Towards Application-level I/O Proportionality with a Weight-aware Page Cache Management

Jonggyu Park
Sungkyunkwan University
Suwon, South Korea
jonggyu@skku.edu

Kwonje Oh
Sungkyunkwan University
Suwon, South Korea
keinoh@skku.edu

Young Ik Eom
Sungkyunkwan University
Suwon, South Korea
yieom@skku.edu

***Abstract*—Cloud systems often use blkio subsystem of Cgroups for controlling I/O resources to guarantee the service-level objective (SLO) of the systems. However, the blkio subsystem of Cgroups is originally designed to achieve block-level I/O proportionality without consideration on the upper layers of the system software stack, such as the page cache layer. Therefore, when an application utilizes buffered I/O, the performance of the application can exhibit unexpected results in its I/O proportionality even though the block-level I/O proportionality is still being guaranteed. To address this problem, we suggest a weight-aware page cache management scheme, called Justitia, which realizes application-level I/O proportionality in the systems that use OS virtualization technologies. Justitia prioritizes higher-weighted applications in the lock acquisition process of the page allocation by re-ordering the lock waiting queue based on their I/O weights. Additionally, it keeps the number of allocated pages for each application proportional to its I/O weight, with a weight-aware page reclamation scheme. Our experiments show that Justitia effectively improves I/O proportionality with negligible overhead in various cases.**

***Index Terms*—Storage systems, scheduling and resource management, performance and quality of service, virtualization**

I. Introduction

Cloud services, which utilize various types of virtualization technologies such as hardware virtualization and OS virtualization, are now pervasive in modern computing environments [1]. In those cloud services, guaranteeing service-level objective (SLO) is crucial in order to meet the needs of clients. The majority of virtualization techniques, such as KVM and Docker, guarantee SLO by precisely managing the host kernel resources using Cgroups [2], [3]. Cgroups represents resources in the form of per-resource-type subsystems [3]. Each subsystem contains several parameters that control the consumption of resources. For example, blkio (block I/O) subsystem provides the block I/O weight [3] parameter that can differentiate the I/O bandwidth of containers in the block layer.

However, Cgroups is designed to achieve just block-level I/O proportionality, rather than application-level I/O proportionality, and therefore, it manages I/O resources only in the block layer without giving consideration to the upper layers of the system software stack, such as the page cache layer. As a result, when an application utilizes buffered I/O which uses 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 exhibit disproportional I/O bandwidth in the holistic point of view.

The page cache has been widely used to mitigate the performance gap between CPU and the underlying storage. It utilizes the main memory space to temporarily store data coming from or going to the secondary storage devices. [4]. The page cache substantially improves I/O performance by reducing the frequency of relatively slow accesses to the storage devices [5]. Due to the I/O performance enhancement by the page cache, Docker containers also utilize the host page cache so as to experience the superior I/O bandwidth [5]. However, when containers try to access data in the page cache, such requests cannot be controlled and managed by Cgroups, since they do not go through the block layer. As a result, the page cache can cause dis-proportionality of I/O performance.

The conventional page cache management performs page allocation and reclamation without any considerations on the I/O proportionality and I/O weights. Page cache allocation consists of several steps including allocating a free page and inserting the page into the page cache. Since page allocation should be mutually exclusive among multiple processes, the conventional page cache management adopts *qspinlock* for synchronization of page allocation, 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 [6], [7]. With the FIFO-queue, regardless of I/O weight, the lock acquisition order follows the order of enqueueing. Therefore, the conventional page cache management does not reflect I/O weight in page allocation.

Page reclamation in the page cache refers to the process that evicts existing pages to secure free pages. In page reclamation, the conventional page cache management considers only recency and frequency of page references without considering the I/O weight, even though reflecting

I/O weight is crucial for guaranteeing SLO in virtualized systems. Thus, the conventional page 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 page cache can distort application-level I/O proportionality.

To solve this problem, we propose a weight-aware page cache management scheme, called `Justitia`, which realizes application-level I/O proportionality in the virtualized systems. `Justitia` consists of two different parts, page allocation and reclamation. The page allocation scheme of `Justitia` prioritizes higher-weighted applications in the lock acquisition process of page allocation. When the lock for page allocation is available, `Justitia` traverses the entire lock waiting queue and finds out the element with the highest I/O weight. Afterward, `Justitia` 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. Throughout this process, `Justitia` can achieve proportional I/O sharing according to the I/O weight of each application.

The page reclamation scheme of `Justitia` prioritizes pages that are used by higher-weighted applications during page reclamation. `Justitia` keeps track of the ownership information of each page, and calculates the I/O proportion of each application based on its I/O weight. During page reclamation, `Justitia` tries to keep the number of pages allocated to each application proportional to its I/O weight by reclaiming pages of the application that has more pages than it should. Therefore, the application with higher I/O weight can have more pages in the page cache.

Re-ordering the lock waiting queue based on I/O weight can incur a problem, called starvation, where an application is denied 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 its chance for lock acquisition, thereby experiencing starvation. To solve the problem, we adopt a conventional well-known technique for starvation, called aging. `Justitia` 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.

In our experiments with the Docker virtualization, we measured the application-level I/O proportionality and the performance of `Justitia`, while comparing them with those of conventional Linux page cache management. Evaluation results with real-world benchmarks indicate that `Justitia` shows up to 36.9% better I/O proportionality than the conventional scheme with only 3.9% overheads at most.

The rest of this paper is organized as follows. Section II elaborates the background and motivation of our work. Section III explains a new page cache management scheme, called `Justitia`, in detail. Experimental results are presented in Section V. Section VI discusses the related work. We conclude this paper in Section VII.

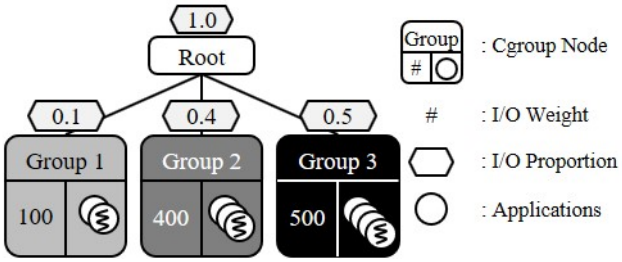


Fig. 1: An Overview of Resource Sharing in Cgroups.

II. Background & Motivation

In this section, we describe some backgrounds on Docker container and Cgroups. Then, a brief overview on the page cache is given with the motivation behind this work. We also present some motivational experimental results.

A. Docker and Cgroups

Docker has been one of the most popular virtualization technologies due to its fascinating advantages compared with the hypervisor-based virtualization frameworks, such as the elimination of duplicated system stack and the resulting high performance [8], [9]. Docker provides virtualized computing environments using container technology, called OS-level virtualization [8], [9]. With this virtualization technique, multiple isolated units (containers) that share a single kernel instance can run on one physical host machine. However, this incurs resource contention among the containers, resulting in unexpected performance variation of each container. Therefore, to satisfy the performance SLO of the containers, Docker manages and regulates the shared system resources with Cgroups.

Cgroups represents system resources such as CPU, memory, and block I/O in the form of subsystems [3]. Each subsystem has various parameters that control or limit resource consumption. Among those subsystems, the blkio subsystem plays the role of monitoring and controlling block I/O requests. Specifically, a blkio resource group is allocated to each container or application, and the blkio subsystem controls the I/O bandwidth ratio of the containers/applications [10] by adjusting the I/O weight values of the corresponding resource groups.

In conventional systems, CFQ (Completely Fair Queuing) I/O scheduler uses this I/O weight value (blkio.weight) [3] when it decides how many I/Os will be dispatched from the request queues in the block layer [10]. By doing so, CFQ scheduler gives more time for processing I/Os to higher-weighted applications so that the block-level I/O proportionality can be achieved according to their I/O weights [11]. Note that, in this paper, I/O proportionality [12] refers to the bandwidth ratios of applications which are proportionally allocated according to their I/O weights. For example, as shown in Fig. 1 let

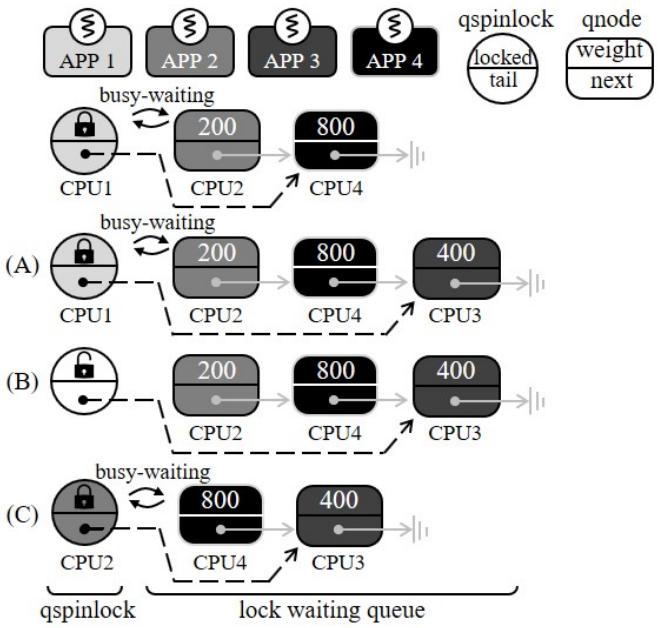


Fig. 2: An Overview of Qspinlock.

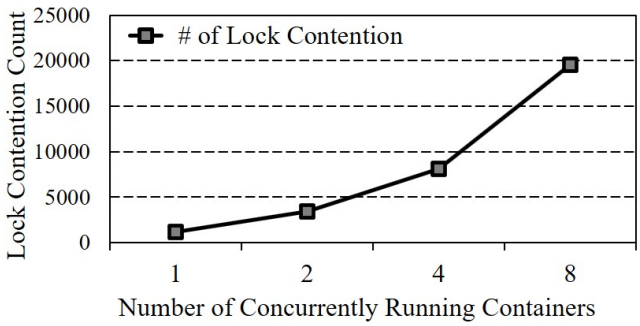


Fig. 3: Lock Contention Count with Varying No. of Containers.

us suppose that the system manager creates three resource groups with I/O weights 100, 400, and 500. Then, 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 buffered I/O because the page cache [4], [11], [13] may exclude the block layer 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 achieve high bandwidth and low latency. However, since such I/O requests do not experience the block layer, the I/O weights of Cgroups cannot be applied. Additionally, the blkio subsystem does not control the page cache layer, and so, the page cache itself cannot achieve the required I/O proportionality.

B. The Conventional Page Cache Management

The page cache works as a caching layer between the user space and the block layer, which supports buffering for write operations and caching for read operations. It reduces the frequency of relatively slow accesses to the storage devices by directly servicing I/O requests when the corresponding data reside in it [5]. Page cache management consists of two tasks, page cache allocation and page cache reclamation.

1) Page Cache Allocation: Page cache allocation is to allocate a clean page for newly accessed data and insert the page into the page cache. Since the page cache are shared among multiple CPUs, page cache allocation should be mutually exclusive [13]. For synchronization, the kernel protects it with a locking mechanism, called *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 instance of lock waiting queue [6]. In the locking mechanism, for page allocation, the conventional page cache management selects the next lock holder from the lock waiting queue in a FIFO manner [6], [7]. Therefore, the conventional page cache allocation cannot prioritize applications with higher I/O weights. For example, in Fig. 2, when CPU3 tries to acquire the qspinlock, it is queued 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 qspinlock in a FIFO manner without consideration on I/O weight. Therefore, even though applications on CPU3 and CPU4 have higher I/O weights than that on CPU2, CPU2 acquires the qspinlock prior to CPU3 and CPU4. Likewise, the conventional page cache management handles page allocation using a FIFO-based queue which does not consider I/O weights, thereby distorting I/O proportionality.

Page allocation and its lock contention are critical to the I/O performance [13], and have an increasing impact on the performance as the number of containers co-running in a system increases. In a multi-container environment, lock contention increases due to the high number of simultaneous page allocation requests. Thus, I/O requests from a container have to wait a significant amount of time for acquiring the lock. We illustrate this problem in Fig. 3, by measuring the lock contention count using `lockstat` [14] while running multiple `fileserver` workloads. 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 containers, mainly due to high contention of page allocation requests from multiple containers. When the number of containers becomes eight

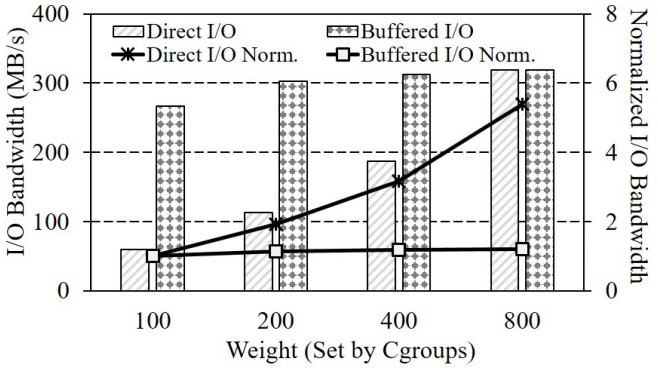


Fig. 4: Comparison of I/O bandwidths and normalized I/O bandwidths between Direct I/O and Buffered I/O, with Fileserver Workload (I/O bandwidth is normalized to the container of weight 100.)

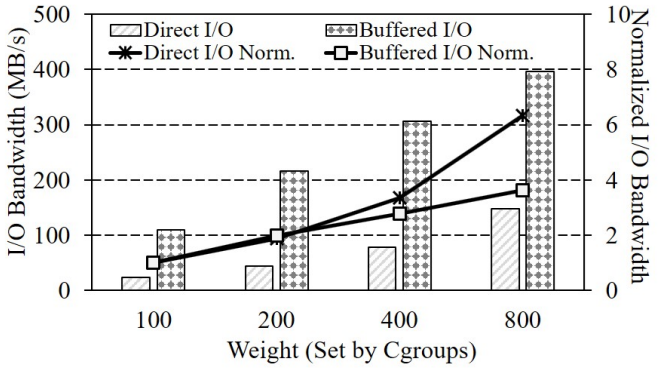


Fig. 5: Comparison of I/O bandwidths and normalized I/O bandwidths between Direct I/O and Buffered I/O, with Re-read Workload (I/O bandwidth is normalized to the container of weight 100.)

in Fig. 3, the number of lock contentions reaches up to around 19,500. Like this, the more containers run in a system, the more important the order of lock acquisition is. Moreover, this problem is exacerbated when the amount of available system memory is low and page allocation induces page reclamation due to it [13]. It is known that, in such cases, allocating free pages can take more than 200ms because dirty pages should be evicted in advance to create free pages [11], [13].

To show the distortion of I/O proportionality caused by the page cache allocation, we conducted an experiment with `Fileserver` workload in Filebench. We ran this workload in each of four containers with different weights (i.e., 100, 200, 400, 800). 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. We used the default `Fileserver` setting except that the number of files was set to 30,000. The experimental setup is described in Table 1 of Section 4.

Fig. 4 shows the 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. Note that, since we only need the relative ratio of the bandwidth of each container in calculating I/O proportionality, all the results for I/O proportionality are normalized in this paper. As shown in Fig. 4, direct I/O exhibits decent I/O proportionality (1 : 1.9 : 3.2 : 5.4). On the other hand, 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 page allocation stage which is related to write performance. Rather, it handles applications in a FIFO manner during the page 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 of direct I/O and buffered I/O shows a clear reason for using the page cache. In the experiment of Fig. 4, the total bandwidth of four containers with direct I/O is 677.3MB/s while that with buffered I/O is 1200.4MB/s. Considering this huge I/O performance gain of the page cache, we cannot ignore the page cache even in cloud systems where SLO is very crucial [5], [15].

2) Page Cache Reclamation: Page reclamation is performed to remove existing pages to secure free pages in the page cache. The conventional page cache management maintains two Least Recently Used (LRU) lists: an active list to keep frequently accessed pages and an inactive list for the other pages [16]. 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 a predefined threshold. However, in this process, the LRU policy only considers the reference counts and the recency of page reference without any consideration on the I/O weights of applications. Therefore, pages used by higher-weighted applications can be evicted prior to those used by lower-weighted ones. This phenomenon degrades the read performance of higher-weighted applications since their read requests should access the underlying storage. Consequently, the conventional page cache reclamation results in disproportionate I/O sharing.

To show the distortion of I/O proportionality caused

by the page cache reclamation, we conducted an experiment with `Re-read` workload of FIO. Like the previous motivational experiment of Fig. 4, we ran this workload in each of four containers with different weights (i.e., 100, 200, 400, 800). Each container creates one 3GB test file to cache all the pages of the file into the page cache. Then, the host writes a 4GB dummy file to trigger excessive page reclamation. Lastly, we executed 1GB `Re-read` operations in each container and measured the I/O bandwidth as shown in Fig. 5. The experiment is also performed with the two different types of I/Os, direct I/O and buffered I/O. As shown in Fig. 5, buffered I/O shows poor I/O proportionality than 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 pages after the dummy writes from the host, it does not consider I/O weights during the page reclamation. Therefore, even though some pages are used by higher-weighted containers, the pages are evicted prior to pages used by lower-weighted ones, resulting in poor I/O proportionality.

The aforementioned analyses on the conventional page cache management lead us to conclude that strictly keeping the FIFO and 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 page cache management scheme that gracefully resolves the aforementioned problems. Note that, in this paper, we regard read/write operations as I/O even if they are not delivered to the layers beneath the page cache.

III. Justitia: Weight-aware Page Cache Management

As discussed in the previous section, the conventional page cache management handles synchronization for page cache allocation using the FIFO *qspinlock* and manages the pages in the page cache based on the LRU policy, whereby I/O weight of each application is not reflected at all. In this section, we propose a weight-aware page cache management scheme, called `Justitia`, which realizes application-level I/O proportionality. `Justitia` consists of two main policies: (1) `Justitia` prioritizes higher-weighted applications in the page allocation stage and (2) tries to balance the number of pages of each application in the page cache according to its I/O weight during the page reclamation stage.

A. Weight-aware Qspinlock for Page Cache Allocation

During page allocation for the page cache, the conventional *qspinlock* manages synchronization by enqueueing the competing applications in the order of entrance (FIFO). However, this mechanism of *qspinlock* distorts I/O proportionality, because the FIFO policy ignores the differentiated bandwidth requirements (i.e., I/O weight) of the applications. Our solution to enhance I/O proportionality during page allocation is to re-order the lock waiting queue based on I/O weights.

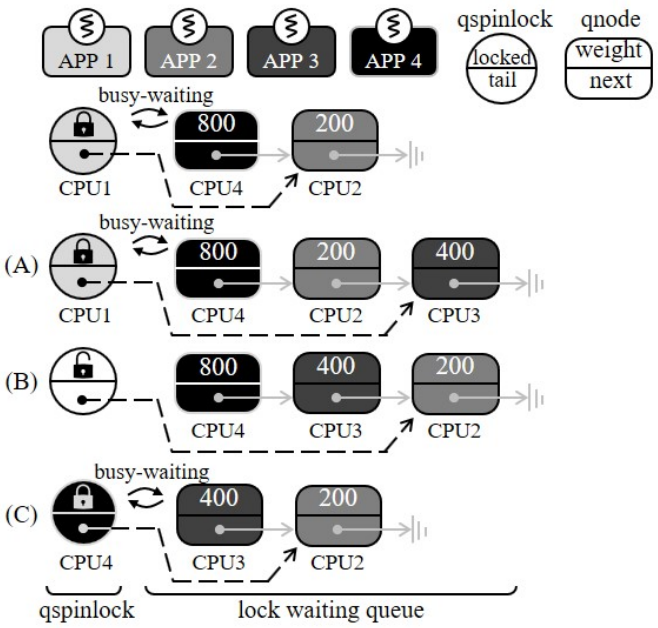


Fig. 6: Weight-aware Qspinlock for Page Cache Allocation.

The overview of the weight-aware *qspinlock* is shown in Fig. 6, and the detailed pseudo-code is presented in Algorithm 1. First, when an application requests a new page for the page cache, `Justitia` creates a qnode structure for the application at the tail of the waiting queue and stores its I/O weight into the qnode structure(A). When the current qspinlock holder releases the lock, the head node of the waiting queue stops busy-waiting for the qspinlock and traverses the waiting queue to find the highest-weighted node (maxNode) among the qnodes that follow the head node. And then, the head node moves the maxNode right after the head node by manipulating its *next* pointer to the address of the maxNode and adjusting the *next* pointers of other qnodes that are affected by the re-ordering(B). Eventually, as the head node holds the qspinlock, the maxNode becomes the new head of the queue and would preferentially acquire the qspinlock when the lock holder releases the qspinlock(C). Therefore, the higher-weighted applications obtain pages relatively faster than the applications with lower weights.

In the example of Fig. 6, when CPU1 releases the qspinlock, CPU4 stops busy-waiting and seeks the highest-weighted node among the qnodes that follow the head node (CPU4). Then, CPU4 shifts CPU3 to its next node and manipulates *next* pointers of other qnodes that are related to the shift. Since the CPU3 is at the tail in the case of Fig. 6, `Justitia` additionally changes the current lock holder's *tail* (CPU1) to point to the previous qnode (CPU2) of the original tail (CPU3). The *next* pointer of CPU3 and

Algorithm 1 Page Allocation with Aging Technique

```
if qspinlock is available then
    qnode == head;
    maxWeight == 0;
    maxNode, prevNode, iterNode == NULL;
    // Find the qnode with the maximum I/O weight
    while qnode → next ≠ tail do
        if maxWeight < qnode →next→weight then
            prevNode = qnode;
            maxNode = qnode → next;
            maxWeight = qnode → weight;
        end
        qnode = qnode → next;
    end
    // Reordering the lock waiting queue
    prevNode → next = maxNode → next;
    maxNode → next = head → next;
    head → next = maxNode;
    iterNode = maxNode;
    // Aging phase
    while IterNode→next ≠ tail do
        iterNode → next → weight += agingValue;
        iterNode = iterNode → next;
    end
    head.acquire(lock);
end
```

CPU2 are modified to point to CPU2's qnode and NULL, respectively. Consequently, when the current head node acquires the qspinlock, the CPU3's qnode becomes the head of the queue and would preferentially acquire the lock in the next phase.

However, `Justitia` can incur the starvation problem, where an application is denied to acquire the lock for very long time, in that lower-weighted applications should repetitively yield their turns of lock acquisition. When there are a large number of higher-weighted applications, lower-weighted applications might not have an opportunity to acquire the lock since it is unlikely to become the maxNode. Building a robust system necessarily requires prevention of such starvation problem because it can incur long-term unfairness and even system failure.

To prevent this, we adopt a conventional well-known technique for starvation, called aging, which gradually increases the priority of a task over time, based on its waiting time in the waiting queue. `Justitia` increases the weights of the qnodes that are not selected (CPU2's qnode in the case of Fig. 6) by a certain value, whenever a lock acquisition happens. By doing so, `Justitia` can consider not only I/O weight but also waiting time in deciding the next lock holder for page allocation.

Algorithm 1 presents the pseudo-code of `Justitia` with the aging technique. After finding the qnode with the maximum I/O weight (maxNode), `Justitia` moves the maxNode next to the current head node so that the maxNode can take the qspinlock in the next turn. Afterward, in order to prevent the starvation problem, `Justitia` increments the I/O weights of all the qnodes after the maxNode, which yielded their turns at the reordering phase. The maximum I/O weight that can be manually set by Cgroups is 1000. On the other hand, the adjusted I/O weights can be higher than the maximum I/O weight (1000). Accordingly, regardless of the I/O weight, any application can eventually become the maxNode due to the aging technique and thus acquire the lock. Consequently, it is guaranteed that applications can eventually possess the qspinlock for page allocation in all cases, thus preventing the starvation problem. In our experiments of Section IV, we set the value of increment to 100, considering the scale of weights of the containers.

B. Weight-aware Page Reclamation

The conventional page cache consists of two LRU lists to effectively enhance the hit ratio, considering reference counts and recency of the references. However, similar to the page allocation process, this mechanism does not reflect I/O weights of Cgroups and thus degrades I/O proportionality. To resolve this problem, `Justitia` tries to keep the number of allocated pages for each application proportional to its I/O weight by reclaiming pages of the applications which have more pages than it should. To achieve this, page reclamation of `Justitia` performs the following jobs.

1) Calculating I/O proportion of each application
2) Recording page ownership information on the page structure
3) Page reclamation considering the I/O proportion

1) Calculating I/O proportion of each application: First, `Justitia` calculates the I/O proportion of each application using its I/O weight and stores the value into the *proportion* variable. For example, in Fig. 7, the system executes four applications in four Cgroups nodes with the I/O weights 100, 200, 400, and 800 each. Then, each of these applications has *proportion* of 0.07, 0.13, 0.27, and 0.53, respectively, when the total amount of page cache resources available in the system is 1.0. `Justitia` also keeps track of the number of pages allocated to each application in *nrp_pages* variable. `Justitia` manages the *proportion* and *nrp_pages* variable in the corresponding Cgroups node of each application.

2) Recording page ownership information on the page structure: When an application allocates a new page in the page cache, `Justitia` stores I/O weight of the application into the page structure. Afterward, `Justitia` links the page to the corresponding Cgroups node in order to clarify the ownership of the page and refer to the variables (*proportion* and *nrp_pages*) of the Cgroups node during page reclamation. Finally, it increments the *nrp_pages* variable of the Cgroups node to keep track of the number of pages each application possesses. For example, in Fig. 7, since

a new page is allocated to APP1, the *nrp_pages* variable increments from 0 to 1, and the corresponding Cgroups node is linked to the newly allocated page structure. Note that when multiple application share the same page, `Justitia` designates the highest-weighted application as the owner of the page.

3) Page reclamation considering the I/O proportion: During page reclamation, `Justitia` tries to keep the number of pages allocated to each application proportional to its I/O weight. To achieve this, `Justitia` performs the following procedure: when pages need to be reclaimed by the kernel reclamation thread, `Justitia` looks up pages for reclamation from the tail of the inactive list. In this process, `Justitia` calculates the threshold values of applications, considering its I/O weight and proportionality, each of which indicates the maximum number of pages an application is supposed to possess. The threshold value can be calculated by the following equation.

$$Threshold = NR\_FILE\_PAGES \cdot proportion \quad (1)$$

Here, *NR_FILE_PAGES* is the number of file-backed page cache entries that the kernel keeps track of. We use this value for the total number of page cache entries. If an application has more pages than its threshold in the page cache, `Justitia` reclaims its pages. By doing so, `Justitia` can keep the number of pages of each application relative to its I/O weight, thereby enhancing the I/O proportionality.

For example, suppose that the four applications have 1, 6, 1, and 2 pages, respectively, as shown in Fig. 7. Since the total number of page cache entries is 10, *NR_FILE_PAGES* holds 10. When page reclamation occurs, `Justitia` seeks pages whose application has more pages than its threshold. In this example, APP2 possesses 6 pages currently, exceeding its threshold value 1.3 (10 * 0.13). Therefore, `Justitia` reclaims a page of APP2 and decrements *nrp_pages* of its Cgroup node from 6 to 5. By performing this process repetitively, `Justitia` can keep the number of pages of each application proportional to its I/O weight.

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

One might concern if the cache hit ratio of `Justitia` can be decreased due to the reflection of I/O weight during page reclamation. However, `Justitia` still maintains 2Q-LRU for the overall reclamation policy to keep the benefits of the conventional page reclamation. Additionally,

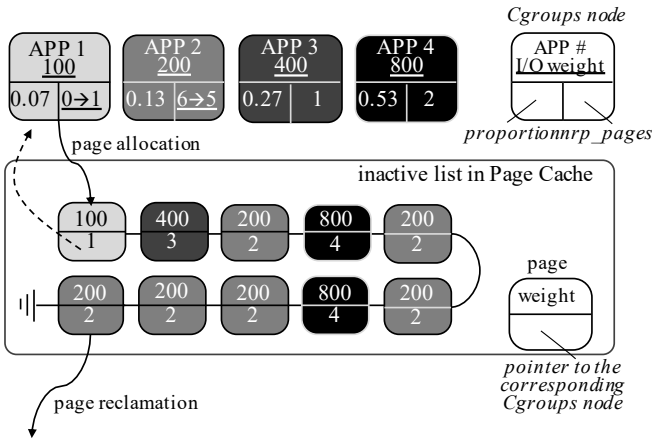


Fig. 7: Weight-aware Page Reclamation.

`Justitia` does not apply I/O weight-awareness to the active list, which is a group of frequently accessed data and contributes significantly to the high cache hit ratio. In our scheme, 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. Also, `Justitia` applies the LRU policy for pages used by the same Cgroup. Finally, `Justitia` performs the conventional page reclamation in the case of the direct reclamation, in which the system has a lack of free pages, in order to prevent the OOM (Out-Of-Memory) problem.

IV. Evaluation

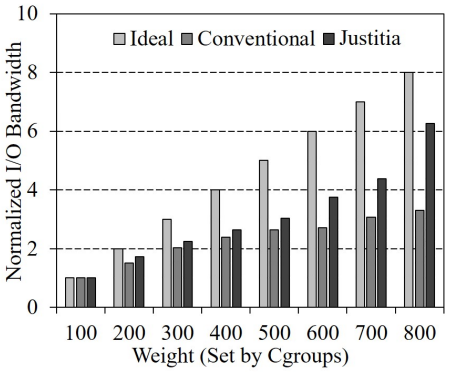
A. Evaluation Setup and Test Settings

TABLE I: Experimental Setup

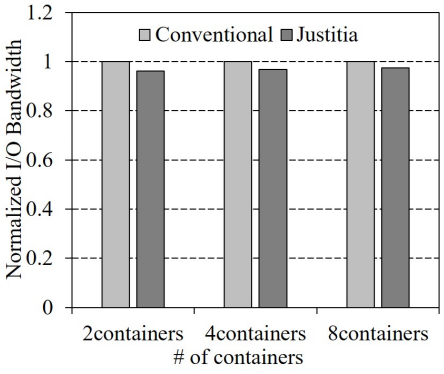
CPU	Intel I7-6700
Memory	16G
Storage	SATA SSD 256G
Docker Base Image	Ubuntu 14.04 LTS
Benchmark	FIO-3.11, Filebench 1.5-alpha3

To implement `Justitia`, we have modified the locking mechanism for page allocation and the page cache management of Linux kernel version 4.19.16. To comprehensively evaluate `Justitia`, we ran both real-world and synthetic benchmarks on a virtualized environment with Docker v18.09.4-CE, as described in Table I.

First, we examined the weight-aware *qspinlock* in page allocation of `Justitia` in terms of application-level I/O proportionality, by running eight `Fileserver` workloads in eight differently weighted containers. The `Fileserver` workload is write-intensive, and so, it incurs frequent page allocation. Secondly, we performed a `Re-read` experiment with `FIO` (Flexible I/O Tester) benchmark to evaluate page reclamation of `Justitia`. In the `Re-read` experiment, four containers with different I/O weights write their own files, pollute the page cache with dummy writes, and then read the files again to examine how many pages

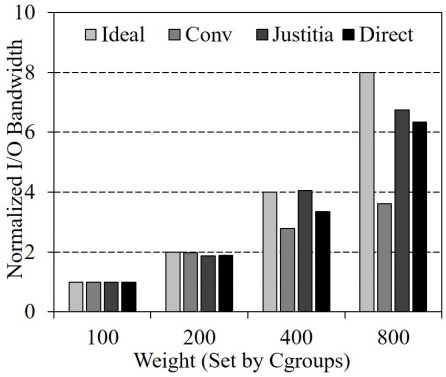


(a) I/O Proportionality on Fileserver



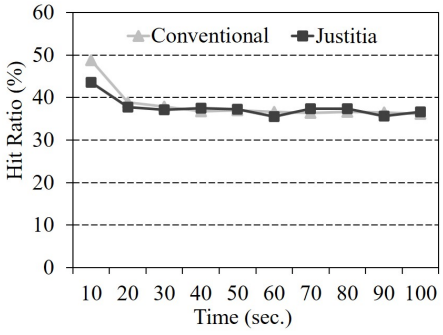
(b) Total Bandwidth (Different Weights)

Fig. 8: Fileserver Results

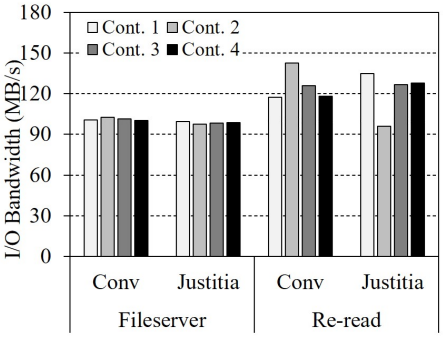


(a) I/O Proportionality on Re-read Case

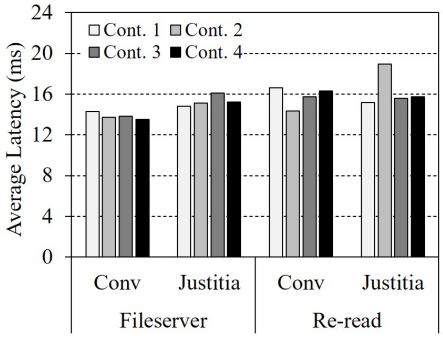
Fig. 9: Re-read Workload Results



(a) Hit Ratio



(b) Bandwidth (Identical Weights)



(c) Average Latency (Identical Weights)

Fig. 10: Overhead of Justitia

of each container reside in the page cache. Thirdly, to investigate the overhead caused by additional procedures of `Justitia`, we executed both write- and read-intensive workloads in four containers with the identical I/O weight, and measured the total bandwidth and the average latency. Additionally, we also measured the page cache hit ratio on read-intensive `Webserver` workload with four containers. Fourthly, to verify that the conventional memory Cgroups is not an alternative to our scheme, we compare our scheme with memory Cgroups by performing both `Fileserver` and `Re-read` experiments. Finally, to confirm the effectiveness of the aging technique, we performed an experiment with an extreme scenario where a lowest-weighted container co-runs along with multiple highest-weighted containers.

Note that all the experiments are performed with the conventional Linux page cache management, called Conventional (or shortly, Conv) in the figures. To quantitatively measure the I/O proportionality, we adopt a new metric called proportionality variation (PV), introduced in [17], as follows.

$$PV = \frac{1}{N} \cdot \sum_{\forall cont} |Ideal - Actual| \tag{2}$$

Here, *cont* is containers, N is the number of containers, *Ideal* is the ideal I/O proportionality, and *Actual* is the actual I/O proportionality obtained from experiments. The lower the value is, the closer the proportionality is to the ideal.

B. Page Allocation

Fig. 8a shows normalized I/O proportionality of eight containers in the write-intensive `Fileserver` workload. As shown in the x-axis of Fig. 8a, we set 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. Ideal represents the ideal I/O proportionality that the containers expect to achieve. From weights 100 to 800, I/O proportionality of `Justitia` 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, `Justitia` shows only 1.74 lower than the ideal case whereas the conventional scheme exhibits 4.69 lower than the ideal case. `Justitia` can achieve better I/O proportionality than the conventional one because it prioritizes higher-weighted containers in the lock acquisition process during page allocation. As a result, higher-weighted con-

tainers can quickly finish their write operations, thereby showing higher I/O bandwidth. In terms of proportionality variation, `Justitia` shows around 36.9% better than the conventional scheme.

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 the page allocation scheme of `Justitia`. Like the previous experiments, we ran `Fileserver` benchmarks with differently weighted containers. As shown in Fig. 8b, the experimental result shows that the total I/O bandwidth decreases only by 3.9% at most when `Justitia` is applied.

C. Page Reclamation

To evaluate the weight-aware page reclamation of `Justitia`, we performed the same `Re-read` experiments as the motivational experiments presented in Section II.B.2). We executed four containers with I/O weight 100, 200, 400, and 800, respectively. In our experimental results, all the performance values are normalized to the result of the lowest-weighted one.

As shown in Fig. 9a, the conventional scheme exhibits poor I/O proportionality because the conventional page reclamation scheme does not consider I/O weight. 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 proportionality variation of 1.4. On the other hand, `Justitia` shows proportionality variation of 0.33, since `Justitia` balances the number of allocated pages of the containers according to their I/O weights, by keeping pages of higher-weighted containers longer in the page cache. This result is even superior to that of direct I/O, in that direct IO shows proportionality variation of 0.61.

D. Overhead of Justitia

To examine the overhead induced by `Justitia`, we measured the hit ratio, while executing the `Webserver` workload in four containers with different I/O weights. 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 weight. As shown in Fig. 10a, the overall hit ratio drops about 0.8% on average, denoting that `Justitia` keeps a satisfactory hit ratio even with deprioritizing pages of lower-weighted containers. This result comes from the fact that `Justitia` reflects I/O weight only in the inactive list.

Fig. 10b shows the write bandwidth of `Fileserver` and the read bandwidth of `Re-read` workloads. They are performed in the same configuration as Fig. 8b and Fig. 9a except that all the containers have the same I/O weight in this experiment. Compared with the conventional scheme, `Justitia` exhibits 2.7% and 3.7% drop in the total bandwidth on `Fileserver` and `Re-read` workloads, respectively. In addition, the average latency of executing `Fileserver` and `Re-read` workload increases by 1.5ms and 0.6ms on average in `Justitia`, respectively. These overheads of `Justitia` originate from searching for a container with higher weight during the page allocation and repeatedly keeping track of the number of allocated pages per application for page reclamation. However, considering the satisfactory results of I/O proportionality, we believe `Justitia` is very practical to help improve I/O proportionality with imperceptible overheads.

Note that when all the containers have identical I/O weight, `Justitia` operates in the same way as the conventional system in that it attempts to equally distribute resources to the containers.

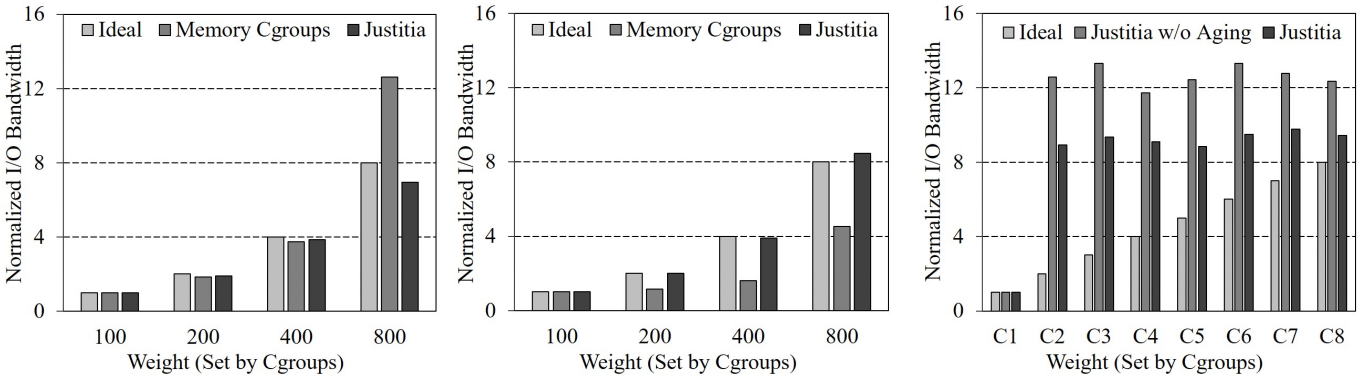
E. Comparison with Memory Cgroups

One might think of the combination of memory and blkio of Cgroups as an alternative to `Justitia`. The memory subsystem of Cgroups provides the ability to control the maximum memory usage of each application and distinguishes between file-backed and anonymous pages when calculating the aggregate memory usage. However, when the memory usage of a resource group exceeds its memory limit, memory Cgroups reclaims not only its file-backed pages but also anonymous pages. Therefore, it cannot solely limit the amount of file-backed pages which affects the I/O performance. Consequently, I/O proportionality still cannot be guaranteed by memory Cgroups.

To confirm this, we run both of `Fileserver` and `Re-read` benchmarks and compared the experimental results of memory Cgroups with those of our scheme. Here, in the case of memory Cgroups, we set the same ratio of memory limit (1 : 2 : 4 : 8) as that of I/O weights of `Justitia`. In the `Fileserver` experiment of Fig. 11a, memory Cgroups shows poor I/O proportionality, showing PV of 4.25, while PV of `Justitia` is 0.33. Especially, with memory Cgroups, 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. 11b of the `Re-read` workload experiment, from weights 100 to 800, I/O proportionality of `Justitia` shows 1 : 2.02 : 3.91 : 8.46, whereas that of memory Cgroups shows 1 : 1.15 : 1.60 : 4.53. PVs of `Justitia` and memory Cgroups are 0.14 and 1.67, respectively. The results come from the fact that memory Cgroups cannot solely control limit file-backed pages (page cache), while excluding anonymous pages.

F. Justitia with Aging Technique

To prevent the starvation problem, we added the aging technique to our weight-aware *qspinlock* scheme. To verify that our scheme is robust in extreme cases, we performed `Fileserver` experiment with eight containers. Here, I/O weight of one container (C1) is 100 and the others (C2 – C8) are 1000. As shown in Fig. 12a, I/O proportionality of `Justitia` 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, `Justitia` without aging technique shows 1 : 12.57 : 13.31 : 11.72 : 12.443



(a) I/O Proportionality on Fileserver (b) I/O Proportionality on Re-read Case

Fig. 11: Comparison with Memory Cgroups

(a) I/O Proportionality on Re-read Case

Fig. 12: Eval. for Aging Scheme

: 13.31 : 12.77 : 13.35 in I/O proportionality with PV of 2.31. With `Justitia` 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 `Justitia` 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 `Justitia` without aging technique. Therefore, even though there are multiple higher-weighted containers, our scheme is still able to guarantee the performance of the lower-weighted container according to its I/O weight.

V. Related Work

J. Kim *et al.* [17] and S. Ahn *et al.* [18] proposed new schemes for precisely controlling I/O resources based on Cgroups. Specifically, J. Kim *et al.* suggested SSD-aware I/O schedulers that improve I/O proportionality in virtualized systems. Similarly, S. Ahn *et al.* proposed an alternative to the conventional I/O subsystem of Cgroups to support NUMA-aware scalable I/O sharing by throttling I/Os above the block layer. However, such schemes still do not consider the existence of the page cache layer in the I/O path, even though the page cache can significantly contribute to the improvement of I/O performance. `Justitia`, on the other hand, considers the page cache as a part of I/O stack and substantially improves I/O proportionality even with buffered I/O.

Cgroups v2 [19], which is an improved version of Cgroups, provides features to control writeback by setting a certain amount of dirty page ratio. However, Cgroups 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, Cgroups v2 cannot solve problems incurred by the locking mechanism in page allocation. Due to the dependency issue, Cgroups v2 is currently unavailable in Docker containers yet [20], [21].

P. Sharma *et al.* [22] proposed a per-VM page cache partitioning scheme. The main contribution of the paper is to increase hit ratio with a small size of memory by isolating VMs in the page cache layer. Most recently, S. Kashyap *et al.* [23] has proposed the shuffling mechanism that reorders 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 `Justitia` is to achieve application-level I/O proportionality by giving weight-awareness to the page cache without incurring long-term unfairness and the starvation problem.

K. Oh *et al.* [24] suggested a weight-aware page cache management that tries to improve I/O proportionality of containers when using the page cache. However, it does not consider the page allocation process and thus cannot provide better I/O proportionality in the case of write-intensive workloads. Moreover, since the scheme uses only I/O weight without clarifying the ownership of pages, it is not applicable when multiple containers have the same weight. On the other hand, `Justitia` prioritizes higher-weighted containers both in page allocation and reclamation stage, and also can be used even when multiple containers/applications have the same weight value because it distinguishes them in the page cache. Finally, [24] simply gives more opportunities to higher-weighted containers in the active list during page reclamation. However, `Justitia` precisely calculates the amount of page cache entries that each container should have, and utilizes them in page reclamation.

VI. Conclusion

The page cache plays a crucial role in providing high I/O performance. However, the conventional Linux page cache management does not reflect I/O weight of Cgroups in both page allocation and reclamation, thereby degrading application-level I/O proportionality. We presented a new page cache management scheme, called `Justitia`, that (1) prioritizes higher-weighted applications during page allocation in the page cache and (2) balances the number of allocated pages of each application in the page cache according to its I/O weight during page reclama-

tion. Experimental results with both synthetic and real-world benchmarks prove that `Justitia` effectively enhance application-level I/O proportionality without noticeable performance degradation. The source code of `Justitia` is available at github.com/kzeoh/Justitia.git.

VII. Acknowledgements

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2017R1A2B3004660).

References

[1] J. Moura and D. Hutchison, "Review and analysis of networking challenges in cloud computing," *Journal of Network and Computer Applications*, vol. 60, no. C, pp. 113–129, Jan. 2016.

[2] (2014) Control groups resource management. [Online]. Available: https://libvirt.org/cgroups.html

[3] (2018) Cgroups. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[4] 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. of 55th Annual Design Automation Conference (DAC '18)*, 2018, pp. 1–6.

[5] J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Docker container scheduler for I/O intensive applications running on NVMe SSDs," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 313–326, Feb. 2018.

[6] (2014) MCS locks and qspinlocks. [Online]. Available: https://lwn.net/Articles/590243

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

[8] (2019) Docker overview. [Online]. Available: https://docs.docker.com/engine/docker-overview

[9] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt, "Improving docker registry design based on production workload analysis," in *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST '18)*, 2018, pp. 265–278.

[10] (2017) CFQ (complete fairness queueing). [Online]. Available: https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt

[11] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O path: A holistic approach for application performance," in *Proc. of 15th USENIX Conference on File and Storage Technologies (FAST '17)*, 2017, pp. 345–358.

[12] Y. Wang and A. Merchant, "Proportional-share scheduling for distributed storage systems," in *Proc. of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, 2007, pp. 47–60.

[13] 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. of 2018 USENIX Annual Technical Conference (ATC '18)*, 2018, pp. 15–27.

[14] (2007) Lockstat: Documentation. [Online]. Available: https://lwn.net/Articles/252835

[15] D. Zheng, R. Burns, and A. S. Szalay, "A parallel page cache: Iops and caching for multicore systems," in *Proc. of 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '12)*, 2012, pp. 1–6.

[16] J. Park, C. Min, and H. Yeom, "A new file system I/O mode for efficient user-level caching," in *Proc. of 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '17)*, 2017, pp. 649–658.

[17] J. Kim, E. Lee, and S. H. Noh, "I/O scheduling schemes for better I/O proportionality on flash-based SSDs," in *Proc. of 24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '16)*, 2016, pp. 221–230.

[18] S. Ahn, K. La, and J. Kim, "Improving I/O resource sharing of Linux cgroup for NVMe SSDs on multi-core systems," in *Proc. of 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)*, 2016, pp. 111–115.

[19] (2015) Cgroups v2. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[20] (2016) Support cgroup v2 (unified hierarchy). [Online]. Available: https://github.com/opencontainers/runc/issues/654

[21] (2019) The future of docker containers. [Online]. Available: https://lwn.net/Articles/788282/

[22] P. Sharma, P. Kulkarni, and P. Shenoy, "Per-vm page cache partitioning for cloud computing platforms," in *Proc. of 8th International Conference on Communication Systems and Networks (COMSNETS '16)*, 2016, pp. 1–8.

[23] S. Kashyap, I. Calciu, X.Cheng, C. Min, and T. Kim, "Scalable and practical locking with shuffling," in *Proc. of 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, 2019, pp. 586–599.

[24] K. Oh, J. Park, and Y. I. Eom, "Weight-based page cache management scheme for enhancing I/O proportionality of cgroups," in *Proc. of 2019 IEEE International Conference on Consumer Electronics (ICCE '19)*, 2019, pp. 1–3.