Weight-Aware Cache for Application-Level Proportional I/O Sharing

Jonggyu Park and Young Ik Eom

Abstract—Virtualization technology has enabled server consolidation where multiple servers are co-located on a single physical machine to improve resource utilization. In such systems, proportional I/O sharing is critical to meet the SLO (Service-Level Objectives) of the applications running in each virtual instance. However, previous studies focus on block-level I/O proportionality without considering the upper-layer I/O caches, which handle I/O requests on behalf of the underlying storage devices, thereby failing to achieve application-level proportional I/O sharing. To overcome this limitation, we propose a new cache management scheme, Weight-aware Cache (WaC), which reflects the I/O weights on cache allocation and reclamation. Specifically, WaC prioritizes higher-weighted applications in the lock acquisition process of cache allocation by re-ordering the lock waiting queue based on I/O weight. Additionally, WaC keeps the number of cache entries of each application proportional to its I/O weight, through weight-aware cache reclamation. To verify the efficacy of our scheme, we implement and evaluate WaC on both the page cache and bcache. The experimental results demonstrate that our scheme improves I/O proportionality with negligible overhead in various cases.

Index Terms—Cache, cloud computing, operating systems, resource management

1 Introduction

Virtualization has become an essential building block for modern sever systems[1], [2], [3], [4], [5]. One of its fascinating benefits is the consolidation of multiple servers into a single physical machine, thereby maximizing resource utilization. However, server consolidation inherently forces multiple servers to share underlying system resources, making it difficult to meet the SLO (Service-Level Objective) of each application[6], [7], [8], [9]. To achieve SLO guarantees, the majority of virtualization techniques, such as KVM and Docker, rely on cgroup for controlling the host kernel resources[10], [11], [12], [13].

Cgroup [11] is a Linux kernel feature that limits and controls various kinds of system resources, including CPU, memory, and block I/O. Particularly, cgroup collaborates with I/O schedulers to proportionally distribute I/O resources using I/O weight. However, cgroup is designed solely to achieve block-level I/O proportionality, rather than application-level I/O proportionality. Accordingly, it regulates I/O activities only in the block layer without giving consideration to the upper layers of the system software stack, such as caching layers. As a result, when an application utilizes I/O caches, such as bcache and the page cache, the application-level I/O proportionality can be distorted. In other words, even though the bandwidths of the applications are proportional to their I/O weights at the block layer, the applications actually experience disproportional I/O bandwidth in the holistic point of view.

The concept of caches is widely utilized to bridge the performance gap between fast and slow media. For example, Linux kernel assigns a certain amount of main memory as page cache to remedy relatively slow access to storage devices. The page cache handles I/O requests on behalf of the underlying storage devices [14], thereby achieving enormous performance improvement. Similarly, bcache[15] has been developed to hide the high latency of slow storage devices in the block layer by utilizing comparatively faster storage devices. For example, the poor random I/O performance of hard disk drives (HDD) can be alleviated by using flash storage devices as bcache. However, the design principles of conventional I/O cache management schemes do not include application-level I/O proportionality, and the ability to cooperate with cgroup is insufficient to realize application-level proportional I/O sharing. Therefore, the adoption of such I/O cache can cause dis-proportionality of I/O performance.

The conventional I/O cache management consists of two phases, cache allocation and reclamation. Cache allocation handles the upcoming I/O operations by allocating a new cache entry and temporarily storing the data inside the cache. Since cache is shared by multiple threads, cache allocation is often protected by synchronization techniques, such as spinlock. For example, the page cache and bcache utilize qspinlock, which manages multiple lock acquisition requests in a lock waiting queue. However, the qspinlock is implemented based on FIFO-queue, where the oldest element of the queue is serviced first [16], [17]. Thus, regardless of I/O weight, the lock acquisition order follows the

- *Jonggyu Park is with the Department of Platform Software, Sungkyunkwan University, Suwon 16419, South Korea. E-mail: jonggyu@skku.edu.*
- *Young Ik Eom is with the Department of Electrical and Computer Engineering/College of Computing and Informatics, Sungkyunkwan University, Suwon 16419, South Korea. E-mail: yieom@skku.edu.*

Manuscript received 27 Apr. 2021; revised 9 Nov. 2021; accepted 14 Nov. 2021. Date of publication 19 Nov. 2021; date of current version 8 Sept. 2022.
This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) Grant funded by the Korea government (MSIT) under Grant 2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays).
(Corresponding author: Young Ik Eom.)
Recommended for acceptance by D.B. Whalley.
Digital Object Identifier no. 10.1109/TC.2021.3129366

order of enqueueing. Accordingly, the conventional cache management schemes cannot reflect the I/O weight in cache allocation.

Cache reclamation secures free cache entries by evicting the existing ones. Since cache reclamation decides which data the cache will keep, it significantly affects the performance of read operations. Conventional cache management schemes often adopt an LRU policy, which considers recency of page references without considering the I/O weight. Thus, the conventional cache reclamation can reclaim pages used by higher-weighted applications ahead of those used by lower-weighted ones, even when the pages have similar reference characteristics. In this way, the use of I/O cache can distort application-level I/O proportionality.

To realize application-level proportional I/O sharing, we introduce a new cache management scheme, called *WaC* (Weight-aware Cache). *WaC* reflects the I/O weight in the process of cache allocation and reclamation. The cache allocation scheme of *WaC* prioritizes higher-weighted applications in the lock acquisition process of cache allocation. When the lock for cache allocation is available, *WaC* traverses the entire lock waiting queue and detects the element with the highest I/O weight. Afterward, *WaC* re-orders the lock waiting queue based on the I/O weights so that the element with the highest I/O weight can possess the lock in the next turn. The reordering job can degrade scalability in the case when the next lock holder is located in a different NUMA node. To prevent this, *WaC* also considers the NUMA topology when deciding the next lock holder. Through this process, *WaC* can achieve proportional I/O sharing according to the I/O weight of each application.

Re-ordering the lock waiting queue based on I/O weight can incur a problem, called starvation, where an application fails to acquire the lock for very long time. When there are many applications with high I/O weights, the lower-weighted applications should keep yielding their chances for lock acquisition, thereby experiencing starvation. To solve the problem, we adopt a conventional well-known technique for starvation, called aging. *WaC* continuously increments I/O weights of the applications that yield their turns of lock acquisition at each re-ordering phase. Therefore, lower-weighted applications can acquire the lock in a finite time, avoiding the starvation problem.

The cache reclamation scheme of *WaC* prioritizes cache entries that are used by higher-weighted applications while considering the recency of the cache references. *WaC* keeps track of the owner application of each cache entry and the number of cache entries owned by each application. During cache reclamation, *WaC* decides cache entries for eviction by considering the I/O weights of the owner applications and their current number of cache entries. By doing so, *WaC* makes the number of cache entries of each application proportional to their I/O weights. Consequently, the higher-weighted applications can possess more cache entries in the I/O cache, which in turn improves the read performance of such applications, compared with lower-weighted ones.

To verify the effectiveness of our scheme, we implement *WaC* in two I/O cache systems: the kernel page cache and a generic block-layer caching kernel module (bcache). In our experiments with the Docker virtualization, we measured the application-level I/O proportionality and the performance of *WaC*, while comparing them with those of conventional cache management schemes. Evaluation results with real-world benchmarks indicate that *WaC* displays up to 36.9% better I/O proportionality than the conventional scheme with only 3.9% overhead at most.

The rest of this paper is organized as follows. Section 2 elaborates on the background, and Section 3 demonstrates the motivation of our work. Section 4 explains the design of *WaC* in detail. The implementation of *WaC* on the page cache and bcache is presented in Section 5. Experimental results are provided in Section 6. Section 7 discusses the related work. We conclude this paper in Section 8.

Overall, this paper makes the following contributions:

- Experimental demonstration of the problem of conventional cache managements, regarding I/O proportionality (Section 3)
- Weight-aware locking mechanism for reflecting I/O weights in cache allocation. (Section 4.1)
- Weight-aware page reclamation for keeping data generated by higher-weighted applications. (Section 4.2)

2 Background

In this section, we describe cgroup and application-level I/O proportionality. Afterward, the mechanism of the conventional I/O cache management schemes is given with their limitation in terms of I/O proportionality.

2.1 Cgroup and Proportional I/O Sharing

Cgroup is widely adopted to control and limit the allocation of system resources, such as CPU, memory, and block I/O. Cgroup manages those resources in the form of subsystems, each of which has various parameters for controlling resource consumption[11]. Particularly, the blkio subsystem monitors and regulates block I/O requests. Cgroup creates a blkio resource group that contains a set of applications as necessary, and the blkio subsystem controls the I/O bandwidth of the groups [18] by adjusting their I/O-related parameters. There are two representative policies to control I/O performance in Cgroup [19]: proportional weight policy and throttling/upper limit policy. The weight policy is to set I/O weight of cgroups and determine their shares. The throttling policy is to set the upper limit of either I/O bandwidth or IOPS. These two policies have both pros and cons. The weight-based controlling can fully utilize the I/O bandwidth that the device can support, but it cannot set the exact I/O throughput. The throttling-based controlling can limit the exact I/O performance but may waste the remaining I/O performance in order to achieve the exact I/O performance. This work focuses on the weight-based policy, which controls I/O performance with I/O proportionality.

In conventional systems, CFQ and BFQ I/O schedulers use this I/O weight value (blkio.weight) [11], [20] when they decide how many I/Os will be dispatched from the request queues in the block layer [18]. Specifically, the CFQ scheduler gives higher-weighted applications more time for processing I/O so that the block-level I/O proportionality can be achieved according to their I/O weights [21]. Note that, in this paper, I/O proportionality [22] refers to making the bandwidth ratio of applications be proportionally allocated

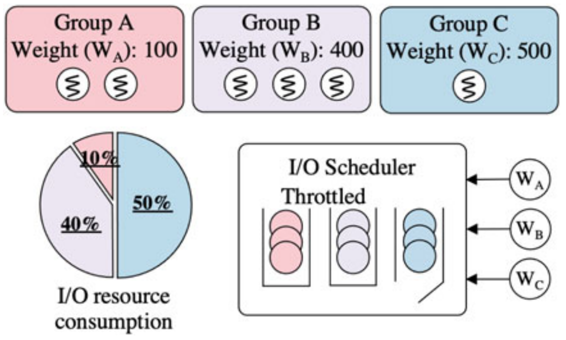


Fig. 1. An overview of I/O resource sharing using cgroup.

according to their I/O weights. For example, as shown in Fig. 1, let us suppose that the system manager creates three resource groups with I/O weights 100, 400, and 500. Then, for I/O proportionality, they should have the I/O bandwidth ratio of 0.1:0.4:0.5, when the total amount of I/O resources available in the system is 1.0.

In practice, however, the application-level I/O proportionality cannot be guaranteed with I/O caches because I/O caches [21], [23], [24] may exclude the I/O scheduler from the critical path except when uncached data are requested. For example, when an application tries to access data already stored in the page cache, the application retrieves the data directly from the page cache, skipping the layers beneath the page cache. This characteristic of the page cache avoids relatively slow access to the underlying storage device and helps to achieve high I/O bandwidth and low latency. However, since such I/O requests do not experience the block layer, the I/O weights of cgroup cannot be applied. Additionally, the blkio subsystem does not directly control I/O caches, and also, the I/O caches cannot achieve the required I/O proportionality by themselves.

2.2 I/O Cache

A cache is a component that temporally stores data on a faster medium in lieu of a slower one. The current computer systems adopt various kinds of caches to bridge the performance gap between two different hardware devices. Particularly, I/O caches effectively improve I/O performance by directly servicing I/O requests on behalf of the slow storage device, when the corresponding data reside in it [25]. For example, Linux kernel utilizes unused space of the main memory as I/O cache, called page cache. Similarly, Linux kernel provides another type of I/O cache in the block layer, called bcache, to remedy the low performance of a slow storage device by means of a faster one.

2.3 Cache Allocation

I/O cache management mainly performs two tasks; cache allocation and cache reclamation. Cache allocation refers to a task that allocates a new cache entry to store incoming data. For example, when an application tries to write uncached data, the cache management allocates a new cache entry to the application and stores the corresponding data inside the entry. Since the cache resource is shared by multiple CPUs, cache allocation should be mutually exclusive [24]. To achieve this, I/O cache management often adopts a locking mechanism to protect the critical section. For example, both the page cache and bcache protect the cache allocation using qspinlock in Linux systems.

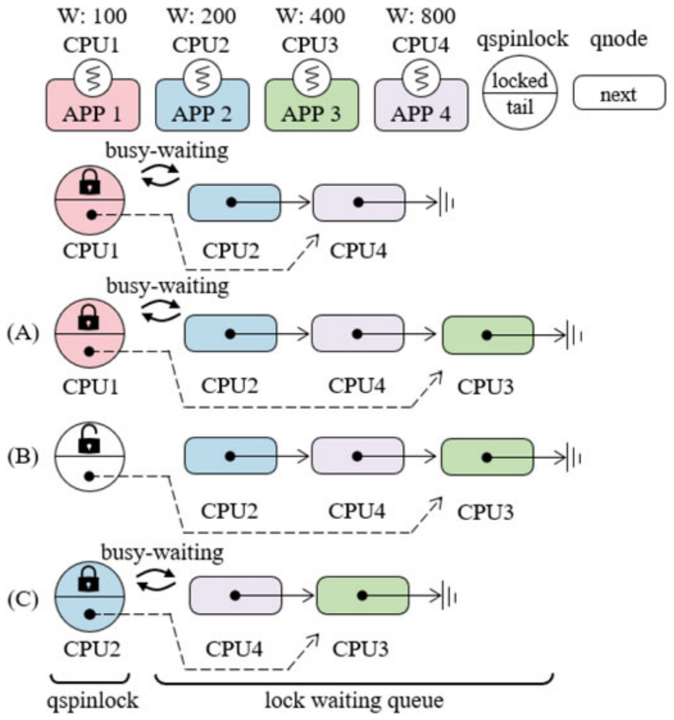


Fig. 2. The conventional Qspinlock.

The overview of qspinlock mechanism is shown in Fig. 2. As described in Fig. 2, qspinlock mechanism consists of one qspinlock structure and multiple per-CPU qnodes, each of which is an element of lock waiting queue[16]. In the locking mechanism, for cache allocation, the conventional cache management selects the next lock holder from the lock waiting queue in a FIFO manner [16], [17]. Therefore, the conventional cache allocation cannot prioritize applications with higher I/O weights. For example, in Fig. 2, when CPU3 tries to acquire the qspinlock, it is inserted at the tail of the lock waiting queue regardless of its I/O weight. Afterward, when CPU1 releases the qspinlock, CPU2 stops busy-waiting and acquires the lock in a FIFO manner without consideration on I/O weight. Therefore, although the applications on CPU3 and CPU4 have higher I/O weights than the one on CPU2, CPU2 acquires the lock ahead of CPU3 and CPU4. Like this, the conventional cache management handles cache allocation using a FIFO-based queue which does not consider I/O weights, thereby distorting the I/O proportionality.

Cache allocation and its lock contention are critical to the I/O performance [24], and have an increasing impact on the performance as the number of applications co-running in a system increases. For example, in a multi-container environment where multiple servers co-run in a single physical system, lock contention increases due to the high number of simultaneous cache allocation requests. Thus, I/O requests from applications have to wait a significant amount of time to acquire the lock. Accordingly, the order of lock acquisition becomes a critical factor on the I/O performance in such systems.

We demonstrate this problem in Fig. 3, by measuring the lock contention count using `lockstat` [26] while running

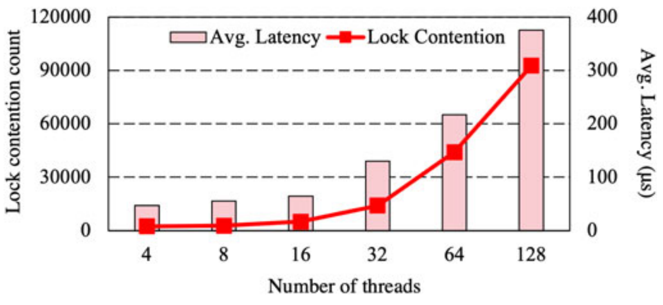


Fig. 3. Lock contention count and the average latency while varying the number of threads. A higher value denotes more severe lock contention and worse performance.

multiple random write workloads on tmpfs. Here, the lock contention denotes the case that a process attempts to acquire the lock that is already held by another process. As shown in Fig. 3, the lock contention count increases along with the number of running threads, mainly due to high contention of cache allocation requests from multiple threads. When the number of threads becomes 128, the number of lock contentions reaches 92,687. In terms of performance, the average latency of the workloads significantly increases as the number of concurrent threads increases. Like this, the more threads run in a system, the more important the order of lock acquisition becomes. Moreover, this problem is exacerbated when the amount of available cache entries is low and cache allocation induces cache reclamation [24]. In the case of page cache, it is known that allocating free pages can take more than 200ms in such cases because dirty pages should be evicted in advance to create free pages [21], [24].

2.4 Cache Reclamation

Since the cache size is usually smaller than the entire data size, the cache management should reclaim the existing cache entries to secure free entries for new data. In this process, cache replacement algorithms such as LRU (Least Recently Used) and FIFO decide which cache entries to evict. There have been various kinds of cache replacement algorithms to maximize the cache hit ratio. For example, LRU policy evicts the cache entries that are not used for a long time under the assumption that recently accessed data will be accessed again soon.

The Linux page cache utilizes a variant of LRU, called 2Q-LRU, which utilizes two LRU queues: an active list to keep frequently accessed pages, and an inactive list for the other pages [27]. When a page is accessed for the first time, it is placed at the head of the inactive list. Afterward, the page can be promoted to the active list when it is accessed again. Pages in the active list are demoted to the inactive list when they are considered unlikely to be accessed again. Finally, pages at the tail of the inactive list are reclaimed when the amount of free memory space gets lower than the predefined threshold.

Bcache provides LRU, FIFO, and random as a configuration parameter. It assigns each cache entry a priority value, which decrements over time, and utilizes the value in cache reclamation. For example, in the case of LRU, bcache restores the priority value of cache entries to the pre-defined value upon each access and evicts cache entries that have the lowest priority value.

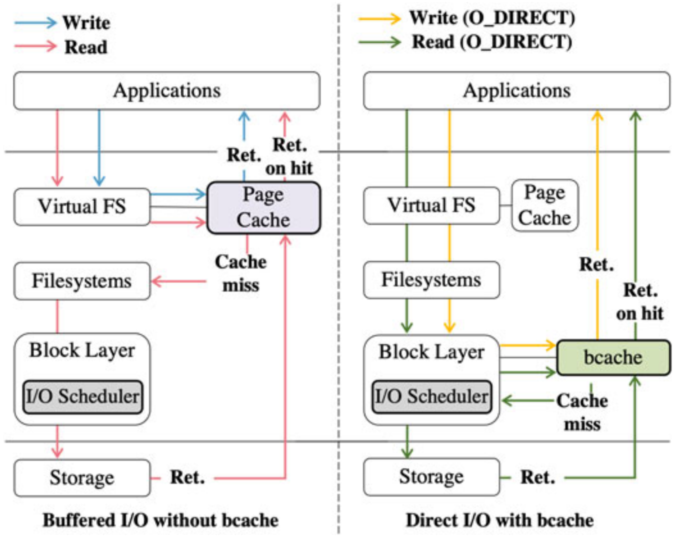


Fig. 4. The critical path of I/O requests depending on I/O types.

Cache reclamation is crucial to the I/O performance, especially to that of read I/Os, because it decides the contents of the cache. Unfortunately, both the page cache and bcache do not consider I/O weight in the process of cache reclamation. Therefore, those cache systems lack the ability to reflect I/O weight in cache reclamation, which in turn leads to failing to achieve application-level proportional I/O sharing.

3 Motivation

A majority of virtualized systems utilize cgroup, which cooperates with I/O schedulers, to achieve proportional I/O sharing. However, the adoption of the I/O cache can impede application-level proportional I/O sharing because I/O requests might not experience the underlying I/O scheduler. As shown on the left side of Fig. 4, buffered I/Os are usually handled by the page cache without going through the I/O scheduler. Only some of the read requests are delivered to the I/O scheduler when the corresponding data do not exist in the cache. Similarly, as shown on the right side of Fig. 4, bcache also handles I/Os on behalf of the underlying storage, thereby hindering the I/O scheduler from controlling the I/O requests. Unfortunately, the conventional I/O cache management cannot reflect I/O weight by itself, as mentioned in Section 2.

To experimentally demonstrate the distortion of I/O proportionality caused by cache allocation, we conducted an experiment with the Fileserver workload in Filebench. We ran this workload in each of four containers with different weights (i.e., 100, 200, 400, 800). Each of the workloads runs for around 300 seconds and generates around 6 GB of data. The experiment is performed with two different types of I/Os: (1) direct I/O whereby I/O requests bypass the page cache layer, and (2) buffered I/O whereby I/O requests use the page cache for buffering and caching. In this experiment, we do not adopt bcache, and thus all of direct I/Os visit the I/O scheduler. The detailed experimental setup is described in Table 1 of Section 6.

Fig. 5 shows the R/W combined I/O bandwidths and the normalized (to the bandwidth of the container with weight 100) I/O bandwidths for the two different I/O types. As

TABLE 1
Experimental Configuration

Hardware Configuration		Machine A
	CPU	Intel I7-6700 CPU @ 3.40GHz (1 socket)
	Memory	16GB
	Storage	SATA Flash SSD 256GB
		Machine B
	CPU	Intel Xeon Gold 6130 CPU @ 2.10GHz (4 Sockets)
	Memory	128GB
	Storage	NVMe Optane SSD 500GB (Partitioned into 20GB) SATA Flash SSD 256GB
Software Configuration	OS	Ubuntu 16.04
	Kernel	Linux Kernel 4.19.16
	Docker	Docker v18.09.4-CE (base image: Ubuntu 14.04)

shown in Fig. 5, while direct I/O exhibits decent I/O proportionality (1:1.9:3.2:5.4), buffered I/O presents poor proportionality in that it shows similar performance in all differently weighted containers. This unsatisfying result of buffered I/O comes from the buffering of I/O operations in the page cache.

Fileserver is a write-intensive I/O workload, and so, the majority of the I/O operations are absorbed by the page cache in the case of buffered I/O. However, the conventional page cache management does not prioritize higher-weighted applications in the cache allocation stage, which has critical impacts on the write performance. Rather, it handles applications in a FIFO manner during the cache allocation process. Therefore, even if the containers have different I/O weights, buffered I/O cannot differentiate the I/O bandwidth of the containers.

Nonetheless, the difference in the total bandwidth between the two cases shows a clear reason for using I/O caches. In the experiment of Fig. 5, the total bandwidth of four containers with direct I/O is 677.3 MB/s, while that with buffered I/O is 1200.4 MB/s. Considering this huge I/O performance gain of I/O caches, we cannot ignore I/O caches even in cloud systems where performance SLO is very crucial [25], [28].

To show the distortion of I/O proportionality caused by cache reclamation, we conducted an experiment with the Re-read workload of FIO. Like the previous motivational experiment of Fig. 5, we ran this workload in each of four containers with different weights (i.e., 100, 200, 400, 800). Each container creates one 3 GB test file to cache all the data of the file into the page cache. Then, the host writes a 4 GB dummy file to trigger excessive page reclamation. Lastly, we executed 1 GB re-read operations in each container and measured the I/O bandwidth. The experiment is also performed with two different types of I/Os, direct I/O and buffered I/O. As shown in Fig. 6, buffered I/O shows poor I/O proportionality than

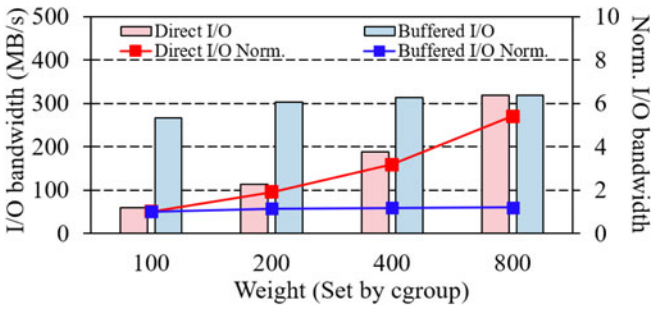


Fig. 5. Comparison of I/O bandwidths and normalized I/O bandwidths between direct I/O and buffered I/O, with the Fileserver workload (I/O bandwidth is normalized to the container of weight 100). Here, the I/O proportionality of direct I/O is 1 : 1.9 : 3.2 : 5.4.

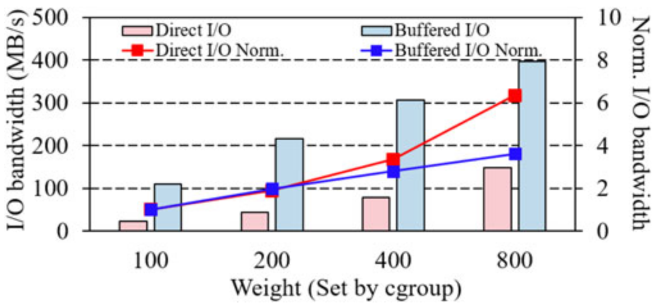


Fig. 6. Comparison of I/O bandwidths and normalized I/O bandwidths between direct I/O and buffered I/O, with the Re-read workload (I/O bandwidth is normalized to the container of weight 100.)

direct I/O. A large number of re-read requests are serviced from the page cache in the case of buffered I/O. Unfortunately, when the conventional page cache management evicts cache entries after the dummy writes from the host, it does not consider I/O weights. Therefore, even though some cache entries are used by higher-weighted containers, the entries are evicted prior to other entries used by lower-weighted ones, resulting in poor I/O proportionality.

The aforementioned analyses on the conventional I/O cache management lead us to conclude that strictly keeping the FIFO or LRU policy without considering the I/O weights is harmful to the I/O proportionality when running weighted applications. Based on the motivational analyses above, we suggest a new cache management scheme that resolves the aforementioned problems.

4 Design

In this section, we present a new cache management scheme, called *WaC* (Weight-aware Cache), to achieve application-level proportional I/O sharing. The overview of *WaC* is presented in Fig. 7. *WaC* has the following goals:

- *WaL* (Weight-aware Lock): Prioritizing higher-weighted applications in the locking mechanism for cache allocation.
- *WaR* (Weight-aware Reclamation): Prioritizing the cache entries used by higher-weighted applications in the cache reclamation.

4.1 Weight-Aware Lock (WaL)

WaL is a new locking mechanism that reflects I/O weight in the decision process of the next lock holder for cache allocation to achieve proportional I/O sharing. The structure of *WaL* is similar to the conventional qspinlock in that *WaL* utilizes the lock waiting queue and qspinlock. However, *WaL* stores the I/O weight of each application inside the qnode

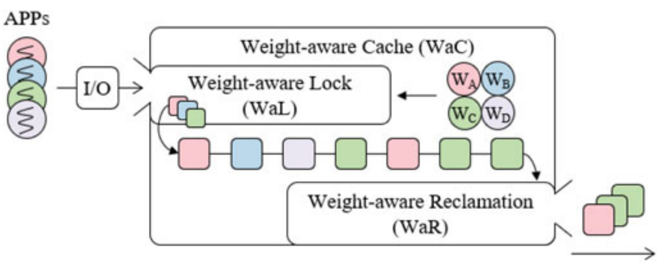


Fig. 7. An overview of WaC.

and utilizes them in deciding the next lock holder. As a result, while the conventional qspinlock chooses the next lock holder in a FIFO manner, *WaL* traverses the lock waiting queue and chooses the one that has the highest I/O weight.

However, simply reordering the lock waiting queue based on I/O weight can incur two problems. First, the starvation problem can take place. Since the lower-weighted applications should yield their turns to higher-weighted ones, such applications may consume a long time to acquire the lock. Especially, when the majority of applications have high I/O weights, the lower-weighted ones might be denied to acquire the lock constantly. Prevention of such starvation is necessary to build a robust system because starvation can produce long-term unfairness and even system failures.

To solve this problem, we adopt a conventional well-known technique for starvation, called aging, which gradually increments the priority of a task over time. *WaL* adjusts the I/O weights of qnodes by a certain value, whenever reordering occurs. Consequently, *WaL* considers not only I/O weight, but also waiting time in deciding the next lock holder.

The second problem is a scalability issue derived from NUMA-blindness. As previously reported[29], [30], [31], [32], [33], NUMA-blind lock management can result in performance degradation on NUMA systems. Specifically, frequent change in the NUMA nodes of the lock holders can induce noticeable overheads, because the memory bandwidth between NUMA sockets is finite and remote access is more expensive than local access[30]. To mitigate such overheads, *WaL* additionally stores the NUMA node ID of each qnode and endeavors to maintain the lock holder on the same NUMA node.

The pseudo-code of *WaL* is presented in Algorithm 1. When the current lock holder releases the qspinlock, *WaL* traverses the lock waiting queue to find the qnode that has the highest I/O weight (search phase). We call this qnode as maxNode. *WaL* examines not only the I/O weight, but also the NUMA node ID to minimize performance overheads on a NUMA system. Therefore, when multiple qnodes have the same highest I/O weight, *WaL* chooses the one on the same NUMA node as the head node. Afterward, *WaL* reorders the lock waiting queue so that the maxNode is located at the next of the head node and can acquire the qspinlock in the next turn. Finally, *WaL* increments the I/O weights of the other qnodes to prevent the starvation problem.

Indeed, the aging phase is inherent in the search phase to minimize the traversing overheads. In other words, *WaL* needs to traverse the lock waiting queue only once. But for ease of understanding, we decoupled the search phase and the aging phase in Algorithm 1. The maximum I/O weight that can be manually set by cgroup is 1,000, while the adjusted I/O weights can be higher than 1,000. Accordingly, regardless of initial I/O weights, any application can eventually become the maxNode due to the aging technique and thus acquire the lock. In our experiments of Section 5, we set the increment value as 100 for the following reason. First, we set the weight values of cgroups in units of 100 (i.e., 100, 200, 400, and 800). I/O weight denotes the proportionality relationship in that a process with weight 800 is supposed to show 8 times higher I/O performance than that with weight 100. Therefore, setting the increment value as 100 follows the concept of I/O weight since a process with weight 100 can acquire the lock after a process with weight 800 acquires the lock around 7–8 times if the lock is contended. In other words, the process with weight 100 becomes weight 800 after yielding 7 times when the increment value is 100. If a system sets the weight values in units of 50, (e.g., 50, 100, 150, and 200), the increment value should be 50.

Algorithm 1. The Pseudo-Code of WaL

```
1:  if qspinlock is available then
        // cur is the qnode of the current lock holder
2:      headNid = head → nid
3:      qnode == head;
4:      maxWeight == 0;
5:      maxNode, prevNode, iterNode == NULL;
        /* Search phase: find the qnode with the maximum
        I/O weight (on the same NUMA node if possible) */
6:      while qnode → next ≠ tail do
7:          nextNode = qnode →next
8:          if maxWeight <nextNode →weight then
9:              prevNode = qnode;
10:             maxNode = nextNode;
11:             maxWeight = nextNode → weight;
12:         end
13:         else if (maxWeight == nextNode →weight) and
            (headNid == nextNode → nid) then
14:             prevNode = qnode;
15:             maxNode = nextNode;
16:             maxWeight = nextNode → weight;
17:         end
18:         qnode = qnode → next;
19:     end
        // Reordering the lock waiting queue
20:     prevNode → next = maxNode → next;
21:     maxNode → next = head → next;
22:     head → next = maxNode;
        // Aging phase
23:     iterNode = maxNode;
24:     while iterNode ≠ tail do
25:         iterNode → next → weight += agingValue;
26:         iterNode = iterNode → next;
27:     end
28:     head.acquire(lock);
29: end
```

An example of WaL is illustrated in Fig. 8. In the initial state, the qspinlock is occupied by CPU1 where APP1 is running, and two qnodes are in the lock waiting queue. CPU4 is busy-waiting for the qspinlock because it is the head node. The I/O weight of CPU2 qnode has been increased from 200 to 400 after reordering twice. When APP3 tries to acquire the qspinlock, *WaL* creates a qnode structure for APP3 at the tail of the lock waiting queue and stores its I/O weight and the NUMA node ID inside the qnode (A). When CPU1 releases the qspinlock, *WaL* traverses the lock waiting queue to find the maxNode. Here, since the head node (CPU4) has been busy-waiting, *WaL* does not reorder the head node and lets it acquire the lock in this turn. In the example, the qnodes of CPU2 and CPU3 have identical I/O weight (400). Therefore, *WaL* additionally investigates the NUMA node information and picks CPU3 node as maxNode, because CPU3 node is located on the same NUMA node as the head node (CPU4). Afterward, *WaL* reorders the lock waiting queue so that the

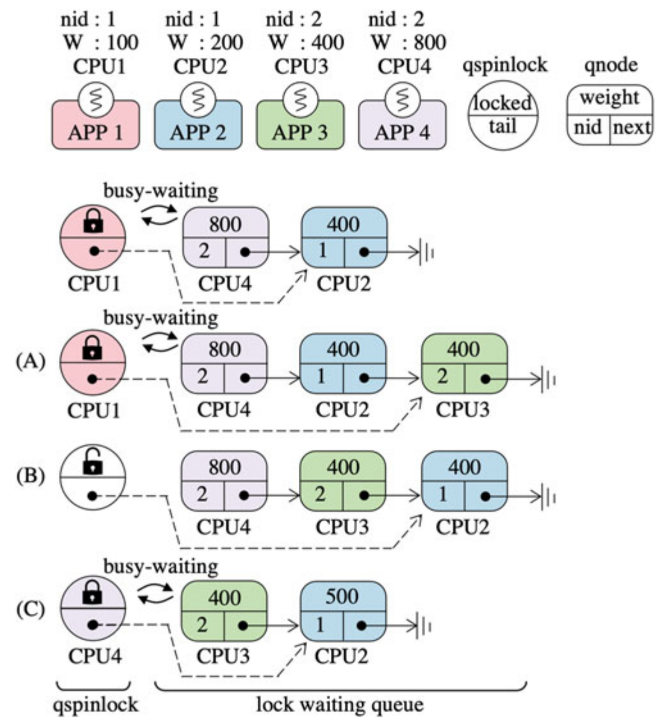


Fig. 8. An example of WaL.

maxNode (CPU3) is located next to the head node (CPU4) (B). Finally, as the head node (CPU4) holds the qspinlock, the maxNode (CPU3) becomes the new head of the queue and will acquire the qspinlock in the next turn. Meanwhile, the aging technique is applied, and thus the I/O weights of qnodes in the queue are incremented (C). Consequently, by use of *WaL*, higher-weighted applications can obtain the lock for cache allocation relatively faster than applications with a lower weight, thereby achieving application-level proportional I/O sharing.

4.2 Weight-Aware Reclamation (WaR)

The conventional cache reclamation usually utilizes FIFO and LRU variants. However, such methods cannot prioritize applications with high I/O weights. To overcome this limitation, we propose *WaR*, which considers both reference recency and I/O weight. The goal of *WaR* is to keep the number of allocated cache entries for each application proportional to its I/O weight. To achieve this, *WaR* calculates the I/O proportion of each application and calculates the threshold number of cache entries that the application should have. Finally, during cache reclamation, *WaR* compares the number of cache entries of each application and its threshold, and decides which cache entries to evict.

Fig. 9 describes the behavior of *WaR*. First, *WaR* calculates the I/O proportion of each application through dividing each I/O weight by the total sum of I/O weights. For example, in Fig. 9, the system executes four applications in four cgroup nodes with I/O weight 100, 200, 400, and 800, respectively. Then, each of these applications has an I/O proportion of 0.07, 0.13, 0.27, and 0.53, respectively, when the total amount of cache resources available in the system is 1.0. Second, *WaR* keeps track of both the total number of cache entries (total # entries in the figure) and the number of cache entries used by each application (p.g. # entries in

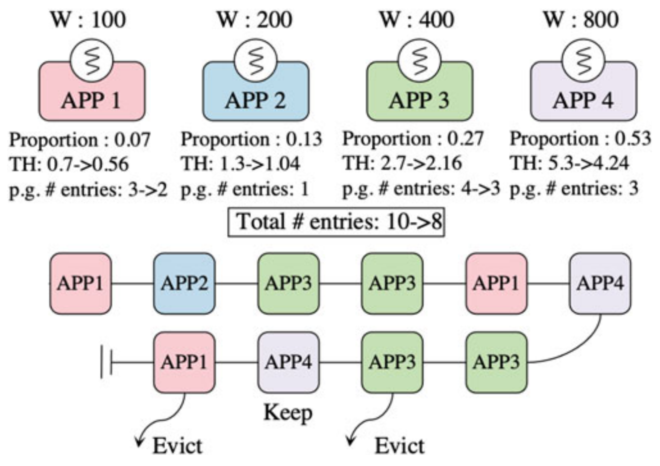


Fig. 9. An overview of WaR.

the figure). When cache reclamation is triggered, *WaR* calculates the threshold (TH in the figure) of each application by multiplying the I/O proportion and the total number of cache entries. Here, the threshold value indicates the maximum number of cache entries an application is supposed to possess. From the tail of the cache entries, *WaR* compares the number of cache entries per group (p.g. # entries in the figure) with the threshold value and evicts the cache entry whose owner application has more cache entries than its threshold. Note that when multiple applications share the same page, *WaC* designates the highest-weighted application as the owner of the page.

For instance, as shown in Fig. 9, the cache entries of APP1 and APP3 will be evicted because their p.g. # entries is higher than their TH, while *WaR* keeps the cache entries of APP4. Afterward, *WaR* decreases the total # entries from 10 to 8, and the TH values become adjusted due to the change of the total # entries. Finally, *WaR* decrements the p.g. # entries of APP1 and APP3 by 1, because of the cache eviction. By performing this process repetitively, *WaR* can keep the number of pages of each application proportional to its I/O weight.

Cache reclamation is closely related to read performance in that it decides the contents of the cache. Read operations can be serviced by the cache, depending on the existence of the corresponding data in the cache. *WaR* tries to keep more data of higher-weighted applications in the cache, which in turn increases the probability that their read requests are processed by the cache without accessing the underlying storage device. As a consequence, *WaR* can prioritize higher-weighted applications, thereby enhancing I/O proportionality.

5 Implementation

We implemented *WaC* in two places in the Linux kernel: the page cache and bcache. The operating system utilizes unused space of the main memory as page cache to accelerate I/O requests. Similarly, bcache is a discrete storage device to cache I/O requests at the block layer. Since they have different internal mechanisms, we explain the implementation details in this section.

5.1 Page Cache

The page cache manages cache entries in the unit of page and maintains the cache entries in two LRU lists: active list and inactive list. The conventional page cache protects cache allocation using FIFO-based qspinlock. To implement *WaL*, we add weight and nid (NUMA node ID) variables into the qnode structure and assign the corresponding values to the variables when an application creates a new qnode. We utilize the *numa_node_id()* function, which is provided by Linux kernel, to obtain the NUMA node ID of the current application.

To implement *WaR* in the page cache, we performed the following tasks. First, we add two new variables, for I/O proportion and the number of cache entries per group, into the cgroup structure. Whenever the cgroup hierarchy changes due to an event, such as the creation of a new cgroup, *WaR* re-calculates the I/O proportion. Upon cache allocation, *WaR* links the new cache entry to the corresponding cgroup node, in order to clarify the ownership of the cache entry. Additionally, *WaR* increments the number of cache entries per group. In this way, *WaR* can refer to the I/O proportion and the number of cache entries per group during cache reclamation. Finally, to obtain the total number of cache entries, *WaR* utilizes the existing *NR_FILE_PAGES* variable, which is the number of file-backed page cache entries that the kernel keeps track of. When the page reclamation is triggered, *WaC* can evict pages from the tail of the inactive list according to the policy of *WaR*.

We only apply *WaR* to the inactive list of the page cache because the active list significantly contributes to the high cache hit ratio. Therefore, moving pages from the active list to the inactive list is performed with LRU to preserve the high hit ratio of the page cache as conventional. Meanwhile, *WaC* performs the conventional cache reclamation in the case of the direct reclamation (foreground reclaim), in which the system has a lack of free pages, in order to prevent the OOM (Out-Of-Memory) problem and minimize performance degradation. *WaR* requires additional CPU overhead for comparing the number of cache entries of each application and its threshold. Therefore, *WaC* performs *WaR* in the case of background reclaim (kswapd), thereby taking the computation off the critical path and minimizing performance degradation.

5.2 Bcache

Bcache manages the cache entries in the unit of bucket and utilizes priority values to implement cache reclamation algorithms such as LRU, FIFO, etc. Bcache also utilizes qspinlock to protect the cache allocation as in the case of page cache. Therefore, the implementation of *WaL* in the page cache is applied to bcache. To implement *WaR* in bcache, in addition to the modification in the cgroup structure, we additionally add a cgroup pointer inside the bucket structure to access I/O proportion and the number of cache entries per group during cache reclamation. Upon cache allocation, *WaC* links the corresponding cgroup pointer to the allocated cache entry and increments the total number of cache entries and the number of cache entries per group. When cache reclamation is needed, *WaR* calculates the threshold and decides which cache entries to evict. After cache reclamation, *WaC* adjusts the number of cache entries per group and the total number of cache entries. The implementation of *WaR* does not require modification on the mechanism for priority control. Therefore, the conventional cache reclamation algorithm is still valid among the cache entries within the same cgroup.

6 Evaluation

6.1 Evaluation Setup and Test Settings

To verify the efficacy of *WaC*, we performed various experiments on the two machines described in Table 1. Machine A is utilized to evaluate the page cache version of *WaC*. To test the scalability of *WaC* and the performance of the bcache version, we utilize machine B, which is equipped with 64 physical cores (hyper-threading disabled) and high performance storage device for bcache. All the benchmarks in the experiments are containerized by Docker v18.09.4-CE and run five times, unless otherwise specified. To quantitatively measure the I/O proportionality, we adopt a new metric called proportionality variation (PV), introduced in [34]. PV is calculated via the following equation.

$$PV = \frac{1}{N} \cdot \sum_{\forall APPs} |Ideal - Actual|. \tag{1}$$

Here, *APPs* are applications, N is the number of applications, *Ideal* is the ideal performance, and *Actual* is the actual performance obtained from experiments. The lower the value is, the closer the proportionality is to the ideal.

6.2 Page Cache

6.2.1 Fileserver Workload

To examine *WaL* on the page cache in terms of application-level I/O proportionality, we ran eight Fileserver workloads in eight differently weighted containers, each of which runs for around 300 seconds and generates around 3 GB of data. The Fileserver workload performs a large amount of buffered writes and thus incurs frequent page cache allocation. Fig. 10a shows the normalized I/O bandwidth of eight containers in the Fileserver experiment. As shown in the x-axis of Fig. 10a, we assign different weights to the containers from 100 to 800. The bandwidths of the containers are normalized to the bandwidth of the container with weight 100. In the figure, Ideal represents the page cache management with ideal I/O proportionality that the containers expect to achieve, i.e., 1:2:3:4:5:6:7:8, and the Conventional is the conventional page cache management of the Vanilla Linux kernel.

As a result of the experiment, from weights 100 to 800, I/O proportionality of *WaC* shows 1:1.73:2.24:2.65:3.04:3.75:4.37:6.26, whereas the conventional scheme shows 1:1.51:2.02:2.40:2.63:2.71:3.07:3.31. Especially in the case of weight 800, *WaC* shows only a 1.74 lower I/O proportion than the ideal case, while the conventional scheme exhibits a 4.69 lower I/O proportion than the ideal case. *WaC* can achieve better application-level I/O proportionality than the conventional scheme, because it prioritizes higher-weighted containers in the lock acquisition process during cache allocation. As a result, higher-weighted containers can quickly finish their write operations and resume the next write operations, thereby showing higher I/O bandwidth. In terms of PV, *WaC* outperforms the conventional scheme by around 36.9%. We also measured the total I/O bandwidth while varying the number of containers from two to eight, in order to analyze the overheads of *WaL* of *WaC*. Similarly to the previous

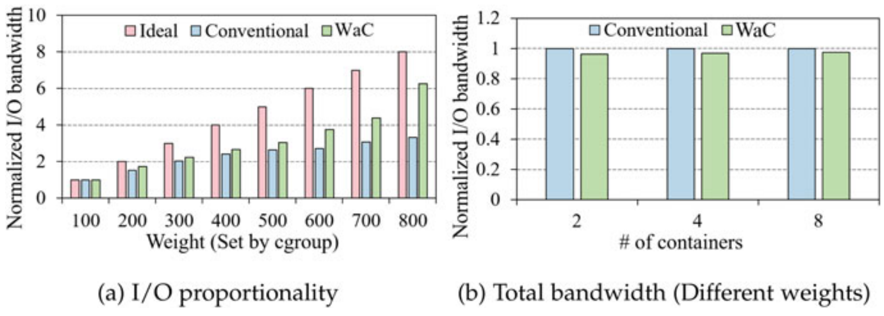


Fig. 10. Results on Fileserver.

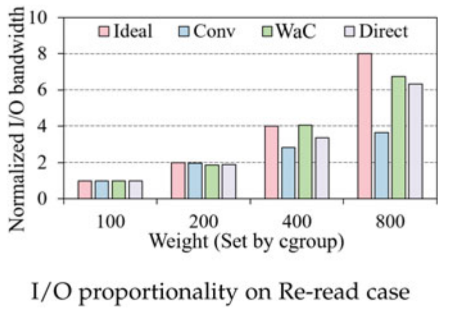


Fig. 11. Results on Re-read workload.

experiment, we ran Fileserver workloads with differently weighted containers. As shown in Fig. 10b, the experimental result shows that the total I/O bandwidth decreases only by 3.9% at most when *WaC* is applied.

6.2.2 Re-Read Workload

To evaluate *WaR* of *WaC*, we performed the same Re-read experiments as the motivational experiments presented in Section 3. In the Re-read experiment, four containers with different I/O weights create their own files, and then the host creates a dummy file to contaminate the page cache. Afterward, the containers read the files again to examine how many pages of each container reside in the page cache. In our experiments, we also ran the workload with direct I/O as well as buffered I/O for better comparison. All the performance results are normalized to the case of weight 100. As shown in Fig. 11, the conventional scheme exhibits poor I/O proportionality because the conventional cache reclamation scheme does not consider I/O weight at all. Therefore, pages of higher-weighted containers can be evicted before those of lower-weighted containers, even when their reference counts are the same. As a result, the conventional scheme shows the PV of 1.4, while *WaC* shows the PV of 0.33. This result stems from the fact that *WaC* balances the number of allocated cache entries of the containers according to their I/O weights, by keeping cache entries of higher-weighted containers longer in the page cache. This result is even superior to that of direct I/O, in that direct I/O shows the PV of 0.61. This is because, in the case of *WaC*, the block-level I/O proportionality is also guaranteed by the underlying I/O scheduler, in addition to the cache-level I/O proportionality.

6.2.3 Overhead of WaC

To investigate the overheads caused by additional processing of *WaC*, we measured the hit ratio, while executing four containers with different I/O weights, each of which runs the Webserver (read-intensive) workload. In addition, we also measured the total bandwidth and the average latency, while performing Fileserver and Re-read workloads in four containers with the same I/O weight. As shown in Fig. 12a, the overall hit ratio drops just by about 0.8% on average, denoting that *WaC* keeps a satisfactory hit ratio even with deprioritizing the pages of lower-weighted containers. This result comes from the fact that *WaC* reflects I/O weight only in the inactive list leaving the active list as it is.

Fig. 12b shows the write bandwidth of Fileserver and the read bandwidth of Re-read workloads. They are performed in the same configuration as in Fig. 10b and Fig. 11 except that all the containers have the same I/O weight in this experiment. Compared with the conventional scheme, *WaC* exhibits 2.7% and 3.7% drop in the total bandwidth on the Fileserver and Re-read workloads, respectively. In addition, the average latency of executing Fileserver and Re-read workloads increases by 1.5ms and 0.6ms on average in *WaC*, respectively. These overheads of *WaC* originate from searching for a container with the highest I/O weight during the cache allocation and repeatedly keeping track of the number of allocated cache entries per application for cache reclamation. However, considering the satisfactory results of I/O proportionality, we believe *WaC* is very practical to help improve the I/O proportionality with imperceptible overheads.

6.2.4 Comparison With Memory Cgroup

One might assume a combination of memory and blkio cgroup could be an alternative to *WaC*. The memory subsystem of cgroup provides the ability to control the maximum memory usage of each application. However, when the memory usage of a resource group exceeds its limit, the memory cgroup reclaims not only its file-backed pages but also anonymous pages. Therefore, it cannot solely limit the

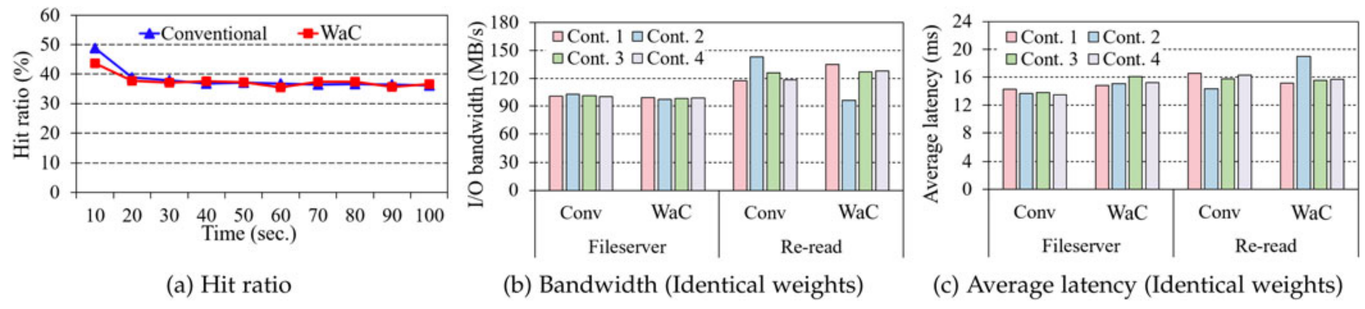


Fig. 12. Overhead of WaC.

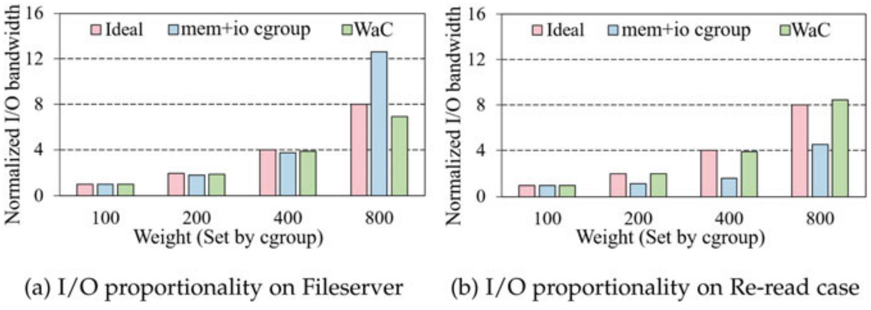


Fig. 13. Comparison with memory cgroup.

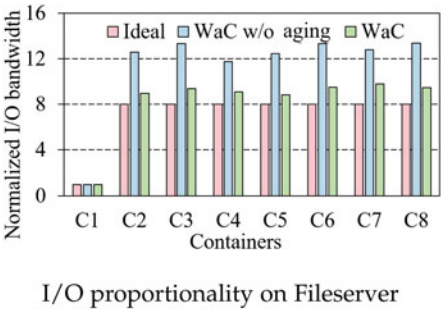


Fig. 14. Eval. for aging scheme.

amount of file-backed pages which affects the I/O performance. Consequently, I/O proportionality still cannot be guaranteed by memory cgroup.

To confirm this, we ran both Fileserver and Re-read benchmarks and compared the experimental results of memory cgroup with those of our scheme. Here, in the case of memory cgroup, we set the same ratio of memory limit (1:2:4:8) as that of I/O weights of *WaC*. In the Fileserver experiment of Fig. 13a, memory cgroup shows poor I/O proportionality, showing PV of 4.25, while PV of *WaC* is 0.33. Especially, with memory cgroup, the container with I/O weight 800 shows 12.6 times higher I/O bandwidth than the lowest-weighted container (100). Also, as shown in Fig. 13b of the Re-read workload experiment, from weights 100 to 800, I/O proportionality of *WaC* shows 1:2.02:3.91:8.46, whereas that of memory cgroup shows 1:1.15:1.60:4.53. PVs of *WaC* and memory cgroup are 0.14 and 1.67, respectively. We believe that these results come from the fact that memory cgroup cannot solely control file-backed pages (page cache) without limiting anonymous pages.

6.2.5 Aging Technique

To prevent the starvation problem, we added the aging technique to *WaL*. To verify that our scheme is robust in extreme cases, we performed the Fileserver experiment with eight containers. Here, the I/O weight of one container (C1) is 100 and the others (C2 – C8) are 1000. As shown in Fig. 14, I/O proportionality of *WaC* is 1:8.94:9.36:9.08:8.83:9.49:9.77:9.43 with PV of 0.64. On the other hand, *WaC* without the aging technique shows 1:12.57:13.31:11.72:12.443:13.31:12.77:13.35 in I/O proportionality with PV of 2.31. In the case of *WaC* without aging technique, C1 should repetitively yield its turn to other containers with higher I/O weight without any reward, thereby showing lower I/O bandwidth than it should. However, since *WaC* increases the I/O weight of C1 whenever it yields its turn for lock acquisition, it shows better I/O proportionality than the case of *WaC* without the aging technique. Therefore, even though there are multiple higher-weighted containers, our scheme is still able to guarantee the performance of the lower-weighted containers according to their I/O weights.

6.2.6 Scalability

To analyze the effect of NUMA-awareness of *WaL*, we performed a scalability experiment using FIO on machine B shown in Table 1. We ran 4 KB buffered writes via four differently weighted containers, each of which execute multiple threads for the write operations. In this experiment, we spread the threads of the same container to different nodes as much as possible to simulate the worst case where the NUMA node of the lock holder frequently changes. As shown in Fig. 15, for all the cases, the total IOPS decreases after the total number of threads reaches 64, which is the number of physical cores. The peak IOPS of *WaC* without NUMA-awareness is around 5% lower than that of the conventional. It is because *WaC* without NUMA-awareness chooses the next lock holder without considering the NUMA topology and thus frequently changes the NUMA node of the lock holder. On the other hand, *WaC* outperforms even the conventional scheme by around 7% due to its NUMA-awareness.

6.3 Bcache

In addition to the page cache, we also evaluate *WaC* on bcache. We utilized machine B to perform experiments with the high performance storage device (Optane SSD) for bcache and ran benchmarks using four differently weighted containers from 100 to 800.

6.3.1 Random Write Test

To test *WaL*, we ran FIO with 4KB random write workloads using four containers. The random write workload performs direct I/O which bypasses the page cache and utilizes bcache. Similarly to the previous experiments, the containers are differently weighted from 100 to 800 so as to inspect proportional I/O sharing. As shown in Fig. 16, the conventional bcache cannot effectively prioritize higher-weighed applications, resulting in 2.59 of PV. On the other hand, *WaC* shows only 0.94 of PV by gracefully overcoming the limitation of the conventional bcache through *WaL*. The computation overhead of *WaL* is imperceptible in the case of bcache, because bcache is a storage device which is much slower than CPU and main memory. Therefore, in this experiment, we did not observe

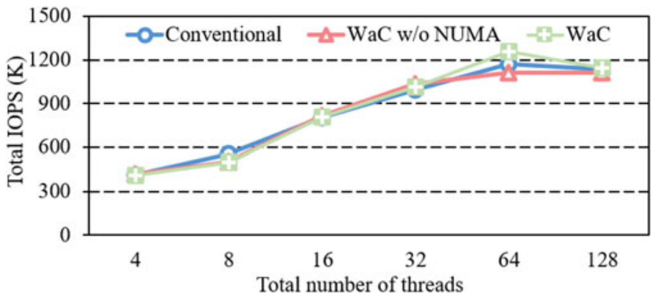


Fig. 15. Scalability of WaC.

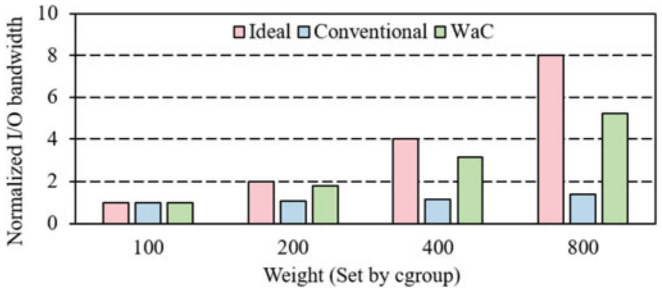


Fig. 16. I/O proportionality on FIO random writes.

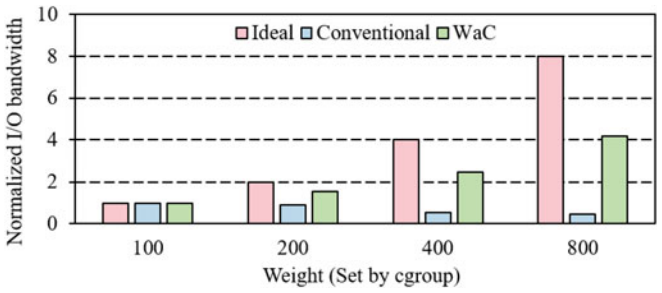


Fig. 17. I/O proportionality on FIO random reads.

any noticeable overheads of *WaC* in terms of the total I/O bandwidth. Specifically, the raw bandwidths (MB/s) of the conventional bcache from weight 100 to weight 800 are 111.14, 116.82, 130.18, and 156.65. The raw bandwidths (MB/s) of *WaC* are 47.01, 84.09, 149.84, and 246.20.

6.3.2 Random Read Test

To examine the I/O proportionality of read performance, we ran random read workloads in four differently weighted containers. Prior to measuring the performance, we ran the benchmark once to warm up the cache. As shown in Fig. 17, the conventional bcache shows the PV of 3.01, while *WaC* exhibits that of 1.45. Moreover, on the conventional bcache, the lower weighted containers show higher read bandwidth because their data are cached more in the warm-up phase. The random read workload is not time-based, and thus the higher-weighted containers finished earlier than the others in the warm-up phase. Therefore, the data for lower-weighted containers evicts the data for higher weighed ones, according to the recency policy of LRU. On the other hand, *WaR* of *WaC* considers not only recency, but also I/O weight. Therefore, bcache keeps more cache entries of higher-weighted containers, thereby showing better I/O proportionality. The total read bandwidth of the conventional and our scheme are 2,047 MB/s and 2,188 MB/s, respectively. In detail, the raw bandwidths (MB/s) of the conventional bcache from weight 100 to weight 800 are 700, 643, 375, and 329, whereas those of *WaC* are 238, 371, 588, and 991.

WaR induces additional CPU computations, compared with the conventional reclamation. Therefore, we additionally measured the CPU utilization of cache reclamation during the evaluation using the perf profiler. Here, 1 CPU utilization means that a thread occupies 1 CPU core during the given period. In the case of the conventional bcache, cache reclamation requires 0.009 CPU utilization during the evaluation whereas *WaR* requires 0.014 CPU utilization. Although *WaR* shows around 1.56 times higher CPU utilization, the CPU utilization is still extremely low. Therefore, we believe the additional computation of *WaR* is negligible.

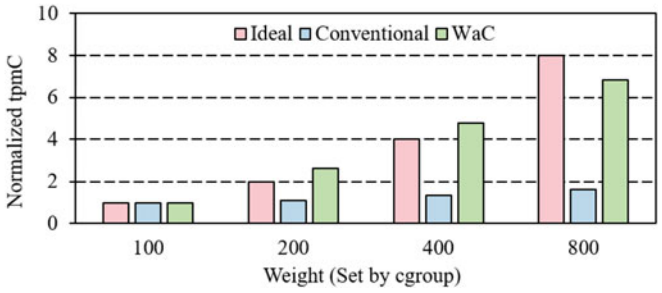


Fig. 18. I/O proportionality of TPC-C.

6.3.3 TPC-C

To extensively verify the efficacy of our scheme, we ran the TPC-C workload with mysql-innodb. The TPC-C workload generates 92% of read/write transactions and 8% of read-only transactions [35]. In terms of I/O requests, the workload issues small-sized random read/write operations, and the ratio of read/write is around 1.9:1 [35]. The TPC-C workload generates around 40 GB of data in total. The performance of TPC-C is measured in transactions per minute (tpmC). As shown in Fig. 18, the conventional bcache could not effectively differentiate the performance of differently weighted containers. As a result, the PV of the conventional bcache is 2.48. The detailed I/O proportionality of the conventional bcache is 1 : 1.09:1.35:1.62. On the contrary, *WaC* prioritizes containers according to their I/O weights during both cache allocation and reclamation. Consequently, *WaC* shows the PV of 0.63 (1:2.63:4.78:6.85). Moreover, the total tpmC of *WaC* is 18% higher than that of the conventional scheme. In the case of *WaC*, higher-weighted containers can utilize sufficient amount of cache resource, thereby greatly improving their performance. As a result, the tpmC of the container with weight 800 is around 66% higher with *WaC* than the conventional scheme.

6.3.4 Heterogeneous Workloads

Finally, we ran two heterogeneous workloads in a time-varying manner to further verify the effectiveness of our scheme. In this experiment, we utilize two different workloads, a random write workload of FIO for a malicious application and a Fileserver workload of filebench for a high-priority application. We ran the FIO and Fileserver workload for 240 and 300 seconds, respectively. To minimize the performance interference of the malicious application, we set the priority of FIO as 100 and that of the Fileserver as 500. First, we run the FIO benchmark prior to the Fileserver in order to occupy cache entries with data generated by FIO. After 30 seconds, we execute the Fileserver workload and measure the I/O throughput using iotop [36].

Fig. 19 shows the I/O throughput of the workloads with the conventional cache management and *WaC*. As shown in the figure, the Fileserver workload with the conventional scheme achieves around 83% lower I/O throughput than that with *WaC* on average. This significant performance difference stems from the fact that the conventional cache management

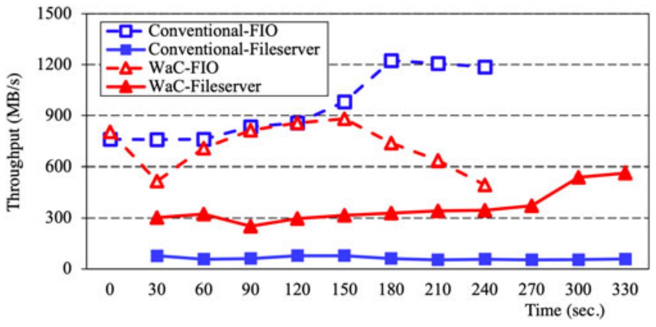


Fig. 19. The I/O throughput of heterogeneous workloads in a time-series manner.

fails to prioritize the Fileserver workload, although the Fileserver workload has 5 times higher priority than the FIO workload. Especially, since we execute the FIO workload ahead of the Fileserver workload, the data from the FIO workload occupy most of the cache entries. Therefore, the Fileserver workload cannot effectively utilize the cache resources. Additionally, the conventional cache allocation is priority-oblivious, thereby failing to prioritize the Fileserver workload.

On the other hand, *WaC* preferentially allocates cache entries to the Fileserver workload over the FIO workload due to its *WaL*. Additionally, its *WaR* tries to keep more data of the Fileserver workload than the FIO workload. As a result, the read and write throughput of the Fileserver with *WaC* are 291 MB/s and 281 MB/s, respectively, while those with the conventional are 66.5 MB/s and 65.9 MB/s. Note that the I/O throughput of FIO with the conventional scheme is 805 MB/s where as that of *WaC* is 663 MB/s.

7 Discussion

Since proportional I/O sharing is an essential requirement to construct a cloud computing environment, there has been several research to improve I/O proportionality. For example, J. Kim *et al.* [34], [37] proposed A+CFQ and H+BFQ which extends Linux I/O schedulers considering internals of SSDs. Additionally, S. Ahn *et al.* [38] proposed a new scheme, which predicts the future I/O demands of each container and collaboratively manages read/write operations. Similarly, Woo *et al.* [39] suggested a light-weight fair-queueing scheme to provide fairness while minimizing CPU consumption. However, such schemes still do not consider the existence of cache layers in the I/O path, which is introduced to improve I/O performance in almost all types of system. *WaC*, on the other hand, is a new cache management scheme that improves application-level I/O proportionality even with the cache layers.

Cgroup v2 [40], which is the next version of cgroup, provides features to control writeback by setting dirty page ratio. However, cgroup v2 still cannot control the page cache with the I/O weight although I/O weight is a straight-forward and user-friendly method. Additionally, cgroup v2 cannot solve problems incurred by the locking mechanism in page allocation. Finally, cgroup v2 has no ability to control bcache for the sake of application-level I/O proportionality.

P. Sharma *et al.* [41] proposed a per-VM page cache partitioning scheme. The main contribution of the paper is to increase hit ratio with small-sized memory by isolating VMs in the page cache layer. Most recently, S. Kashyap *et al.* [29] has proposed the shuffling mechanism that re-orders lock waiting queue based on a certain policy. They mainly focused on solving the conventional lock problems, such as memory footprint, with NUMA-awareness. On the other hand, the contribution of this paper is to propose a new cache management design to achieve application-level I/O proportionality.

In this paper, we address the I/O proportionality problem of virtualized environments, particularly with Docker virtualization, because proportional I/O sharing is a critical factor in such environments. However, since the page cache and bcache mechanism are identical in a system without virtualization, *WaC* is also applicable to non-virtualized environments.

We extended the previous version of our research [12], [42] in two ways. First, *WaC* extended *WaL* to consider the NUMA topology when deciding the next lock holder. The NUMA architecture has been widely adopted in many servers due to its superior performance. Therefore, NUMA-aware lock design is necessary to build a high-performance system. Second, there are various I/O caches in the current computer systems. However, the previous version of our work focused only on the page cache. In this paper, we re-designed our scheme to be general so that it can be adopted to various I/O caches. Additionally, we implemented and evaluated our idea on both the page cache and bcache in this paper.

8 Conclusion

In this paper, we proposed a new I/O cache management scheme, called *WaC* (Weight-aware Cache), to achieve application-level proportional I/O sharing. *WaC* prioritizes higher-weighted applications both in cache allocation and reclamation. For cache allocation, *WaC* utilizes *WaL* which considers I/O weight and NUMA-topology when deciding the next lock holder. For cache reclamation, *WaR* of *WaC* tries to keep the number of cache entries proportional to I/O weight. We implemented and evaluated our idea on both the page cache and bcache, and the experimental results demonstrate that *WaC* effectively improves application-level I/O proportionality, compared with the conventional cache management.

References

[1] J. Moura and D. Hutchison, "Review and analysis of networking challenges in cloud computing," *J. Netw. Comput. Appl.*, vol. 60, pp. 113–129, Jan. 2016.
[2] R. Uhlig *et al.*, "Intel virtualization technology," *IEEE Comput.*, vol. 38, no. 5, pp. 48–56, May 2005.
[3] P. Barham *et al.*, "Xen and the art of virtualization," in *Proc. ACM Symp. Oper. Syst. Princ.*, 2003, pp. 164–177.
[4] M. Marzolla, O. Babaoglu, and F. Panzieri, "Server consolidation in clouds through gossiping," in *Proc. IEEE Int. Symp. World Wireless Mob. Multimed. Netw.*, 2011, pp. 1–6.
[5] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proc. ACM SIGOPS Eur. Conf. Comput. Syst.*, 2007, pp. 275–287.
[6] J. Kim, D. Lee, and S. H. Noh, "Towards SLO complying SSDS through OPS isolation," in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 183–189.
[7] M. Kwon, D. Gouk, C. Lee, B. Kim, J. Hwang, and M. Jung, "Dc-store: Eliminating noisy neighbor containers using deterministic I/O performance and resource isolation," in *Proc. USENIX Conf. File Storage Technol.*, 2020, pp. 183–191.
[8] A. Tavakkol *et al.*, "Flin: Enabling fairness and enhancing performance in modern NVME solid state drives," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2018, pp. 397–410.

[9] J. Huang *et al.*, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized SSDS," in *Proc. USENIX Conf. File Storage Technol.*, 2017, pp. 375–390.

[10] Control groups resource management. Accessed: Mar. 23, 2021. [Online]. Available: https://libvirt.org/cgroups.html

[11] Cgroups. Accessed: Mar. 11, 2021. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[12] J. Park, K. Oh, and Y. I. Eom, "Towards application-level I/O proportionality with a weight-aware page cache management," in *Proc. Int. Conf. Massive Storage Syst. Technol.*, 2020, pp. 1–11.

[13] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," *J. Softw. Eng. Appl.*, vol. 5, no. 4, pp. 277–290, Jan. 2012.

[14] J. Park and Y. I. Eom, "Fragpicker: A new defragmentation tool for modern storage devices," in *Proc. ACM Symp. Oper. Syst. Princ.*, 2021, pp. 280–294.

[15] A block layer cache (bcache). Accessed: Mar. 10, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/bcache.html

[16] MCS locks and qspinlocks. Accessed: Feb. 10, 2021. [Online]. Available: https://lwn.net/Articles/590243

[17] S. Kashyap, C. Min, and T. Kim, "Opportunistic spinlocks: Achieving virtual machine scalability in the clouds," *SIGOPS Oper. Syst. Rev.*, vol. 50, no. 1, pp. 9–16, Jan. 2016.

[18] CFQ (complete fairness queueing). Accessed: Dec. 10, 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt

[19] Block io controller. Accessed: Mar. 11, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/blkio-controller.html

[20] K. Oh, J. Park, and Y. I. Eom, "H-BFQ: Supporting multi-level hierarchical cgroup in BFQ scheduler," in *Proc. IEEE Int. Conf. Big Data Smart Comput.*, 2020, pp. 366–369.

[21] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O path: A holistic approach for application performance," in *Proc. Conf. File Storage Technol.*, 2017, pp. 345–358.

[22] Y. Wang and A. Merchant, "Proportional-share scheduling for distributed storage systems," in *Proc. Conf. File Storage Technol.*, 2007, pp. 47–60.

[23] S. Chen, T. Chen, Y. Chang, H. Wei, and W. Shih, "Enabling union page cache to boost file access performance of NVRAM-based storage device," in *Proc. Des. Autom. Conf.*, 2018, pp. 1–6.

[24] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim, "Fasttrack: Foreground app-aware I/O management for improving user experience of Android smartphones," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 15–27.

[25] J. Bhimani *et al.*, "Docker container scheduler for I/O intensive applications running on NVMe SSDs," *IEEE Trans. Multi-Scale Comput. Sys.*, vol. 4, no. 3, pp. 313–326, Feb. 2018.

[26] Lockstat: Documentation. Accessed: Feb. 11, 2021. [Online]. Available: https://lwn.net/Articles/252835

[27] J. Park, C. Min, and H. Yeom, "A new file system I/O mode for efficient user-level caching," in *Proc. IEEE/ACM Int. Symp. Clust., Cloud Grid Comput.*, 2017, pp. 649–658.

[28] D. Zheng, R. Burns, and A. S. Szalay, "A parallel page cache: IOPS and caching for multicore systems," in *Proc. USENIX Conf. Hot Top. Storage File Syst.*, 2012, pp. 1–6.

[29] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim, "Scalable and practical locking with shuffling," in *Proc. ACM Symp. Oper. Syst. Princ.*, 2019, pp. 586–599.

[30] Z. Radovic and E. Hagersten, "Hierarchical backoff locks for nonuniform communication architectures," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2003, pp. 241–252.

[31] S. Kashyap, C. Min, and T. Kim, "Scalable Numa-aware blocking synchronization primitives," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 603–615.

[32] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing numa locks," *ACM Trans. Parallel Comput.*, vol. 1, no. 13, pp. 1–42, Feb. 2015.

[33] D. Dice and A. Kogan, "Compact Numa-aware locks," in *Proc. ACM SIGOPS Eur. Conf. Comput. Syst.*, 2019, pp. 1–15.

[34] J. Kim, E. Lee, and S. H. Noh, "I/O scheduling schemes for better I/O proportionality on flash-based SSDs," in *Proc. IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2016, pp. 221–230.

[35] S. Chen *et al.*, "TPC-E VS. TPC-C: Characterizing the new tpc-e benchmark via an I/O comparison study," *ACM SIGMOD Rec.*, vol. 39, no. 3, pp. 5–10, Feb. 2011.

[36] iotop. Accessed: Mar. 10, 2021. [Online]. Available: https://linux.die.net/man/1/iotop

[37] J. Kim, E. Lee, and S. H. Noh, "I/O schedulers for proportionality and stability on flash-based ssds in multi-tenant environments," *IEEE Access*, vol. 8, pp. 4451–4465, 2020.

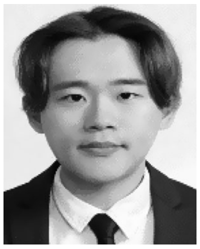
[38] S. Ahn, K. La, and J. Kim, "Improving I/O resource sharing of Linux cgroup for NVMe SSDs on multi-core systems," in *Proc. USENIX Workshop Hot Top. Storage File Syst.*, 2016, pp. 111–115.

[39] J. Woo, M. Ahn, G. Lee, and J. Jeong, "D2FQ: Device-direct fair queueing for NVME SSDS," in *Proc. USENIX Conf. File Storage Technol.*, 2021, pp. 403–415.

[40] Cgroups v2. Accessed: Feb. 16, 2021. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[41] P. Sharma, P. Kulkarni, and P. Shenoy, "Per-vm page cache partitioning for cloud computing platforms," in *Proc. Int. Conf. Commun. Syst. Netw.*, 2016, pp. 1–8.

[42] K. Oh, J. Park, and Y. I. Eom, "Weight-based page cache management scheme for enhancing I/O proportionality of cgroups," in *Proc. IEEE Int. Conf. Consum. Electron.*, 2019, pp. 1–3.



Jonggyu Park received the BS degree in software from Sungkyunkwan University, South Korea, in 2014 and the MS degree in 2016 in platform software from Sungkyunkwan University, where he is currently working toward the PhD degree with the Department of Platform Software. His research interests include storage systems and operating systems.

Young Ik Eom received the BS, MS, and PhD degrees in computer science and statistics from Seoul National University, South Korea, in 1983, 1985, and 1991, respectively. Since 1993, he has been a professor with Sungkyunkwan University, South Korea. From 2000 to 2001, he was a visiting scholar with the Department of Information and Computer Science, University of California at Irvine. He was also the president of the Korean Institute of Information Scientists and Engineers in 2018. His research interests include virtualization, operating systems, file and storage systems, cloud systems, and UI or UX system.