Front.Comput.Sci.
DOI

RESEARCH ARTICLE

Precise Control of Page Cache for Containers

Kun WANG, Song WU (✉), Shengbang LI, Zhuo HUANG, Hao FAN, Chen YU, Hai JIN

National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China

Abstract Container-based virtualization is becoming increasingly popular in cloud computing due to its efficiency and flexibility. Resource isolation is a fundamental property of containers. Existing works have indicated weak resource isolation could cause significant performance degradation for containerized applications and enhanced resource isolation. However, current studies have almost not discussed the isolation problems of page cache which is a key resource for containers. Containers leverage memory cgroup to control page cache usage. Unfortunately, existing policy introduces two major problems in a container-based environment. First, containers can utilize more memory than limited by their cgroup, effectively breaking memory isolation. Second, the OS kernel has to evict page cache to make space for newly-arrived memory requests, slowing down containerized applications. This paper performs an empirical study of these problems and demonstrates the performance impacts on containerized applications. Then we propose *pCache* (*precise control of page cache*) to address the problems by dividing page cache into private and shared and controlling both kinds of page cache separately and precisely. To do so, *pCache* leverages two new technologies: *fair account* (f-account) and *evict on demand* (EoD). F-account splits the shared page cache charging based on per-container share to prevent containers from using memory for free, enhancing memory isolation. And EoD reduces unnecessary page cache evictions to avoid the performance impacts. The evaluation results demonstrate that our system can effectively enhance memory isolation for containers and achieve substantial performance improvement over the original page cache management policy.

Keywords page cache, memory cgroup, container isolation, cloud computing

Received month dd, yyyy; accepted month dd, yyyy

E-mail: *wusong@hust.edu.cn*

1 Introduction

An important recent trend in cloud computing is the rise of container-based virtualization, such as Docker [1], because of its high performance, low footprint, simplicity of design, and natural support for the micro-service [2, 3] and serverless computing [4, 5]. Because containers share the underlying operating system (OS) kernel, it is critical the kernel provides isolation for containers. Typically, containers attain isolation through various kernel isolation primitives, such as control groups (cgroups) and namespaces [6] in the Linux kernel.

Resource isolation, which is realized by allocating, metering and enforcing resource usage, is a fundamental property of containers. Recent studies have indicated weak resource isolation could cause significant performance degradation for containerized applications and aimed to enhance resource isolation [7–10]. For example, Khalid *et al.* [9] demonstrated containers can use more CPU than their respective cgroups allocate when sending or receiving network traffic, causing co-located containers to suffer an almost 6× slowdown. To address this problem, the authors propose to allocate container CPU resources in a fine-grained manner by monitoring, charging and enforcing CPU usage for processing network traffic. However, existing works have almost not discussed the isolation problems of page cache which is a key resource for containers. Page cache is a memory cache of files on the disk used to accelerate access to files, and it is

charged to the memory limit of containers.

In this paper, we show containers suffer from two major problems when working with page cache. First, containers can utilize more memory than limited by their respective cgroup, effectively breaking memory isolation. Page cache is charged on the basis of the first touch approach, which refers to that memory cgroup charges a page to the container that first brings the page into physical memory. And a page is uncharged from the container only if the page is fully unmapped with the physical memory. This accounting approach introduces unfairness between containers, which means that only the first container pays for the page cache and non-first containers can use the page cache for free. Even worse, one container can repeatedly accumulate the "free page cache", exhausting system memory and breaking memory isolation. Second, the operating system (OS) kernel has to evict page cache to make space for newly-arrived memory requests, significantly slowing down containers. Because page cache is a part of memory limit, when one container reaches its memory limit, the OS kernel evicts page cache to avoid OOM (out of memory). As a result, the containers sharing the evicted page cache would suffer from significant performance impacts, including latency of memory requests and page cache miss. A typical scenario where these two problems often occur in a real cloud system is that different containers access many same library files and share plenty of page cache.

We perform an empirical study to demonstrate that these problems can cause performance degradation for containerized applications. Experimental results show that one container can break the memory limit and use page cache for free, exhausting system memory and starving other containers. In addition, the page cache eviction can slow down memory requests of containers by 65.7%. Compared to the case of running one container, the page cache miss rates of running 4 and 8 concurrent containers fluctuate more sharply. The average cache miss rate increases to 8.8× and 10.2×, respectively. These results clearly show that the current OS kernel cannot provide precise management of page cache for containers, which causes isolation issues and significant performance degradation, suggesting the necessity of optimizing page cache management for containers.

We design and implement a system named *pCache* (*precise control of page cache*) to address the problems. The key idea is to divide the page cache of containers into private and shared, then control both kinds of page cache separately and precisely. To this end, we propose *fair account* (f-account) and *evict on demand* (EoD). F-account splits the shared page cache charging for containers based on their shares to guarantee fairness, and it evicts the private page cache when containers exit to prevent containers from using memory for free. However, splitting page cache charging will introduce possible information leakage and security problems. For instance, attackers can conduct a side-channel attack by inferring the memory usage of other containers from the change of their charged share. To address this challenge, we charge the per-container share lazily and randomly, preventing attackers from inferring the real-time memory usage. In addition, we observe that only uncharging page cache is necessary to release space for containers' new memory requests. And evicting page cache, which causes performance impacts, is unnecessary if the system memory is sufficient. Inspired by that, EoD is proposed to evict page cache on demand to reduce unnecessary evictions.

In summary, our major contributions are as follows:

- First, we perform an empirical study to show two problems: 1) the current page cache charging policy introduces unfairness and breaks memory isolation, and 2) page cache eviction slows down containers. Then we reveal the reasons for these problems and the impacts on containerized applications.
- Next, we propose *pCache* to address these problems by dividing page cache into private and shared for containers and controlling both kinds of page cache separately and precisely. To account for page cache fairly and prevent containers from using memory for free, *pCache* splits the shared page cache charging based on the shares of containers and evicts private page cache when containers exit. In addition, *pCache* reduces unnecessary page cache eviction to avoid performance impacts.
- Last, we implement and evaluate *pCache* in Linux kernel 5.4.2. Experimental results show that *pCache* can account for page cache fairly, enhance memory isolation effectively, and achieve substantial performance improvement over the original page cache management policy for various applications.

The rest of this paper is organized as follows: Section 2 introduces the background of this work. Section 3 analyzes the problems and their impacts. Section 4 presents the design and implementation of *pCache*. Section 5 discusses evaluation results and Section 6 reviews the related work. Section 7 concludes this paper.

2 Background

Containers, popularized by Docker [1], are quickly adopted in cloud computing. A container is essentially a group of processes running on the host, where multiple containers share the same underlying host OS kernel. Docker containers mainly rely on three kernel primitives: seccomp, namespaces and cgroups, to realize isolation. As Docker is the most popular container system, this research describes these mechanisms in the context of Docker and Linux. Other container systems including LXC [11] and Rkt [12] adopt similar technologies.

Seccomp checks system call to protect the OS kernel against untrusted containers [13]. It allows containers to specify which system calls they can invoke and which argument values such system calls can use. Namespaces allow containers to have their own isolated views of the host system. A namespace wraps a global system resource in an abstraction that makes it appear to processes within the corresponding container. Changes to the global system resource are visible to processes that are members of the namespace. Cgroups enable resource isolation for containers. Processes belonging to the same container are assigned to one control group, which has the ability to account for and enforce resource consumption of this container. Such that guarantees and fair shares are preserved for other containers.

Page cache is a critical component of the current operating system (e.g., Linux), and it is usually used to accelerate access to files on disk [14]. When first reading or writing data on disk, the OS kernel stores the data in memory, which acts as a cache. This cache is shared among all processes, so any process can reuse it without accessing the disk. Specifically, when the OS kernel initiates a read request (for example, a process invokes a `read` system call), it will first check whether the requested data is cached in memory. If yes (cache hit), the data is read directly from page cache without accessing the disk. If there is no requested data in page cache (cache miss), the OS kernel will first read the requested data from the disk and then caches the data into page cache so that later requests can hit the cache. When the OS kernel initiates a write request (for example, a process invokes a `write` system call), it writes data directly to page cache and marks the written page as dirty. Then, a kernel thread periodically and asynchronously writes dirty pages back to the disk to ensure the consistency between page cache and disk. However, when system memory is insufficient, page cache may be evicted and written back synchronously, which is based on a least recently used (LRU) policy [15].

Containers work with page cache. As efficiently accessing data on disk is critical for the performance of containers, containers normally use page cache to accelerate the accessing. Containers adopt memory cgroup to account for and limit consumption of page cache. Firstly, memory cgroup accounts for page cache as the memory consumption for containers, such that the allocation of page cache can be limited by memory cgroup. However, page cache is shared among containers and accounted for on the basis of the first touch approach. Specifically, memory cgroup charges a page to the container that first brings the page into physical memory. And the page is uncharged from the container only if the page is fully un-mapped with the physical memory. In other words, the page will not be uncharged, even if the container has not actively used it. As a result, this accounting approach may introduce unfairness and break the memory isolation between containers (the details are described as Problem 1 in Section 3).

The first touch approach is only a page cache charging policy of memory cgroup. It can account for the page cache consumption of one container and then charge such consumption to the container's memory usage. In fact, the first touch approach can not help accelerate the access speed of containers. So, we can determine that the first touch approach introduces unfairness in different application scenarios. As reading data from page cache is much faster than disk, Sharing page cache between containers can greatly accelerate the access speed of containers. For each container, if they only read the data they need from the disk, it will solve the unfairness of containers. However, such an approach will make page cache useless, decelerating the access speed of containers significantly. So, it is necessary to analyze and address the unfairness problem.

Secondly, when a container reaches its memory limit, which is configured by memory cgroup at startup, the current policy either tries to evict page cache to make space for new memory requests or invokes an OOM (out of memory) routine to kill the bulkiest process in the container. This page cache eviction is synchronous. Thus it delays the memory requests. In addition, Due to this eviction, the containers that are sharing this page cache suffer from cache miss and have to access data from the disk, resulting in a significant impact on the performance of the containerized applications (the details are described as Problem 2 in Section 3). This motivates us to analyze the problems and their impacts on containerized applications.

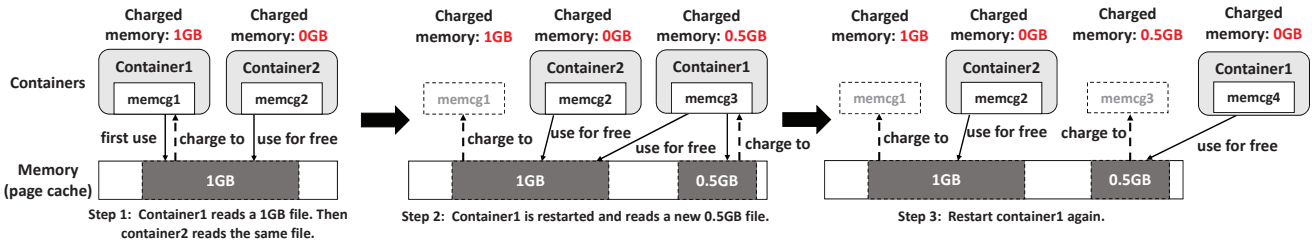


Fig. 1 An example of containers using memory for free. When container 1 is restarted, the kernel creates a new memory cgroup for the container which is charged with zero memory.

3 Analysis of Problems and Impacts

Here we analyze the problems of containers working with page cache and demonstrate the impacts the containerized applications.

Problem 1: *Containers can utilize more memory than limited by their respective cgroup, effectively breaking memory isolation.* Page cache is charged to containers based on the first touch approach and uncharged only if the physical memory is released. There are two disadvantages of this accounting policy. First, it introduces unfairness between containers. The first container that brings page cache into physical memory undertakes all charges of the shared page cache. And the non-first containers are not charged at all and use page cache for free. This is obviously unfair. Second, the current policy cannot provide hardened memory limitation for containers. Because one container can use page cache without charging, it is charged less than its actual usage, breaking the memory limitation. As a result, one container can exhaust system memory and starve other containers by repeatedly accumulating the "free page cache". This problem may cause performance degradation for the cloud platform.

An example of this problem is shown in Fig. 1. The workflow consists of three steps. In step 1, container 1 first traverses a 1GB file. Now, the file is cached in memory (page cache), and this page cache is charged to memcg1 (memory cgroup1), which is created for container 1. Then container 2 reads the same file from the page cache without accessing the disk. But the OS kernel does not charge to container 2. Therefore, container 1, whose charged memory is 1GB, is suffering from unfairness caused by the accounting policy based on the first touch approach. In step 2, container 1 is restarted and reads a new 0.5GB file. The kernel creates a new memory cgroup (memcg3) for container 1. At this time, the page cache of the 1GB file is still charged to memcg1, and the charged memory of container 1 changes to 0.5GB, which is the newly-read file. In this situation, container 1 uses the page cache of the 1GB file for free. In step 3, we restart container 1 again. Similarly, container 1 can use the page cache of the 0.5GB file for free. As a result, we can exhaust the system memory by repeating step 2 and step 3, even though container 1 has a memory limit. The root cause of this issue is that page cache is uncharged from memory cgroup only if the physical memory is released.

Problem 2: *OS kernel has to evict page cache for newly-arrived memory requests, significantly slowing down containers.* Because page cache is counted to the memory limit for containers, when a container reaches its memory limit, new memory requests from the container will cause the eviction of page cache. This eviction delays the new memory requests and leads to a high cache miss rate for containers that are accessing the shared page cache, severely degrading the performance of the containerized applications.

Even worse, the page cache is constantly buffered into or evicted from memory by containers, resulting in performance fluctuations for containers. Fig. 2 shows an example. First, container 1 reads a file that is not buffered in page cache. Now, page cache miss happens, and the file is buffered into memory by container 1. Thus, container 2 hits the page cache when reading the same file later. Then if container 1 reaches its memory limit, the OS kernel will evict the page cache to make space for new memory requests. At this time, container 2 re-accesses the file, suffering from a page cache miss. Similarly, container 2 then brings the file into page cache, and the accessing from container 1 hits the page cache. However, container 2 may also reach its memory limit and evict the page cache, which will cause cache miss too. As a result, the two containers constantly buffer or evict the page cache due to memory limit, causing increases and fluctuations of cache miss rate.

We further demonstrated the impact of this problem on the performance of containerized applications through a set of experiments. We ran containers on a Linux host, which was

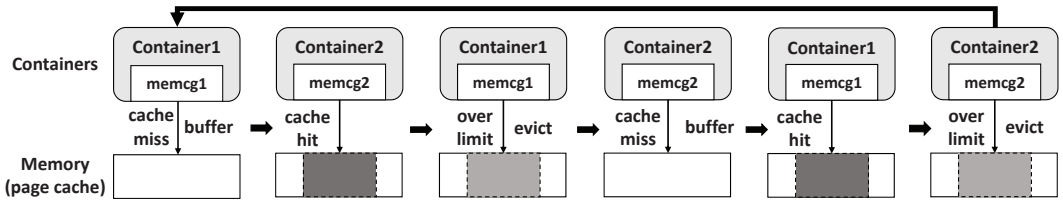


Fig. 2 An example of containers suffering from cache miss problem. If container 1 reaches its memory limit, the kernel evicts the page cache to make space for new memory requests, causing page cache miss for container 2.

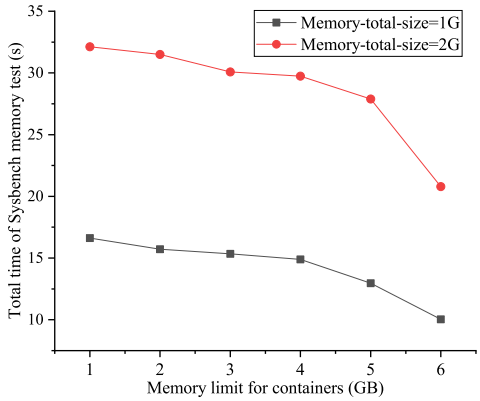


Fig. 3 The performance impact of page cache eviction. When reaching memory limit, containers must wait for kernel to evict page cache to request new memory, significantly decreasing the performance.

equipped with 20 CPU cores, 128GB memory and 2TB HDD storage. To demonstrate the performance impact on memory requests, we ran two containers. In each container, we ran a Webserver workload using Filebench benchmark [16]. And the Webserver workload was set with the default configuration of Filebench. In this case, containers would share the page cache, and only one container paid for the page cache. Then we used Sysbench benchmark to measure the performance of memory allocation for the payer. To demonstrate the cache miss problem, we ran Webserver into various numbers of containers on the same host. Moreover, we used a tool named cachestat that is included in perf-tools to test the page cache miss rate of the whole system.

Fig. 3 shows the performance of the memory test from Sysbench with various memory limits. We set the total size of allocated memory (denoted as memory-total-size in Fig. 3) to 1GB and 2GB, respectively. In the case of 1GB memory-total-size, the memory test is completed faster as the memory limit increases. For instance, when the memory limit is set to 6GB, the completed time is 10.03s. But when the memory limit is set to 1GB, the completed time is 16.62s, which is 65.7% slower than that of the 6GB memory limit. This is because the shared page cache occupies the memory limit, and the memory test has to evict the page cache before requesting memory, decreasing the performance. In addition, the available memory becomes less with the decrease of the memory limit, so the kernel needs to evict more page cache, leading to more performance degradation. Results of 2GB memory-total-size are much similar to the above.

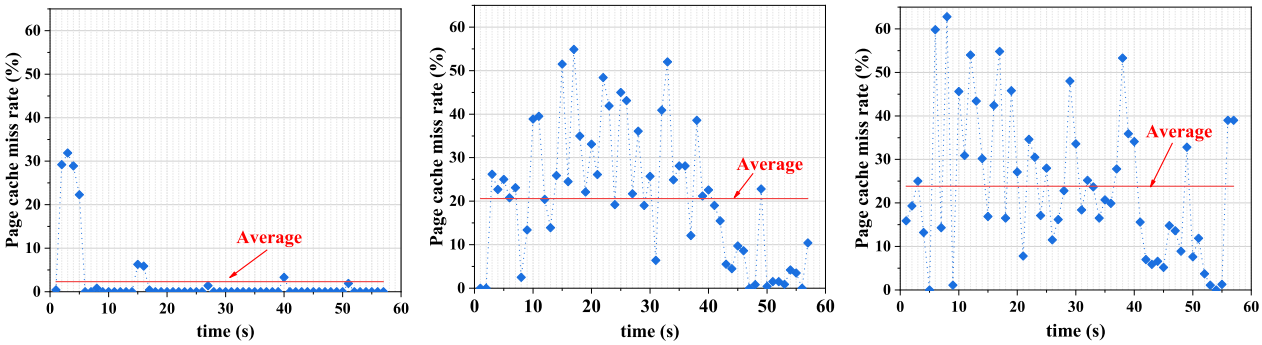
For the cache miss problem, we compared the cache miss rate of three cases: (1) run 1 Webserver container, (2) co-run 4 Webserver containers, and (3) co-run 8 Webserver containers. As shown in Fig. 4(a), in the case of 1 container, at the beginning of the container (first 5 seconds), cache miss happens due to the first access to the files. And in the subsequent process, the cache miss rate is almost 0. The average page cache miss rate is 2.33%. This suggests that page cache works well for one container. However, As shown in Fig. 4(b) and Fig. 4(c), for 4 and 8 concurrent containers, in the whole process of containers running, the cache miss rate increases and fluctuates sharply. The average page cache miss rate is 20.58% and 23.84%, which are 8.8× and 10.2× against the result of 1 container case, respectively. These results demonstrate the significance of addressing the cache miss problem.

4 *pCache* Design & Implementation

In this section, we present *pCache*, a system for optimizing page cache management for containers. The key idea of *pCache* is to leverage two new technologies: f-account and EoD, to provide precise control of page cache for containers. Specifically, f-account accounts for the shared page cache fairly and prevents containers from using memory for free to address problem 1. EoD is used to address problem 2 by reducing unnecessary evictions. Hereafter, we first summarize the overview of *pCache* and then describe f-account and EoD in detail.

4.1 *pCache* Overview

pCache is designed with the following goals:



(a) 1 container. (b) 4 concurrent containers (c) 8 concurrent containers

Fig. 4 When sharing page cache among containers, page cache miss rate increases and fluctuates as the number of concurrent containers increases.

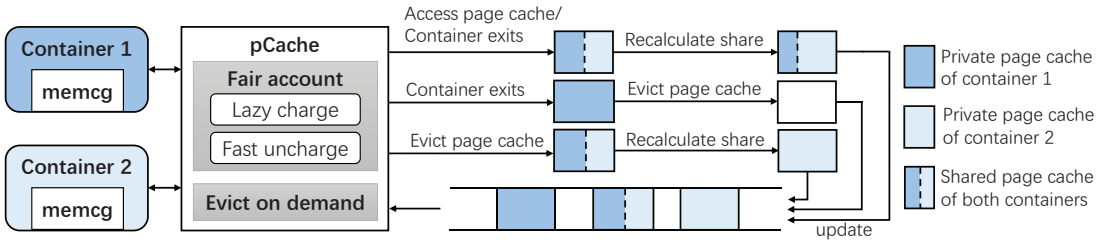


Fig. 5 *pCache* architecture showing f-account in charge of preventing containers from using memory for free and EoD used to reduce unnecessary page cache evictions.

- **Prevent containers from using memory for free:** Page cache is charged to the first container that brings it into physical memory and uncharged only if the physical memory is released, thus causing containers can utilize more memory than limited by their respective cgroup, effectively breaking memory isolation. *pCache* aims to prevent containers from using memory for free by fairly accounting page cache.
- **Avoid performance impacts of page cache eviction:** While containers reach the memory limit, the kernel will evict page cache to make space for new memory requests, resulting in latency of memory requests, as well as increases and fluctuations of page cache miss rate. *pCache* tries to avoid these performance impacts by reducing unnecessary evictions.

To achieve the above goals, a straightforward idea is to refer to the management mechanism of CPU cache. For the sharing mechanism, the most popular method is using Intel Cache Allocation Technology (CAT) to control CPU cache allocation [17, 18]. CAT is a hardware technology and implements way partition for L3 cache [19]. Referring to it, *pCache* controls page cache charging based on partition. Because CAT is designed based on cache way partition and cannot be used for page cache charging, we implement *pCache* in software and build it as a Linux kernel component. For dirty data eviction, there are many replacement algorithms, such as LRU [20]. Our system uses similar algorithms to evict dirty data. Before such algorithms are executed, *pCache* will check whether the page cache eviction is necessary to reduce unnecessary eviction, improving performance. To avoid disruptive changes to the kernel, the design of *pCache* is simple and straightforward.

As shown in Fig. 5, *pCache* divides the page cache of containers into private and shared, then controls both kinds of page cache separately and precisely. Specifically, *pCache* is made up of two components: f-account (fair account) and EoD (evict on demand). In order to address problem 1, f-account splits the charging of shared page cache to containers that are sharing the page cache, based on the per-container share. However, with the splitting policy, containers can infer memory usage from the change of their accounted share, leading to security problems, such as side-channel attacks [21]. To address this challenge, f-account updates the per-container charge randomly and lazily (denoted as "Lazy charge" in Fig. 5) to reduce the leakage of real-time information. For private page cache, when containers exit, f-account evicts page cache (denoted as "Fast uncharge" in Fig. 5 to prevent containers from using page cache for free. In addition, we observe that only uncharging page cache is necessary to make space for containers'

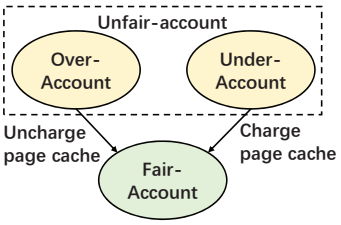


Fig. 6 Container state transition. *Over-Account*, *Under-Account* and *Fair-Account* refer to that the charged page cache is greater than, less than and equal to the calculated share, respectively.

newly-arrived memory requests, and evicting, which causes the performance impacts, is needless. Based on that, EoD is designed to address problem 2 by uncharging page cache from one container when reaching its memory limit but lazily evicting page cache based on the memory pressure of the whole system. As a result, unnecessary evictions would be avoided, reducing performance impacts.

pCache acts as a middleware between containers and page cache and works as follows: When containers access shared page cache or container exit, f-account counts the total number of accesses and re-calculates the per-container share. If the share of containers is changed, *pCache* charges their share to the corresponding memory cgroups after a period, which is randomly generated for security. And when containers exit, their private page cache will be evicted to prevent them from using memory for free. Moreover, when containers reach the memory limit, the OS kernel uncharges page cache as usual and re-calculates per-container share. The shared page cache will not be evicted and will remain in memory to serve other containers if the system memory is sufficient. Otherwise, *pCache* evicts the page cache with EoD. Other page cache operations except the above are as usual.

4.2 Fair Account

To address problem 1, we introduce f-account to guarantee fairness and isolation between containers. For shared page cache, f-account splits the page cache charging for containers, rather than charging to the first user only. For private page cache, f-account evicts page cache when containers exit. In what follows, we describe f-account in detail.

F-account splits the page cache charging based on the per-container share of the page cache. To calculate the per-container share precisely, f-account records the access count of containers to the shared page cache and recalculates the per-container share when increasing or decreasing the access count. Take the Linux kernel as an example, f-account counts the number of accesses at a page-grained level because the physical page is the basic unit to complete page cache access. After obtaining the per-container share, we define three states for containers: *Over-Account*, *Under-Account* and *Fair-Account*, which refer to that the charged page cache is greater than, less than and equal to the calculated share, respectively. State transition is illustrated as Fig. 6. *Fair-Account* is the ideal state for containers. Therefore, if one container is in the *Over-Account* state, we uncharge the over-charged page cache to transform the container state into *Fair-Account*. Conversely, if one container is in the *Under-Account* state, we charge the miss-accounted page cache. Such that page cache charging is split among containers that are sharing it, guaranteeing the fairness between containers.

Algorithm 1 shows the methods of calculating the per-container share and container state transition. When the current process accesses page cache, we test whether the process is from host or from a container. If the process is from a container and hits the page cache, *pCache* guarantees fairness by finding unfair charging and transforming the container state. First, we update `num_containers` and `total_access_count`. These two variables refer to the number of containers that are sharing the page cache and the total access count of all containers to the page cache. They are used to determine the fair share of each container. Then if the page is not charged to the current memory cgroup, unfairness happens and we we update the number of accesses (`unfair_access_count`) that lead to the unfairness. Using the above three variables, we calculate the number of pages that should be re-charged (`miss_charge_pages`) to guarantee fairness. Last, we transform the container state by uncharging page cache from the original containers and charging the page cache to the current container. In addition, when containers exit, we will recalculate the share, which is similar to the above process. Normally unfairness will not happen if the current process is from host or it does not hit the page cache, so *pCache* charges page cache as usual in this case.

Unfortunately, the above splitting policy causes possible security problems. Once splitting the page cache charging among containers, one container can infer the page cache consumption of others by observing the changes of its charged share. In other words, the splitting policy causes additional information leakage. The attacker can leverage this to conduct a side-channel attack that aims to gather information from a system by measuring indirect effects of the system. For instance, similar to the typical timing attack, the attacker can attempt to extract secrets by analyzing the page

Algorithm 1 Calculate the per-container share and split page cache charging for containers

```
/*total_access_count: the access count of all containers to page cache;
unfair_access_count: the access count leading to unfairness;
num_containers: the number of containers sharing the page cache;
miss_charge_pages: the number of pages missing charge;
nrpages: the number of pages for the total page cache*/
function SPLIT_PAGE_CACHE(nrpages)
    if current process in one container then
        if access file and page cache in memory then
            total_access_count ++
            if current memory cgroup is different from the memory cgroup the page cache charged to then
                unfair_access_count ++
            end if
            re-calculate num_containers
            miss_charge_pages = unfair_access_count / (num_containers+total_access_count) × nrpages
            uncharge miss_charge_pages pages from their original memory cgroups (containers)
            charge miss_charge_pages pages to current memory cgroup (container)
            change the state of the corresponding containers
        end if
    end if
end function
```

cache a system uses to execute cryptographic algorithms. Therefore, the main challenge here is how to split the shared page cache accounting securely.

In order to address the above challenge, we propose to charge the per-container share lazily instead of in real-time. With the lazy charge, containers cannot observe their real-time accounted share, preventing attackers from inferring the real-time page cache consumption. Thus, the possible security problems can be avoided. Now we need to decide when to charge the per-container share. On the one hand, there is a tradeoff between the effectiveness of the splitting policy and the security of the system. If we set the charge period too short, the real-time page cache consumption might be inferred, reducing the system security. But setting a too long period will lead to the invalidation of the splitting policy and cannot guarantee fairness. On the other hand, using a constant period cannot avoid the security problem completely. That is because once attackers obtain the constant period, they can infer how much page cache other containers consume at a certain point in time (at the beginning of the period), leading to possible side-channel attacks.

Inspired by the typical randomization-based defense schemes [22], we charge the per-container share based on period which is generated randomly. As Algorithm 2 shows, first we determine the period (`period_t`) randomly using the existing random function. Then we only charge the per-container share in two cases. The first one is when the per-container share has not been charged for `period_t` time.

Algorithm 2 charge per-container share lazily

```
/*period_t: the charge period*/
function LAZY_CHARGE( )
    period_t = random()
    if one container requests memory then
        charge per-container share
    else if have waited for period_t time then
        charge per-container share
    end if
end function
```

Because the period is generated randomly, attackers can not infer when other containers use the page cache. Although they obtain the page cache usage of other containers, they can not conduct attacks successfully. The second one is when one container requests memory. In order to prevent containers from using memory for free, we must charge the accurate usage and fair share of page cache before one container requests memory. As a result, this random method can guarantee the system security and enhance memory isolation.

In addition, as shown in the example in Fig 1, one container can repeatedly accumulate its private page cache by restarting, such that the container can use memory for free and exhaust system memory. To address this problem, we propose fast uncharge, which evicts the private page cache of one container when it exits. Due to the eviction, containers have to reload the page cache and pay for it when re-accessing it after restarting. Because fast uncharge is designed for pri-

vate page cache, there are no impacts on other containers.

There are two sources of overhead associated with f-account. The first one is the calculation of the per-container share. As the calculation is done along with the page cache accessing routines, it does not need additional synchronization or locking. So, the calculation is simple and lightweight. The second one is charging (uncharging) pages to (from) memory cgroup. This operation may be more expensive because it needs to look up the corresponding pages and cgroups. But this occurs less frequently due to the lazy charge. As a result, we do not anticipate the overheads to affect the overall performance.

4.3 Evict on Demand

When containers reach their memory limit, the kernel will uncharge and evict page cache to make space for new memory requests, resulting in impacts on the performance of containerized applications, including latency of memory requests and page cache miss. In fact, the root cause of there being no available memory for containers is the memory limit, not the system memory pressure. So, only uncharging is necessary, and evicting (the root cause of the cache miss problem) is needless. Based on that, we propose to evict page cache on demand (EoD) and reduce the unnecessary eviction to reduce the performance impacts. Suppose one container reaches its memory limit and its new memory requests are arriving. In that case, EoD first uncharges page cache as usual except labeling the page cache with "uncharged", and then we recalculate the per-container share to ensure fairness. Last, EoD evicts page cache in two cases: 1) no containers share the page cache, and 2) the system memory is insufficient.

As shown in Algorithm 3, we monitor whether the above two cases happen through the number of containers that are sharing the page cache (`num_containers`) and the available memory of the system (`avai_mem`). For the first case, when EoD uncharges page cache and labels the page cache with "uncharged", we check whether there are containers accessing the page cache by checking whether `num_containers` is zero. If no containers access the page cache, evicting the page cache will not hurt the performance of any containers. So at this time, we can evict all uncharged pages of the page cache. For the second case, we use the memory watermark [23] in the kernel to monitor the system memory pressure. Specifically, if the available memory is below the high watermark of memory (`high`), we determine the system memory is insufficient. To avoid the occurrence of "Out of Memory", we must start to evict the uncharged page cache. If the available memory is more than $1.25 \times$ `high`, we determine there is enough memory in the system. To reduce the performance impacts of page cache eviction, we stop it. We set the threshold to $1.25 \times$ `high` by referring to the default settings of the high and low watermarks in the kernel. Obviously, when there is no uncharged page cache in the system, we will stop the eviction too.

Algorithm 3 Evict page cache on demand

```
1: /*num_containers: the number of containers sharing the page cache;
2: avai_mem: the available system memory;
3: high: the high memory watermark configured in the OS kernel*/
4: function EVICT_ON_DEMAND( )
5:     get num_containers and avai_mem
6:     if num_containers == 0 then
7:         evict all uncharged pages of the page cache
8:     else if avai_mem ≤ high then
9:         repeat
10:            evict uncharged pages
11:            avai_mem ++
12:            if avai_mem ≥ high × 125% then
13:                break loop
14:            end if
15:        until there is no uncharged pages
16:    end if
17: end function
```

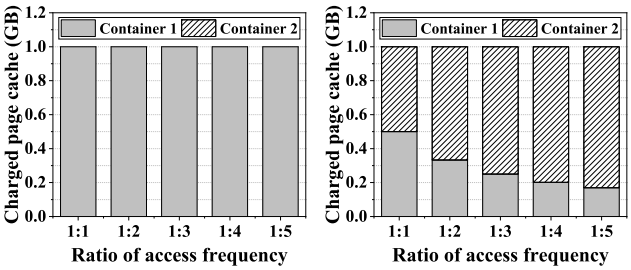
5 Evaluation

In this section, we present the experimental evaluation of *pCache*. Our evaluation answers the following questions:

- Are the f-account and EoD effective for addressing the isolation issues and performance impacts, respectively? (Section 2)
- How about the overall performance improvement of *pCache* against the original page cache design in shared environments? (Section 3)
- What is the performance overhead of our system? (Section 4)

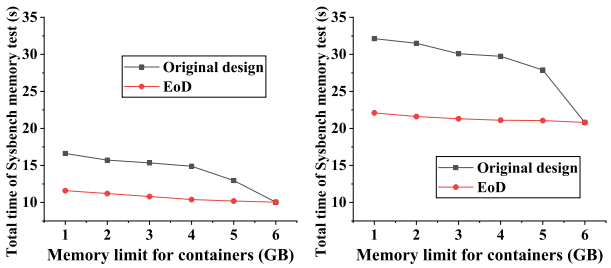
5.1 Experimental Settings

We implemented *pCache* on the Linux 5.4.2 kernel. We used Docker 20.10.5 and Ubuntu 16.04 to deploy containers on a server, which was equipped with a 20-core Intel Xeon E5-2650 CPU, 128GB memory and 2TB HDD storage. Sim-



(a) Original design (b) *pCache*

Fig. 7 Charged page cache of containers under various ratios of page cache access frequency (*pCache* versus original Linux kernel design).



(a) Request 1GB memory (b) Request 2GB memory

Fig. 8 Performance of memory requests for requesting 1GB and 2GB memory under various memory limit for containers.

ilar to the cloud production environment, cgroups, namespaces and seccomp were enabled to isolate containers. Each container was set as non-privileged, configured with limited hardware resources, and pinned to specific cores. The experimental settings and benchmark details of the individual evaluation were slightly different and further discussed in the respective sections.

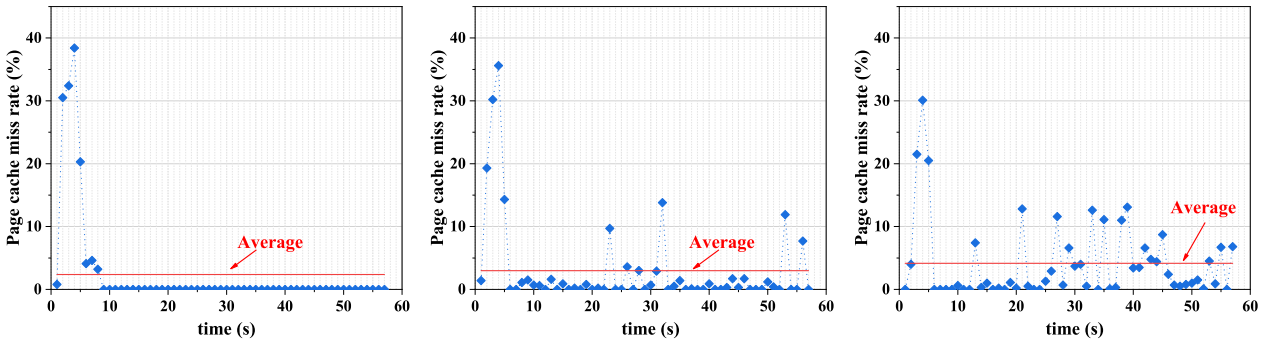
5.2 Effectiveness of f-account and EoD

Effectiveness of f-account. We first demonstrate the effectiveness of f-account on guaranteeing fairness between containers and preventing containers from using memory for free. In this experiment, we ran two containers, which were denoted as container 1 and container 2. Container 1 first read a 1GB file. Then we varied the access frequencies of container 2 to the same file and observed their charged page cache. Results are shown in Fig. 7. The X-axis represents the ratio of access frequency between container 1 and container 2. And the Y-axis represents the amount of charged page cache for two containers. As shown in Fig. 7(a), for the original page cache design, no matter how many times container 2 accesses, all page cache is only charged to container 1, due to the first touch accounting policy. And as shown in Fig. 7(b), our design charges page cache to both containers based on the ratios of access frequency between them because of our splitting policy. These results clearly suggest that f-account can guarantee fair page cache accounting between containers. In addition, we read a new 2GB file in container 1 (private page cache for container 1) and restarted container 1. Then we observed that the page cache of this 2Gb file was evicted, and container 1 could not use it for free. The above results demonstrate that f-account can guarantee fairness between containers and prevent containers from using memory for free effectively.

Effectiveness of EoD. EoD is designed to reduce the performance impacts of page cache eviction, including latency of memory requests and page cache miss. To evaluate its effectiveness, we repeated the experiments described in Section 3, which were used to demonstrate the performance impacts. And we evaluated the performance gain of EoD against the original page cache design. As shown in Fig. 8, we separately measured the performance of requesting 1GB memory and 2GB memory with various memory limits. Compared to the original page cache design, EoD improves the performance of memory requests by 43.1% and 45.5% for requesting 1GB memory and 2GB memory, respectively. And for both cases, the memory requests are slowed down slightly (negligibly) as the memory limit decreases. This is because EoD reduces the synchronous page cache eviction that delays memory requests. As shown in Fig. 9, we measured the page cache miss rate of system with various numbers of containers to demonstrate the effectiveness of EoD. Compared to the results of 1 container (Fig. 9(a)), the page cache miss rate of 4 containers (Fig. 9(b)) and 8 containers (Fig. 9(c)) fluctuate slightly, and the average page cache miss rate increases little (from 2.35% to 2.98% and 4.15%, respectively). Compared to the results of the original page cache design (Fig. 2), our design achieves 6.9× and 5.7× performance improvement for 4 containers and 8 containers, respectively. All the above experimental results clearly indicate that EoD can reduce the unnecessary page cache eviction and improve performance effectively.

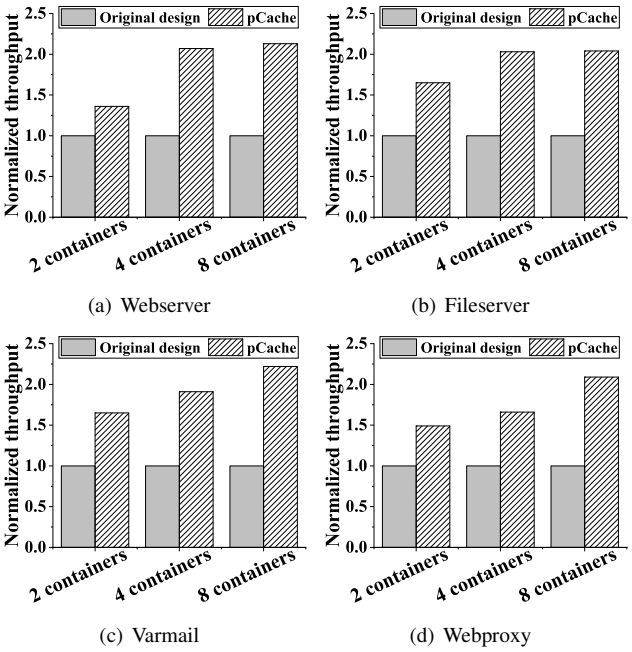
5.3 Overall Performance

To evaluate the effectiveness of *pCache* in more realistic scenarios, we ran Webserver, Fileserver, Varmail and Webproxy in containers using Filebench. All workloads were configured with the default configuration of Filebench. First, we ran same workloads with various number of containers for the 4 workloads separately. Then we mixed the 4 workloads in containers. Fig. 10 and Fig. 11 show the relative throughput



(a) 1 container (b) 4 concurrent containers (c) 8 concurrent containers

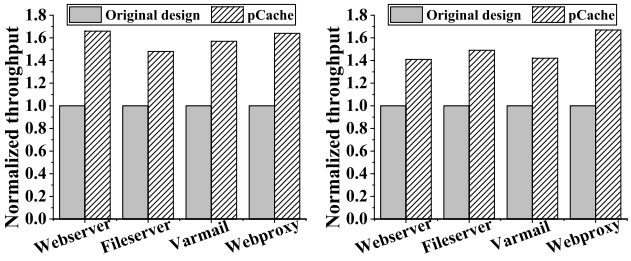
Fig. 9 The system page cache miss rate for *pCache* (lower is better). Compared to original Linux kernel design (Figure 4), our system has lower and stabler page cache miss rate.



(a) Webserver (b) Fileserver

(c) Varmail (d) Webproxy

Fig. 10 Normalized throughput of Webserver, Fileserver, Varmail and Webproxy when running the same workloads in various number of containers (higher is better).



(a) 4 concurrent containers (b) 8 concurrent containers

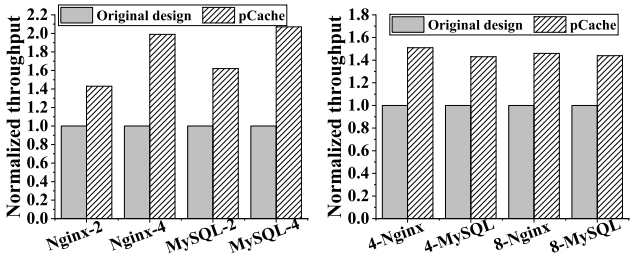
Fig. 11 Normalized throughput of Webserver, Fileserver, Varmail and Webproxy when mixing workloads in various number of containers (higher is better).

of workloads normalized to the original page cache design.

Fig.10 shows the results of running same workloads in containers. For Webserver workload, *pCache* achieves 36%, 107% and 113% higher throughput in comparison to the original page cache design for 2 containers, 4 containers and 8 containers, respectively. The performance improvement is because that *pCache* can enhance memory isolation and reduce performance impacts for containers. Moreover, *pCache* shows higher performance improvement for 4 containers and 8 containers compared to that of 2 containers. The reason is that there are more unnecessary page cache evictions for more containers, and our system can effectively reduce such needless evictions. The results of Fileserver, Varmail and Webproxy are much similar to the above.

Fig. 11 shows the results of mixing the 4 different workloads in containers. When 4 containers are running 4 different workloads, the performance of Webserver, Fileserver, Varmail and Webproxy is improved by 66%, 48%, 57% and 64%, respectively. And when 8 containers are running 4 different workloads (2 containers for each workload), the performance of Webserver, Fileserver, Varmail and Webproxy is improved by 41%, 49%, 42% and 67%, respectively. The performance is a little lower compared to that of running same workloads in containers. This is because there is less shared page cache when running different workloads. But these results still clearly indicate the effectiveness of our design.

To further demonstrate the performance improvement of our system, we conducted some experiments with two real applications: Nginx and MySQL. Similarly, we ran same and different workloads with various number of containers. As shown in Figure 12, the results are much similar to that of Filebench. This clearly suggests *pCache* can address the page cache problems and can be used in realistic scenarios.



(a) running same workloads (b) running different workloads

Fig. 12 Normalized throughput of Nginx and MySQL when running same and different workloads in various number of containers (higher is better). "Nginx-2" denotes running Nginx in 2 containers (same workloads). "4-Nginx" denotes the average performance of Nginx containers when running 2 Nginx containers and 2 MySQL containers together (different workloads).

5.4 Overhead

Lastly, we evaluated the overhead of our system on micro operations, containerized applications and memory footprint, comparing with the original page cache design (Docker). As Kata containers are widely used for isolation, we also compared *pCache* with it to show the significance of our study.

Micro operations. We implemented *pCache* on the path of accessing page cache, uncharging page cache and exiting containers, so we mainly evaluated these operations. We ran an Ubuntu container and used the *time* command to print out how long it takes for executing these operations with 1GB page cache. Fig. 13(a) shows the results. For accessing page cache, our design only adds 0.7% additional time, which is spent on calculating the share of page cache charging. For uncharging page cache, *pCache* introduces 0.6% overhead, which is spent on determining whether eviction is needed. And for exiting the container, *pCache* incurs 1.3% performance overhead, which is spent on evicting its private page cache. Compared to Docker and *pCache*, Kata container takes more time to execute these micro operations. For exiting the container, Kata container introduces 7.34× overhead, which is spent on exiting the per-container kernel. Overall, the performance overhead of *pCache* on micro operations is negligible.

Containerized applications. To evaluate the performance overhead of our system on containerized applications, we ran Webserver, Fileserver, Varmail and Webproxy in a container. As shown in Fig. 13(b), the throughput overhead of Webserver, Fileserver, Varmail and Webproxy is 1.5%, 1.3%, 2.1% and 1.7%, respectively. These overheads are introduced by our precise control of page cache and do not contribute much to the overall performance. Compared to Kata container, the throughput of Webserver, Fileserver, Varmail and Webproxy in *pCache* is 6.58×, 5.92×, 5.99× and 5.6×, respectively. These results show our system introduces much less overhead on containerized applications than existing popular container isolation methods.

Memory footprint. To measure the memory footprint, we ran an Ubuntu container and a Webserver container, then leveraged the *docker stats* command to collect memory usage of both containers. As illustrated in Fig. 13(c), the Ubuntu container and Webserver container consume 1.5% and 0.3% additional memory compared to the original page cache design. The memory overhead of Webserver is much less than Ubuntu because the Webserver container consumes more memory. Compared to Kata container, *pCache* can decrease memory footprints by 4.23× and 4.58× for Ubuntu and Webserver, respectively. Because Kata boot a dedicated kernel for each container, introducing more overhead. Overall, the memory footprint overhead of *pCache* is remarkably negligible.

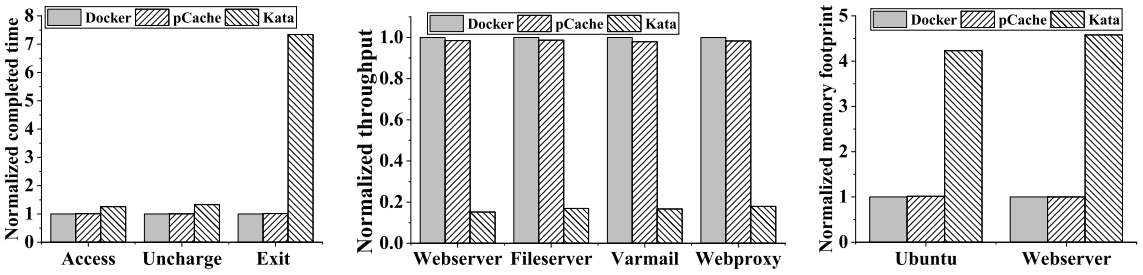
5.5 Limitation and Discussion

Our results show *pCache* can enhance page cache isolation and achieve performance improvement for many applications. Nevertheless, there are potential limitations in this study. Our approach may not work well for workloads that share lots of small files. As the physical page is the basic memory management unit, we split the page cache charging at a page-grained level. In this case, one page can be charged to only one container. This may introduce charging unfairness when containers share lots of small files. As an extreme example, if two containers share 100 files and the size of each file is one page, all the page cache of these files may be charged to the first container and the other container can use the page cache for free. In the future, we plan to design a more fine-grained splitting mechanism to improve the soundness of our study.

6 Related Work

In this section, we review the most relevant work that inspires our work and highlight the differences between our work and previous research. We mainly discuss research works in the following areas.

Analyzing container isolation. There is a large body of work focusing on analyzing the isolation of container-based virtualization [24–32]. These researches demonstrated the



(a) micro operations (b) containerized applications (c) memory footprint

Fig. 13 The performance overhead of *pCache* (versus original Linux kernel) for different micro operations, different containerized applications, and memory footprint.

weak isolation of containers by comparing containers with virtual machines through various workloads. And they show the significance of container isolation in the cloud computing environment. However, none of the above research has fully investigated the risks and impacts of weak page cache isolation. In this work, we demonstrate containers can break memory isolation by using page cache for free and reveal the root cause as well as its impacts.

Enhancing container isolation. There are two ways to enhance container isolation. One is dedicated to optimizing the kernel isolation mechanisms. For view isolation, Huang *et al.* [6] focused on the resources (e.g. CPU and memory) view of containers and designed a new namespace to export the available resources to containers. For security isolation, Sun *et al.* [33] enabled containers to have autonomous control over their security policies with a security namespace. Gao *et al.* [34] explored the possible power attack by exploiting leaked host information and proposed a power-based namespace to defend against the attack. For CPU isolation, Gao *et al.* [35] explored several exploiting strategies to escape the resources control by de-associating processes from their original CPU cgroups. Khalid *et al.* [9] were dedicated to similar problems and optimized cgroup to charge CPU resource for softirq processing. Huang *et al.* [36] proposed efficient oversubscription to isolate thread for containers. For I/O isolation, PINE [10] was another cgroup optimization designed to address the issue of misallocation of storage resource. Wu *et al.* [37] proposed container-aware I/O stack to enhance I/O isolation for containers. IOCost [38] provided scalable, work-conserving and low-overhead IO control for containers to address isolation failures in datacenter environments. For network isolation, Gu *et al.* [39] proposed a container network traffic control framework to provide strong isolation of network resource isolation. However, the above optimizations were used to address specific isolation problems, which did not include the page cache isolation. This work focuses on page cache isolation and proposes *pCache* for addressing the problem by providing each container with precise control of page cache.

The other one trends to wrap containers within a dedicated OS kernel. Kata containers [40], LightVM [41], Mavridis [42], X-Containers [43] and μKontainer [44] all leveraged hypervisor to run containers in a light virtual machine to make containers not share a same kernel, enhancing isolation. Similarly, gVisor [30] ran containers within a user-space kernel which is written in Go language. Although some works [45, 46] have been proposed to improve the performance, compared to native container systems (e.g. Docker), these platforms could cause significant performance overhead due to the individual kernels. Instead, *pCache* optimizes the native containers and incurs negligible performance overhead.

Optimizing memory cgroup and page cache. Several works have been dedicated to analyzing the performance pitfalls of memory cgroup and page cache as well as proposing practical solutions. Huang *et al.* [47] studied the Linux memory management and pointed out that the `page_cgroup` lock is unnecessary, which affected the concurrency of memory cgroup. Kim *et al.* [48] leveraged dynamic memory request throttling and cgroup to reduce memory interference for critical applications. Zhuang *et al.* [49] found four performance issues of memory cgroup but not addressed them. Oh *et al.* [50] proposed a weight-based page cache management scheme that reflects I/O weight of cgroups to improve I/O proportionality. Park *et al.* [51] proposed weight-aware cache to reflect the I/O weights on page cache allocation and reclamation, achieving application-level proportional I/O sharing. Zheng *et al.* [52] focused on addressing the scalability issues within the page cache layer. Finer-LRU [53] optimized the page reclamation process to reduce lock contention on the sharing memory space. Unlike these works, we optimized memory cgroup to charge page cache fairly, isolate

page cache, and reduce performance impacts for containers. And the above studies are orthogonal and complementary to *pCache*.

7 Conclusion

In this paper, we first perform an empirical study of containers working with page cache. Our results reveal two major problems: 1) containers can utilize more memory than limited by their respective cgroup, effectively breaking memory isolation, and 2) OS kernel has to evict page cache for newly-arrived memory requests, significantly slowing down containers. Then we demonstrate the significant performance impacts on containerized applications. Next, we design and implement a system named *pCache* to address these problems for containers by dividing page cache into private and shared and controlling both kinds of page cache separately and precisely. We propose f-account to account page cache fairly and prevent containers from using memory for free. F-account splits the shared page cache charging based on per-container share and evicts the private page cache when containers exit. To address the security challenge caused by the splitting policy, we propose to update per-container charging lazily and randomly. In addition, *pCache* leverages EoD to reduce unnecessary page cache eviction to avoid additional performance overhead. Last, experimental results show that *pCache* can account for page cache fairly, enhance page cache isolation and achieve substantial performance improvement over the original page cache management policy for various applications.

References

1. Merkel D. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014, 239(2):1-5
2. Zeng R, Hou X F, Zhang L, Li C, Zheng W L, Guo M Y. Performance optimization for cloud computing systems in the microservice era: state-of-the-art and research opportunities. Frontiers of Computer Science, 2022, 16(6):1-13
3. Hou X F, Li C, Liu J C, Zhang L, Ren S L, Leng J W, Chen Q, Guo M Y. AlphaR: learning-powered resource management for irregular, dynamic microservice graph. In: Proceeding of IEEE International Parallel and Distributed Processing Symposium. 2021, 797-806
4. Suo K, Zhao Y, Chen W, Rao J. An analysis and empirical study of container networks. In: Proceedings of IEEE Conference on Computer Communications. 2018, 189-197
5. Zhang Y Q, Goiri Í, Chaudhry G I, Fonseca R, Elnikety S, Delimitrou C, Bianchini R. Faster and cheaper serverless computing on harvested resources. In: Proceedings of ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021, 724-739
6. Huang H, Rao J, Wu S, Jin H, Suo K, Wu X F. Adaptive resource views for containers. In: Proceedings of International Symposium on High-Performance Parallel and Distributed Computing. 2019, 243-254
7. Soltesz S, Pötzl H, Fiuczynski M E, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: Proceedings of ACM european conference on computer systems. 2007, 275-287
8. Laadan O, Nieh J. Operating System virtualization: practice and experience. In: proceedings of Annual Haifa Experimental Systems Conference. 2010, 1-12
9. Khalid J, Rozner E, Felter W, Xu C, Rajamani K, Ferreira A, Akella A. Iron: Isolating network-based CPU in container environments. In: Proceedings of USENIX Symposium on Networked Systems Design and Implementation. 2018, 313-328
10. Li Y H Z, Zhang J C, Jiang C F, Wan J, Ren Z J. PINE: Optimizing performance isolation in container environments. IEEE Access, 2019, 7(1): 30410-30422
11. Senthil K S. Practical LXC and LXD: linux containers for virtualization and orchestration. 1st ed. New York: Apress, 2017
12. Xie X L, Wang P, Wang Q. The performance analysis of Docker and rkt based on Kubernetes. In: Proceedings of International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery. 2017, 2137-2141
13. Skarlatos D, Chen Q R, Chen J Y, Xu T Y, Torrellas J. Draco: Architectural and Operating System Support for System Call Security. In: Proceedings of IEEE/ACM International Symposium on Microarchitecture. 2020, 42-57
14. Do H D, Hayot-Sasson V, Da Silva R F, Steele C, Casanova H, Glatard T. Modeling the Linux page cache for accurate simulation of data-intensive applications. In: Proceedings of IEEE International Conference on Cluster Computing. 2021, 398-408
15. Eklov D, Hagersten E. StatStack: Efficient modeling of LRU caches. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software. 2010, 55-65
16. Tarasov V, Zadok E, Shepler S. Filebench: A flexible framework for file system benchmarking. The USENIX Magazine, 2016, 41(1):6–12
17. Xiang Y C, Wang X L, Huang Z H, Wang Z Y, Luo Y W, Wang Z L. DCAPS: Dynamic cache allocation with partial sharing. In: Proceedings of EuroSys Conference. 2018, 1-15
18. Xu M, Thi L, Phan X, Choi H Y, Lee I. vCAT: Dynamic cache management using CAT virtualization. In: Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium. 2017, 211-222
19. Sohal P, Bechtel M, Mancuso R, Yun H, Krieger O. A Closer Look at Intel Resource Director Technology (RDT). In: Proceedings of International Conference on Real-Time Networks and Systems. 2022, 127-139
20. Chaudhuri M. Zero inclusion victim: Isolating core caches from inclu-

sive last-level cache evictions. In: Proceeding of ACM/IEEE International Symposium on Computer Architecture. 2021, 71-84

21. Delimitrou C, Kozyrakis C. Bolt: I know what you did last summer... in the cloud. ACM SIGARCH Computer Architecture News, 2017, 45(1):599-613
22. Volckaert S. Randomization-based Defenses against Data-Oriented Attacks. In: Proceedings of ACM Workshop on Moving Target Defense. 2021, 1-2
23. Love R. Linux Kernel Development. 3rd ed. New York: Pearson Education, 2010
24. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and Linux containers. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software. 2015, 171-172
25. Sharma P, Chaufournier L, Shenoy P, Tay Y. Containers and virtual machines at scale: A comparative study. In: Proceedings of International Middleware Conference. 2016, 1-13
26. Plauth M, Feinbube L, Polze A. A performance survey of lightweight virtualization techniques. In: Proceedings of European Conference on Service-Oriented and Cloud Computing. 2017, 34-48
27. Matthews J N, Hu W, Hapuarachchi M, Deshane T, Dimatos D, Hamilton G, McCabe M, Owens J. Quantifying the performance isolation properties of virtualization systems. In: Proceedings of the workshop on Experimental Computer Science. 2007, 6-14
28. Xavier M G, De Oliveira I C, Rossi F D, Passos R D D, Matteussi K J, Rose C A D. A performance isolation analysis of disk-intensive workloads on container-based clouds. In: Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. 2015, 253-260
29. Yang N Z, Shen W B, Li J K, Yang Y T, Lu K J, Xiao J T, Zhou T Y, Qin C G, Yu W, Ma J F, Ren K. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security. 2021, 764-778
30. Caraza-Harter T, Swift M M. Blending containers and virtual machines: A study of firecracker and gVisor. In: Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2020, 101-113
31. Sartakov V A, Vilanova L, Eyers D, Shinagawa T, Pietzuch P. CAP-VMs: Capability-based isolation and sharing in the cloud. In: proceedings of USENIX Symposium on Operating Systems Design and Implementation. 2022, 597-612
32. Hua Z C, Yu Y, Gu J Y, Xia Y B, Chen H B, Zang B Y. TZ-container: Protecting container from untrusted OS with ARM TrustZone. Science China Information Sciences. 2021, 64(9):1-16
33. Sun Y Q, Safford D, Zohar M, Pendarakis D, Gu Z S, Jaeger T. Security namespace: making linux security frameworks available to containers. In: Proceedings of USENIX Security Symposium. 2018, 1423-1439
34. Gao X, Gu Z S, Kayaalp M, Pendarakis D, Wang H. Containerleaks: Emerging security threats of information leakages in container clouds. In: Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks. 2017, 237-248
35. Gao X, Gu Z S, Li Z F, Jamjoom H, Wang C. Houdini's escape: Breaking the resource rein of Linux control groups. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security. 2019, 1073-1086
36. Huang H, Rao J, Wu S, Jin H, Jiang S, Che H, Wu X F. Towards exploiting CPU elasticity via efficient thread oversubscription. In: Proceedings of International Symposium on High-Performance Parallel and Distributed Computing. 2021, 215-226
37. Wu S, Huang Z, Chen P F, Fan H, Ibrahim S, Jin H. Container-aware I/O stack: Bridging the gap between container storage drivers and solid state devices. In: Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2022, 18-30
38. Heo T, Schatzberg D, Newell A, Liu S, Dhakshinamurthy S, Narayanan I, Bacik J, Mason C, Tang C Q, Skarlatos D. IOCost: Block IO control for containers in datacenters. In: Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022, 595-608
39. Gu L, Guan J, Wu S, Jin H, Rao J, Suo K, Zeng D Z. CNTC: A container aware network traffic control framework. In: Proceeding of International Conference on Green, Pervasive, and Cloud Computing. 2019, 208-222
40. Randazzo A, Tinnirello I. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In: Proceedings of International Conference on Internet of Things: Systems, Management and Security. 2019, 209-214
41. Manco F, Lupu C, Schmidt F, Mendes J, Kuenzer S, Sati S, Yasukata K, Raiciu C, Huici F. My VM is lighter (and safer) than your container. In: Proceedings of Symposium on Operating Systems Principles. 2017, 218-233
42. Mavridis I, Karatza H. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. Future Generation Computer Systems, 2019, 94:674-696
43. Shen Z M, Sun Z, Sela G E, Bagdasaryan E, Delimitrou C, Renesse R V, Weatherspoon H. X-Containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2019, 121-135
44. Tazaki H, Moroo A, Kuga Y, Nakamura R. How to design a library OS for practical containers? In: Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2021, 15-28
45. Li Z J, Cheng J, Chen Q, Guan E, Bian Z Z, Tao Y, Zha B, Wang Q, Han W D, Guo M Y. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In: Proceeding of USENIX Annual Technical Conference. 2022, 53-68
46. Lim J T, Nieh J. Optimizing nested virtualization performance using direct virtual hardware. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2020, 557-574
47. Huang J, Qureshi M K, Schwan K. An evolutionary study of Linux memory management for fun and profit. In: Proceedings of USENIX

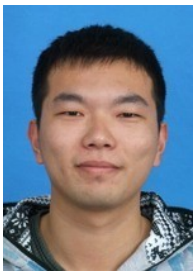
Annual Technical Conference. 2016, 465-478
48. Kim J, Shin P, Noh S, Ham D, Hong S. Reducing memory interference latency of safety-critical applications via memory request throttling and Linux Cgroup. In: Proceedings of IEEE International System-on-Chip Conference. 2018, 215-220
49. Zhuang Z, Tran C, Weng J, Ramachandra H, Sridharan B. Taming memory related performance pitfalls in linux Cgroups. In: Proceedings of International Conference on Computing, Networking and Communications. 2017, 531-535
50. Oh K, Park J, Eom Y I. Weight-based page cache management scheme for enhancing i/o proportionality of cgroups. In: Proceedings of IEEE International Conference on Consumer Electronics. 2019, 1-3
51. Park J, Eom Y I. Weight-aware cache for application-level proportional i/o sharing. IEEE Transactions on Computers, 2021, 71(10):2395-2407
52. Zheng D, Burns R, Szalay A S. Szalay. Toward millions of file system IOPS on low-cost, commodity hardware. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 2013, 1-12
53. Bang J, Kim C, Kim S, Chen Q C, Lee C, Byun E K, Lee J, Eom H. Finer-LRU: A scalable page management scheme for hpc manycore architectures. In: Proceeding of IEEE International Parallel and Distributed Processing Symposium. 2021, 567-576

Kun Wang received the B.S. from Huazhong University of Science and Technology (HUST) in 2015. Currently he is a Ph.D. candidate student in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), Huazhong University of Science and Technology (HUST) in China. His current research interests include container virtualization and kernel resource isolation.

Song Wu received the PhD degree from Huazhong University of Science and Technology (HUST) in 2003. He is a professor of computer science at HUST in China. He currently serves as the vice dean of the School of Computer Science and Technology and the vice head of Service Computing Technology and System Lab (SCTS) and the Cluster and Grid Computing Lab (CGCL) in HUST. His current research interests include cloud resource scheduling and system virtualization.

Shengbang Li received the B.S. from Shandong University(SDU) in 2021. Currently he is a M.S. candidate student in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), Huazhong University of Science and Technology (HUST) in China. His current research interest is kernel resource isolation.

Zhuo Huang received the B.S. from Huazhong Agricultural University (HZAU) in 2014. Currently he is a Ph.D. candidate student in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), Huazhong University of Science and Technology (HUST) in China. His current research interests include container virtualization, serverless computing optimization, and storage system.



Hao Fan received the PhD degree from Huazhong University of Science and Technology (HUST) in 2021. Currently he is working as a post-doctor in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), Huazhong University of Science and Technology (HUST) in China. His current research interests include container technology and storage system.

Chen Yu received the Ph.D. degree in information science from Tohoku University in 2005. From 2005 to 2006, he was a Japan Science and Technology Agency Postdoctoral Researcher with the Japan Advanced Institute of Science and Technology, Nomi, Japan. In 2006, he was with Japan Society for the Promotion of Science Postdoctoral Fellow with the Japan Advanced Institute of Science and Technology. Since 2008, he has been with the School of Computer Science and Technology, Huazhong University of Science and Technology, where he is currently a Professor and Special Research Fellow, working in the areas of wireless sensor networks, ubiquitous computing, edge computing, and edge intelligence.

Hai Jin is a Chair Professor of computer science at Huazhong University of Science and Technology (HUST). Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz. Jin worked at The University of Hong Kong between 1998 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is a Fellow of IEEE, Fellow of CCF, and a life member of the ACM. He has co-authored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security.