FinerDedup: Sifting Fingerprints for Efficient Data Deduplication on Mobile Devices

Xianzhang Chen, Xingjie Zhou, Wei Li*, Xi Yu, Duo Liu, Yujuan Tan, Ao Ren
College of Computer Science, Chongqing University, Chongqing, China
{xzchen,yuxi,liuduo,tanyujuan,ren.ao}@cqu.edu.cn,{zhouxingjie,liwei}@stu.cqu.edu.cn

Abstract

Data deduplication is promised to extend the lifetime and capacity of storage on mobile devices. However, existing data deduplication works show high memory consumption and indexing costs for maintaining a fingerprint for each data block, especially when the duplicate ratio of data blocks on mobile systems is about 10% to 30%. In this paper, we propose a novel approach called FinerDedup to optimize the memory costs and retrieval efficiency of data deduplication. FinerDedup drastically reduces the number of fingerprints by screening out the duplicate data blocks via random forest and Bloom filter. We implement FinerDedup on real mobile devices with Android 10 and evaluate it with real workloads. Extensive experimental results show that FinerDedup can reduce 85% of fingerprints and 20% of I/O latency over the widely-used DmDedup.

Keywords

Data deduplication, machine learning, mobile systems

ACM Reference Format:
Xianzhang Chen, Xingjie Zhou, Wei Li*, Xi Yu, Duo Liu, Yujuan Tan, Ao Ren. 2024. FinerDedup: Sifting Fingerprints for Efficient Data Deduplication on Mobile Devices. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657307

1 Introduction

With the ever-increasing storage consumption of mobile applications, both users and manufactures of mobile devices are bothered by the limited storage space on the devices [4, 9]. In this case, data deduplication is promised to extend the storage limitations of mobile devices by saving the storage space of duplicated data [11–13, 16]. The mainstream manufactures of mobile devices, such as Apple [5] and Huawei [6], are trying to provide data deduplication services for users.

There have been several data deduplication approaches for mobile systems in the past decade. DmDedup [14] is one of the most widely-used data deduplication mechanisms, which builds a fingerprint for each data block by hashing and avoids redundant writes of duplicate data blocks by retrieving the fingerprint table. SmartDedup [16] uses online deduplication in the foreground to reduce redundant writing and runs offline deduplication in the background to find omissions. APP-Dedupe [12] divides the entire fingerprints into different groups and only maintains the fingerprints of the foreground application in the memory. In general, existing approaches rely on a huge fingerprint table that precisely records a fingerprint for each data block to find out the duplicate data blocks.

Unfortunately, the overall duplication ratio of data on mobile systems is merely about 10% to 30% [12, 16]. It means that about 70% to 90% of the fingerprints in the fingerprint table of existing data deduplication systems are useless. These useless fingerprints not only bring large memory footprints but also degrade performance for loading and retrieving them before data accesses. Therefore, suppose we can find out the duplicate data blocks and build a fingerprint table only for them. We can drastically reduce the overhead for maintaining and searching the fingerprints of data blocks.

In this paper, we propose an efficient and low-overhead data deduplication method, namely FinerDedup, for mobile systems by cutting off the useless fingerprints. FinerDedup recognizes data blocks as two types: 1) *unique* data blocks that are unlikely to be duplicated in the future and 2) *duplicate* data blocks that will be duplicated in the future. Given a new write request, FinerDedup predicts the type of the corresponding data block using a pre-trained random forest [3], which is a lightweight classification model that consumes minimal performance on mobile devices. To prepare the model, FinerDedup collects the features of writes, transforms them into datasets, and trains the random forest when the system is idle.

Moreover, FinerDedup designs a Bloom filter [2, 8] along with the fingerprint table to correct the possible false predictions. The Bloom filter can filter out the duplicate data blocks that are predicted to be unique. Then, FinerDedup will save fingerprints for these data blocks in the fingerprint table. The counters in the fingerprint table are used to periodically remove those fingerprints of the unique data blocks that are predicted to be duplicated. Finally, FinerDedup uses Linux kernel's device mapper framework to formulate a mapping strategy. Compared with other deduplication systems, FinerDedup can keep fewer fingerprints and achieve lower I/O latency.

We implement FinerDedup on the Google Pixel 3 smartphones. We evaluate the performance and overhead of FinerDedup at different duplication rates using typical benchmark, Flexible I/O (FIO) [1], and real-world applications. The experimental results show that FinerDedup outperforms DmDedup [14] when the duplication rate is lower than 30%. In real workloads, compared with DmDedup, we can achieve more than 92% deduplication and reduce more than 85% of fingerprints. In summary, FinerDedup can minimize the overhead caused by useless fingerprints while ensuring efficient data deduplication.

*Wei Li is the corresponding author.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3657307

The main contributions of this paper include:

- We propose a feature-based data deduplication method for mobile devices, FinerDedup, for saving memory footprints and reducing I/O latency.
- We implement Finerdedup on a real mobile system running with Android.
- Extensive experimental results show that FinerDedup significantly reduces memory costs and improves write performance over the existing data deduplication approach.

2 Background and Motivation

2.1 Data Duplicates in Mobile Systems

The ability of data deduplication technologies has drawn significant attention from the manufacturers of mobile devices in extending the storage capacity of mobile systems and thereby enhancing user engagement. For example, the smartphones using HarmonyOS of Huawei have already supported data deduplication [6]. Before deploying a data deduplication mechanism on mobile devices, it is essential to understand the features of data on mobile systems.

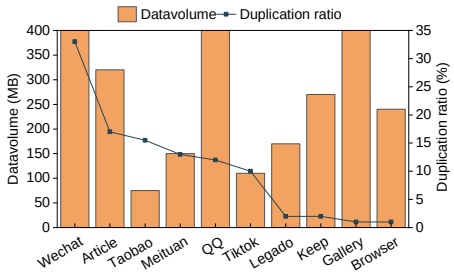


Figure 1: Application duplication ratio of mobile device

To this end, we collect user data from real-world applications to ascertain data duplication on mobile devices. Figure 1 displays the data duplication ratios of typical mobile applications with varied data volumes (a mere 400M data points are visible, and applications like WeChat and QQ have even larger volumes). In our experiments, the majority of duplication rates are below 20%, averaging 12.27%. Existing works, such as App-dedup [12] and SmartDedup [16] introduce that the overall duplication rate of mobile device applications is about 20% to 33%. Consequently, we estimate that the data duplication ratio of mobile devices is approximately 10% to 30%, which is significantly lower than the data load in traditional deduplication scenarios, such as backup systems. Hence, suppose the mobile applications store 100GB data on a mobile system. It may contain 10GB to 30GB of duplicate data. Due to the highly limited capacity, generally smaller than 256GB, and lifespan of mobile devices, it is meaningful for users to have tens of gigabytes more of storage space, taking advantage of data deduplication techniques.

2.2 High Costs of Existing Data Deduplication

There are many data deduplication technologies available for mobile systems [7, 10, 15, 17, 18]. For example, the widely-used DmDedup [14] is an efficient block-level deduplication solution for Linux-based systems, such as Android [13]. DmDedup saves the fingerprints of all data blocks to ensure that every write instance coming to DmDedup is deduplicated against the previously written data. Similarly, SmartDedup [16] also maintains the fingerprints of all data blocks. Differently, SmartDedup uses a two-level architecture to reduce the run-time memory footprints of fingerprints by caching part of the fingerprints in the memory and storing the whole fingerprint table in the storage.

Nevertheless, existing data deduplication methods cause a high overhead for managing the useless fingerprints of unique data in mobile systems with low duplication ratios. Taking DmDedup as an example, when writing 100GB of data that the deduplication ratio is 20%, DmDedup has to maintain a 540MB fingerprint table where each entry of a data block is 24B for all the data blocks. In the fingerprint table, the size of fingerprints of unique data blocks is 480MB, which will not be used in the future. As a result, it is inevitable to cause high overhead for data deduplication with such a big table in mobile systems by either maintaining the whole table in the memory or swapping the required fingerprints from the storage. We believe that a promising low-overhead data deduplication for mobile systems with a low duplication ratio is to cut off the overhead of the useless fingerprints of unique data.

3 FinerDedup Design

3.1 Overview of FinerDedup

The main idea of FinerDedup is to reduce the costs of fingerprints by predicting and sifting out the unique data blocks that will never appear again in write operations. Specifically, FinerDedup uses the random forest [3] to predict the potential labels (i.e., *duplicate* or *unique*) of data blocks and decides whether to write the data blocks and save their corresponding fingerprints. Figure 2 shows the architecture of FinerDedup, which consists of four major components: *Data block classifier*, *Bloom filter*, *block address mapper*, and *fingerprint table in main memory*.

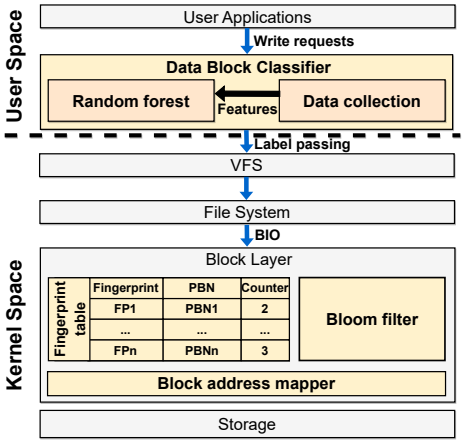


Figure 2: Architectural view of FinerDedup

Data block classifier. FinerDedup classifies the incoming data blocks into two categories, i.e., duplicate data blocks and unique data blocks, using the data block classifier. The data block classifier

takes part of the metadata attributes of a data block and its associated directory as features. Based on these features, we collect daily write data from users for incrementally training the model. The training of the model of the data block classifier only occurs when the device is idle. During runtime, when receiving a write request from a user application, the classifier first collects the feature of the corresponding data block in *libc*. Then, the classifier predicts whether the data block is a potential duplicate one. Finally, the data block classifier transmits the predicted label of the requested data block to the block layer via a critical write path.

Bloom filter. Similar to other learning models, the model in the data block classifier may make false predictions, especially for the data blocks marked as *unique* by the model but actually are duplicate ones. FinerDedup handles such false predictions using a dedicated Bloom filter [2, 8]. The Bloom filter is a space-efficient probabilistic data structure used to test whether an element is a set member. It mixes a long binary vector and a series of random hash functions. The Bloom filter receives the fingerprints of data blocks with *unique* labels as input to check whether these fingerprints have appeared in the past.

Block address mapper. Situated at the device mapper layer, the block address mapper eliminates the need to write data blocks with identical fingerprints into the storage. FinerDedup modifies the mapping from the logical block numbers (LBNs) to the physical block numbers (PBNs) to establish a mapping where multiple logical blocks with the same fingerprint are associated with the same physical block.

Fingerprint table in main memory. The fingerprint table is used to store and retrieve the hash values, i.e., fingerprints, associated with the stored data blocks. Additionally, it adds the counters for handling *False positive* predictions and maintains the Hash-PBN mapping, which represents the relationship between the fingerprints of data blocks and their corresponding physical block numbers. FinerDedup relies on querying and comparing the fingerprints of the blocks to identify identical data. The fingerprint table empowers FinerDedup to determine whether the content of a data block has appeared before or is appearing for the first time.

3.2 Feature-based Data Deduplication

In this section, we introduce the methods for collecting training data, selecting features, and labeling data, as well as the training of machine learning models and the metrics for analyzing the model.

Data collection. The initial training data for the data block classifier is sourced from the daily application usage of device users. We capture the features of write data blocks from all applications and calculate their fingerprints. With a large number of applications on mobile devices that evolve quickly, various apps may exhibit unique user behavior patterns. Therefore, customizing models for particular applications is costly and unnecessary. We modify the I/O critical path to support tracing users' daily data writing. We save the captured data features and fingerprints in the form of files.

Feature extracting and data labeling. After exploring various input features, we have determined to use the metadata attributes of data blocks and their respective directories as training features. Considering the limited resources of mobile devices, the number of features should not be excessive. Additionally, not all features are essential for model prediction, so we use Recursive Feature Elimination (RFE) to help us filter features. During our design iterations, we identify the three most essential features, which are: *d_ino* (directory inode number), *who* (owner of the file), and *pos* (current writing position offset in a file). All the features mentioned above are associated with user behavior. The accuracy can be improved by adding features about *file_mode* (permissions mode for a file or directory), *dentry_flags* (flags indicating dentry attributes), and *open_flag* (flags for file open operations). Furthermore, we traverse the fingerprints in the file to determine whether the label for each block is *duplicate* or *unique*. According to our tests, it takes about 1 minute to label 4GB of user write data. Finally, using the aforementioned features and labels can enable the random forest to achieve an overall accuracy rate of over 92%.

Machine learning based categorization. Considering the limited resource constraints on mobile devices, we need a high-precision categorization model with acceptable overhead. After evaluating a multitude of machine learning models, the data block classifier uses random forest as the categorization model to learn the input features. A random forest is a combination of multiple decision trees that harnesses the capabilities of multi-class decision trees for decision-making purposes. The model's training typically occurs during the device's idle time, so it does not affect normal user operations. We incrementally train and update the model with user data collected daily to ensure the model adapts to the users' daily behavior patterns. According to our tests, training with 4GB write data takes about 2 minutes on average, which is acceptable.

To avoid unnecessary overhead caused by extensive feature passing, we modify the critical write path to only pass the labels to the block layer. When subsequent write requests occur, we use the trained model to predict the labels of write data blocks. The transmission of data labels consists of two parts: from the user space to the kernel file system and from the kernel file system to the block layer. In user space, we modify the *libc write* function to pass the labels through a system call to the kernel and write them into the cache. The file system dispatches a *BIO* structure to the block layer, through which we convey the labels to the block layer.

Metrics for analyzing categorization model. The categorization model marks the data blocks as two types: *unique* or *duplicate*. In FinerDedup, we label data as *unique* if there is a greater probability that it will not appear again in the future. Otherwise, we label it as *duplicate* data. Unfortunately, the actual properties of the data blocks may be different. Hence, we divide the data blocks into four situations, as shown in Figure 3.

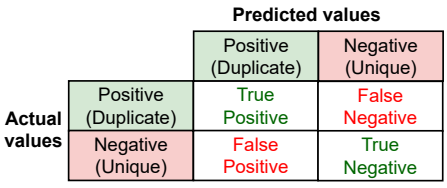


Figure 3: Confusion matrix

The true positive (TP) data blocks are marked as *duplicate* and will be written again in the future. The true negative (TN) data

blocks marked as *unique* and will not be written in the future. For both TP and TN data blocks, it is correct for us to record or discard the fingerprints of the data blocks. The false positive (FP) data blocks are marked as *duplicate* and will never be written in the future. FinerDedup erroneously records their fingerprints, which offers no benefits for future deduplication. Additionally, the increase in the fingerprint table affects the overall effectiveness of the entire deduplication process. We need to eliminate the redundant fingerprints caused by the FP data blocks. The false negative (FN) data blocks are marked as *unique* and will be written again in the future. FinerDedup incorrectly discards their fingerprints, leading to data blocks being redundantly written multiple times. Thus, we need a method to find out the FN data blocks to save storage space.

3.3 Handling False Predictions

As Section 3.2 mentions, there are two types of false predictions of data blocks: FP and FN. To minimize the cost of these two types of false predictions, FinerDedup uses the *counters* and the *Bloom filter* to handle the false predictions properly.

The Counters for handling false positive predictions. In the case of FP predictions, the categorization model of FinerDedup mistakenly regards the *unique* data blocks as duplicate data. As a result, the fingerprint table will consume more memory, and the retrieval time of fingerprints will be longer. To reduce the memory overhead caused by FP predictions, as shown in Figure 4, FinerDedup sets a reference counter for each fingerprint entry in the fingerprint table. This counter indicates whether the PBN (physical block numbers) has been shared. We modify the LBN-PBN mapping in the block address mapper to enable duplicate data blocks to share the same physical block. When a *duplicate* data block arrives, FinerDedup searches for the corresponding entry in the fingerprint table based on its fingerprint and then increments the entry's counter by 1. FinerDedup creates a new fingerprint entry and initializes the counter to 0 when a data block arrives for the first time. When a data block is modified, FinerDedup decrements the counter of its corresponding fingerprint entry by 1. Finally, FinerDedup sets a clearing period for the fingerprint table, regularly clearing entries with counter values 0. The duration of this period can be aligned with the frequency of model updates, such as one day.

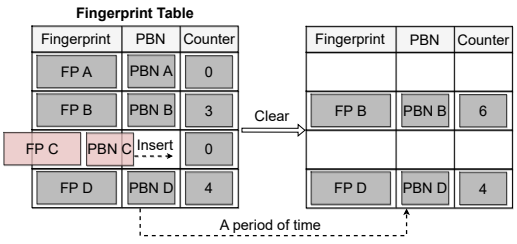


Figure 4: FP prediction handling

We provide an example to illustrate the function of the counter, as shown in Figure 4. ① The categorization model inaccurately predicts the *unique* data block associated with fingerprint C as *duplicate* data. Because this data block appears for the first time, FinerDedup inserts fingerprint C into an available entry in the fingerprint table. ② Next, the fingerprint table sets the clearing period to one day. ③ For fingerprints A and C, if their counter values remain at 0 after undergoing a clearing period, FinerDedup will clear their fingerprint entries.

A Bloom filter for handling false negative predictions. For the FN predictions, the categorization model of FinerDedup mistakenly marks the duplicate data as *unique*. The fingerprint table will not record the fingerprints of this type of data block, leading to redundant writing of duplicate data. Redundant writing exacerbates IO latency, consumes storage space, and diminishes system efficiency. FinerDedup uses the Bloom filter to identify FN predictions. The Bloom filter ascertains the presence of an input element through a binary vector and a series of hash functions. The Bloom filter takes as input the fingerprint of the data block labeled *unique* and maps it to certain bit positions in the vector through some hash functions. If all the bit positions value 1, it indicates that the *unique* data block has been previously saved, implying an erroneous prediction by the categorization model. Then, we consider to save the fingerprint for the data block. If the values at some bit positions are 0, indicating that the *unique* data block appears for the first time, then the Bloom filter needs to change these bit values to 1.

Bloom filter can accurately determine that a certain element is not in the set. However, there is a certain false positive rate when it determines an element is in the set, meaning elements not in the set are mistakenly identified as being in the set. This false positive rate depends on the size of the Bloom filter, the volume of elements input, and the quantity of hash functions. We can reduce the false positive rate by adjusting the size of the Bloom filter and the quantity of hash functions. Even if a misjudgment occurs, the only cost is storing an extra fingerprint.

3.4 Write Workflow

When FinerDedup receives a write request, as illustrated in Figure 5, it executes the following actions:

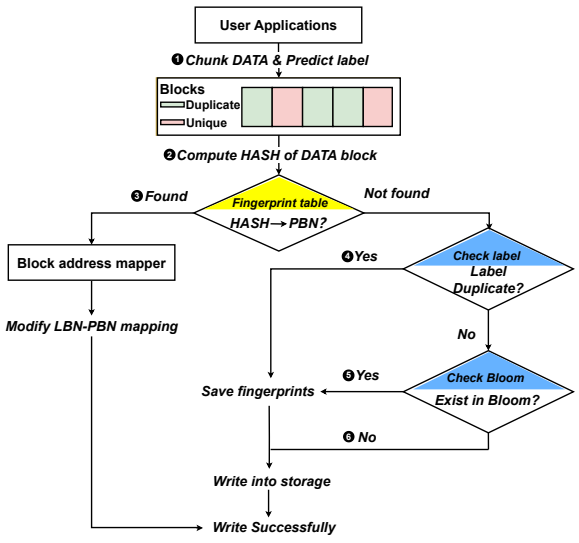


Figure 5: Write workflow of FinerDedup

❶ Upon an application sending a write request in user space, we collect the features of each data block. Using the categorization model, we obtain the labels of the data blocks. Next, we actively pass the label results to the block layer via the critical path of data writing. ❷ After the write data blocks arrive at the block layer, we compute their fingerprints and retrieve them in the fingerprint table. ❸ For data blocks with fingerprints already recorded in the fingerprint table, we only need to obtain their physical addresses from the table entries and establish the mapping relationship from logical addresses to physical addresses in the block address mapper. These types of data blocks will not continue to be written to storage. ❹ For data blocks with fingerprints not recorded in the fingerprint table, if the data block label is a *duplicate*, we save its fingerprint. ❺ If not, we check the Bloom filter to see whether the fingerprint has appeared previously. For fingerprints found in the Bloom filter, we save them in the fingerprint table and then modify the bit values in the Bloom filter. ❻ For fingerprints not found in the Bloom filter, we do not save their fingerprints. Finally, for step ❹, ❺ and ❻, we write the blocks into storage.

4 Evalution

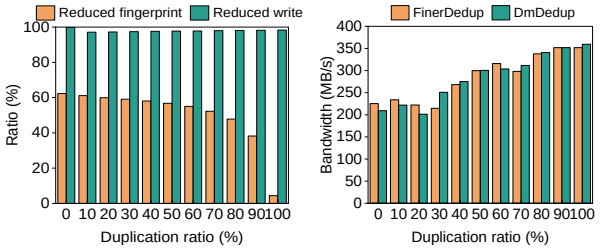
4.1 Environment Setup

To evaluate the effectiveness of FinerDedup, we implement FinerDedup on Google Pixel 3 smartphones composed of SDM 845, 4GB RAM, and 64GB flash storage. FinerDedup runs on Android 10 with Linux kernel version 4.9. We deploy FinerDedup and DmDedup [14] on these devices for evaluation. To comprehensively present the characteristics of FinerDedup, we compare the performance of FinerDedup to DmDedup, which is a traditional data deduplication method. We first evaluate and analyze the performance of FinerDedup with micro-benchmarks. Then, we evaluate the performance breakdown of FinerDedup. Finally, we run real-world applications on the smartphone to evaluate the performance of FinerDedup.

4.2 Micro-benchmarks

Utilizing FIO, we generate datasets with duplicate distributions ranging from 0% to 100%, divided into 11 distinct datasets. Each dataset comprises 300,000 blocks, each 4KB in size. As shown in Figure 6 (a), *Reduced fingerprint* represents the ratio of the difference in the number of fingerprints saved by FinerDedup and DmDedup to the number of fingerprints saved by DmDedup. *Reduced write* represents the ratio of the number of writes reduced by FinerDedup to the number of writes reduced by DmDedup. Regarding the volume of deduplicated data, Finerdedup reduces at least 50% of fingerprints compared to DmDedup while achieving at least 98% reductions in write operations. The reason for not achieving 100% reduced write is that DmDedup keeps the fingerprints of all write data, ensuring its reduced write rate is always 100%. In contrast, FinerDedup may have FP predictions, leading to a small amount of redundant writing.

As shown in Figure 6 (b), regarding write bandwidth, Finerdedup is superior to Dmdedup when the duplication rate is below 30%. Considering the common 20% data duplication rate on mobile devices, Finerdedup emerges as a superior option.

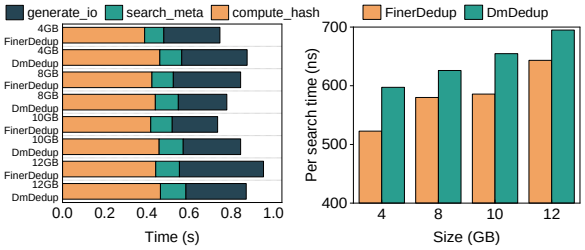


(a) Reduced fingerprints and writes (b) Bandwidth

Figure 6: Deduplication ratio and bandwidth using FIO.

4.3 Performance Breakdown

Figure 7 (a) illustrates the write latency of FinerDedup under various conditions. We write data sizes of 4GB, 8GB, 10GB, and 12GB to both systems to expand the fingerprints in memory. Subsequently, we calculate the time allocation for each phase.



(a) Write latency (b) Search time

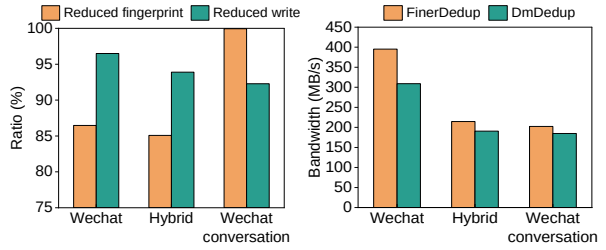
Figure 7: Breakdown of write latency and search time.

Figure 7(b) shows that the search time for each block in FinerDedup demonstrates a reduction over that in DmDedup, ranging between 50 and 70 ns. The main reason for the reduced search time is that FinerDedup maintains fewer fingerprints in memory, thereby enhancing retrieval efficiency. Despite the increase in data writing volume, the total search time increases only slightly because FinerDedup uses hash tables in memory to speed up indexing.

4.4 Real-world Application

We collect real application write data from users for analysis. As shown in Figure 8, we analyze three datasets: WeChat application write data, hybrid applications write data, and WeChat conversation write data. Furthermore, their data duplication rates are 28.2%, 7.8%, and 0.6%, respectively.

Figure 8 (a) shows the ratio of fingerprints and writes reduced by FinerDedup compared to DmDedup. The calculation methods for *Reduced fingerprint* and *Reduced write* are the same as those in Section 4.2. FinerDedup achieves a fingerprint saving rate of over 85% in real applications. The fingerprint saving rate for WeChat conversation is nearly 100% due to a high proportion of *unique* data blocks and a high recall rate of the model in predicting *unique* data blocks, resulting in relatively fewer fingerprints saved. Moreover, FinerDedup has attained over 90% in the reduced write operations



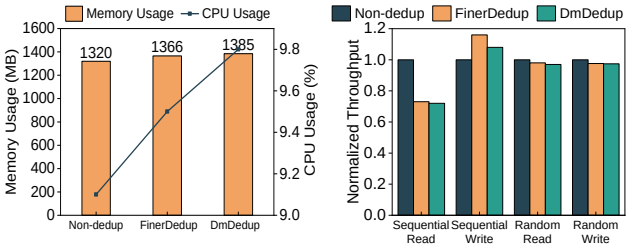
(a) Reduced fingerprints and writes (b) Bandwidth

Figure 8: Comparison of deduplication ratio and bandwidth using real-world applications.

ratio. Figure (b) shows that in scenarios with low data duplication rates on mobile devices, Finerdedup outperforms Dmdedup in write bandwidth across all worksets.

4.5 Overhead Analysis

System overhead. We set up a non-dedup device to compare the system overhead after equipping it with FinerDedup and DmDedup, respectively. The non-dedup device, a bare machine without data deduplication systems, uses the same configuration as in the experimental setup. We monitor the system CPU utilization using the *iostat* command and track the system memory usage with the *dumpsys* command. As shown in Figure 9 (a), the results indicate that, compared to the non-dedup device, FinerDedup incurs minimal memory overhead, not exceeding 3.4%, with an average of 2.2%. It occurs because FinerDedup selectively loads some of the data's fingerprints into memory. FinerDedup imposes minimal extra processing overhead on CPU utilization compared to the prototype system, with an average not exceeding 3.3%. Given the low system overhead, IO data deduplication could be a viable option for Android-based smartphones.



(a) Memory and CPU overhead (b) The normalized system throughput

Figure 9: Overhead analysis

Performance overhead. Figure 9 (b) presents the normalized system throughput under various conditions, contrasting the performance of the prototype system, FinerDedup, and Dmdedup. FinerDedup enhances sequential write throughput by 11.5% but reduces sequential read throughput by 30.0%. It illustrates the prevalent issue of read amplification in the data deduplication field. Data deduplication technology typically rearranges the layout of original data blocks to facilitate data sharing and conserve storage space. However, such rearrangement may result in dispersed data distribution across the storage space, consequently elevating the cost of read operations.

5 Conclusion

In this paper, we proposed a novel approach called FinerDedup to optimize the memory costs and retrieval efficiency of data deduplication. FinerDedup largely reduced the fingerprints by ignoring the predicted non-duplicate data blocks. We implemented and evaluated FinerDedup on real mobile systems. The experimental results proved that FinerDedup can effectively reduce the overhead of data deduplication on mobile systems.

Acknowledgments

This work is partially supported by Natural Science Foundation of China (Project No.62072059, 62102051, and 62372073), the Fundamental Research Funds for the Central Universities under Grant (Project No.2023CDJXY-039), and Chongqing Post-doctoral Science Foundation (Project No.2021LY75 and cx2023080). We would like to thank the anonymous reviewers for their valuable comments and improvements to this paper.

References

[1] 2017. *Fio: flexible i/o tester.* https://fio.readthedocs.io/en/latest/fio_doc.html
[2] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. 13, 7 (1970).
[3] L. Breiman. 2001. Random Forests. *Machine Learning* 45 (2001), 5–32.
[4] Aaron Carroll and Gernot Heiser. 2010. An analysis of power consumption in a smartphone. In *ATC*.
[5] Apple Inc. 2023. *Merge duplicate photos and videos on iPhone.* https://support.apple.com/guide/iphone/merge-duplicate-photos-and-videos-iph1978d9c23/ios
[6] Huawei Inc. 2023. *Clean up phone storage.* https://consumer.huawei.com/en/support/content/en-us15921348/
[7] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *FAST*. 183–197.
[8] Guanlin Lu, Young Jin Nam, and David HC Du. 2012. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *MSST*. 1–11.
[9] Aqeel Mahesri and Vibhore Vardhan. 2004. Power consumption breakdown on a modern laptop. In *PACS*. Springer, 165–180.
[10] N. Mandagere, P. Zhou, and M. A. Smith. 2008. Demystifying Data Deduplication. In *The ACM/IFIP/USENIX Middleware Conference Companion*. 12–17.
[11] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. 2016. Using hints to improve inline block-layer deduplication. In *FAST*. 315–322.
[12] Bo Mao, Jindong Zhou, Suzhen Wu, Hong Jiang, Xiao Chen, and Weijian Yang. 2019. Improving Flash Memory Performance and Reliability for Smartphones With I/O Deduplication. *IEEE TCAD* 38, 6 (2019), 1017–1027.
[13] Chunlin Song, Xianzhang Chen, Duo Liu, Xiaoliu Feng, Xi Yu, Jiali Li, Yujuan Tan, and Ao Ren. 2022. CADedup: High-performance Consistency-aware Deduplication Based on Persistent Memory. In *ICCD*. 726–729.
[14] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. 2014. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*. Citeseer.
[15] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proc. IEEE* 104, 9 (2016), 1681–1710.
[16] Qirui Yang, Runyu Jin, and Ming Zhao. 2019. SmartDedup: Optimizing Deduplication for Resource-constrained Devices. In *ATC*. 633–646.
[17] Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, and Yaping Wan. 2010. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. In *IPDPS*. 1–12.
[18] Ke Zhou, Shaofu Hu, Ping Huang, and Yuhong Zhao. 2017. LX-SSD: Enhancing the lifespan of NAND flash-based memory via recycling invalid pages. In *MSST*. 1–13.