

Optimizing the Copy-on-Write Mechanism of Docker by Dynamic Prefetching

Yan Jiang, Wei Liu, Xuanhua Shi*, and Weizhong Qiang

Abstract: Docker, as a mainstream container solution, adopts the Copy-on-Write (CoW) mechanism in its storage drivers. This mechanism satisfies the need of different containers to share the same image. However, when a single container performs operations such as modification of an image file, a duplicate is created in the upper read-write layer, which contributes to the runtime overhead. When the accessed image file is fairly large, this additional overhead becomes non-negligible. Here we present the concept of Dynamic Prefetching Strategy Optimization (DPSO), which optimizes the CoW mechanism for a Docker container on the basis of the dynamic prefetching strategy. At the beginning of the container life cycle, DPSO pre-copies up the image files that are most likely to be copied up later to eliminate the overhead caused by performing this operation during application runtime. The experimental results show that DPSO has an average prefetch accuracy of greater than 78% in complex scenarios and could effectively eliminate the overhead caused by the CoW mechanism.

Key words: Docker; container; Copy-on-Write (CoW); storage driver; prefetch strategy

1 Introduction

Cloud computing provides users with convenient access to computing resources, which are allocated by virtual machines and consistently inefficiently scheduled. Container technology satisfies the need for efficient resource scheduling in cloud computing^[1,2] as it only contains the files necessary for its startup and shares the operating system kernel with the host.

Docker^[3] is the most popular open-source container solution today; it is based on image sharing to minimize the container volume, but the runtime files required by an individual container must be stored separately. Several Docker storage drivers featuring tiered storage have been proposed, such as OverlayFS, AUFS^[4], DeviceMapper^[5], Btrfs^[6], and zFS^[7]. Although these different storage drivers have

their own characteristics, all of them support the Copy-on-Write (CoW) mechanism.

The CoW mechanism allows multiple callers obtain the same pointer and share the same file; the system does not actually provide a dedicated copy until the caller attempts to modify this file. This mechanism is utilized when Docker's image file sharing feature is implemented. Docker sets the image file to read-only; therefore, an image can be used to start multiple containers, and each container only needs to save its own private data. The CoW mechanism can considerably reduce the space requirement of the container. However, when a single container performs operations such as modification of an image file, the file must be copied up to the upper read-write layer. This copy operation results in performance degradation of the application inside the container, particularly when a massive number of large files must be modified. Therefore, the overhead caused by the CoW mechanism is an urgent problem that must be solved^[8,9].

In this paper, we present the concept of Dynamic Prefetching Strategy Optimization (DPSO), which optimizes the CoW mechanism for a Docker container

• Yan Jiang, Wei Liu, Xuanhua Shi, and Weizhong Qiang are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: xhshi@hust.edu.cn.

*To whom correspondence should be addressed.

Manuscript received: 2019-07-19; accepted: 2019-07-26

on the basis of the dynamic prefetching strategy. DPSO avoids large overheads by pre-copying up image files at the beginning of the container life cycle. Considering that the internal behavior of containers launched by the same image is similar, we prefetched image files that were most likely to be copied up later. This work makes the following contributions.

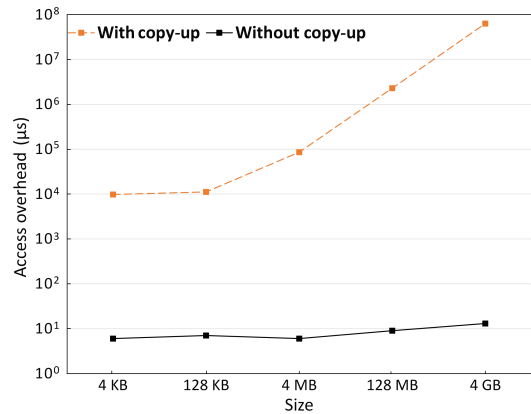
- We explored the pattern of the copy-up operation performed by a container in the CoW mechanism.
- We developed a record-based approach to guide the new container to perform a pre-copy-up operation when it started.
- We developed a dynamic prefetching strategy-based method to filter image files that must be pre-copied up from a large number of records.

The rest of this paper is organized as follows. Section 2 introduces the motivation of our work, while Section 3 presents the system architecture and design module of the DPSO system. We evaluate this strategy in Section 4, discuss related work in Section 5, and finally, summarize this research in Section 6.

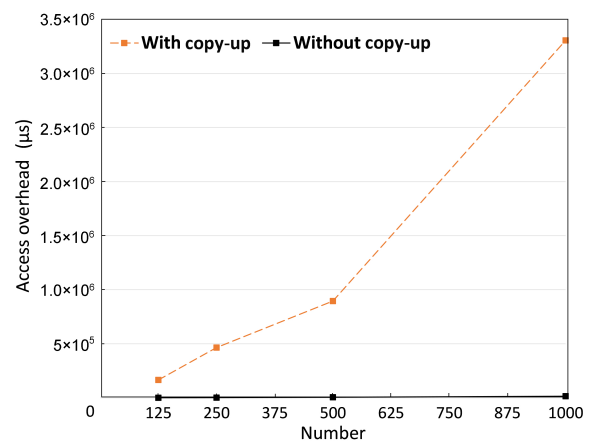
2 Motivation

Image files exist as a read-only layer in a Docker container instance. When the image file must be modified, it is copied up to the private read-write layer belonging to an individual container. We used the base image *docker.io/Ubuntu:latest* to customize specific images and simulate the container instance in real scenarios and performed two sets of experiments to illustrate the additional overhead caused by the copy-up operation.

In the first set of experiments, we created an image with five log files with sizes of 4 KB, 128 KB, 4 MB, 128 MB, and 4 GB inside the image and accessed these files via the internal application. In the second set of experiments, we customized an image containing 1000 files of 4 KB each and then periodically updated the timestamps of these files by using the internal application. Our experiments evaluated the containers that were started by these two sets of images. As shown in Fig. 1, the additional overhead of the copy-up operations accounted for most of the overall overhead of file access. Moreover, this overhead increased with increasing size and the number of files. In some cases, the overhead resulted in significant performance degradation in the application within the container. In this study, we aim to eliminate the overhead caused by



(a) Effect of file size



(b) Effect of the number of files

Fig. 1 Overhead of copy-up operation.

the CoW mechanism.

In the CoW mechanism, reading of the file does not trigger a copy-up operation, and the internal application of the container does not modify all of the image files that need to be copied up in its life cycle within a short time after startup. In Fig. 2, we assume that the container starts at t_0 and the copy-up operation of the image file occurs at t_2 . The core idea of our optimization is to pre-copy up the image file during the idle time between t_0 and t_2 .

3 Design and Implementation

3.1 Overall architecture

The overall architecture of DPSO is shown in Fig. 3.

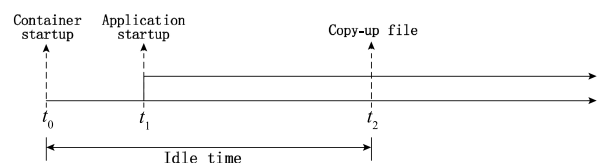


Fig. 2 Timeline of the container life cycle.

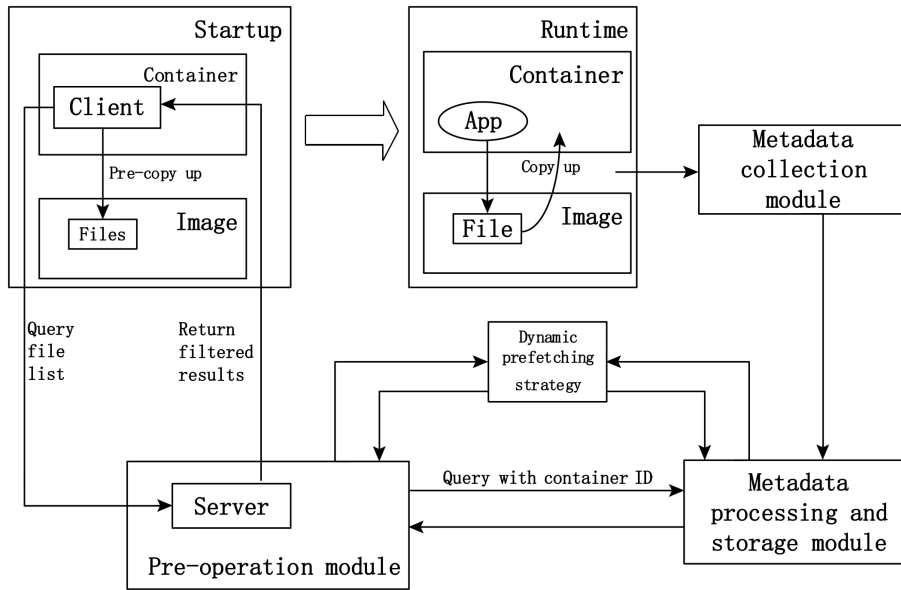


Fig. 3 Overall architecture of DPSO.

Our strategy includes four internal modules: the metadata collection module, the metadata processing and storage module, the pre-operation module, and the dynamic prefetching strategy. The metadata collection module is responsible for collecting the metadata information of the file throughout the life cycle of the container and sending it to the metadata processing and storage module, which stores and analyzes the records. The pre-operation module performs a pre-copy-up operation on part of the image files at the beginning of the container life cycle. The dynamic prefetching strategy is an abstraction module that is responsible for guiding the processing strategy and the processing flow of the other modules.

3.2 Collection of metadata

Docker adopts OverlayFS as its default storage driver on account of various reasons, such as improved performance and Linux kernel support^[10]. Moreover, the copy-up operation described earlier is a special operation implemented by OverlayFS. Thus, we collected the metadata information from OverlayFS.

In OverlayFS, the copy-up operation of the image file is in `ovl_copy_up_data()`. The DPSO system obtains the file handle of the image file in this function and takes the handle as a parameter to acquire the full path of the image file, which is used to uniquely mark the image file. One of the goals of DPSO is to analyze the mode in which Docker images perform copy-up operations internally; thus, correlating the image file with the image ID is necessary. We can easily obtain

the full path to the image file, but we cannot directly get the image ID on which the container is based. However, the characteristics of Docker allow us to get the unique *containerID*, which can be used to query the corresponding image ID on the host side later.

After the metadata collection module has obtained the necessary information, it will compose this information into a complete record and use NETLINK to send this record to the metadata processing and storage module. Here, we only collected metadata information for image files larger than 256 KB; the explanation for doing so is given in Section 4.4.

3.3 Processing and storage of metadata

This subsection describes the processing and final storage structure of records. To use the records directly, we designed additional properties for them on the basis of the parameters necessary for the prediction module. We also used MySQL to store historical records for lower overhead access and updated the database with dynamic policies.

Docker provides commands for various functions. The *docker inspect* command can effectively query specific information based on the container ID sent by the metadata collection module. By parsing the output of *docker inspect \$containerID*, we can get the unique ID of the image on which the container was started. Thus, the metadata information is associated with the corresponding image. We then added two properties to the record: Count and Timestamp. Count indicates the number of times the same record occurred, while

Timestamp refers to the last time the record occurred. These two properties play a decisive role in the analysis and prediction phase discussed later.

To establish correspondence between an image and the metadata information of the image file, we followed several principles when designing the data table as follows: each Docker image corresponds to a table and each record in the table represents the metadata information of an image file.

The structure of the data table includes *Id*, *Filename*, *Count*, and *Timestamp*. Besides *Count* and *Timestamp*, we added an *Id* column as the primary key to indicate the sequential number of the record in data table. *Filename* consisted of the image ID and the full path to the file, and it was also given a UNIQUE attribute and was used to identify the image file. The data table is inserted/updated upon arrival of some metadata information. During insert operations, *Count* was set to 1 and *Timestamp* was set to the current system time. When records in the table were updated, *Timestamp* was refreshed first. However, *Count* was updated only if the total number of records in the table is greater than a preset value of N ; the reason for doing so is given in Section 3.4.

3.4 Dynamic prefetching strategy

The dynamic prefetching strategy is one of the core designs of the DPSO system, and its main function is to manage updates to historical records and filter them in the pre-operation module.

Because of the highly customized characteristics of a container image, containers often have the functional attributes of specific applications, which means containers launched from the same image are highly likely to access the same image files. For example, in our experiment in Section 2, we started the container multiple times and observed that the files that were copied up inside the container were always the same. Our prefetch method is based on the observation that the internal behavior of containers launched by the same image is the same. However, small differences in the container instances launched by the same image may occur in more complex scenarios. Taking these scenarios into consideration, we require a flexible prefetch strategy to predict which files are most likely to be copied.

We could pre-copy up all of the recorded image files to obtain the highest prediction accuracy when only a few records are available. However, individual

differences between container instances can bring about a massive increase in the number of records over time, and large amounts of space and time may be necessary to copy up all records completely. Thus, we set a preset value N to mark an upper limit on the number of predicted files. The value of N is set by the administrator based on experience and container type. If the number of records exceeds N , we propose the following schemes for filtering.

- Filter the latest N records, on the basis of the value of *Timestamp*.
- Perform a further analysis by using the probability model on the basis of the value of *Count*.

In the second scheme, we used a probability model based on the value of *Count* to randomly predict N different records. The implementation of the probability model is shown in Algorithm 1. As described above, the *Count* value indicates the number of times the image file is copied up. Therefore, the image file with a larger *Count* value is the most likely to be copied later. If we make all *Counts* form a linear interval and the size of *Count* corresponds to the subinterval size, we can express the probability of the subinterval i being selected follows:

$$P_i = \frac{\text{count}_i}{\sum_0^{n-1} \text{count}_i} \quad (1)$$

where n is the total number of records in the table and count_i is the *Count* value of the i -th record.

As mentioned in Section 3.3, DPSO implements a dynamic strategy to update *Count*, i.e., when the number of records is less than or equal to the preset value N , *Count* will not be updated to avoid the additional accumulation of records entering the table.

Algorithm 1 Probability model

```

1: records[] ← {(id.count), ...}
2: records_sum[] ← empty
3: records_sum[0] ← records[0]
4: for each item in records_sum do
5:   records_sum[i] = records_sum[i - 1] + records[i]
6: end for
7: for  $i$  in range( $N$ ) do
8:   index ← rand()%records_sum[n - 1] + 1
9:    $k$  ← binarysearch(records_sum, index)
10:  if  $k$  has been chosen then
11:    continue
12:  end if
13:  res[i] ←  $k + 1$ 
14:   $i++$ 
15: end for
16: return res

```

This dynamic strategy can guarantee the prediction accuracy of the probability model when it is just enabled. However, it may also lose the earliest part of the information. In the next section, we prove that this loss has a limited effect on the prediction results.

The proposed strategy is based on the observation that the internal behavior of different containers launched by the same image is similar. Thus, subinterval accountings for most of the entire linear interval could be obtained, and records from this subinterval are expected to be predicted. These records can be expressed as follows:

$$R = R_i + R_c, R_i \gg R_c \quad (2)$$

where R_i represents the record that is expected to be predicted, R represents the total records, and R_c represents the record resulting from the difference between the container instances.

When the number of records $\text{count}(R)$ is greater than N , the property can be expressed as follows:

$$\text{count}(R) = N + a \quad (3)$$

where a represents the portion that the number of records $\text{count}(R)$ exceeds N . The number of records predicted by the probability model is equal to the preset value N . We used R_{pre} to represent the actual predicted records as Eq. (4).

$$\text{count}(R) = \text{count}(R_{\text{pre}}) + a \quad (4)$$

From the above equations, we obtained the following:

$$\text{count}(R_i) + \text{count}(R_c) = \text{count}(R_{\text{pre}}) + a \quad (5)$$

When the system starts to enable the probability model (i.e., Count just starts updating and $\text{count}(R)$ just exceeds N), $a \ll \text{count}(R_{\text{pre}})$. Moreover, $\text{count}(R_i) \gg \text{count}(R_c)$; thus, $\text{count}(R_i)$ can be expressed as follows:

$$\text{count}(R_i) \approx \text{count}(R_{\text{pre}}) \quad (6)$$

Because R_i accounts for the majority of the total records, we can speculate that most of R_{pre} also falls within this interval, which achieves the goal of prediction. As the proportion of a gradually increases to the point where it cannot be ignored, the weight (Count) of each record is basically formed; thus, we can predict the target record with high accuracy.

3.5 Pre-operation of the image files

In Section 3.4, we illustrated the feasibility of prefetching. In the current section, we introduce the specific implementation of prefetching.

Prefetch is a key step in our proposed optimization system; it is based on a dynamic strategy that screens out image files requiring pre-operation and pre-copies

up the image files in the idle time described in Section 2 to eliminate the additional runtime overhead caused by the CoW mechanism.

The pre-operation of an image file is divided into two sub-processes, i.e., the client process and the server process. As shown in Fig. 3, the client resides within the container, and the server is on the host. The client initiates a pre-copy-up operation at the beginning of the container life cycle via the following process:

- Parse the container ID from `/proc/self/cgroup` inside the container and write it to `/etc/hostname`.
- Create socket to connect with the server and send the container ID for query.
- Perform a copy-up operation on each file in the returned image file list. The specific command for this step is `touch $filename`.

The server queries and returns the image file information according to the container ID sent by the client via the following steps:

- Create a socket and wait for the client's request.
- Obtain the image ID based on the container ID sent from the client.
- Query the database, and filter the data in the table according to the prefetching strategy. Then, return the filtered results to the client.

4 Evaluation

4.1 Experiment environment

We evaluated DPSO on a Linux high-performance server with a 2.6 GHz Intel Xeon E5-2670 CPU and 64 GB memory. The implementation of DPSO was based on Docker version 1.13.1. In addition, the system collected the historical information of OverlayFS by instrumentation.

4.2 Performance evaluation

DPSO was optimized by spreading the copy-up overhead to the idle time of the container runtime. To test the performance improvements the system could bring about, we evaluated the additional access cost of image files under different idle times.

Because no standard benchmarks and applications exist for testing in this area, our experiments were based on a set of synthetic simulations; for the sake of simplicity and efficiency, we only simulated the copy-up operation of the image file triggered by the application inside the container. We used the base image `docker.io/Ubuntu:latest` to customize a specific image with four target image files with sizes of 4 GB, 256 MB,

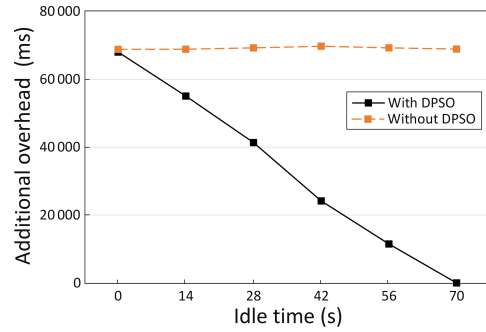
16 MB, and 1 MB for testing; the internal application must access these image files and add a record to the end of each file. In this experiment, the variables were the idle time between the container startup and the application execution. Note that the idle time here should actually be the time between the container startup and the copy-up operation of the target image file.

Figure 4 shows that, when the idle time is set to zero, the overhead required to copy up a 4 GB image file is approximately 70 s. As the idle time increased to 70 s, the overhead of DPSO gradually approached zero (0.067 ms) because the image file had already been copied up to the upper read-write layer in advance. Results from the three experiments were similar. These findings prove that the DPSO system could effectively eliminate the overhead caused by the copy-up operation when the container has enough idle time. Indeed, the system does not increase the overhead even in some extreme cases.

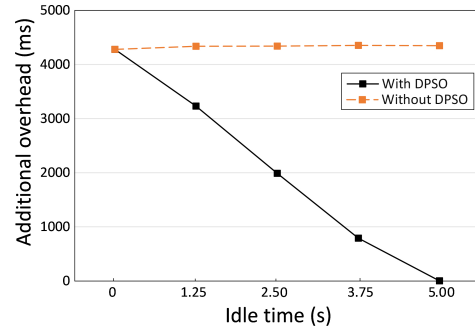
4.3 Accuracy of the dynamic prefetching strategy

As described earlier, DPSO predicts the files that are most likely to be copied up on the basis of the observation that the internal behaviors of containers launched by the same image are similar. To simulate the similarities and differences between containers, we abided by the following principles when designing the internal behavior of each container: all of the containers performed a copy-up operation on the same set of files to simulate their similarity and each container also performed a copy-up operation on a few other different image files to simulate individual differences between container instances. Based on these principles, we designed a reasonable set of inputs, which are listed in the second column of Table 1. In this experiment, the behavior of the application and the contents of the image file are identical to those in Section 4.2. Since nearly all of containers modified files 1–10, we set N to 10. We evaluated the accuracy of the dynamic prefetching strategy by experimenting with 10 containers successively launched from the same image. The history of all previous containers was used to predict the behavior of the next container.

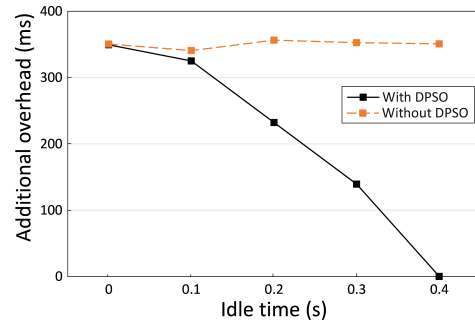
Table 1 reveals an average prediction accuracy of 78.6% (range 60%–90%) despite the experiment using a small data set, leading to differences between different container instances accounting for a large proportion (approximately 10%). Moreover, as the



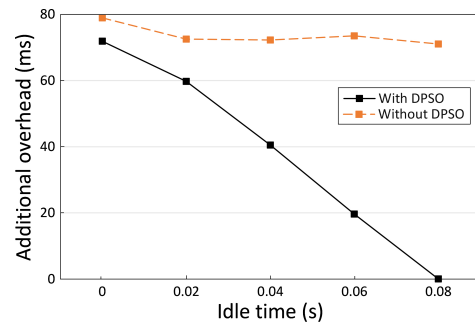
(a) Additional overhead of 4 GB file



(b) Additional overhead of 256 MB file



(c) Additional overhead of 16 MB file



(d) Additional overhead of 1 MB file

Fig. 4 Overhead of different sized image files under different idle times.

number of containers launched by the image increased, our probability model based on the value of Count appeared to improve the accuracy of prefetching.

Table 1 Experimental results of the proposed dynamic prefetching strategy.

Container ID	Copy-up file	Predicted file	Accuracy (%)
1	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11		
2	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12	File:1, 2, 4, 5, 6, 7, 8, 9, 10, 11	81.8
3	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 23	File:2, 3, 4, 5, 6, 7, 8, 9, 10, 11	75.0
4	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	File:1, 2, 3, 4, 5, 6, 8, 10, 11, 12	81.8
5	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15	File:1, 3, 6, 7, 8, 9, 10, 11, 12, 23	63.6
6	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 11	81.8
7	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 11	File:1, 2, 3, 5, 6, 7, 8, 9, 11, 23	90.0
8	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10	File:1, 2, 4, 5, 6, 7, 10, 11, 12, 23	70.0
9	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	File:1, 2, 3, 4, 7, 8, 9, 10, 11, 12	81.8
10	File:1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20	File:1, 2, 3, 4, 5, 7, 8, 9, 10, 11	81.8

4.4 System overhead analysis

In this subsection, we measured and analyzed the overall cost of the DPSO system. The overhead mainly concentrates on two processes: metadata collection and pre-operation. During the metadata collection process, a complete record is combined and transmitted in OverlayFS when a copy-up operation occurs. The pre-operation occurs during the idle time of the container. Thus, the metadata collection behavior interferes with the application inside the container.

We estimated the additional cost of DPSO by switching the metadata collection function on and off. We tested our strategy on 1000 files with sizes of 4 KB each and show the results in Table 2. The average cost of collecting and transmitting metadata information was approximately 10 ms. When the image file was small (4 KB or less), the additional cost of DPSO accounted for a large proportion of the total overhead. However, this cost was independent of the file size. The overhead of the copy-up of small files did not have a considerable effect on the application. Therefore, only files larger than 256 KB were collected in our flexible metadata collection method, which implies that DPSO's overhead exerts a minimal effect on the entire system.

5 Related Work

Given the increasing popularity of Docker^[11], several researchers have conducted studies on the storage drives of the software. Several Docker storage drivers were comprehensively evaluated in Refs. [12, 13] and

Table 2 Overall cost of metadata collection.

Number of files accessed	Metadata collection off (μ s)	Metadata collection on (μ s)
1	12 459	23 595
10	130 486	224 302
100	1 265 765	2 320 724
1000	12 391 793	22 423 136

a number of key findings were obtained. For example, BtrFS performs better under Docker commands but has poor performance under real workloads, DeviceMapper has poor performance under Docker commands but performs well under actual workloads. Interestingly, Advanced multi-layered Unification Filesystem (AUFS) performs well in both cases.

Some research has focused on the memory usage of specific storage drivers (e.g., DeviceMapper and BtrFS) for Docker. File systems based on the block storage mode, such as BtrFS and zFS, bear the disadvantage of redundant disk reads. Wu et al.^[14] designed TotalCOW to create a read-only cache layer between the block device and the file system, thereby eliminating redundant disk reads and caching. Similarly, Zhao et al.^[15] designed a DRR strategy that includes a DRR metadata management layer between the file system and the disk to reduce redundant disk reads by memory copying between containers.

Another study focused on the optimization of the performance of UnionFS. Copying files from disk to memory and synchronizing from memory to disk are the most time-consuming overheads caused by the CoW mechanism. In Ref. [16], Mizusawa et al. found the most time-consuming fsync operation by analyzing the OverlayFS stack. The group then reduced the number of unnecessary fsync operations to lower the overhead caused by the CoW mechanism; despite the improvements provided by this solution, however, it still impacts the application. Our work can completely eliminate this impact.

6 Conclusion

In the present paper, we introduced DPSO, an optimization of the CoW mechanism for a Docker container based on the dynamic prefetching strategy, to spread the overhead of the CoW mechanism

to the idle time of the container life cycle. We analyzed the execution flow of the copy-up operation performed by OverlayFS and realized efficient metadata information collection on the basis of this analysis. We also designed a flexible dynamic prefetch strategy to select different prefetching schemes according to different needs. DPSO proved to be effective in various scenarios, although several aspects of our approach could still be improved. In future work, we plan to build more complex models or use machine learning strategies to prefetching. As the implementation of DPSO does not depend on a specific code of Docker, we will also consider integrating it into the Docker source code as a plug-in.

Acknowledgment

The work was supported by the National Key Research and Development Program of China (No. 2018YFB1003203), the National Natural Science Foundation of China (Nos. 61772218 and 61433019), the Outstanding Youth Foundation of Hubei Province (No. 2016CFA032), and the Chinese Universities Scientific Fund (No. 2019kfyRCPY030).

References

- [1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, An updated performance comparison of virtual machines and Linux containers, presented at IEEE Int. Symp. Performance Analysis of Systems and Software, Philadelphia, PA, USA, 2015.
- [2] B. Xavier, T. Ferreto, and L. Jersak, Time provisioning evaluation of KVM, Docker and unikernels in a cloud platform, in *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing*, Cartagena, Colombia, 2016.
- [3] C. Anderson, Docker [Software engineering], *IEEE Softw.*, doi: 10.1109/MS.2015.62.
- [4] J. Okajima, Aufs5-advanced multi layered unification filesystem version 5.x, <http://aufs.sourceforge.net/>, 2013.
- [5] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok, Dmddedup: Device mapper target for data deduplication, presented at Ottawa Linux Symposium, Ottawa, Canada, 2014.
- [6] O. Rodeh, J. Bacik, and C. Mason, BTRFS: The Linux B-tree filesystem, *ACM Trans. Storage*, vol. 9, no. 3, p. 9, 2013.
- [7] O. Rodeh and A. Teperman, zFS—A scalable distributed file system using object disks, in *Proc. 20th IEEE/11th NASA Goddard Conf. Mass Storage Systems and Technologies*, San Diego, CA, USA, 2003.
- [8] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, Slacker: Fast distribution with lazy Docker containers, in *Proc. 14th USENIX Conf. File and Storage Technologies*, Santa Clara, CA, USA, 2016.
- [9] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors, in *Proc. 2nd ACM SIGOPS/EuroSys European Conf. Computer Systems*, Lisbon, Portugal, 2007.
- [10] N. Mizusawa, K. Nakazima, and S. Yamaguchi, Performance evaluation of file operations on OverlayFS, in *Proc. 15th Int. Symp. Computing and Networking*, Aomori, Japan, 2017.
- [11] D. Jaramillo, D. V. Nguyen, and R. Smart, Leveraging microservices architecture by using Docker technology, presented at SoutheastCon 2016, Norfolk, VA, USA, 2016.
- [12] R. Dua, V. Kohli, S. Patil, and S. Patil, Performance analysis of union and cow file systems with Docker, presented at Int. Conf. Computing, Analytics and Security Trends, Pune, India, 2016.
- [13] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. J. Li, R. Rangaswami, and M. Zhao, In search of the ideal storage configuration for Docker containers, in *Proc. 2nd Int. Workshops on Foundations and Applications of Self Systems*, Tucson, AZ, USA, 2017.
- [14] X. B. Wu, W. G. Wang, and S. Jiang, TotalCOW: Unleash the power of copy-on-write for thin-provisioned containers, in *Proc. 6th Int. Asia-Pacific Workshop on Systems*, Tokyo, Japan, 2015, p. 15.
- [15] F. Zhao, K. Xu, and R. Shain, Improving copy-on-write performance in container storage drivers, presented at Storage Developers Conference, Santa Clara, CA, USA, 2016.
- [16] N. Mizusawa, J. Kon, Y. Seki, J. Tao, and S. Yamaguchi, Performance improvement of file operations on OverlayFS for containers, presented at IEEE International Conference on Smart Computing, Taormina, Italy, 2018.



Yan Jiang received the bachelor degree from Huazhong University of Science and Technology, China, in 2017. He is a master student at National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology, China. His

research interests focus on the HPC I/O, burst buffer, and parallel filesystem.



Wei Liu received the bachelor degree from Huazhong University of Science and Technology, China, in 2016. He is a master student at National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology, China. His

current research interests focus on the HPC I/O, cloud computing, and parallel filesystem.



Xuanhua Shi is a professor in National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology, China. He received the PhD degree from Huazhong University of Science and Technology, China, in 2005.

From 2006, he worked as an INRIA Post-Doc in PARIS team at Rennes for one year. His current research interests focus on the cloud computing and big data processing. He has published over 100 peer-reviewed publications.



Weizhong Qiang is an associate professor in National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology, China. He received the PhD degree from Huazhong University of Science and Technology,

China, in 2005. He has authored or coauthored about 30 scientific papers. His research interests include system security about virtualization and cloud computing.