sectors that $\mathcal{C}_k$ can request (either by reads or writes) during the j-th throttling window I_{tw}^j. The used credit $\mathcal{U}_k^j$ of the container $\mathcal{C}_k$ represents the total number of credits consumed by $\mathcal{C}_k$ during I_{tw}^j. The residual credit $\mathcal{R}_k^j$ of the container $\mathcal{C}_k$ indicates the remaining credits not consumed during I_{tw}^j. $\mathcal{R}_k^j$ is carried over to the next throttling window I_{tw}^{j+1}. Whenever an I/O request of $\mathcal{C}_k$ is serviced, $\mathcal{U}_k^j$ is incremented by the number of sectors serviced.

In order to decide whether the current I/O request should be issued or throttled under I/O proportionality requirements, we check if $\mathcal{U}_k^j$ is smaller than the sum of $\mathcal{B}_k^j$ and $\mathcal{R}_k^j$. If $\mathcal{U}_k^j$ is smaller, that is, if there are remaining credits available, the current I/O request is issued. Otherwise, it is throttled until the next throttling window.

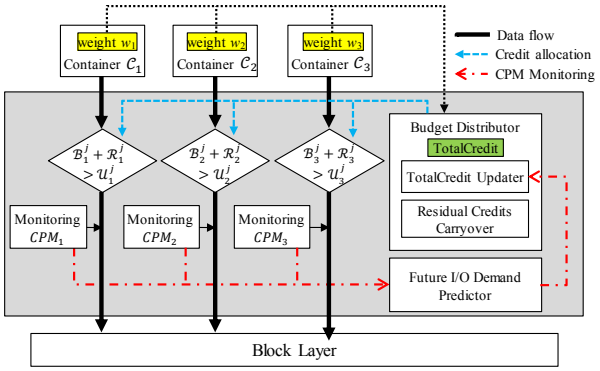

Fig. 5: An organizational overview of *WDT*.

An overview of the proposed *WDT* scheme is shown in Fig. 5. The *WDT* scheme consists of two main functions. The future I/O demand predictor is responsible for estimating a future I/O demand of the container $\mathcal{C}_k$. *WDT* monitors the I/O service rate of $\mathcal{C}_k$ for I_{tw}^j, which we denote as CPM_k^j (Credits per Millisecond), and computes the future I/O demand of $\mathcal{C}_k$ based on the cumulated past CPM_k values. Once future I/O demands of the containers are predicted for the next throttling window I_{tw}^{j+1}, the total amount of credits required for the next throttling interval, *TotalCredit*, is computed. The budget distributor then updates $\mathcal{B}_k^{j+1}$ values for the containers by distributing *TotalCredit* to each container based on its I/O weight.

3.3 Future I/O Demand Predictor

The key step of *WDT* is to compute *TotalCredit* for each throttling window. Since the budget distributor simply divides *TotalCredit* based on I/O weights of containers, the efficiency of *WDT* largely depends on the accuracy of predicting *TotalCredit*. If *TotalCredit* is overestimated by a larger amount than an actual total number of credits necessary for the next throttling window, it may be difficult to meet proportional I/O requirements because some containers may consume too many credits while others have no usable credits left. On the other hand, if *TotalCredit* is underestimated, the overall I/O performance may be degraded because it may throttle I/O requests more than necessary. Therefore, accurately predicting *TotalCredit* is important in *WDT*. Furthermore, in order to reduce the overhead of updating *TotalCredit*, *WDT* only updates *TotalCredit* every N throttling windows (which we call the update window of *TotalCredit*). We denote the length of this update window as $size_{upd}$. Note that since we recompute *TotalCredit* every update window, $\mathcal{B}_k^j$'s are also updated only once per update window. However, $\mathcal{U}_k^j$'s and $\mathcal{R}_k^j$'s are still updated every throttling window.

Let $TotalCredit_p$ represent *TotalCredit* computed at the p-th update window. In order to compute $TotalCredit_{p+1}$ close to an actual I/O demand, we first es-

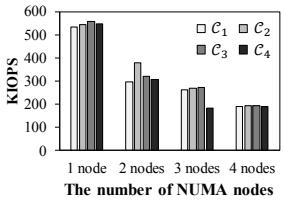

Fig. 6: I/O throughput of containers with varying number of NUMA nodes.

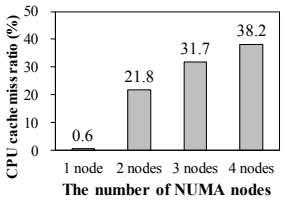

Fig. 7: CPU cache miss ratio with varying number of NUMA nodes.

timate the future credit budget $\mathcal{B}_{max}^{j+1}$ of the container $\mathcal{C}_{max}$ which had the highest I/O weight at the p-th update window.

Assuming that $\mathcal{C}_{max}$ is also the container with the highest I/O weight at the $(p+1)$-th window[2], we can estimate $TotalCredit_{p+1}$ as follows:

$$TotalCredit_{p+1} = TotalCredit_p \times \frac{\mathcal{B}_{max}^{j+1}}{\mathcal{B}_{max}^{j}}$$

$\mathcal{B}_{max}^{j+1}$ can be conservatively estimated as follows:

$$\mathcal{B}_{max}^{j+1} = CPM_{max}^{Nth} \times size_{tw}$$

where CPM_{max}^{Nth} is the Nth percentile of a cumulative distribution of CPM_{max} values. In the current *WDT* scheme, we used the 80th percentile value based on our empirical evaluation[3]. Since maintaining an entire cumulative distribution histogram incurs a large overhead inside the kernel, we instead use the *probit* function [10], a well-known quantile function associated with the standard normal distribution. Using the probit function, the 80th percentile CPM_k^{80th} of CPM_k can be calculated as follows:

$$CPM_k^{80th} = Mean(CPM_k) + Std(CPM_k) * probit(0.8)$$

where $Mean(CPM_k)$ and $Std(CPM_k)$ are the average and standard deviation of a cumulative distribution of CPM_k values.

3.4 Residual Credit Carryover

Although the current *WDT* scheme focuses on achieving locally-proportional I/O sharing among locally I/O-active containers, *WDT* tries to improve the overall I/O performance by reducing wasted credit budgets of containers. In order to minimize wasted credits allocated for a container $\mathcal{C}_k$ for I_{tw}^j, the container maintains the residual credit $\mathcal{R}_k^j$ for each I_{tw}^j. $\mathcal{R}_k^{j+1}$ is computed as $\mathcal{B}_k^j + \mathcal{R}_k^j - \mathcal{U}_k^j$. When the I/O behavior of $\mathcal{C}_k$ suddenly changes (for example, almost no I/O requests for I_{tw}^j), most of $\mathcal{B}_k^j$ are wasted unless they are carried over for future usage. By using $\mathcal{R}_k^j$, *WDT* can use the unused credits in a future throttling window when $\mathcal{C}_k$ needs higher I/O bandwidth.

3.5 Per-container Lock for Performance Scalability

While developing the *WDT* scheme, we discovered a scalability problem of the current Cgroup throttling layer implementation on a NUMA machine with high performance NVMe SSDs. Fig. 6 illustrates the NUMA scalability problem using a four-container example where each container runs three FIO processes and each FIO process intensively generates 4-KB random rad requests. As shown in Fig. 6, the read bandwidth sharply drops when more than one NUMA nodes are used on a Dell R920 machine (with four NUMA nodes).

The main source of this scalability problem is that a single request-queue lock is shared among all containers (i.e., all FIO processes) whenever an I/O bandwidth threshold is checked. Since multiple containers running on different NUMA nodes will continuous incur expensive cacheline invalidation operations when the shared lock is updated, the read bandwidth is very quickly degraded as the number of containers running on different NUMA nodes increases. Fig. 7 shows that CPU cache miss ratio sharply increases when multiple containers issue I/O requests from more than one NUMA node. The performance impact of the increased cache misses, however, depends on the performance level of a target storage system. For example, in slower HDDs, the performance penalty from the increased cache misses was insignificant because HDDs performed slowly. On the other hand, for NVMe SSDs, this penalty directly affects the I/O throughput as shown in Fig. 6.

In order to solve the scalability problem, we adopted per-container locks instead of a single request-queue lock at the I/O throttling layer of Linux Cgroup. Since *WDT* requires container-local information only, it is not necessary to use a global lock shared by all the containers. Fine-grained per-container locks make the I/O throttling layer operate independently from other containers. The experimental result (in Sec. 4) shows that our simple modification significantly reduces performance degradation from the I/O scalability problem.

4 Experiment Results

4.1 Experimental Setup

The proposed *WDT* scheme was implemented in Linux kernel 4.0.4 and evaluated on a Dell R920 machine configuration described in Sec. 2. In evaluations, four real-world workloads (described in Table 1) are used as well as a synthetic workload (based on FIO [11]). We set $size_{tw}$ to be 100 ms (which is the throttling window size used in the original Linux Cgroup). Since CPM_k^{80th} tends to be changed slowly, we set $size_{upd}$ to the 10 times of $size_{tw}$ (i.e., 1 s).

We evaluate three schemes, *ST*, *WDT,* and *WDT-*, where *WDT-* works in the same way as *WDT* except that a single request-queue lock is used for all containers.

4.2 Results

Fig. 8 shows how *WDT* satisfies different I/O weight combinations for four containers using read-world workloads of Table 1. For four different cases, *WDT* very accurately satisfies the proportional sharing requirements.

[2] In most cases, this assumption holds for our experiments. For a few cases where $\mathcal{C}_{\max}$ changes at the next throttling window, $\mathcal{B}_k^{j+1}$ values may be inaccurate. However, *WDT* quickly catches up this mistake within several subsequent throttling windows.

[3] Choosing a right CPM_{max}^{Nth} value is not trivial. Since we estimate the future budget for C_{max} using CPM_{max}^{Nth}, the best CPM_{max}^{Nth} is workload-dependent. Designing a better solution (e.g., choosing CPM_{max}^{Nth} values in a workload-adaptive fashion) is one of our future *WDT* extensions.

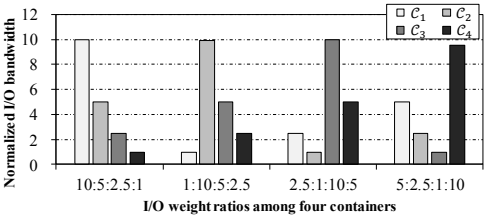

Fig. 8: Evaluation results of proportional I/O sharing under *WDT* with real-world workloads.

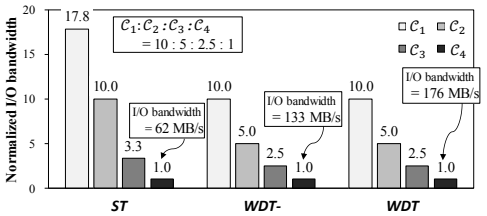

Fig. 9: Evaluation results of proportional I/O sharing in *ST*, *WDT-* and *WDT* schemes with FIO processes.

Fig. 9 compares *ST*, *WDT-* and *WDT* for their proportional I/O support using synthetic workloads. In this evaluation, each container generates 4-KB random read and write requests intensively by using FIO processes. A ratio of read to write in each container was set differently from 90% (in $\mathcal{C}_1$), 80% (in $\mathcal{C}_2$), 70% (in $\mathcal{C}_3$) and 60% (in $\mathcal{C}_4$). Both *WDT* and *WDT-* can meet the proportional sharing requirement while *ST* cannot. It is because *ST* cannot properly handle asymmetric bandwidth consumption behaviors.

Moreover, as shown in Fig. 9, *WDT* achieves much higher I/O bandwidth for four containers over *WDT-*. For example, $\mathcal{C}_4$ achieves an I/O bandwidth of 176 MB/s under *WDT* while $\mathcal{C}_4$ reaches only up to an I/O bandwidth of 133 MB/s under *WDT-*. This difference in the achieved I/O bandwidth between *WDT* and *WDT-* shows that *WDT* significantly reduces the overhead of a shared lock at the Cgroup throttling layer. The cache miss ratio under *WDT* was 12.8 % only while that under *WDT-* was 32.4%.

5 Related Work

Several research groups have proposed I/O resource control schemes based on credit allocation and throttling such as SLEDS [12], RW(D) [13] and SARC [14]. Unlike our scheme, SLEDS and RW(D) are not fully work-conserving because they lack a mechanism for utilizing spare bandwidth. Although SARC is work-conserving, it is rather ineffective in meeting the I/O proportionality because residual credits are not accounted in future credit allocation. Our scheme, on the other hand, is fully work-conserving while satisfying the required I/O proportionality very accurately by updating the total credit amounts depending on estimated future I/O demands and fully accounting residual credits for each throttling window.

6 Conclusions

We have presented an I/O resource management technique, *WDT*, for supporting proportional I/O resource sharing in Linux Cgroup on NUMA multi-core machines with NVMe SSDs. In order to overcome the shortcomings of the existing throttling policy, *WDT* employs a dynamic throttling approach by intelligently predicting the future I/O demands of each container and manages reads and writes in a combined fashion. Our evaluation results show that the *WDT* technique achieves very accurate proportional I/O resource sharing. By employing per-container locks, *WDT* also achieves NUMA-scalable high I/O performance as well. The proposed *WDT* can be extended in several directions. For example, as described in [15,16], the problem of proportional I/O sharing should be solved in a cross-layer fashion. Although the current version of *WDT* has focused on the block layer only, we plan to extend the *WDT* scheme to consider multiple layers (e.g., a file system and a page cache) in an integrated fashion.

7 Acknowledgments

We would like to thank, Vijay Chidambaram, our shepherd, and anonymous reviewers for their valuable suggestions. Jihong Kim was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2013R1A2A2A01068260), and the Next-Generation Information Computing Development Program through the NRF funded by the MSIP (NRF-2015M3C4A70656 45).

References

[1] Cgroups, https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2016.
[2] LXC, https://linuxcontainers.org/lxc/introduction/, 2016.
[3] R. Rosen, "Linux containers and the future cloud," Linux Journal, 2014.
[4] M. Bjørling *et al*., "Linux block IO: introducing multi-queue SSD access on multi-core systems," in Proc. of the 6th Int. Systems and Storage Conf., 2013.
[5] Samsung XS1715 NVMe PCIe SSD, http://www.samsung.com/us/business/oem-solutions/pdfs/XS1715_ProdOverview_2014_October_v1.pdf Oct, 2014.
[6] J. Axboe, "Linux block IO - present and future," in Proc. of Ottawa Linux Symp., 2004.
[7] Trace-replay, https://bitbucket.org/yongseokoh/trace-replay, 2016.
[8] UMass trace repository, http://skuld.cs.umass.edu/traces/storage/SPC-Traces.pdf, 2002.
[9] D. Narayanan *et al*., "Migrating server storage to SSDs: analysis of tradeoffs," in Proc. of the 4th ACM European Conf. on Computer Systems, 2009.
[10] C. I. Bliss, "The Method of probits," Science, vol. 79, no. 2037, pp. 38-39, 1934.
[11] FIO, http://freecode.com/projects/fio, 2016.
[12] D. D. Chambliss *et al*., "Performance virtualization for large-scale storage systems," In Proc. of the 22nd Int. Symp. on Reliable Distributed Systems, 2003.
[13] W. Jin *et al*., "Interposed proportional sharing for a storage service utility," in Proc. of Int. Conf. on Measurement and Modeling of Computer Systems, 2004.
[14] J. Zhang *et al*., "Storage performance virtualization via throughput and latency control," in Proc. of 13th IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005.
[15] J. Kim *et al*., "Towards SLO complying SSDs through OPS isolation," in Proc. of the 13th USENIX Conf. on File and Storage Technologies, 2015.
[16] S. Yang *et al*., "Split-level I/O scheduling," in Proc. of the 25th Symp. on Operating Systems Principles, 2015.