HP-Mapper: A High Performance Storage Driver for Docker Containers

Fan Guo
University of Sci. and Tech. of China
lps56@mail.ustc.edu.cn

Yongkun Li
University of Sci. and Tech. of China
ykli@ustc.edu.cn

Min Lv
University of Sci. and Tech. of China
lvmin05@ustc.edu.cn

Yinlong Xu
University of Sci. and Tech. of China
ylxu@ustc.edu.cn

John C. S. Lui
The Chinese University of Hong Kong
cslui@cse.cuhk.edu.hk

ABSTRACT

Docker containers are widely deployed to provide lightweight virtualization, and they have many desirable features such as ease of deployment and near bare-metal performance. However, both the performance and cache efficiency of containers are still limited by their storage drivers due to the coarse-grained copy-on-write operations, and the large amount of redundancy in both I/O requests and page cache. To improve I/O performance and cache efficiency of containers, we develop HP-Mapper, a high performance storage driver for Docker containers. HP-Mapper provides a two-level mapping strategy to support fine-grained copy-on-write with low overhead, and an efficient interception method to reduce redundant I/Os. Furthermore, it uses a novel cache management mechanism to reduce duplicate cached data. Experiment results with our prototype system show that HP-Mapper significantly reduces copy-on-write latency due to its finer-grained copy-on-write scheme. Moreover, HP-Mapper can also reduce 65.4% cache usage on average due to elimination of duplicated data. As a result, HP-Mapper improves the throughput of real-world workloads by up to 39.4%, and improves the startup speed of containers by 2.0×.

CCS CONCEPTS

• **Computer systems organization → Availability**.

KEYWORDS

Docker containers, storage drivers, I/O performance

ACM Reference Format:
Fan Guo, Yongkun Li, Min Lv, Yinlong Xu, and John C. S. Lui. 2019. HP-Mapper: A High Performance Storage Driver for Docker Containers. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3357223.3362718

1 INTRODUCTION

Virtualization technologies have become increasingly popular in recent years [45]. However, traditional virtual machine (VM) based virtualization solutions [38] need to emulate an entire set of hardware components, and run an individual operating system (OS) for each VM, so they introduce numerous overheads and significantly degrade the performance of applications running in VM [30]. To reduce the virtulization overhead, container-based virtualization has been proposed recently [35]. Specifically, containers run directly on the host operating system and perform as host processes without emulating hardware, so containers can achieve near bare-metal performance [33]. Moreover, containers use OS-level virtualization mechanisms (e.g., namespaces [12] and cgroups [5]) to provide isolation between containers, which introduce mild impact on containers' performance.

Docker [35] is the most popular container engine and is widely deployed in many cloud platforms (e.g., Google Cloud [6], Microsoft Azure [3], and AWS [2]). Specifically, Docker containers use images to hold their data and state, including binaries, input files and configuration parameters to run applications within the containers [10]. To improve storage efficiency and deployment speed of containers, Docker stores images in many layers, and enable each layer to be read-only and sharable between multiple containers. Thus, the storage driver of Docker containers, which is used to provide a unified view for multiple image layers and support copy-on-write for read-only files, plays a critical role in the performance of Docker container, and has also been the focus of some recent studies [21, 27, 28, 40, 44].

Existing storage drivers include Overlay2, AUFS, DeviceMapper, BtrFS and so on [10], and they work in different storage levels with different mechanisms. Specifically, Overlay2 and AUFS use the *file-based mechanism*, which puts multiple directories (i.e., images) on a single mount point and presents them as a single directory, so different containers have a unified view of the file system and they can share cached files easily. On the other hand, DeviceMapper and BtrFS employ the *block-based mechanism*, which stores images as logical volumes and manages them at the block level. As a result, they can perform copy-on-write at a finer granularity of blocks instead of the entire file, so the copy-on-write overhead can be greatly reduced.

We observe that I/O performance and cache efficiency of existing storage drivers are still limited for both file-based and block-based mechanisms. First, the file-based copy-on-write needs to copy the entire file before update. As a result, this kind of coarse-grained copy-on-write operations incur a large write overhead and degrade I/O performance. Second, for block-based drivers, when multiple containers read data from a shared file, a large number of redundant I/O requests may be introduced. This is because when using block-based drivers, each container has an individual file system and page cache. Last but not least, both file-based and block-based drivers introduce large amount of redundancy in page cache, and thus significantly degrades cache efficiency. Specifically, when containers read the same block, block-based drives may generate many copies of the block in page cache due to its inefficiency in sharing cached data. Although file-based drivers support sharing cached data when reading shared files, it also causes many duplicate data after performing copy-on-write on shared files.

Our experiment results validate the above problems. In particular, using current storage drivers, they generate up to 93.8% duplicated data in page cache, and cause up to 99.0% redundant read requests (see Section 2.2). Furthermore, current storage drivers may also bring as much as 616× extra overhead when performing copy-on-write operations. All these problems not only waste memory space and I/O bandwidth, but also cause a significant slowdown on the performance of containers. *Therefore, it is important to provide an efficient mechanism to detect and reduce redundant I/O requests and duplicate cached data, as well as enable a finer-grained copy-on-write so as to improve I/O performance and cache efficiency of Docker containers.*

However, it is non-trival to achieve the above goals for several challenges exist. First, current storage drivers and previous studies do not provide any method to timely detect redundant I/O requests, and it is challenging to develop such a method with low overhead. Second, it is also challenging to efficiently detect duplicate cached pages and evict appropriate cached copies. Finally, simply reducing data block size (i.e., copy-on-write unit) will incur a lot of metadata space and lookup overhead. Thus, how to support a fine-grained copy-on-write method with small overhead is also challenging. To address the above issues, we design and implement an efficient storage driver for Docker containers, and call it HP-Mapper. HP-Mapper can accurately intercept redundant I/O requests with low overhead, and significantly reduces duplicate data in page cache to improve cache efficiency, as well as supports a finer-grained copy-on-write method. Our contributions are summarized as follows.

- We propose a two-level mapping strategy to support writing data in different block sizes so as to enable "finer-grained" copy-on-write. Specifically, copying data is performed at the granularity of small blocks (i.e., 4KB blocks), and new writes are performed at the granularity of large blocks (i.e., 512KB blocks). Experiment results show that our finer-grained approach reduces the copy-on-write latency by up to 99.8% compared with file-based drivers.
- We develop a *lightweight* method to detect redundant I/O requests and intercept them by reading data from page cache to reduce I/Os. In particular, we use a memory-efficient hash table to record information of recent I/O requests, and so detecting redundant I/O requests can be achieved by looking up physical block number of each new I/O request from the hash table in memory. Experiment results show that our intercepting scheme can detect and intercept more than 92.6% redundant read requests.
- We further develop an adaptive page cache management scheme to improve cache efficiency. We take into account the memory utilization, the number of cache copies, as well as the access frequency of pages, in the design of page eviction scheme so as to better utilize memory and improve container performance. Experiment results show that our cache management mechanism can reduce 65.4% - 75.5% cache usage.
- We implement a prototype and conduct extensive experiments to show the overall efficiency of HP-Mapper. Experiment results show that compared with both file-based and block-based drivers, HP-Mapper improves the throughput of filebench workloads by up to 39.4% and improves the startup speed of containers by 2.0×. Moreover, HP-Mapper performs much better than current storage drivers when memory is scarce.

The rest of the paper is organized as follows. In Section 2, we first introduce the background of containers and their storage drivers, then analyze their limitations, finally we motivate the design of HP-Mapper. In Section 3, we present the design details of HP-Mapper. In Section 4, we describe the experiment setup and present the evaluation results of HP-Mapper. Section 5 reviews related work and Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Docker Storage Drivers

Docker provides lightweight virtualization by running containers directly on host operating system. It provides resource isolation between containers with a number of lightweight techniques that are present in the Linux kernel [35], such as namespaces, cgroup, and storage drivers. In particular, Docker uses images to store all requirements for running containers, and uses its own storage drivers to manage images. As shown in Fig. 1, container images are organized into many layers, and image layers are read-only so to enable sharing between different containers to improve storage efficiency of the images. As a result, any changes to container images need to be performed with copy-on-write operations by copying data to a writable layer. Besides supporting copy-on-write operations, Docker also uses its storage drivers to support data lookup across image layers. All existing storage drivers can be classified as either *file-based drivers* or *block-based drivers*. Next, we briefly describe these storage drivers and their key features.

File-based Drivers. AUFS [1] and Overlay2 [13] are both file-based drivers, which stack image layers to provide a single unified view at a single mount point. As a result, different containers can share data in page cache due to their use of the same file system. That is, when multiple containers read the same file, file-based drivers only generate one copy in page cache and share it among different containers, and this improves cache efficiency. However, when containers update a read-only file, file-based drivers need to create a copy of the entire file in the top writable layer, and then perform updates to the newly copied file. Such file-level copy-on-write method not only introduces a lot of I/O overheads by reading

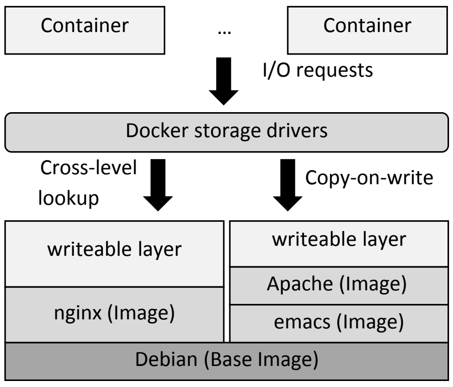


Figure 1: Layered structure of Docker storage driver

and writing unchanged data blocks, but also causes many duplicate data in both page cache and disk.

Block-based Drivers. Unlike file-based drivers, block-based drivers (e.g., DeviceMapper [7], BtrFS [37] and ZFS [15]) manage container images at the granularity of blocks. For example, DeviceMapper stores base level of an image as a logical volume and stores upper levels and containers as snapshots of the lower level. This way enables copy-on-write operations to be performed at the granularity of blocks, which only need to copy the changed blocks rather than the entire file. As a result, such a block-level copy-on-write method not only significantly reduces writing overhead, but also reduces duplicate data brought by copy-on-write operations. However, block-based mechanism also introduces other problems, and the most serious one is that cached data cannot be shared between different containers. This is because when using block-based mechanism, each container has an individual file system and page cache, and it cannot read data from other containers' page cache. Thus, block-based drivers usually generate a lot of redundant I/O requests and duplicate cached pages when they read data from the same file, which further degrade both I/O performance and cache efficiency.

In conclusion, current storage drivers possess design tradeoffs and they suffer from various problems in I/O performance and cache efficiency. Specifically, compared with block-based drivers, file-based drivers provide better I/O performance due to the ease of cache sharing when reading data from the same files, but their copy-on-write methods introduce larger overhead. Even worse, both the block-based drivers and file-based drivers generate a lot of duplicate data in page cache, and thus reduce cache hit ratio and degrade both I/O performance and cache efficiency when host memory becomes scarce.

2.2 Inefficiency of Docker Storage Drivers

To validate the performance degradation and cache inefficiency problem with current storage drivers, we conduct experiments with many container images and various storage workloads. Here, we like to point out that as demonstrated by many previous researches [40, 44], Overlay2, DeviceMapper and BtrFS always have the best

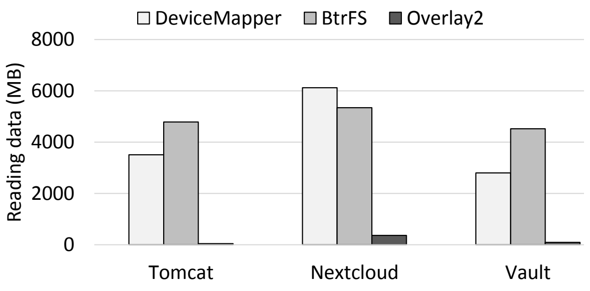


Figure 2: Total amount of reading data during the startup of containers. Block-based drivers (DeviceMapper and BtrFS) incur a lot of redundant I/Os due to sharing inefficiency.

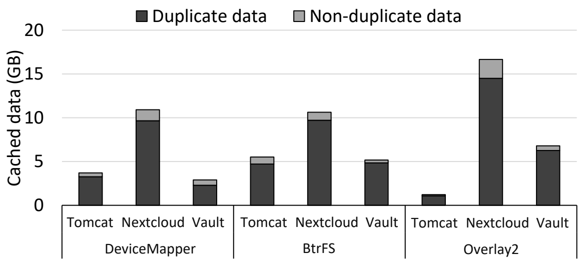


Figure 3: Total amount of duplicate/non-duplicate data.

overall performance and stability in most use scenarios, and they are also highly recommended by Docker Inc [14]. Thus, we take them as representatives of two kinds of storage drivers to demonstrate the issues.

First, to validate sharing inefficiency of blocked-based drivers, i.e., DeviceMapper and BtrFS, we compare them with Overlay2, which supports file sharing. Specifically, we concurrently launch 64 containers from a single image (see Section 4.1 for more detailed description about the system setup), and evaluate the total amount of data read during the startup of containers. The results are shown in Fig. 2, and we can see that compared with Overlay2, DeviceMapper and BtrFS need to read up to 76.3× and 104.1× more data during the startup of containers, respectively. As a result, they prolong the startup time of containers by up to 3.7× compared with Overlay2 (see Section 4.3). This is because block-based drivers usually generate a lot of redundant I/O requests due to the unshareable nature of cached data.

Next, we demonstrate the redundancy problem in page cache for both file-based and block-based drivers. In particular, we launch 64 containers from the same image and scan the cache to count the amount of duplicate pages and non-duplicated pages. The results are shown in Fig. 3. From the figure, we find that both file-based and block-based drivers generate a large amount of duplicate data in page cache, and the proportion of duplicate data reaches up to 88.6% when using DeviceMapper, 93.8% when using BtrFS, and 92.2% when using Overlay2. We point out that the duplicate data of DeviceMapper and BtrFS are mainly generated by their redundant

File Size	4KB	64KB	1MB	16MB
DM	0.12	0.74	0.96	1.39
BtrFS	0.09	0.09	0.09	0.10
Overlay2	1.99	2.49	7.14	61.7

Table 1: Copy-on-write latency (ms) of DeviceMapper (DM) and Overlay2 (test on SSD). File-based driver (Overlay2) introduces a large copy-on-write overhead.

read requests, and the duplicate data of Overlay2 are mainly brought by its file-level copy-on-write operations.

Finally, we demonstrate the inefficiency of coarse-grained copy-on-write operations by using filebench [41] to evaluate the latency. Specifically, we first generate a number of read-only files with the same size, then write 4KB data to each file. We also vary the size of files from 4KB to 16MB to measure the impact of file size on copy-on-write latency. As shown in Table 1, we can find that the copy-on-write latency of file-based driver Overlay2 increases significantly with the file size, and it reaches up to 61.7ms on 16MB files, which is up to 617× latency compared with the block-based BtrFS. Moreover, the copy-on-write latency of DeviceMapper is also up to 13.9× compared with BtrFS. The main reason why Overlay2 and DeviceMapper perform much worse than BtrFS is that Overlay2 needs to copy the entire file when performing copy-on-write operations, and DeviceMapper needs to copy the changed data at the granularity of large blocks (≥ 64KB), while BtrFS only copies the changed small blocks (4KB).

In summary, from the above results, we find that there is no single storage driver which can achieve both high I/O performance and high cache efficiency. There is also a tradeoff between copy-on-write performance and file sharing efficiency. Even worse, both the file-based and block-based drivers suffer from poor cache efficiency. Thus, there is still a large room to further improve the performance of storage drivers for Docker containers.

2.3 Challenges of HP-Mapper

To further improve the I/O performance and cache efficiency of Docker containers, we develop HP-Mapper, which is a high-performance and cache efficient storage driver by supporting finer-grained copy-on-write operations and intercepting redundant I/Os, as well as efficiently managing duplicate cached data. However, various challenges exist to provide these features and they are summarized as follows.

Finer-grained copy-on-write operations. HP-Mapper provides fine-grained copy-on-write to improve I/O performance, but how to support this feature without introducing extra overhead is challenging. Note that simply decreasing block size (i.e., unit of copy-on-write) will introduce a lot of overhead, such as memory overhead and lookup overhead of metadata . To address this challenge, we design a two-level mapping strategy in HP-Mapper with a careful design to reduce the memory overhead and disk fragments.

Intercepting redundant I/Os. HP-Mapper intercepts redundant I/O requests so as to retrieve data from memory instead of disks as much as possible. However, current storage drivers do not provide a mechanism to detect and intercept redundant I/O requests, and there are two challenges. The first challenge is how to accurately detect redundant I/O requests with low overhead, e.g., traditional content-based comparison methods must introduce a large overhead. The second one is how to locate the needed data from page cache requested by redundant I/Os. Note that cached pages are often moved and modified by the host kernel and containers, so how to obtain correct data from memory is non-trivial.

Managing duplicate cached data. Current storage drivers usually introduce a lot of duplicate data in page cache and thus reduce the cache effectiveness, and two challenges exist to efficiently manage duplicate cached data. First, how to find duplicate pages in page cache is difficult, because traditional memory deduplication methods (e.g., KSM [17]) do not support deduplication in page cache, and they will introduce a large overhead by comparing pages' content. Second, how to determine which cached pages need to be evicted is also challenging, because we need to consider not only the access frequency of cached pages, but also the amount of duplicate data and the memory utilization of the host.

3 DESIGN OF HP-MAPPER

HP-Mapper is an efficient storage driver for Docker containers. In this section, we first introduce the overall design of HP-Mapper, then present the details of its key components.

3.1 Overview of HP-Mapper

HP-Mapper works at the block level and follows block-based mechanisms to store and manage images. It provides fine-grained copy-on-write and high cache efficiency. As shown in Fig. 4, HP-Mapper consists of three modules to realize its key features: *Address mapper*, *I/O interceptor* and *cache manager*. First, to support fine-grained copy-on-write with low memory overhead, the address mapper employs a two-level mapping strategy to support two different block sizes in logical volumes, and adopts an on-demand block allocation mechanism for different write requests to achieve both high copy-on-write performance and low overhead. Next, to efficiently reduce redundant I/O requests, the I/O interceptor provides a lightweight intercepting mechanism, which can accurately detect redundant I/O requests so to read data from page cache instead of disks. Finally, the cache manager is used to reduce redundant cached data based on an efficient monitoring method and eviction policy.

The work flow of HP-Mapper is also illustrated in Fig. 4. When containers read/wirte a data block with its virtual block number (VBN), the address mapper first translates it into physical block number (PBN) by using its mapping trees. Then the address mapper uses an on-demand mechanism to allocate new blocks with different sizes for both copy-on-write operations and new writes. Finally, the I/O interceptor checks the I/O request and intercepts it if it is redundant. The cache manager is created as a daemon thread, which periodically scans all cached pages and decides which pages need to be evicted. We introduce the three modules in details in the following subsections.

3.2 The Address Mapper

To obtain low copy-on-write latency of small blocks and low lookup overhead of large blocks at the same time, HP-Mapper develops a two-level mapping tree to support two kinds of block sizes (i.e.,

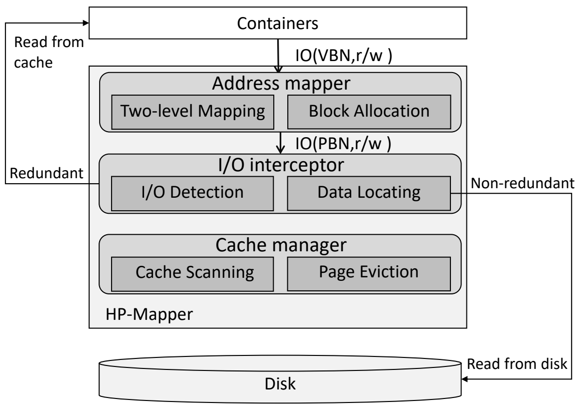


Figure 4: HP-Mapper architecture

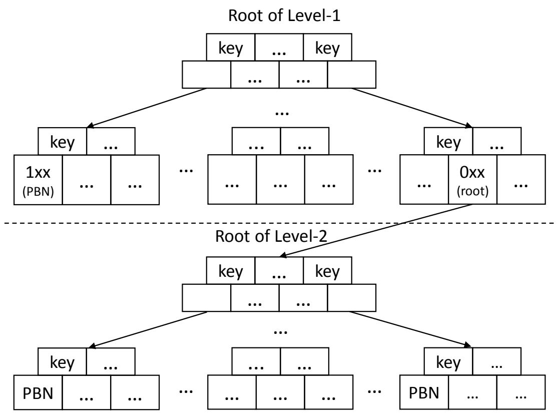


Figure 5: Structure of the two-level mapping tree

large blocks and small blocks), and it also provides an on-demand allocation mechanism to selectively allocate large blocks or small blocks for different I/O requests. In particular, HP-Mapper performs copy-on-write at the granularity of small blocks to reduce the update overhead, and allocates large blocks for new writes to lower the lookup overhead and space overhead of the mapping tree. However, several key technical issues need to be addressed: (1) How to design the two-level mapping tree to avoid high memory overhead and high lookup overhead? (2) How to design a selective block allocation policy for different write requests to achieve both high storage efficiency and high I/O performance? (3) How to reduce fragmentation brought by the two-level mapping strategy?

The two-level mapping tree. The one-level B-tree [24] adopted by DeviceMapper uses a fixed-length part of the VBN to index a physical block. As a result, DeviceMapper only supports one kind of block size. As shown in Fig. 5, we develop a two-level B-tree in HP-Mapper to support indexing two kinds of blocks: large blocks and small blocks. Specifically, we use values of the leaf nodes at the first level to differentiate the mappings. If the first bit of the value is set as 1, then the remaining bits of the value store the PBN of a large block. Otherwise, i.e., the first bit is set as 0, then we use the value to store the root of a second-level B-tree so as to index small blocks.

To perform the two-level mapping, the VBN of an I/O request is divided into three parts. The first part is used as an index to look up in the first level of the mapping tree. If the value in the first level points to a large block, then both the second and third parts are used as the offset in a large block. Otherwise, the second part of the VBN acts as the index of the second-level tree, and the third part is used as the offset in a small block. Moreover, HP-Mapper can make flexible conversion from large blocks to small blocks by simply adding a second-level mapping tree. Note that we set the block sizes of large blocks and small blocks as 512KB and 4KB respectively. By using these two block sizes, we can ensure that all information of a second-level tree can be exactly accommodated by one physical block, which reduces the storage overhead and lookup overhead of the mapping tree.

To support fast lookup of the mapping tree, HP-Mapper also splits I/Os to align with the block size, and the work flow of the address mapper is as follow. First, HP-Mapper splits large I/O requests to make sure that each split I/O can fit in one large block. That is, both the start and the end addresses of each split I/O could be located at the same large block. Then, HP-Mapper searches the VBN of the split request and translates it to PBN if it targets a large block. Otherwise, HP-Mapper further splits the I/O to fit in small blocks. After splitting, HP-Mapper ensures that the target sectors of each I/O are continuous on the physical devices.

The on-demand allocation mechanism. Next, we propose an on-demand allocation mechanism to allocate new blocks for I/O requests which target new virtual blocks, such as copy-on-write operations and new writes. Specifically, for copy-on-write operations, the goal is to avoid copying unnecessary data, so HP-Mapper chooses the appropriate block size based on the request size. For example, HP-Mapper allocates small blocks if the request size is less than half of the large block size. Otherwise, it allocates large blocks. With this policy, HP-Mapper can perform fine-grained copy-on-write operations to achieve high I/O performance. On the other hand, to reduce the frequency of triggering block allocation and lower the overhead of the mapping tree, HP-Mapper allocates large blocks immediately for new writes to improve the I/O performance. We note that new writes usually perform writing on contiguous virtual blocks, so they can make full use of the allocated large blocks. More importantly, most image files are written to disk with new writes. Hence, such an allocation mechanism can ensure that most blocks are allocated as large blocks, which significantly reduces the lookup overhead and the memory overhead of the mapping tree. Additionally, after the allocation of new blocks, HP-Mapper needs to update the mappings in the mapping tree to point to the newly allocated blocks. Specifically, when performing a fine-grained copy-on-write operation on a large block, HP-Mapper needs to split the block and remap it into several small blocks.

Reduction of disk fragments. By adopting two block sizes, HP-Mapper may generate a lot of fragments on disk, which further

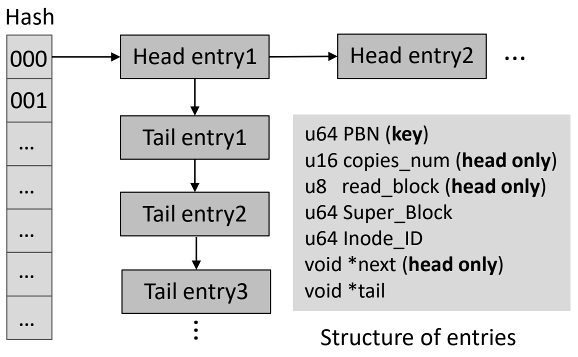


Figure 6: Structure of the hash table

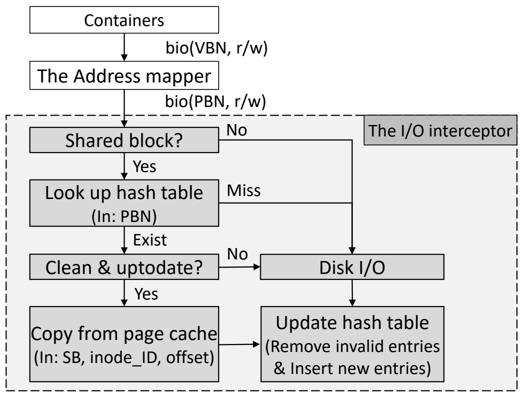

Containers
bio(VBN, r/w)
The Address mapper
bio(PBN, r/w)
Shared block?
No
The I/O interceptor
Yes
Look up hash table (In: PBN)
Miss
Exist
Clean & uptodate?
No
Disk I/O
Yes
Copy from page cache (In: SB, inode_ID, offset)
Update hash table (Remove invalid entries & Insert new entries)

Figure 7: Workflow of the I/O interceptor

leads to the under-utilization of disk space. To alleviate this problem, we develop a new block placement policy for HP-Mapper. To be specific, HP-Mapper divides the storage space of a physical device into two partitions. The first partition is allocated and used at the granularity of large blocks, and the second partition is allocated and used at the granularity of small blocks. Such a separated placement of blocks with different sizes can significantly reduce the fragments and improve the storage efficiency. Furthermore, we also propose a defragmentation policy to avoid the second partition to be depleted. In particular, HP-Mapper triggers defragmentation when the physical device is at idle, and it copies the scattered small blocks which once belonged to a same large block, combines them as large blocks, and places them to a contiguous physical space in the first region.

3.3 The I/O Interceptor

The I/O interceptor is responsible for detecting redundant read requests and intercepting them to read from the page cache so as to improve read performance on shared data. To achieve this goal, three key problems need to be addressed: (1) How to accurately detect redundant I/O requests and obtain the needed data from page cache with an accuracy guarantee? (2) How to manage the metadata to achieve high lookup performance with low memory overhead? (3) How to design the workflow to accelerate the interception of redundant I/O requests?

Detecting redundant I/Os and obtaining data. First, HP-Mapper provides a lightweight method to detect redundant I/O requests. If two or more I/O requests read data from the same data blocks, we treat them as redundant I/Os. To achieve this, we first record the PBN of the recent read requests in a hash table, then check each new read request by looking up their PBN in the hash table. If an I/O request is found in the hash table, we treat it as a redundant I/O and then intercept it by reading from the page cache immediately.

However, accurately retrieving the needed data of a redundant I/O from cache is difficult, because the cached pages are changeable and moveable in memory. To handle this problem, HP-Mapper records the metadata of the file system and files which can be used to locate the cached pages, such as super block of the file system, inode ID of files, and the offset in files. After finding the needed cached pages, HP-Mapper checks flags of these pages and copies data from them only if they are clean and up-to-date.

Management of metadata. To support fast lookup, HP-Mapper maintains the above mentioned metadata in a hash table, and uses the PBN of I/O requests as the key to find the corresponding entries in the hash table as shown in Fig. 6. The entries in the same bucket, which have the same hash value, are linked in a two-dimensional list. In particular, HP-Mapper uses two kinds of entries to store the metadata of cached pages: head entries and tail entries. A data block only has a single head entry to store the metadate of its latest cached copy, and the metadata of other copies are maintained in tail entries. HP-Mapper links all entries of the same block with "tail" pointers, and links head entries of different blocks with "next" pointers. Moreover, some other metadata are stored as value of the entries, such as super block, inode ID and so on. Here, we emphasize that HP-Mapper does not store offset of blocks in the hash table, because the offset of a shared block is fixed for different I/Os, and it can be obtained from the current I/O request. Our experiment results (see Section 4.3) show that HP-Mapper can quickly and accurately intercept redundant I/Os by using our hash table based design with a little memory overhead.

Workflow of the I/O interceptor. Note that blocks in the writable layer cannot be shared, so requests accessing these blocks are clearly non-redundant. To accelerate the interception of redundant I/O requests, we carefully design workflow of the I/O interceptor. As shown in Fig. 7, for each I/O, the I/O interceptor first checks whether target block of the request is read-only and sharable. By taking this step, HP-Mapper quickly excludes some non-redundant I/O requests without searching them in the hash table. Then, HP-Mapper checks whether the remaining I/O requests are redundant by looking up their PBNs in the hash table. If target blocks of a request cannot be found in the hash table, we treat it as a non-redundant I/O, and read data from disks. Otherwise, we treat it as a redundant I/O. Finally, HP-Mapper locates the needed data from other containers' page cache, and read data from page cache if they exist and are clean. Here we note that if an I/O request is redundant to multiple previous requests, HP-Mapper will check all corresponding cached pages until we find a clean one. If the corresponding cached

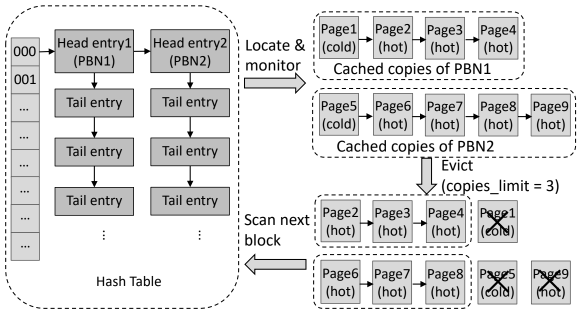


Figure 8: Workflow of the cache manager

pages have been evicted or modified, then HP-Mapper removes the entries from the hash table. Additionally, after the completion of I/O requests, HP-Mapper adds new entries for read requests and removes old entries of write requests to guarantee the correctness of the hash table.

We point out that several important issues are also addressed to guarantee the accuracy of the workflow. First, HP-Mapper uses a variable (i.e., *read_block*) in the hash table to avoid a disk block being read by multiple I/Os at the same time. Specifically, when a read request is not found in the hash table, HP-Mapper inserts a head entry into the hash table and sets *read_block* before reading from disk, and unsets it after completion of the I/O. If a read request hits in the hash table but the *read_block* is one, it will wait for the completion of previous request on the same block. Second, if some blocks of an I/O request hit in the hash table but others do not, HP-Mapper will split the I/O request into multiple small I/O requests and read data separately from memory and disk. Finally, HP-Mapper uses a read-write semaphore to control read-write permission of the hash table, which can avoid conflicts between read and write operations, and it only causes a little impact on I/O performance.

3.4 The Cache Manager

To efficiently remove redundant data in page cache and also improve the cache hit ratio, HP-Mapper takes into consideration of both pages' redundancy and pages' hotness in cache manager. As illustrated in Fig. 8, it first monitors all pages' characteristics, including the hotness information and the number of copies for each block, then chooses candidate pages for eviction based on their characteristics. The cache manager also provides a mechanism to adaptively adjust the eviction strategy so as to balance cache redundance and cache hit ratio. Several key issues must be addressed in cache management: (1) How to monitor pages' characteristics with low overhead? (2) How to develop a good eviction policy to achieve both low cache redundancy and high cache hit ratio? (3) How to adaptively adjust the eviction strategy based on the utilization of host memory?

Monitoring cached pages. To monitor real-time characteristics of cached pages with low overhead, the cache manager is implemented as a daemon thread, and scans all the cached pages periodically. That is, the cache manager scans *num_to_scan* pages in each scan period, and sleeps $scan\_interval$ between two contiguous periods. Besides, the cache manager uses metadata in the hash table to scan pages, and it scans all entries of the hash table and locates the corresponding pages of each entry in order. After locating all cached copies of a block, the cache manager can obtain the number of clean copies of the scanning block, as well as the hotness of each cached copy, which can be obtained by calling *page_referenced()* provided by Linux kernel. Thus, the cache manager can use very little overhead to monitor all pages' characteristics in real time (see Section 4.2). An example of the monitored information is shown in Fig. 8.

Page eviction. After scanning all cached copies of a data block, the cache manager will choose candidate pages for eviction based on their hotness and the total number of copies. To remove unnecessary redundancy, HP-Mapper limits the maximum number of cached copies for each block defined by *copies_limit*. Meanwhile, to improve cache hit ratio, HP-Mapper also takes hotness into consideration in the eviction. Fig. 8 also shows the eviction policy, if the number of copies is not greater than *copies_limit*, then HP-Mapper only evicts cold pages which has low access frequency. Otherwise, HP-Mapper first evicts cold pages, then evicts hot pages from tail of the list. The rationale is that tail page in the list is always the earliest one to be cached and has the longest lifetime. Thus, evicting tail page can improve the cache efficiency. Note that our eviction strategy only causes a negligible slowdown when containers read the evicted pages again, because the I/O interceptor of HP-Mapper enable the data to be read from other cached copies. Moreover, we emphasize that our eviction strategy only works on clean pages which are not locked or mapped into the page table, because locked and mapped pages are being used by other processes and cannot be evicted. Additionally, HP-Mapper will remove all invalid entries after the completion of cache eviction, and this also reduces lookup overhead of the hash table.

Adaptive adjustment of *copies_limit*. HP-Mapper also enables to adaptively adjust the value of *copies_limit* based on the utilization of host memory so as to balance the cache redundancy and cache hit ratio. Meanwhile, to avoid frequent adjustment, HP-Mapper classifies memory utilization into three states (low, normal and high) by using two thresholds (i.e., $Thresh_{low}$ and $Thresh_{high}$), and adjusts the number of copies accordingly. Specifically, if memory utilization is low (i.e., lower than $Thresh_{low}$), the cache manager will double the value of *copies_limit* to cache more copies for each block so as to improve the cache hit ratio for each block. If memory utilization is high (i.e., higher than $Thresh_{high}$), the value of *copies_limit* will be halved so as to reduce duplicate cached data and enable more blocks to have pages being cached. If memory utilization is at the normal state (i.e., between $Thresh_{low}$ and $Thresh_{high}$), then *copies_limit* will keep unchanged so as to avoid ping-pong effect. With this adaptive scheme, HP-Mapper achieves a consistent improvement on cache efficiency and I/O performance under different levels of memory utilization.

4 EVALUATION

To evaluate HP-Mapper, we implement a prototype in Linux kernel 3.10.0 to act as a plug-in module. In particular, we emphasize

	Average File Size	Num. of Files	Execution Time	Preallocated Files	I/O size				Proportion			
					read	new-write	append	update	read	new-write	append	update
Seqread	4GB	1	60s	100%	1MB	–	–	–	100%	–	–	–
Seqwrite	4GB	1	60s	0%	–	1MB	–	–	–	100%	–	–
Randread	4GB	1	60s	100%	4KB	–	–	–	100%	–	–	–
Randwrite	4GB	1	60s	100%	–	–	–	4KB	–	–	–	100%
Mongo	16KB	250000	60s	100%	WF	–	16KB	–	14.3%	–	14.3%	–
Varmail	16KB	250000	60s	80%	WF	16KB	16KB	–	15.4%	7.7%	7.7%	–
OLTP	10MB	440	60s	100%	2KB	–	–	2KB	50.5%	–	–	49.0%

Table 2: Configuration of Filebench workloads. WF stands for reading or writing the entire file.

that HP-Mapper works transparently to containers. We then conduct experiments on CentOS 7.5, and compare HP-Mapper with DeviceMapper (abbr. DM), BtrFS and Overlay2, which are state-of-the-art storage drivers for Docker containers. Here, we set the block size of DeviceMapper as 64KB, which is the minimal and default block size supported by DeviceMapper. We emphasize that using this block size can have a better copy-on-write performance, but incurs larger metadata overhead. Moreover, we set the backing file system of these storage drivers as Ext4 (except BtrFS) to avoid its impact on our experiment results. In the evaluation, we address the following questions.

- How large is the overhead of CPU cycles and memory introduced by HP-Mapper (Section 4.2)?
- How much improvement can HP-Mapper achieve in reducing redundant I/O requests and reducing start-up time of containers (Section 4.3)?
- How much improvement can HP-Mapper achieve for the I/O performance of containers? (Section 4.4)?
- How much memory can be saved with HP-Mapper and how does HP-Mapper perform in memory-scarce systems (Section 4.5)?

4.1 Setup

We conduct experiments on a server with an Intel Xeon E5-2650 v4 2.20GHz processor, 64GB memory, a 1TB hard disk (WD10EZEX), and a 512GB SSD (Intel 545s). We conduct our experiments by using a wide range of images and workloads. In particular, we first test containers' I/O performance with multiple workloads in Filebench [41], which is a popular storage benchmark and has been widely used in previous works [16, 22, 31, 43]. We list the configuration of these test workloads in Table 2. Note that in this table, we classify the write operations into three categories: "append", "new write" and "update". The append operations add some new data at the end of existing files, the new write operations write data to newly created files, and the update operations perform updating on existing data blocks. We note that most of the configurations are set as default values in Filebench, except for the number of files and the running time. Next, we use three different container images (i.e., Tomcat, Nextcloud and Vault) to evaluate HP-Mapper's effect on reducing redundant I/O requests (Section 4.3), reducing start-up time (Section 4.3), and improving cache efficiency (Section 4.5). We emphasize that these images are selected from the most popular ones in Docker Hub [8].

For the parameters of HP-Mapper, we choose the settings which achieve high performance and low overhead. Specifically, we set *scan_interval* of the cache manager as 20ms, and set *num_to_scan* as 2^{14}. These settings can achieve both high scanning speed and low scanning overhead. For *copies_limit*, we set its initial value as half of the total number of containers. Finally, we set $Thresh_{low}$ as 60% and set $Thresh_{high}$ as 80% to achieve a consistent improvement on cache efficiency.

4.2 Overhead of HP-Mapper

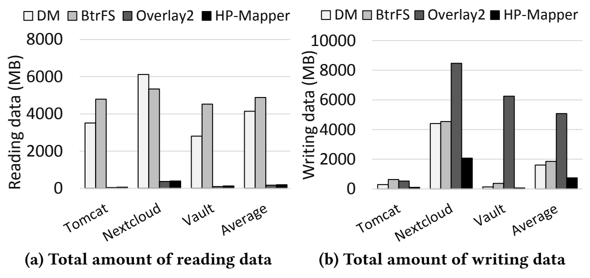
We first evaluate the overhead of HP-Mapper when simultaneously launching 64 "Vault" containers, including CPU overhead, memory overhead of metadata, and lookup overhead of the mapping tree and hash table.

CPU Overhead. We first evaluate the overhead of CPU cycles, which is mainly caused by the cache manager of HP-Mapper. Table 3 shows the results. We see that HP-Mapper only uses 4.1% CPU cycles to monitor all cached pages and perform eviction. This result demonstrates the lightweight nature of our cache management mechanism. This is achieved mainly because HP-Mapper locates all duplicate pages by only scanning metadata maintained in the hash table, and obtains their characteristics directly from pages' flags maintained by Linux kernel. As a result, the cache manager only introduces a little CPU overhead.

CPU Overhead	**Mem. overhead**		**Lookup overhead**	
	MT	HT	MT	HT
4.1%	3.3MB	10.6MB	1.1 us	< 0.1 us

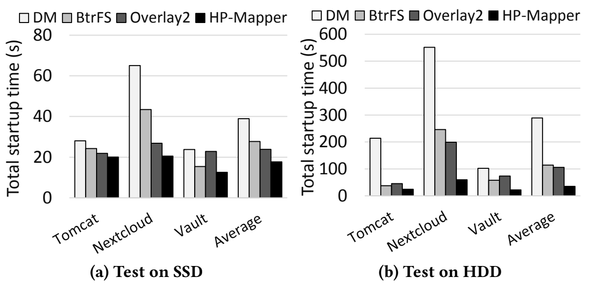
Table 3: Overhead of HP-Mapper (MT represents Mapping Tree, HT represents Hash Table)

Memory Overhead. We now evaluate memory overhead caused by the metadata of HP-Mapper, including the mapping tree (MT) and the hash table (HT). From Table 3, we find that HP-Mapper only uses 3.3MB memory to store the mapping tree, this is because most of the data blocks are mapped into large blocks (i.e., 512KB). Thus, HP-Mapper requires very little memory space to store the mapping tree. Moreover, we also see that HP-Mapper only uses 10.6MB memory space to store the hash table. In summary, the memory overhead brought by HP-Mapper only accounts for less than 1% of the total memory usage of containers, so it has a negligible impact on containers' performance.



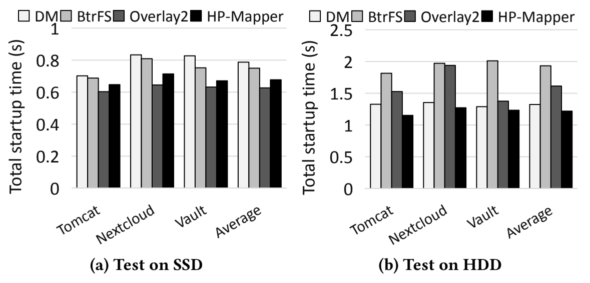
(a) Total amount of reading data (b) Total amount of writing data

Figure 9: Total amount of reading/writing data when launching 64 containers from a single image



(a) Test on SSD (b) Test on HDD

Figure 10: Total startup time when launching 64 containers from a single image on SSD/HDD



(a) Test on SSD (b) Test on HDD

Figure 11: Total startup time when launching a container from a image on SSD/HDD

Lookup Overhead. Finally, we evaluate the overhead of looking up the mapping trees and hash table. Table 3 shows that the average cost of looking up the hash table and mapping tree are within 0.1*us* and 1.1*us* respectively. Note that these lookup costs are far less than the latency of an I/O request, so they have a negligible impact on containers' I/O performance (see Section 4.4 for the I/O performance).

4.3 Reduction of I/O Redundancy

One major benefit of HP-Mapper is in reducing unnecessary I/O requests during the startup of containers, which further reduces containers' startup time. Here, we note that startup time is an important metric in many short-life containers [28]. In particular, we evaluate the improvement of HP-Mapper in reducing container' startup time by launching multiple containers simultaneously from a single image. We note that we use this setup because starting multiple containers from the same image is quite common in high-density use cases such as CaaS [4]. We use iostat [11] to monitor the total amount of data which is read from (or wrote to) disk during the startup of containers.

The results are shown in Figure 9. From Figure 9(a), we find that compared with DeviceMapper and BtrFS, HP-Mapper can reduce 92.6% - 98.7% of data which need to be read from disk during the startup of containers. This is because HP-Mapper can intercept redundant I/O requests when multiple containers read the same data block. On the other hand, for the total amount of writing data shown in Figure 9(b), we can see that HP-Mapper can reduce 53.6% and 85.3% writing data on average when comparing with DeviceMapper and Overlay2, respectively. We note that the reduction of writing data comes from our finer-grained copy-on-write method. Additionally, HP-Mapper reduces 59.5% writing data on average compared with BtrFS, and this improvement mainly comes from the reduction of unnecessary copy-on-write operations on the files in writable layers.

Finally, we test total startup time of 64 containers when launching them from a single image on SSD or HDD, and show the results in Figure 10. From Figure 10(a) we see that HP-Mapper achieves the fastest startup speed in most test cases. In particular, HP-Mapper reduces the average startup time of containers by 25.6% - 54.6% compared with the other three storage drivers. These improvements are mainly due to the reduction of unnecessary I/O requests. Moreover, we also test startup time of containers when launching from HDD. The results in Figure 10(b) show that HP-Mapper can improve the startup speed of containers by 2.0× - 7.2× compared with the other three storage drivers. This is because I/Os become more expensive on HDD, so the reduction of unnecessary I/Os can save more time.

Moreover, we also test startup time of a single container, and show the results in Figure 11. From Figure 11(a) we see that startup speed of containers when using HP-Mapper is faster than using DeviceMapper or BtrFS, but it is a little slower than using Overlay2. This is because HP-Mapper spends more time on the initialization of storage, such as creating virtual devices and mounting them for the launching container. However, when launching a container from HDD, HP-Mapper can always achieve the fastest startup speed due to its better I/O performance. Overall, HP-Mapper also can perform well in low density scenarios.

4.4 Improvement of I/O Performance

Another major benefit of HP-Mapper is improving containers' I/O performance by using two-level mapping strategy and on-demand allocation mechanism. To validate this, we evaluate the I/O performance of HP-Mapper. Specifically, we first evaluate the performance of basic I/O operations and then evaluate I/O performance of real-world workloads.

First, we evaluate the copy-on-write latency of different storage drivers. We emphasize that copy-on-write latency is one of the most important metrics to measure containers' I/O performance.

For example, the process of starting up containers introduces a lot of copy-on-write operations. To evaluate copy-on-write performance, we perform copy-on-write operations by writing 4KB data to each read-only file, whose size varies from 4KB to 16MB. The results are shown in Table 4, and we find that HP-Mapper always achieve optimal or near-optimal copy-on-write performance on all test cases. In particular, HP-Mapper reduces up to 90.6% and 99.8% copy-on-write latency comparing with DeviceMapper and Overlay2, respectively. This is because that HP-Mapper can perform copy-on-write operations at the granularity of 4KB blocks, which is usually the smallest unit of I/O length, so it does not need to read or write unnecessary data.

File Size	4KB	64KB	1MB	16MB
DM	0.13	0.74	0.96	1.39
BtrFS	0.09	0.09	0.09	0.10
Overlay2	1.99	2.49	7.14	61.7
HP-Mapper	0.07	0.08	0.10	0.13

Table 4: Copy-on-write latency (ms)

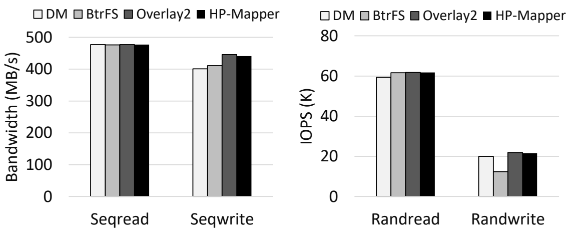


Figure 12: Performance of basic I/O operations (on SSD)

Next, we also evaluate the performance of basic I/O operations (i.e., sequential read, random read, sequential write and random write) by using the micro workloads in Filebench as listed in Table 2. We point out that to avoid the impact of copy-on-write operations, we issue random write to writable files. The results are shown in Figure 12. We can see that HP-Mapper always achieve the optimal or near-optimal performance on all basic I/O operations, so improving copy-on-write performance in HP-Mapper does not sacrifice the performance of other basic I/Os. In particular, HP-Mapper even improves sequential write bandwidth by 10.1% comparing with DeviceMapper. This is because HP-Mapper always allocates large blocks (512KB) for sequential writes, which can reduce the allocation frequency and thus improves I/O performance. Moreover, BtrFS performs much worse than other storage drivers for random write, and this slowdown mainly comes from its copy-on-write strategy, which performs copy-on-write on all existing data in both writable and read-only layers [44], and thus causes a large overhead on the update of metadata.

Lastly, we evaluate the overall I/O performance of containers by running filebench workloads, “Mongo”, “Varmail”and “OLTP”, and the workload configurations are listed in Table 2. In particular, we package the pre-allocated data of these workloads into the images to act as initial data. The results are shown in Figure 13. From Figure 13a, we find that HP-Mapper performs much better than Overlay2 when running “Mongo” and “Varmail” on SSD. Because there are many append operations in these two workloads, and HP-Mapper can handle append operations by simply adding new blocks at the end of files, but Overlay2 needs to copy the entire file and append data at the end of the new file, which is more expensive than block-based drivers. Furthermore, compared with DeviceMapper, HP-Mapper improves the throughput by 18.9% and 18.2% for “Mongo” and “Varmail” respectively. We note that the improvements come from the low allocation overhead of new blocks and low lookup overhead of the mapping tree. Furthermore, for the update-intensive workload (i.e., “OLTP”), HP-Mapper improves the throughput by 39.4% comparing with the other three drivers due to its low copy-on-write overhead and its in-place update strategy for writable layers. From Figure 13b, we find that HP-Mapper can also get improvements when running workloads on HDD. In summary, HP-Mapper can achieve the best I/O performance in most scenarios, especially for write-intensive workloads.

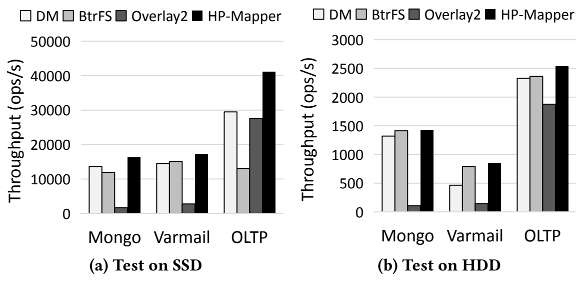


Figure 13: Performance of filebench workloads

4.5 Improvement of Cache Efficiency

Now we evaluate the improvement of cache efficiency with HP-Mapper. We first conduct experiments to evaluate the cache usage of containers when launching 64 containers from a single image, and the results are shown in Figure 14. We find that HP-Mapper always achieve the minimal cache usage in all test cases, e.g., HP-Mapper reduces 65.4% - 75.5% cache usage, when comparing with the other three storage drivers. This is because DeviceMapper and BtrFS generate many cached copies when multiple containers read the same block, and Overlay2 also generates duplicate pages due to the file-based copy-on-write. HP-Mapper, on the other hand, monitors all cached pages, and evicts cold copies and unnecessary hot copies, so it can significantly reduce the cache usage. We emphasize that the reduction of cache usage does not cause any slowdown on containers' performance, as shown in Sec. 4.4 and Sec. 4.3.

We further conduct experiments to evaluate the performance in a memory-scarce scenario. Specifically, we limit the host memory by running an in-memory file system (hugetlbfs [9]) to occupy certain amount of memory space on the host, and pages held by hugetlbfs cannot be swapped out. This way, we can flexibly adjust the size of host memory for running containers. In particular, we first gradually reduce the total available memory on the host to simulate a memory-scarce system, and then evaluate the startup

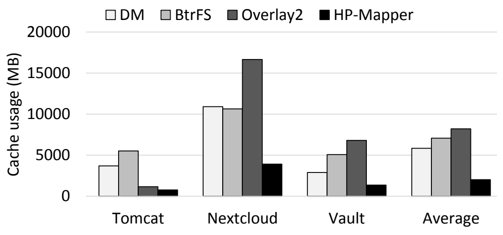
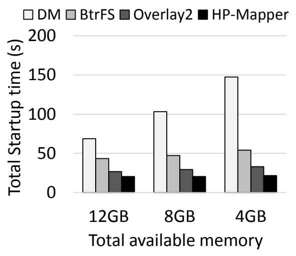
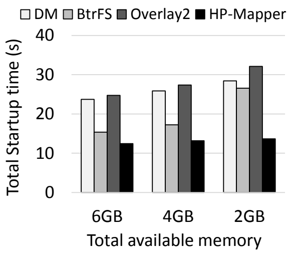


Figure 14: Page cache usage when launching 64 containers from a single image



(a) Launch 64 Nextcloud containers



(b) Launch 64 Vault containers

Figure 15: Total startup time of launching 64 containers in memory-scarce systems (test on SSD)

time of containers. The results are shown in Figure 15. We can see that as the total available memory space decreases, the startup speed of containers using HP-Mapper drops much slower than that of using the other two drivers. For example, when reducing the host memory from 12GB to 4GB, the startup time of "Nextcloud" containers increases by up to 114.5% when using DeviceMapper, but it only increases by 5.9% when using HP-Mapper. This is because HP-Mapper can significantly reduce duplicate data in page cache, so it increases the hit rate of page cache and reduces the frequency of page swapping and disk I/Os.

5 RELATED WORK

Container-based virtualization is an emerging solution that can be considered as a lightweight alternative to the traditional VM-based virtualization (e.g., KVM [32] and Xen [20]), and it has many representative implementations, such as Docker [35], LXC [29] and Linux-Vserver [25]. Moreover, many researchers focus on container-based virtualization and propose various optimization techniques to improve containers' performance [36, 42] and security [18, 39]. However, we note that none of the above works focus on containers' storage drivers to study the inefficiency problem in both I/O performance and memory management.

There are also some works paying attention to the I/O performance of containers. For example, Xu et al. [44] conduct extensive experiments to compare the performance of different Docker storage drivers, and they later propose an efficient I/O scheduling mechanism for containers that are concurrently executing [21]. Harter et al. [28] propose a Docker storage driver to enable fast container deployment by lazily pulling image data from remote registry so as to reduce network I/O. Du et al. [26] introduce a rapid container deployment system based on sharing network storage, which can reduce transferred data during the deployment. Unlike these works, HP-Mapper focuses on the native I/O performance of storage drivers, and it improves their native I/O performance by reducing redundant I/O requests and providing finer-grained copy-on-write.

On the other hand, memory efficiency is also an important issue in container-based systems, and it also attracts the attentions of researchers. For instance, Chen et al. [23] propose a resource allocation mechanism for container-based clouds to improve memory efficiency, and it is implemented based on a combination of auction and simulated annealing algorithms. Awada et al. [19] and Mao et al. [34] focus on the placement of containers in a cluster so as to improve the efficiency of both memory and CPU. Different from the above works, HP-Mapper pays attention to the cache inefficiency problem of containers, and it realizes an efficient cache management mechanism to reduce duplicate cached data so as to improve memory efficiency.

6 CONCLUSION

In this paper, we propose HP-Mapper, a high performance Docker storage driver, to address the inefficiency problem of current storage drivers on I/O performance and cache management. HP-Mapper supports finer-grained copy-on-write by using a two-level mapping strategy. It also significantly reduces redundant I/Os and duplicate cached data by using a lightweight interception method and an efficient cache management mechanism, respectively. Our experiment results show that both the I/O performance and cache efficiency of containers can be greatly improved with HP-Mapper.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Liuba Shrira, for their valuable comments and suggestions. Yongkun Li is USTC Tang Scholar, and he is the corresponding author. This work was supported in part by NSFC 61772484, NSFC 61772486, Huawei YB-N2018115197, and Youth Innovation Promotion Association CAS. The work of John C.S. Lui was supported in part by GRF 14200117.

REFERENCES

[1] 2019. AUFS: Another Union Filesystem. http://aufs.sourceforge.net.

[2] 2019. AWS containers. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[3] 2019. Azure containers. https://azure.microsoft.com/en-us/overview/containers.

[4] 2019. CaaS, the foundation of next generation PaaS. https://kubernetes.io/blog/2017/02/caas-the-foundation-for-next-gen-paas/.

[5] 2019. Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[6] 2019. Container services on google cloud. https://cloud.google.com/containers/.

[7] 2019. Device Mapper. https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt.

[8] 2019. Docker Hub. https://hub.docker.com/explore/.

[9] 2019. hugetlbfs. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[10] 2019. Images and storage drivers of Docker containers. https://docs.docker.com/storage/storagedriver/.

[11] 2019. iostat. https://linux.die.net/man/1/iostat.

[12] 2019. Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html.

[13] 2019. Overlay filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[14] 2019. Select a storage driver. https://docs.docker.com/storage/storagedriver/select-storage-driver/.

[15] 2019. ZFS. https://en.wikipedia.org/wiki/ZFS.

[16] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. 2017. Evolving Ext4 for Shingled Disks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 105–120.

[17] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. Citeseer, 19–28.

[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 689–703.

[19] Uchechukwu Awada and Adam Barker. 2017. Improving resource efficiency of container-instance clusters on clouds. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 929–934.

[20] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 164–177.

[21] Janki Bhimani, Zhengyu Yang, Ningfang Mi, Jingpei Yang, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan. [n.d.]. Docker Container Scheduler for I/O Intensive Applications running on NVMe SSDs. *IEEE Transactions on Multi-Scale Computing Systems* 1 ([n. d.]), 1–1.

[22] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. 2017. On the Performance Variation in Modern Storage Stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 329–344.

[23] Chaoquan Chen, Zhengzheng Zhang, and Xiaolan Xie. 2018. Container Cloud Resource Allocation Based on Combinatorial Double Auction. In *Proceedings of the 3rd International Conference on Intelligent Information Processing*. ACM, 146–151.

[24] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[25] Benoit des Ligneris. 2005. Virtualization of Linux based computers: the Linux-VServer project. In *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*. IEEE, 340–346.

[26] Lian Du, Tianyu Wo, Renyu Yang, and Chunming Hu. 2017. Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. In *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on*. IEEE, 332–339.

[27] Rajdeep Dua, Vaibhav Kohli, Sriram Patil, and Swapnil Patil. 2016. Performance analysis of Union and CoW File Systems with Docker. In *Computing, Analytics and Security Trends (CAST), International Conference on*. IEEE, 550–555.

[28] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers.. In *FAST*, Vol. 16. 181–195.

[29] Matt Helsley. 2009. LXC: Linux container tools. *IBM devloperWorks Technical Library* 11 (2009).

[30] Luis Herrera-Izquierdo and Marc Grob. 2017. A performance evaluation between Docker container and Virtual Machines in cloud computing architectures. *Maskana* 8 (2017), 127–133.

[31] Hyukjoong Kim, Dongkun Shin, Yun Ho Jeong, and Kyung Ho Kim. 2017. SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 271–284.

[32] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[33] Zhanibek Kozhirbayev and Richard O Sinnott. 2017. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems* 68 (2017), 175–182.

[34] Ying Mao, Jenna Oak, Anthony Pompili, Daniel Beer, Tao Han, and Peizhao Hu. 2017. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International*. IEEE, 1–8.

[35] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[36] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC' 18)*. USENIX Association.

[37] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 9.

[38] James E Smith and Ravi Nair. 2005. The architecture of virtual machines. *Computer* 38, 5 (2005), 32–38.

[39] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 1423–1439.

[40] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. 2017. In search of the ideal storage configuration for Docker containers. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*. IEEE, 199–206.

[41] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking.; login: The USENIX Magazine, 41 (1): 6–12.

[42] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC' 18)*. USENIX Association.

[43] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 59–72.

[44] Qiumin Xu, Manu Awasthi, Krishna T Malladi, Janki Bhimani, Jingpei Yang, Murali Annavaram, and Hsieh Ming. 2017. Performance Analysis of Containerized Applications on Local and Remote Storage. In *International Conference on Massive Storage Systems and Technology*.

[45] Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications* 1, 1 (2010), 7–18.