SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

April 2026, Vol. 69, Iss. 4, 142104:1–142104:17
https://doi.org/10.1007/s11432-024-4517-1

RefineDedup: efficient deduplication for mobile systems via application-wise learning

Wei LI[1], Xianzhang CHEN[1*], Xingjie ZHOU[1], Duo LIU[2], Yujuan TAN[1], Ao REN[1], Kan ZHONG[2] & Lei QIAO[3]

[1]*College of Computer Science, Chongqing University, Chongqing 400044, China*
[2]*School of Big Data & Software Engineering, Chongqing University, Chongqing 400044, China*
[3]*Beijing Institute of Control Engineering, Beijing 100190, China*

Received 7 October 2024/Revised 3 May 2025/Accepted 10 July 2025/Published online 11 February 2026

Abstract With the rapid increase in the size of mobile applications, data deduplication technologies are used to improve the storage capacity of mobile systems. However, existing data deduplication approaches show high memory footprints and performance costs on mobile systems due to the relatively low duplication ratios, which generally range from 10% to 35%, of mobile applications. The highly individualized and evolving data accesses of applications make data deduplication on mobile devices even more challenging. Thus, we propose RefineDedup, an efficient data deduplication method for mobile systems that leverages the application-wise data access behaviors. The basic idea of RefineDedup is to avoid the overhead of useless fingerprints of applications using tiny application-specific learning models. First, RefineDedup uses a forecasting sliding window to collect and analyze data accesses of applications. Then, RefineDedup trains application-specific models for the applications with high data duplication ratios, along with a shared model for the applications with low data duplication ratios. These models are effective in predicting the data access behaviors of applications. We implement RefineDedup on a smartphone running Android 10 and evaluate it with real-world applications. Extensive experimental results show that RefineDedup stores 33.4% fewer fingerprints than FinerDedup, the state-of-the-art data deduplication scheme for mobile systems. With a duplication ratio below 35%, RefineDedup's write bandwidth is on average 12.2% higher than that of FinerDedup.

Keywords data deduplication, application-wise learning, mobile systems, fingerprint optimization, storage design

Citation Li W, Chen X Z, Zhou X J, et al. RefineDedup: efficient deduplication for mobile systems via application-wise learning. Sci China Inf Sci, 2026, 69(4): 142104, https://doi.org/10.1007/s11432-024-4517-1

1 Introduction

The storage performance, capacity, and durability of mobile devices are typically constrained [1–5]. In this case, data deduplication technology effectively addresses these issues by eliminating redundant data and I/O operations [2, 6–11]. The mainstream manufacturers of mobile devices, such as Apple and Huawei, are trying to provide data deduplication services for users.

Traditional deduplication techniques, such as the widely used DmDedup [12], rely on extensive fingerprint tables to record and match data block hash fingerprints [9, 10, 13–15], thereby reducing redundant writes. However, as the volume of data written on devices escalates, the memory burden and retrieval costs associated with the fingerprint table are becoming increasingly untenable [1, 15–19]. To address this challenge, innovative techniques like APP-Dedupe [7] and SmartDedup [2] optimize memory usage of fingerprint tables. APP-Dedupe segments the fingerprint table into different groups, maintaining only the fingerprints of active foreground applications in memory. Meanwhile, SmartDedup facilitates rapid fingerprint queries in memory storage and supports comprehensive data deduplication through its disk-based storage.

However, studies indicate that the overall duplication ratio in mobile systems typically ranges from 10% to 35% [2], suggesting that most fingerprints will not be accessed again in the future. Although existing approaches reduce memory overhead, a significant number of invalid fingerprints may still be retained unnecessarily. Recently, FinerDedup [1] points out that the effectiveness of data block fingerprints is related to the data access behaviors of applications. It introduces a global categorization model to predict the duplicity of data blocks, selectively retaining fingerprints and thereby further optimizing the deduplication process. Nevertheless, FinerDedup is limited

* Corresponding author (email: xzchen@cqu.edu.cn)

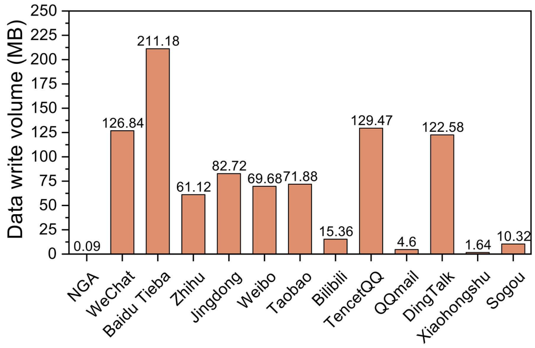

Figure 1 (Color online) The daily average volume of data writes across multiple mobile applications.

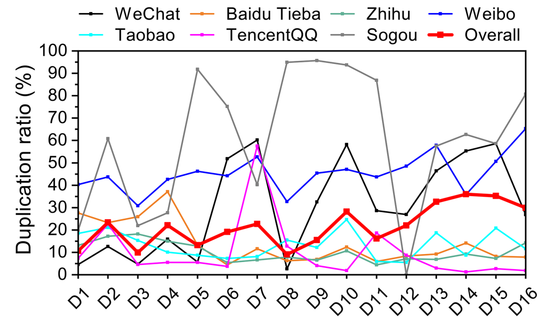

Figure 2 (Color online) Trends of duplication ratios of daily write data for multiple mobile applications. The *Overall* curve represents the total duplication ratio of daily write data.

in capturing the long-term data access behaviors [20, 21], which may affect the accuracy of deduplication and the overall system performance.

In this paper, we propose an efficient application-wise data deduplication method named RefineDedup, which is based on analyzing long-term data access behaviors of applications on mobile devices. Unlike traditional methods, RefineDedup employs a forecasting sliding window to analyze users' multi-day write data and application-specific categorization models of data blocks for various mobile applications. This approach can accurately identify the write data characteristics and usage trends of each application, thereby optimizing the deduplication process. The categorization models label data blocks as *unique* (unlikely to repeat) or *duplicate* (likely to repeat). RefineDedup then transmits these labels through the write critical path to the *Deduplicator*, retaining only the fingerprints of data blocks labeled as *duplicate*. We introduce the $(1+\epsilon)$-approximation algorithm to minimize performance loss due to false predictions. Additionally, we manage multiple models through the *Model Manager* using a dynamic loading strategy, ensuring efficient scheduling and minimal memory usage.

We implement RefineDedup on the Google Pixel 3 smartphone running Android 10. By tracking user write data for 15 d from 10 common mobile applications, we evaluate RefineDedup in terms of write effect, performance impact, and overhead. Experimental results demonstrate that RefineDedup stores 33.4% fewer fingerprints than FinerDedup and 88.1% fewer than DmDedup. With a duplication ratio below 35%, RefineDedup's write bandwidth is on average 12.2% higher than that of FinerDedup and 16.5% higher than DmDedup's. In terms of memory overhead, the memory usage of RefineDedup's individual categorization model is reduced by approximately 97.7% on average compared with FinerDedup's global model, with a maximum reduction of nearly 99%. The major contributions of this paper include the following.

• We collect user write data from multiple popular applications on mobile devices and analyze data access behaviors from both temporal and spatial dimensions.

• We propose an efficient application-wise data deduplication method, RefineDedup, to save memory costs and reduce I/O latency by learning the long-term data access behaviors of mobile applications.

• We devise a set of efficient strategies to manage the models of applications, such as handling the false predictions of models and dynamically scheduling the models for applications during runtime.

• We implement RefineDedup on a real mobile system and evaluate it with real-world applications. Experimental results show that RefineDedup outperforms FinerDedup, the state-of-the-art data deduplication approach for mobile systems.

2 Background and motivations

2.1 Data duplication in mobile systems

We collect real write data in 4 kB units from users on mobile devices over a period of 16 d. We chart the daily average volume of data writes for various applications and the trends of the duplication ratio of daily write data, as illustrated in Figures 1 and 2. The applications with near-zero duplication ratios are omitted from Figure 2 to enhance readability.

First, mobile devices host a large and diverse array of applications, each exhibiting significant differences in data access behaviors. In terms of data write volume, social applications like TencentQQ and WeChat exhibit significantly higher data write volumes compared with other applications, largely due to their frequent daily use by most users. From the perspective of duplication ratios, individual applications such as Sogou may exhibit a write duplication ratio as high as 90%, whereas QQmail displays a ratio close to 0%. Additionally, as depicted in the *Overall* curve in Figure 2, the daily overall duplication ratio exhibits a certain periodic trend. Moreover, the overall duplication ratio generally varies between 10% and 35%, averaging at 22.3%.

Second, as existing applications are updated and new applications are introduced, their data access behaviors may change over time. For example, TencentQQ exhibited an average duplication ratio of approximately 34.1% in its 8th version [6]; after upgrading to the 9th version, which introduces enhanced file transfer functions, its average duplication ratio decreases to 10.1%. It is important to note that such changes are often driven by new features introduced during the applications' updates, including the integration of third-party applications. These new features, such as enabling file transfer applications like DingTalk and WeChat to interact with third-party file editing applications, such as WPS and Office, can introduce new data access patterns. Therefore, fluctuations in duplication ratios are closely related to whether new data behaviors are introduced through application updates or functionality additions.

Third, for some applications, the duplication ratio trends exhibit noticeable fluctuations over time. These variations are attributed to differences in user frequency and the features utilized. For example, if WeChat users frequently engage the file transfer feature on a particular day, this can result in a higher data duplication ratio due to multiple file copies generated during the transfer process. The variability in the usage behaviors of applications complicates the analysis of data access behaviors.

2.2 Motivation

Many deduplication technologies are currently employed on mobile devices. A common characteristic of early deduplication methods is the maintenance of fingerprints for all data blocks, ensuring that each write instance deduplicates the previously written data [13–15, 22], even though the average duplication ratio on mobile devices is only 22% [23, 24]. However, these methods result in the creation of many unused fingerprints, leading to unnecessary memory overhead. The inefficiency arises because previous deduplication methods focus solely on the data itself [17, 25–28], without considering the impact of mobile applications' data access behaviors on deduplication performance. In fact, the duplication ratios of different applications are not fixed and usually change with application updates and usage behaviors across different times. Therefore, focusing solely on the redundancy of the data itself does not fully improve the deduplication efficiency. Although an advanced method [1] predicts the duplication of data blocks and selectively saves fingerprints to reduce memory overhead, it focuses primarily on global data deduplication and overlooks the factors mentioned above. This results in less refined fingerprint storage, leading to unnecessary fingerprints and additional memory overhead.

Considering the importance of mobile storage space and performance [29–31], and our observations of long-term data access behaviors of applications on mobile devices, we propose the RefineDedup method. RefineDedup trains application-specific categorization models for multiple mobile applications, specifically aiming to capture the behavioral characteristics each application exhibits during user engagement. This approach significantly reduces memory overhead by filtering fingerprints while improving the storage efficiency of mobile devices.

3 RefineDedup design

In this section, we first outline the main design objectives of RefineDedup. Then, we present the overall architecture and the design details of RefineDedup.

3.1 Design goals

RefineDedup aims to achieve the following three objectives.

Enhancing storage space utilization. Application-specific categorization models are designed to identify and remove duplicate data blocks, enhancing the space efficiency of storage in mobile devices.

Improving mobile applications' performance. By using application-wise I/O deduplication to remove a significant amount of redundant write data, RefineDedup enhances the I/O bandwidth and overall performance of applications.

Reducing memory overhead. The memory overhead from fingerprint storage can be reduced by RefineDedups prediction of application-specific models that filter out unnecessary fingerprints. This increases the overall system's response speed and efficiency.

3.2 Overview of RefineDedup

The main idea of RefineDedup is to use predictions from machine learning models to identify and exclude unique data blocks that will never be written again, thereby reducing the memory overhead of fingerprints and minimizing redundant data writes to storage as much as possible. Figure 3 shows an overview of our proposed RefineDedup system architecture in Android-based smartphones.

RefineDedup consists of three main parts: *Data Collector*, *Model Manager*, and *Deduplicator*. The *Data Collector* primarily collects user data and selects features. The *Model Manager* is primarily responsible for training and scheduling multiple application categorization models. Unlike FinerDedup [1], which maintains a global model in memory, RefineDedup adopts a customized approach by maintaining an application-specific model for each application. This approach allows the model to more effectively capture the unique data access behaviors of each application. In terms of *Deduplicator*, RefineDedup transmits the data block labels predicted by the models to the block level for I/O deduplication. Simultaneously, it uses a Bloom filter and counters to minimize the additional overhead caused by false predictions. We will next discuss the design details of each part.

3.3 Data collection and feature selection

The *Data Collector* is responsible for gathering training data. The training data for categorization models in RefineDedup originates from the daily use of mobile devices. When users actively browse or run background tasks such as playing music or videos, these activities generate varying amounts of write data. We modify the libc write function to trace each write request made by an application, saving the metadata attributes of 4 kB data blocks as a daily trace file. Unlike FinerDedup, which employs the method of previous-day collection [1], RefineDedup employs a forecasting sliding window to collect data and train models, as illustrated in Figure 4.

RefineDedup gathers traces over a specified period, known as the data collection window, to train the model and predict the next day's labels. We can set an upper limit for the window size, such as 7 d. The window expands daily until it reaches this limit, then maintains a fixed size and slides forward. This ensures that the new user trace is incorporated daily and the old trace is discarded. During the data collection, we use a random sampling strategy to ensure that the amount of data in the window is within an acceptable range.

RefineDedup selects four features for model training: *fingerprint* (unique identifier of data block content), *filename* (name of a file), *d_ino* (directory inode number), and *date* (trace collection date). Our analysis of user write data reveals that most duplicate data blocks are found in specific directories, files, and content types. For instance, duplicates in the WeChat application frequently appear in the *Userdatafiles* directory, which holds all files received through network transmissions. Notably, RefineDedup selects the date on which the user trace is generated as the *date* feature, enabling the model to identify trends in data access behaviors over time. Upon completing trace collection, we label each written data block based on its *fingerprint* feature. Data blocks sharing the same fingerprint are marked as *duplicate* and labeled as 1, while those with a unique fingerprint are marked as *unique* and labeled as 0.

3.4 Application-wise model training

One of the responsibilities of *Model Manager* is to train the categorization models of data blocks for each application. When write requests occur, the trained models are employed to predict the labels of the write data blocks. We adopt random forest [32], which is well-suited for resource-constrained mobile devices, providing high accuracy with low overhead [1], as our basic categorization model.

The duplication ratio observed in mobile applications reflects data access behaviors of applications [33–36]. For example, most duplicated data blocks in WeChat have a *d_ino* that corresponds to the inode number of the *Userdatafiles* directory. This suggests that users frequently utilize the file transfer assistant, which tends to create multiple backups in the *Userdatafiles* directory. Therefore, to capture the data access behaviors of applications, we adopt a customized approach for mobile applications. As shown in Figure 5, RefineDedup trains application-specific models of random forest based on the data duplication ratio of each application. RefineDedup first calculates the global duplication ratio of all user write traces within the data collection window. Then, it segments the write trace by UID [37] and computes the duplication ratio for each application. Each application on Android has a unique and unchangeable UID, which identifies the source application of a data block when examining its UID field. Then, RefineDedup divides the applications into two categories according to their respective duplication ratios.

Application-specific models for high-redundancy applications. Applications with a duplication ratio higher than the global ratio are classified as high-redundancy. For high-redundancy applications, RefineDedup trains a dedicated random forest model using their specific write traces. Each high-redundancy random forest

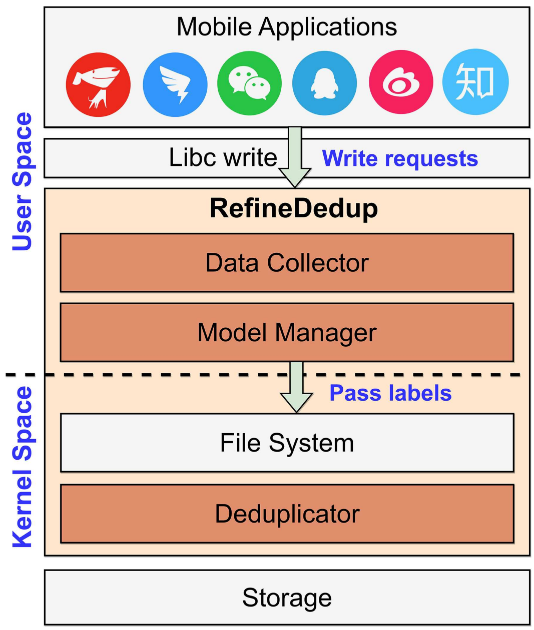

Figure 3 (Color online) System architecture of RefineDedup.

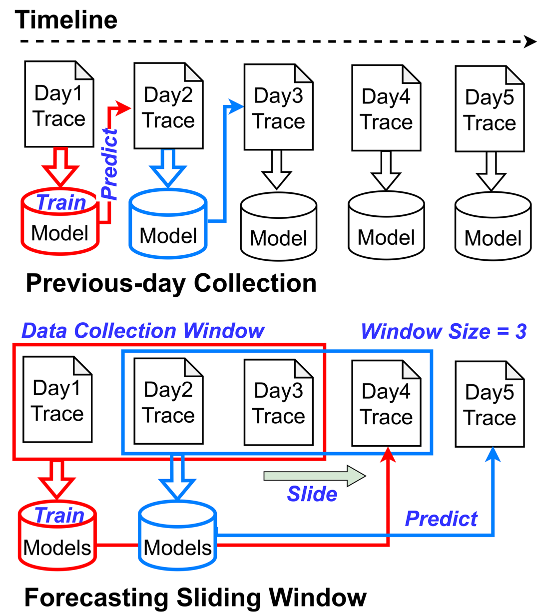

Figure 4 (Color online) Comparison of two data collection methods. RefineDedup uses a forecasting sliding window, while FinerDedup uses a method of previous-day collection. FinerDedup collects the prior day's write trace to train the model and predict the next day's label of the data block.

model has its own unique parameters. Those write data blocks from high-redundancy applications are fed into their respective application-specific models for inference.

Shared model for low-redundancy applications. Applications with ratios below the global ratio are classified as low-redundancy. For low-redundancy applications, RefineDedup aggregates their write traces to train a shared random forest model. Those write data blocks from low-redundancy applications are fed into the shared model for inference. According to our assessment, low-redundancy applications often have a duplication ratio of less than 10%, leading to a highly imbalanced class distribution within the training data. This imbalance can cause the model to overlook minority classes. Combining training traces from multiple low-redundancy applications helps enrich the sample of duplicate data, addressing the issue of class imbalance.

RefineDedup trains and updates models daily during device idle times, typically at night. Upon completion, RefineDedup saves the models as inference binary trees, coded in the C language, which reduces the overhead associated with model inference and loading. When a new application is installed, RefineDedup collects at least one day's write data from this application before categorizing it by duplication ratio and training a corresponding model. If an application undergoes version iteration, the fact we observed in Subsection 2.1 may occur, namely, the data access behaviors of the application change with its version iteration. In this case, RefineDedup treats the application after iteration as a newly installed application.

3.5 Dynamic model loading

The other responsibility of *Model Manager* is to load the corresponding model for the current application. We adopt a dynamic model loading strategy, as illustrated in Figure 6. RefineDedup maintains a mapping table in memory that associates application UIDs with model loading paths. Since low-redundancy applications use a shared model, this mapping supports many-to-one relationships. The table is updated daily to reflect changes in the models.

When the libc write function intercepts a write request from an application, RefineDedup retrieves the UID field of the write data block and consults the mapping table to determine which model to load into the model pool for inference. The model pool, a dedicated segment of memory, manages the loading and releasing of models. If a model is loaded for the first time, RefineDedup sets its *is_used* field from 0 to 1, indicating that the model is active in the model pool, and records the model's memory address in the *add* field. The model is loaded into the model pool only once. Subsequent accesses by the application are made through the *add* field.

As some active applications do not consistently generate write requests, keeping their models resident increases

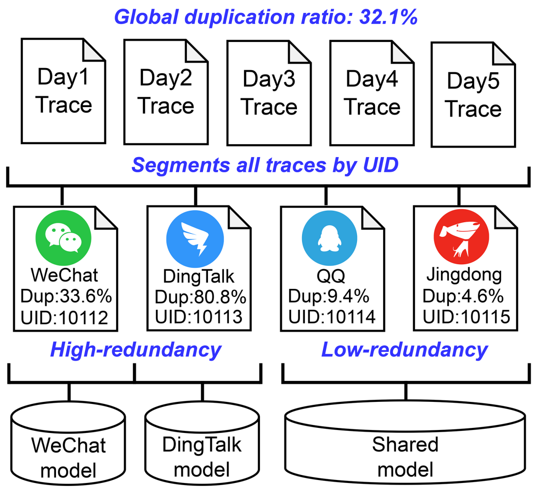

Figure 5 (Color online) RefineDedup segments traces and trains application-specific models based on the data duplication ratio of each application.

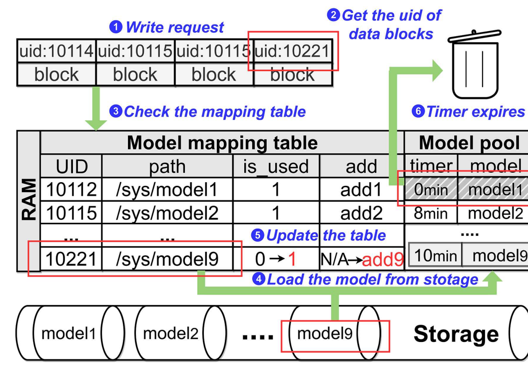

Figure 6 (Color online) RefineDedup's dynamic model loading.

memory overhead. Consequently, we employ a low-overhead timer for every model to precisely control its residency period in the model pool, setting the countdown to 10 min. When the timer expires, RefineDedup releases the model and sets its *is_used* field to 0. If an application continuously generates write requests, RefineDedup will reset the timer. Simply put, if an active application fails to generate write data for up to 10 min, its model's memory will be cleared.

Typically, the model is initially loaded during the cold start of an application. To prevent this loading from extending the cold start time, we configure the random forest model with fewer parameters, yet ensure it maintains effective prediction performance. RefineDedup's random forest consists of only 100 decision trees and has a maximum depth of 10, balancing complexity and performance efficiently. Consequently, with such a significantly lower parameter count, RefineDedup ensures that the increase in cold start time is limited to approximately 0.15 s.

3.6 Learning-based data deduplication

The *Deduplicator* is responsible for learning-based data deduplication. The core concept involves recording only the fingerprints of data blocks labeled as *duplicate*, which are expected to be rewritten in the future, and discarding those labeled as *unique*, which are not anticipated to be reused.

3.6.1 *Application-wise data deduplication*

The *Deduplicator* mainly comprises three components: application fingerprint tables in memory, a block address mapper, and a Bloom filter [38–40], as illustrated in Figure 7.

To achieve application-wise data deduplication, we allocate fingerprint tables to multiple applications, using UID as the indices of each table. The application fingerprint tables primarily store and retrieve hash values, which serve as the fingerprints of data blocks. They maintain a hash-PBN (physical block number) mapping, which represents the relationship between the fingerprints of data blocks and their corresponding physical blocks. By querying and comparing data block fingerprints in fingerprint tables, RefineDedup can identify identical data, determining whether the content of a data block has previously appeared or is appearing for the first time.

The block address mapper, located in the device's mapper layer, avoids writing redundant data blocks to storage by modifying the mapping from logical block numbers (LBNs) to physical block numbers (PBNs). This adjustment allows multiple logical blocks with the same fingerprint to map to the same physical block.

The Bloom filter, a space-efficient probabilistic data structure, combines a long binary vector with a series of random hash functions. RefineDedup uses it to test whether an element is part of a set. It receives fingerprints of data blocks labeled as *unique* to check whether these have previously appeared.

To pass labels to the *Deduplicator* in the block layer, we modify the critical write path. The transmission of data labels consists of two stages: from user space to the kernel file system, and from the kernel file system to the block layer. In the user space, we modify the libc write function through a system call to pass labels to the kernel and record them in the cache. Once data reaches the page cache, the file system transmits labels and UIDs to the block

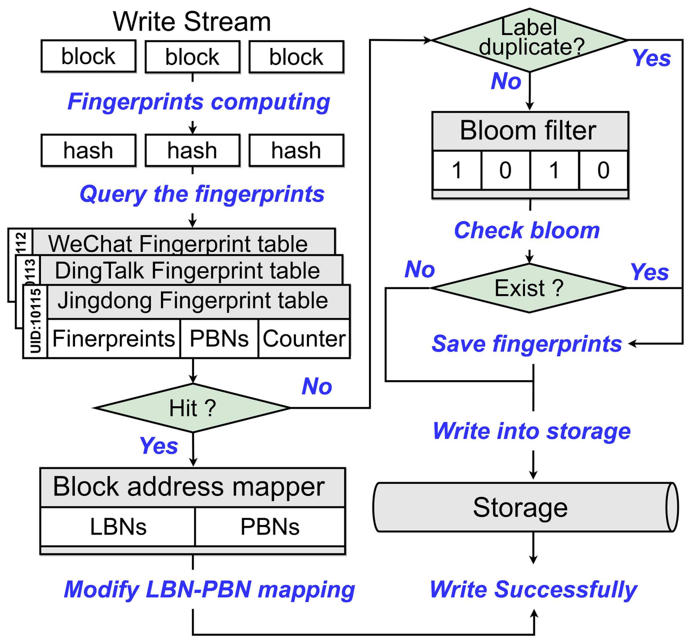

Figure 7 (Color online) The workflow of the *Deduplicator*. The *Deduplicator* consists of three parts: application fingerprint tables in memory, a block address mapper, and a Bloom filter.

layer through BIO (block I/O) structures, which represent I/O operations in the Linux kernel and facilitate data transfer between the kernel and block devices. We pass a data block's UID to the block layer to clearly distinguish the source of the data block. The BIO structure is modified and recompiled to enable the implementation of the above functions. Next, block chopping is performed on these BIOs. After chopping, the *Deduplicator* determines the next write operation for the data block according to the deduplication strategy shown in Figure 7.

3.6.2 *Handling false predictions*

RefineDedup categorizes erroneously predicted data blocks into two types: false positive (FP), where blocks are incorrectly labeled as *duplicate*, and false negative (FN), where duplicates are incorrectly labeled as *unique*. The other two desired categories are true positives (TP) and true negatives (TN). TP data blocks are accurately marked with a *duplicate* label and are expected to be written in the future, whereas TN data blocks are correctly marked with a *unique* label and will not be reused. For TP and TN data blocks, it is appropriate to either record or discard the fingerprints of the blocks, depending on their future use.

FP data blocks, erroneously marked with a *duplicate* label and not expected to be written in the future, have their fingerprints recorded, resulting in additional memory overhead. To minimize this overhead, RefineDedup employs a reference counter for each fingerprint entry in the application fingerprint table, as shown in Figure 7. This counter reflects the number of times the corresponding physical block is shared. RefineDedup clears entries in the fingerprint table with a share count of zero when releasing an application model's memory in the model pool. This strategy significantly reduces the memory overhead caused by false positive predictions for a single application model.

FN data blocks are those erroneously marked with a *unique* label but are expected to be written again in the future. RefineDedup mistakenly discards these blocks' fingerprints, which leads to redundant writing. To reduce the redundant writes caused by FN predictions, RefineDedup must determine whether to store the fingerprints for the FN data blocks that have been written once or multiple times. We adopt the false negative remedy (FNR) strategy, as shown in Algorithm 1, which aims to minimize the cost associated with FN predictions.

The write threshold $X_{\rm w}$ represents the maximum number of consecutive writes allowed for a certain FN data block before data deduplication occurs. This threshold dictates when a fingerprint should be recorded to prevent redundant writes. $w(p)$ represents the number of times data block p has been written. If $w(p)$ is less than $X_{\rm w}$, RefineDedup performs a simple write operation and increments the write count. Once $w(p)$ reaches $X_{\rm w}$, RefineDedup not only writes the data but also records the fingerprint of the data block to facilitate future deduplication. The write threshold $X_{\rm w}$ is crucial in RefineDedup, as it influences both the write strategy and the deduplication efficiency of data blocks. We propose a $(1+\epsilon)$-approximation algorithm to determine the value of $X_{\rm w}$, optimizing RefineDedup's performance and maximizing its benefits. The $(1+\epsilon)$-approximation algorithm is employed to find

Algorithm 1 FNR strategy in RefineDedup.

Require: The fingerprint r of data block p, predicted label $\hat{l}$, write threshold X_{w};
Ensure: The processed data block p;
if r is in fingerprint table **then**
  Modify LBN-PBN mapping for p;
else
  if $\hat{l}$ is a *unique* label **then**
    if The write count $w(p) < X_{\mathrm{w}}$ **then**
      Increment $w(p)$ by 1;
    else
      Record r in fingerprint table;
    end if
  else
    Record r in fingerprint table;
  end if
  Write p;
end if

near-optimal solutions to optimization problems where calculating an exact solution is computationally expensive. In our case, directly calculating the optimal X_{w} requires analyzing the entire lifecycle of the data block, including all previous writes and deduplication history. This process can become prohibitively expensive in terms of time and computational resources, especially for large writes and in mobile systems where performance is critical. Therefore, we use the $(1+\epsilon)$-approximation algorithm, which provides a solution close to the optimal threshold with significantly lower computational overhead, ensuring that the deduplication process remains efficient.

Assuming that the time costs for writing one data block into storage, calculating one block's hash value, and updating the fingerprint table once are T_{w}, T_{h}, and T_{m}, respectively. The worst cumulative time cost for a duplicate data block judged as an FN block n consecutive times over its entire lifecycle can be summarized as follows:

$$C_{\text{worst}} = n \times (T_{\mathrm{w}} + T_{\mathrm{h}}) + \sum_{i=1}^{n} f(g(i)). \tag{1}$$

The data block has undergone n write operations in total. We use i (where $i \leqslant n$) to represent the sequence number at which this data block appears. The function $g(x)$ describes the relationship between the sequence number and the size of the fingerprint table. When the data block appears for the ith time, the fingerprint table size maintained by the system is $g(x)$. Additionally, $f(y)$ correlates the size of the fingerprint table with the time required to retrieve a fingerprint, representing the search time when the table contains y entries. For this duplicate data block, if there are X_{w} instances of FN predictions, its fingerprint will be recorded at the next occurrence. Consequently, the cumulative time cost of this data block can be expressed as follows:

$$C(n, X_{\mathrm{w}}) = n \times T_{\mathrm{h}} + (X_{\mathrm{w}} + 1) \times T_{\mathrm{w}} + T_{\mathrm{m}} + \sum_{i=1}^{n} f(g(i)). \tag{2}$$

Cost $C(r, X_{\mathrm{w}})$ is a monotonically increasing function of X_{w}; if it is possible to accurately determine that an FN data block is a duplicate upon its first arrival and record its fingerprint, then the cost can be reduced. We consider the impact of the value n on the optimal cumulative cost; when $X_{\mathrm{w}} = 0$, the optimal cumulative cost is

$$C_{\text{opt}} = \begin{cases} n \times T_{\mathrm{h}} + T_{\mathrm{w}} + T_{\mathrm{m}} + f(g(1)) + f(g(2)), & \text{if } n = 2, \\ n \times T_{\mathrm{h}} + T_{\mathrm{w}} + T_{\mathrm{m}} + \sum_{i=1}^{n} f(g(i)), & \text{if} n > 2. \end{cases} \tag{3}$$

However, in RefineDedup, the C_{opt} is unachievable because we cannot immediately determine if an FN data block is a duplicate upon its first arrival. When $n > 2$, the best practice is to record the fingerprint of the data block at the second writing to minimize the impact of misjudgments as much as possible. When $n = 2$, the optimal strategy involves not recording the fingerprint for the FN data block, as its fingerprint will not be used again.

According to (2) and (3), when $n = 2$, the ratio of RefineDedup's cumulative cost to the optimal cumulative cost is

$$\begin{aligned} R_1 &= \frac{2 \times (T_{\mathrm{w}} + T_{\mathrm{h}}) + \sum_{i=1}^{2} f(g(i))}{T_{\mathrm{w}} + 2 \times T_{\mathrm{h}} + T_{\mathrm{m}} + \sum_{i=1}^{2} f(g(i))} \\ &= 1 + \frac{T_{\mathrm{w}} - T_{\mathrm{m}}}{T_{\mathrm{w}} + 2 \times T_{\mathrm{h}} + T_{\mathrm{m}} + \sum_{i=1}^{2} f(g(i))}. \end{aligned} \tag{4}$$

Hence,

$$\epsilon_1 = \frac{T_{\mathrm{w}} - T_{\mathrm{m}}}{T_{\mathrm{w}} + 2 \times T_{\mathrm{h}} + T_{\mathrm{m}} + \sum_{i=1}^{2} f(g(i))}. \tag{5}$$

When $n > 2$, the ratio of RefineDedup's cumulative cost to the optimal cumulative cost is

$$\begin{aligned} R_2 &= \frac{2 \times T_{\mathrm{w}} + n \times T_{\mathrm{h}} + T_{\mathrm{m}} + \sum_{i=1}^{n} f(g(i))}{T_{\mathrm{w}} + n \times T_{\mathrm{h}} + T_{\mathrm{m}} + \sum_{i=1}^{n} f(g(i))} \\ &= 1 + \frac{T_{\mathrm{w}}}{T_{\mathrm{w}} + n \times T_{\mathrm{h}} + T_{\mathrm{m}} + \sum_{i=1}^{n} f(g(i))}. \end{aligned} \tag{6}$$

Hence,

$$\epsilon_2 = \frac{T_{\mathrm{w}} - T_{\mathrm{m}}}{T_{\mathrm{w}} + n \times T_{\mathrm{h}} + T_{\mathrm{m}} + \sum_{i=1}^{n} f(g(i))}. \tag{7}$$

In RefineDedup, the value of ϵ is a fixed constant, reflecting the maximum cost deviation permitted by the system. According to our estimates, T_{w} is approximately 2029 ns, T_{h} is approximately 3546 ns, T_{m} is approximately 0.89 ns, and the average search time is about 470 ns. At $n = 2$, because T_{m} and T_{w} differ by several orders of magnitude, the cost difference—whether adding a fingerprint or not—is minimal. This makes ϵ_1 close to ϵ_2 and approximately equal to 0.2. This indicates that the cost of recording the fingerprint for the second write of an FN data block and not recording the fingerprint for misjudged data blocks is essentially the same. At $n > 2$, as n increases, the gap between the cumulative cost and the optimal cumulative cost becomes progressively smaller. Although data blocks may appear multiple times, recording the fingerprint during the second write remains the most effective strategy. This reduces the cost of redundant writes and processing, thus approaching the C_{opt} while keeping the error within the permissible range of ϵ_2. Therefore, RefineDedup selects a write threshold $X_{\mathrm{w}} = 1$ to minimize the performance impact caused by FN predictions as much as possible.

As shown in Figure 7, RefineDedup uses a Bloom filter to detect FN predictions by hashing the fingerprints of data blocks labeled as *unique*. When a data block labeled as *unique* passes through the Bloom filter's hash mapping and all corresponding bits are set to 1, it indicates that the block has reached the write threshold. Consequently, RefineDedup records its fingerprint. If any bits are 0 after hashing, this indicates that the block is appearing for the first time, prompting the Bloom filter to update these bits to 1.

3.6.3 *Handling reads and deletes*

In non-deduplicated systems, read and deletion operations are straightforward and rely solely on the block address mapper. However, RefineDedup maintains additional fingerprint tables for data deduplication, which makes these operations more complex to implement.

In the case of read operations, RefineDedup does not perform deduplication on data blocks. It uses an in-memory read buffer to optimize data accesses. The buffer caches recently read data blocks, allowing faster reads from the buffer compared to storage. When a read request is issued, RefineDedup first checks whether the requested data block is present in the buffer. If it is, the data is read directly from the buffer. If not, RefineDedup reads the data from storage. In this case, RefineDedup looks up the block address mapper to find the PBN corresponding to the LBN. It then reads the data block from storage using the PBN and returns it to the user.

For deletion operations, RefineDedup handles two scenarios. RefineDedup includes a reference counter in the fingerprint entry, as discussed in Subsection 3.6.2. The counter is primarily used to determine whether the physical block is shared during the deduplication process. If a physical block is not duplicated, the corresponding counter remains 0. This indicates a one-to-one mapping between the LBN and PBN of the data block. If duplication occurs, the counter is incremented by 1. A non-zero counter indicates a many-to-one mapping, meaning multiple LBNs map to the same PBN. RefineDedup handles deletion operations based on different mappings.

The LBN and PBN are in a one-to-one mapping. This situation happens when the label of the data block is *duplicate* but no duplication has occurred, which means the physical block is not shared. When a data block needs to be deleted, RefineDedup first determines the PBN corresponding to the LBN in the block address mapper. The PBN is then retrieved from the fingerprint tables to check whether its reference count is 0 or if it does not exist. The PBN may not exist in the fingerprint tables because RefineDedup periodically removes fingerprint entries with a reference count of 0 to handle FP prediction. Finally, the physical block corresponding to the PBN is marked as invalid, and its entries in both the block address mapper and fingerprint tables are cleared.

The LBN and PBN are in a many-to-one mapping. This situation indicates that data blocks are duplicated, which means multiple logical blocks share the same physical block. When a data block needs to be deleted,

Table 1 Information about 10 mobile applications.

Application	Version number	Duplication ratio (%)
WeChat	8.0.48	33.6
TencentQQ	9.0.30	9.4
Weibo	14.4.1	46.9
Bilibili	7.75.0	0.4
Jingdong	12.6.6	4.6
Zhihu	10.3.0	14.5
Taobao	10.35.0	12.5
Baidu Tieba	12.59.1	26.1
DingTalk	7.5.20	80.8
Sogou	11.45	60.9

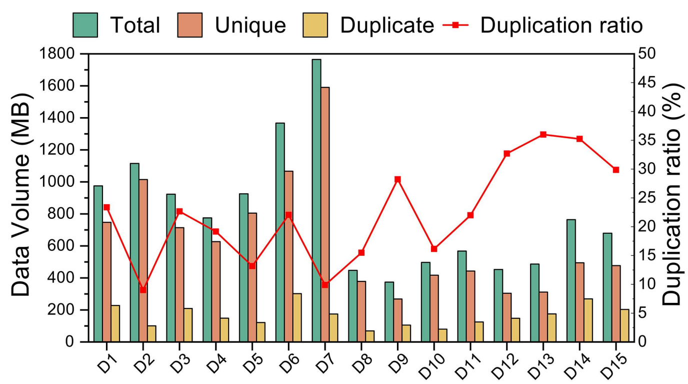

Figure 8 (Color online) Over a period of 15 d, user write data from 10 applications are gathered on a mobile device, displaying daily totals of overall data, unique data, duplicate data, and duplication ratio.

RefineDedup only needs to decrement the reference count of the PBN by 1 for the block. At the same time, RefineDedup removes the LBN entry in the block address mapper. As long as it is not a one-to-one mapping, RefineDedup does not mark the physical block as invalid.

4 Evaluation

In this section, we first describe the experimental setup. Then we analyze the saving performance and the latency performance of RefineDedup. Finally, we conduct the overhead analysis.

4.1 Experimental setup

To evaluate the effectiveness of RefineDedup, we implement it on Google Pixel 3 smartphones, which are equipped with an SDM 845, 4 GB RAM, and 64 GB flash storage. RefineDedup runs on Android 10 with Linux kernel version 4.9. We deploy RefineDedup, FinerDedup [1], and DmDedup [12] on these devices for evaluation and collect user write data in a real environment by installing the ten common mobile applications. Details about these applications are presented in Table 1.

To comprehensively present the characteristics of RefineDedup in real-world applications, we compare the performance of RefineDedup and FinerDedup with that of DmDedup. As shown in Figure 8, we collect 15 d of user write data on the same smartphone, applying RefineDedup and FinerDedup for training and inference. RefineDedup employs the method of forecasting a sliding window, starting from Day1. The size of the data collection window gradually increases until it reaches the preset size of seven on Day8, after which it remains fixed for continuous sliding training. FinerDedup continues to use the previous-day collection method for training and inference. We primarily evaluate their performance starting from Day8.

First, we evaluate and analyze the effects on writes of RefineDedup, focusing on stored fingerprints and reduced writes. Next, we evaluate the impact on performance. Finally, we provide an overhead analysis of RefineDedup.

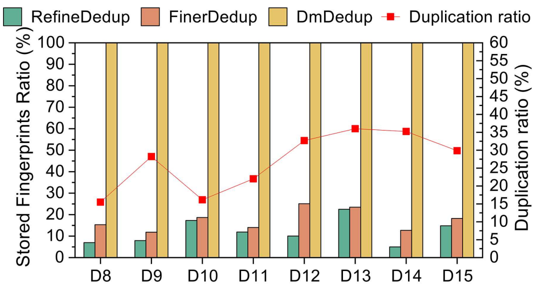

Figure 9 (Color online) Daily comparison of stored fingerprint ratios between RefineDedup and FinerDedup against DmDedup.

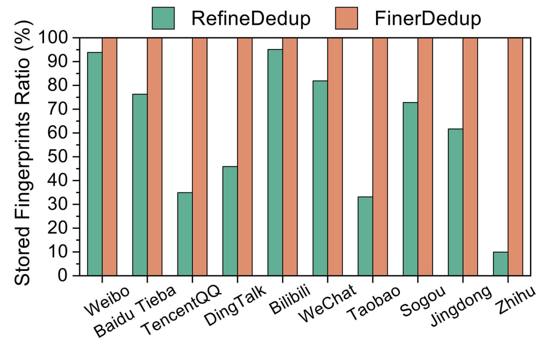

Figure 10 (Color online) Comparison of different model training methods between RefineDedup and FinerDedup across 10 applications.

4.2 Effect on writes

4.2.1 *Stored fingerprints*

During the time RefineDedup writes fingerprints to fingerprint tables, we can monitor the memory using the *dumpsys* command and monitor I/O conditions with the *iostat* statistics. Next, we present the impact of RefineDedup on fingerprint storage performance through three sets of experiments.

Figure 9 shows the daily ratio of fingerprints stored by RefineDedup, FinerDedup, and DmDedup. The vertical axis represents the ratio of fingerprints stored by RefineDedup and FinerDedup compared to those stored by DmDedup, with DmDedup always set as the baseline at 100%. Unlike DmDedup, which stores all fingerprints, RefineDedup and FinerDedup effectively reduce fingerprint storage by using machine learning models to filter fingerprints. RefineDedup stores 33.4% fewer fingerprints than FinerDedup and 88.1% fewer than DmDedup.

To demonstrate the effectiveness of application-wise model training, as shown in Figure 10, we evaluate the stored fingerprint ratio of RefineDedup and FinerDedup on ten mobile applications. The vertical axis represents the ratio of fingerprints stored by RefineDedup compared to those stored by FinerDedup, with FinerDedup always set as the baseline at 100%. The device deployed with RefineDedup uses the application-specific categorization model to predict the write data for each application, while the device deployed with FinerDedup uses the global categorization model to predict the write data. To eliminate the impact of varying data volumes caused by different sizes of data collection windows, both deduplication methods use a write trace of one day for training. The experimental results show that RefineDedup's model training method can reduce the stored fingerprint ratio for each mobile application by an average of 39.45% over FinerDedup. By customizing the categorization model to each application's unique data access behaviors, RefineDedup can more accurately identify which data blocks need to be deduplicated, thereby optimizing fingerprint storage. In contrast, FinerDedup cannot be tailored for each application, leading to rougher fingerprint storage.

In addition, Figure 11 illustrates the impact of the data collection window on fingerprint storage performance from Day1 to Day8. The horizontal axis represents the daily growth of the data collection window in RefineDedup. During the expansion of the data collection window, we monitor the daily ratios of fingerprints stored by RefineDedup, FinerDedup, and DmDedup, with DmDedup serving as the baseline (set to 100% in all cases). Since FinerDedup uses a previous-day collection method, its window size remains fixed at 1. It is important to note that both RefineDedup and FinerDedup require at least one day's write trace to train the model, so the stored fingerprints on Day1 for both methods are 0%. As the window size increases, the ratios of fingerprints stored by RefineDedup gradually decrease and remain consistently lower than those of FinerDedup. After Day8, the data collection window begins to slide, and the ratio stabilizes at an average of around 12%. Interestingly, except for Day1 and Day2, the ratio of fingerprints stored by RefineDedup closely follows the trend of the daily duplication ratio, demonstrating that the method of forecasting the sliding window effectively captures the trend of data access behaviors over time. There are two reasons behind such an observation. First, RefineDedup's data collection method provides richer training data than FinerDedup's method [1]. The forecasting sliding window supplies the model with write traces from multiple time periods, enabling it to learn a broader range of application data access behaviors during training. As a result, the model avoids overfitting to the data characteristics of a single time point. Second, to enhance the model's classification performance, RefineDedup introduces the *date* feature for each day's write trace during data collection. This feature converts timestamps into delta encoding, enabling RefineDedup to adjust the model

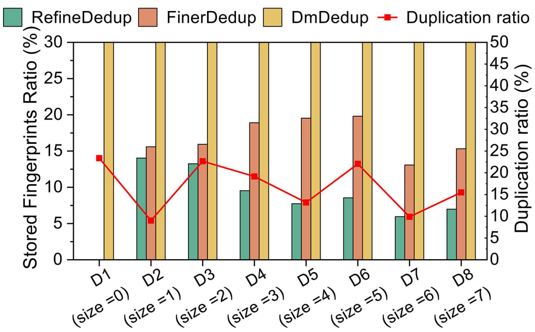

Figure 11 (Color online) Comparison of stored fingerprint ratios during the growth of the data collection window across three methods. Among them, the window size of FinerDedup remains fixed at 1.

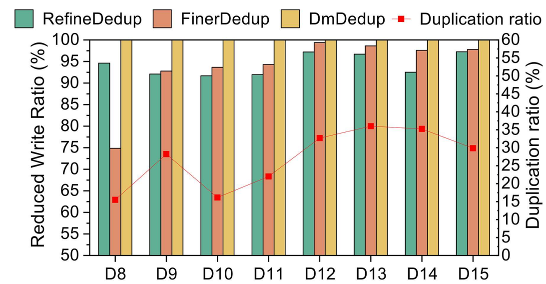

Figure 12 (Color online) Daily comparison of reduced write ratios between RefineDedup and FinerDeDup against DmDedup.

promptly. Hence, RefineDedup can adapt to the dynamic trends in the data access behaviors over time.

4.2.2 *Reduced writes*

We evaluate the daily reduced write ratio of RefineDedup and FinerDedup as shown in Figure 12. Note that *Reduced Writes* represents the ratio of write operations reduced by RefineDedup and FinerDedup compared with those reduced by DmDedup. Therefore, for DmDedup, the reduced write ratio is always 100%. During this eight-day period, RefineDedup achieves an average reduced write ratio that is 0.63% higher than FinerDedup. On Day8, the reduction in writes by RefineDedup is remarkable, exceeding that of FinerDedup by 19.74%.

The strength of RefineDedup lies in its adaptability to the latest data and its enhancement of predictive accuracy. Figure 8 demonstrates that the daily duplication ratio and data volume of mobile applications exhibit significant periodicity and volatility. FinerDedup uses only data from Day7 to train the model for predicting Day8's user data. However, RefineDedup captures the cyclical patterns of the previous seven days before predicting the eighth day's user data. The effectiveness of the model's predictions correlates with data access behaviors on that day, indicating that models trained solely with data from the previous day struggle to handle the uncertainty of data access behaviors. When a sudden change in duplication ratio and data volume occurs on Day8 compared with Day7, FinerDedup cannot withstand the negative impact of this change on model prediction performance. In contrast, RefineDedup adapts to such changes, helping the model maintain higher accuracy in real-world applications.

It is noteworthy that neither RefineDedup nor FinerDedup can achieve a 100% reduction in writes, as DmDedup stores the fingerprints of all written data, ensuring its reduced write ratio remains at 100%. However, both RefineDedup and FinerDedup involve false positive (FP) predictions that lead to some redundant writes [1]. Additionally, we find that applications with low duplication ratio levels exhibit an average ratio of only 5%. RefineDedup trains a shared model specifically for these applications, aiming for this model to focus more on the performance of *unique* data block predictions, which directly affects the size of the in-memory fingerprint table. Concurrently, this introduces a challenge where the FP predictions by RefineDedup may slightly increase. Generally, more FP predictions mean more *duplicate* data blocks are misjudged as *unique* and written to storage, thus increasing redundant writes. We hope that RefineDedup can further reduce the memory overhead of the fingerprint table while striving to match FinerDedup's average reduced write ratio.

4.3 Impact on performance

4.3.1 *Bandwidth performance*

Figures 13 and 14 respectively display the daily write bandwidth for two applications: Zhihu, which has a 14.5% low duplication ratio, and Weibo, with a 46.9% high duplication ratio. Every day, we generate 300000 4 kB write data blocks on both applications to assess the write bandwidth. We find that RefineDedup offers the best bandwidth for low-redundancy applications, followed by FinerDedup, which slightly outperforms DmDedup. For high-redundancy applications, DmDedup performs the best, followed by RefineDedup, which is nearly comparable to DmDedup when the duplication ratio is between 35% and 50%. In contrast, FinerDedup performs the worst, while RefineDedup shows the best bandwidth performance when the duplication ratio is below 35%. Considering that the duplication ratio on mobile devices is typically around 22%, RefineDedup is undoubtedly a superior option.

There are primarily three reasons for these bandwidth results. First, RefineDedup exhibits lower model inference latency compared with FinerDedup (see Subsection 4.4.2). During a write operation, both must go through the

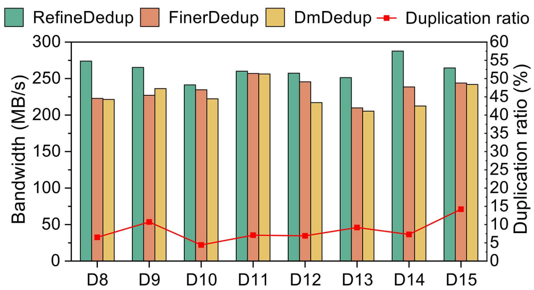

Figure 13 (Color online) Compare the daily write bandwidth of the three methods on the low-redundancy Zhihu application.

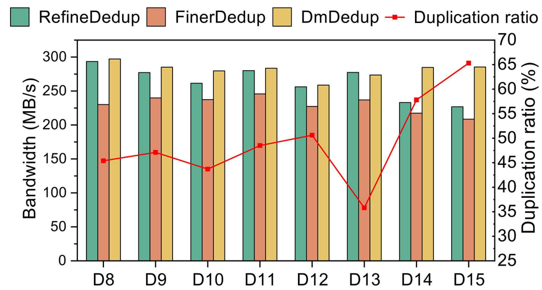

Figure 14 (Color online) Compare the daily write bandwidth of the three methods on the high-redundancy Weibo application.

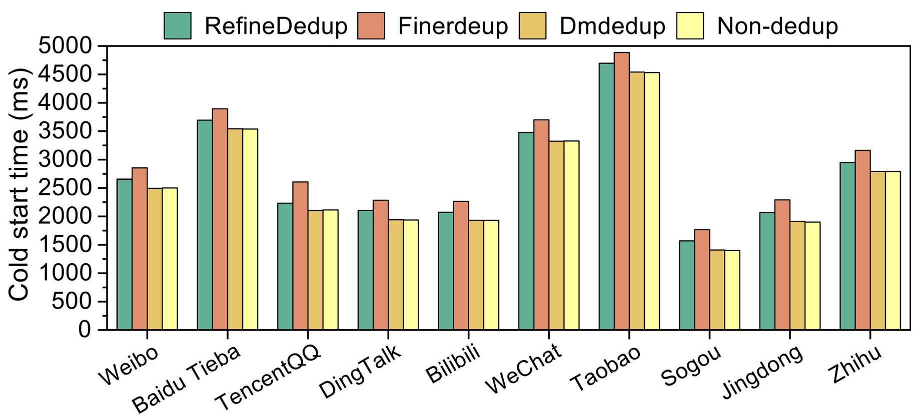

Figure 15 (Color online) Comparison of cold start latencies for 3 deduplication methods across 10 applications.

following steps. (1) Obtain labels through inference; (2) pass labels down the write critical path; (3) retrieve the fingerprint table; (4) write the data blocks to storage. RefineDedup's inference latency is only about 30% of FinerDedup's, thus RefineDedup achieves faster I/O speeds. Second, when the duplication ratio is low, RefineDedup stores fewer fingerprints than other methods. Fewer fingerprints accelerate the matching and retrieval processes. This expedites data writing. Third, in scenarios with a high duplication ratio, data block writing does not require complex deduplication decisions. In such cases, the model's inference latency hinders the speed of data block writing, which explains why DmDedup excels in high-redundancy applications.

4.3.2 *Cold start latency*

Akamai's statistics show that 53% of mobile website visitors will leave a page that takes more than 3 s to load. Users expect applications to load quickly and respond promptly. Particularly in mobile device applications, launch latency is crucial for maintaining user attention and ensuring continued use [41]. For applications with slow startup times, users may leave negative reviews in the app store or even abandon the app entirely. To clearly demonstrate the effect of model loading latency on the cold start of applications, as illustrated in Figure 15, we use the *am start* command in Android to monitor the cold start latency of ten applications. In addition to devices equipped with three deduplication methods, we also set up a non-dedup device that lacks any deduplication methods.

Since both RefineDedup and FinerDedup require loading a categorization model during application cold starts, their startup times are longer than DmDedup. DmDedup operates mainly at the block layer; therefore, the cold start latencies for applications on devices with DmDedup are essentially the same as those on non-dedup devices. On average, startup times for RefineDedup are only about 150 ms slower than those of non-dedup devices, while FinerDedup is about 370 ms slower. For applications that start quickly on non-dedup devices, such as Zhihu, with a latency of 2790 ms, FinerDedup's startup time exceeds 3000 ms. Such latency is unacceptable for mobile applications that are extremely sensitive to startup times.

4.3.3 *Response time*

We develop a script based on the MonkeyRunner tool [7] to measure the response times of mobile applications. The MonkeyRunner tool automates user interactions—such as button taps and screen swipes—and our script records

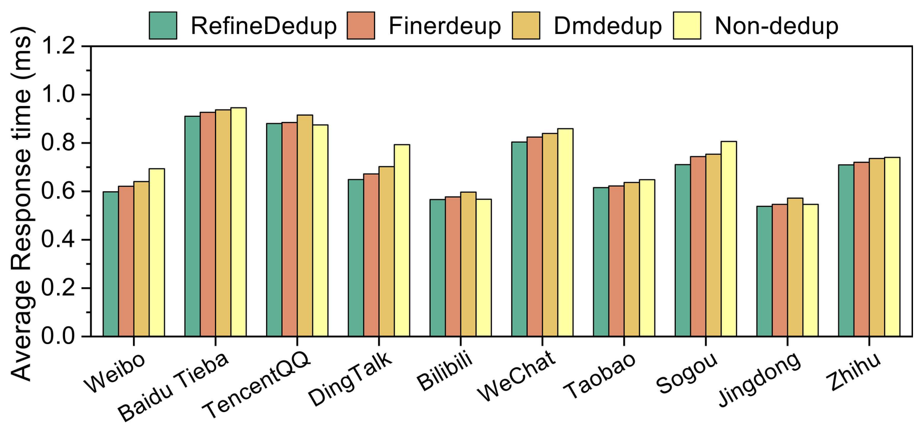

Figure 16 (Color online) Comparison of average response times for 3 deduplication methods across 10 applications.

timestamps immediately before and after each action. Using these timestamps, we compute the response time for each interaction. Figure 16 presents the average response times across ten mobile applications.

Experimental results show that RefineDedup reduces average response times by up to 18.20% compared to non-dedup devices, with an average of 6.55%. It also reduces response times by up to 3.71% compared with FinerDedup, with an average of 2.23%. The reason is that write operations are usually time-consuming, and RefineDedup removes most of the write redundancy to reduce write time. For redundant writes, it modifies block address mapping without disk I/O, greatly lowering write latency. Notably, for applications with a high duplication ratio—DingTalk exhibits up to 80.8% redundant writes, where RefineDedup stands out. By eliminating redundant writes and retrieving the minimum number of fingerprints, it achieves the largest reduction in average response times. In addition, for applications with an extremely low duplication ratio, e.g., Bilibili, which exhibits only 0.4% redundant writes, all three deduplication methods provide limited improvement in response times. Specifically, DmDedup delivers the poorest response times, since each write incurs the overhead of an exhaustive search through the entire fingerprint table. Although RefineDedup and FinerDedup maintain few fingerprints, their response times slightly increase due to model inference latency (see Subsection 4.4.2). Overall, RefineDedup improves the average response times of mobile applications.

4.4 Overhead analysis

4.4.1 *Memory overhead of model*

As shown in Table 2, we use the *dumpsys* command in Android to precisely monitor the daily model memory overhead for ten applications over 15 d with the RefineDedup and FinerDedup methods. The unit of daily model memory consumption is megabyte (MB). It is important to note that the data collection window for RefineDedup expands from Day1 to Day7. Since Day1 falls within the training stage, no memory overhead from the model is recorded, and the overhead stabilizes at a window size of seven from Day8 onwards. We observe that the model memory overhead of RefineDedup for a single application is substantially lower than that of FinerDedup. This is particularly evident on Day6, where the average overhead for RefineDedup is approximately only 0.86% of that for FinerDedup. There are two main reasons for the substantial difference in costs. First, RefineDedup has fewer model parameters, which results in lower memory overhead. Second, RefineDedup employs a random sampling strategy during training, which not only reduces the memory overhead for model training and inference but also ensures robust predictive performance.

Additionally, from a temporal perspective, we observe that the daily trend of model memory overhead for FinerDedup, as presented in Table 2, aligns closely with the daily trend of user data volume shown in Figure 8. This correlation exists because the memory overhead of the FinerDedup model is influenced by the volume of data trained on the previous day. If the training volume from the previous day is excessive, the overall size of the FinerDedup model may increase significantly, potentially offsetting the memory savings achieved from reduced fingerprints. Although FinerDedup can also employ a sampling strategy to reduce the memory overhead caused by training data volume, the limitations of its method of data collection—which relies solely on the previous day's data—can lead to a considerable decline in model prediction performance.

When managing scenarios with multiple applications open on mobile devices, RefineDedup accumulates the model memory overhead of all applications performing I/O operations. However, this overhead does not exceed

Table 2 Comparison of daily model memory overhead for 10 applications between RefineDedup and FinerDedup. Note: All values are expressed in megabytes (MB). The bolded values represent the average memory overhead of a single application on that day.

Apps	Method	Day1	Day2	Day3	Day4	Day5	Day6	Day7	Day8	Day9	Day10	Day11	Day12	Day13	Day14	Day15
WeChat	RefineDedup	–	1.3	1.1	1.6	5.8	0.6	3.5	0.04	0.04	2.2	0.03	1.5	0.04	0.5	0.6
	FinerDedup	–	28.4	29.2	35.4	40.5	52.4	47.1	36.4	6.6	18.5	8.2	13.1	12.6	10.6	19.2
TencentQQ	RefineDedup	–	1.3	0.03	0.5	0.7	0.5	4.3	0.4	1.3	0.07	0.04	0.3	0.4	0.02	0.05
	FinerDedup	–	25.1	26.9	35.3	40.1	55.8	53.7	38.4	9.3	12.8	8.3	10.6	11.6	15.3	19.5
Weibo	RefineDedup	–	0.03	0.4	0.02	0.04	0.07	0.3	0.1	0.04	0.06	0.02	0.1	0.2	0.1	0.4
	FinerDedup	–	26.8	29.8	37.2	46.5	51.8	47.7	37.8	7.5	13.5	7.3	10.7	11.8	9.4	19.3
Bilibili	RefineDedup	–	0.05	0.04	0.04	0.3	0.03	0.2	1.5	0.06	0.02	0.01	0.04	0.02	0.03	0.01
	FinerDedup	–	25.3	27.9	35.4	45.2	61.1	44.8	38.8	7.4	13.8	11.3	11.7	12.2	15.6	20.2
Jingdong	RefineDedup	–	1.4	0.9	0.3	1.1	0.05	0.04	0.04	0.9	1.6	1.02	0.04	0.03	0.1	0.3
	FinerDedup	–	25.8	27.4	42.3	39.6	59.4	55.0	38.5	7.2	13.5	9.2	10.3	11.9	10.6	17.5
Zhihu	RefineDedup	–	2.3	0.6	0.4	0.9	0.1	0.4	0.05	1.9	0.03	2.7	0.12	0.05	1.2	0.2
	FinerDedup	–	27.7	27.7	43.6	40.2	58.4	51.3	37.6	10.6	11.6	8.4	12.1	11.5	11.5	20.3
Taobao	RefineDedup	–	1.2	1.6	0.1	0.2	0.7	0.8	0.05	0.3	0.9	1.9	0.02	0.05	1.3	0.04
	FinerDedup	–	24.9	27.2	43.1	45.2	65.4	54.6	37.7	7.5	14.1	15.9	10.1	12.6	12.1	20.7
Baidu Tieba	RefineDedup	–	2.6	0.3	1.1	3.5	2.9	4.3	0.7	0.4	0.3	1.2	0.1	0.9	1.5	0.02
	FinerDedup	–	27.3	26.7	43.9	43.7	64.9	49.3	39.1	7.9	10.5	9.1	10.3	12.6	16.2	20.6
DingTalk	RefineDedup	–	0.9	1.2	0.3	0.7	0.4	2.1	0.06	0.3	0.1	0.05	0.03	0.13	1.4	1.1
	FinerDedup	–	26.7	30.1	35.8	47.4	53.6	46.6	40.4	8.1	14.2	9.8	8.2	10.9	17.5	20.3
Sogou	RefineDedup	–	0.02	0.05	0.02	0.03	0.02	0.5	0.03	0.05	0.6	0.03	0.03	0.03	0.05	0.02
	FinerDedup	–	25.3	29.1	30.1	36.3	56.7	47.3	38.7	8.3	16.0	10.1	10.1	12.1	12.2	20.4
Average	RefineDedup	–	**1.1**	**0.6**	**0.4**	**1.3**	**0.5**	**1.6**	**0.3**	**0.5**	**0.6**	**0.4**	**0.2**	**0.2**	**0.5**	**0.2**
	FinerDedup	–	**26.3**	**28.2**	**38.2**	**42.5**	**57.9**	**49.7**	**38.3**	**8.1**	**13.9**	**9.8**	**10.7**	**11.9**	**13.1**	**19.8**

that of FinerDedup. First, due to the limit of 32 background processes in Android and the low memory killer (LMK) mechanism [42, 43], Android terminates some applications or restricts their I/O operations. Consequently, RefineDedup frees the model memory of these applications, preventing the cumulative memory overhead from surpassing that of FinerDedup. Second, the single model memory overhead of RefineDedup is far lower than that of FinerDedup. For instance, on Day6 as shown in Table 2, the average memory overhead for a single application model is 0.5 MB for RefineDedup compared with 57.9 MB for FinerDedup. This suggests that devices with RefineDedup would need to maintain over 115 applications performing I/O operations simultaneously in a short time to potentially exceed the global model memory overhead of FinerDedup. Clearly, this scenario is unrealistic. Third, RefineDedup releases model memory as users actively kill applications. Android allows users to enable memory cleaning settings, such as automatically clearing the memory of applications 10 min after the screen turns off. This feature enables RefineDedup to maintain significantly lower memory overhead than FinerDedup, even with multiple application instances active.

4.4.2 *Latency overhead*

We add timestamps before and after the model loading process to evaluate the latency of RefineDedup and FinerDedup during application cold starts. Figure 17 illustrates the daily comparison of average model loading latencies for 10 applications between RefineDedup and FinerDedup. The average model loading latency for RefineDedup is approximately 0.16 s, while for FinerDedup it is 0.35 s. RefineDedup achieves approximately a 50% reduction in model loading latency compared to FinerDedup. This improvement is due to RefineDedup using smaller random forest hyperparameters than those used by FinerDedup. The selection of these hyperparameters significantly affects the model size, loading latency, and ultimately, prediction performance. Generally, a smaller model size results in reduced loading times, enabling faster initialization and loading of models on resource-constrained mobile devices.

Figure 18 illustrates the average inference latency for a single user data entry across 10 applications, comparing RefineDedup and FinerDedup. The inference latency for a single data entry in each application is approximately one-third with RefineDedup compared to that with FinerDedup. First, RefineDedup's random forest model utilizes fewer trees and less depth, with its model parameters amounting to only about 0.1% of those of FinerDedup. In random forest models, deeper trees create longer decision paths, requiring each sample to traverse more nodes during inference, which extends the inference time per tree. Second, RefineDedup transforms the Python-implemented scikit-learn random forest model into inference binary trees coded in C during integration, significantly accelerating

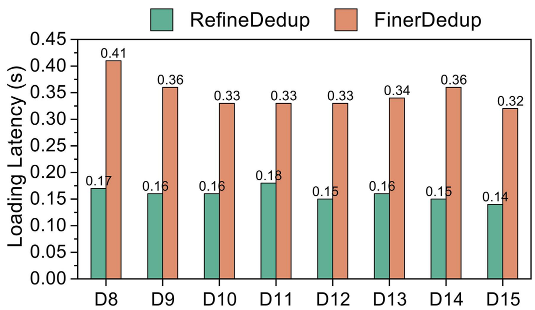

Figure 17 (Color online) Daily comparison of average model loading latency for 10 applications between RefineDedup and FinerDedup.

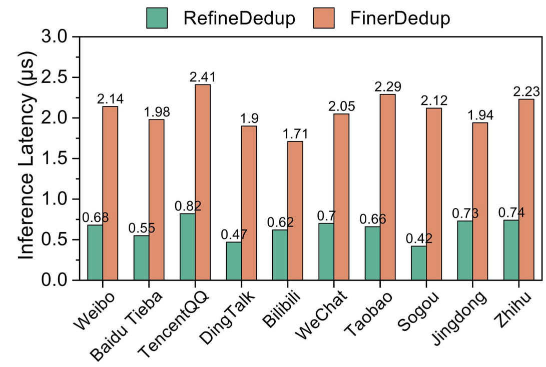

Figure 18 (Color online) Comparison of average inference latency for a single user data entry between RefineDedup and FinerDedup across 10 applications.

the inference process.

5 Conclusion

We propose RefineDedup, an innovative data deduplication method for mobile devices. This method integrates a forecasting sliding window with application-specific categorization models, optimizing the deduplication process based on data access behaviors and application characteristics. Comprehensive evaluations in real mobile systems demonstrate that RefineDedup surpasses existing technologies by reducing fingerprint count, increasing write bandwidth, and decreasing memory usage. RefineDedup not only maintains efficient data deduplication but also minimizes the overhead associated with this process in mobile systems. In future work, we will optimize RefineDedup for other resource-constrained devices, such as smartwatches and smart bracelets, to broaden its applicability across a wider range of mobile devices. Additionally, we will explore alternative machine learning models for deduplication in RefineDedup to uncover new insights and potentially further improve its performance.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 62372073, 62032004, 62402070, 62472058) and New Chongqing Youth Innovative Talents Project (Grant No. CSTB2024NSCQ-QCXMX0079). We would like to thank the reviewers for their valuable comments and improvements to this paper.

References

1 Chen X, Zhou X, Li W, et al. FinerDedup: sifting fingerprints for efficient data deduplication on mobile devices. In: Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC 24), San Francisco, 2024
2 Yang Q, Jin R, Zhao M. SmartDedup: optimizing deduplication for resource-constrained devices. In: Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019. 633–646
3 Carroll A, Heiser G. An analysis of power consumption in a smartphone. In: Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 10), 2010
4 Mahesri A, Vardhan V. Power consumption breakdown on a modern laptop. In: Proceedings of the 4th International Workshop on Power-Aware Computer Systems, 2005. 165–180
5 Liu Y, Ma Y, Xiao X, et al. LegoDroid: flexible Android app decomposition and instant installation. Sci China Inf Sci, 2023, 66: 142103
6 Mandal S, Kuenning G, Ok D, et al. Using hints to improve inline block-layer deduplication. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16), 2016. 315–322
7 Mao B, Zhou J, Wu S, et al. Improving flash memory performance and reliability for smartphones with I/O deduplication. IEEE Trans Comput-Aided Des Integr Circuits Syst, 2018, 38: 1017–1027
8 Song C, Chen X, Liu D, et al. CADedup: high-performance consistency-aware deduplication based on persistent memory. In: Proceedings of the 2022 IEEE 40th International Conference on Computer Design (ICCD), 2022. 726–729
9 Chen F, Luo T, Zhang X. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 11), 2011
10 Chen Z, Shen K. OrderMergeDedup: efficient, failure-consistent deduplication on flash. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16), 2016. 291–299
11 Gupta A, Pisolkar R, Urgaonkar B, et al. Leveraging value locality in optimizing NAND flash-based SSDs. In: Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 11), 2011
12 Tarasov V, Jain D, Kuenning G, et al. Dmdedup: device mapper target for data deduplication. In: Proceedings of the 2014 Ottawa Linux Symposium, 2014
13 Sharma P C, Bansal S, Raja R, et al. Concepts, strategies, and challenges of data deduplication. In: Data Deduplication Approaches. New York: Academic Press, 2021. 37–55
14 Muthitacharoen A, Chen B, Mazieres D. A low-bandwidth network file system. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, 2001. 174–187
15 Xia W, Jiang H, Feng D, et al. A comprehensive study of the past, present, and future of data deduplication. Proc IEEE, 2016, 104: 1681–1710
16 Shilane P, Chitloor R, Jonnala U K. 99 deduplication problems. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16), 2016

17 Debnath B, Sengupta S, Li J. ChunkStash: speeding up inline storage deduplication using flash memory. In: Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 10), 2010
18 Kim J, Lee C, Lee S, et al. Deduplication in SSDs: model and quantitative analysis. In: Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012. 1–12
19 Cheng W, Cai R, Zeng L, et al. IMCI: an efficient fingerprint retrieval approach based on 3D stacked memory. Sci China Inf Sci, 2020, 63: 179101
20 Sun Z, Kuenning G, Mandal S, et al. A long-term user-centric analysis of deduplication patterns. In: Proceedings of the 2016 32nd Symposium on Mass Storage Systems and Technologies (MSST), 2016. 1–7
21 Kim S, Jeong J, Lee J. Selective memory deduplication for cost efficiency in mobile smart devices. IEEE Trans Consumer Electron, 2014, 60: 276–284
22 He Q, Li Z, Zhang X. Data deduplication techniques. In: Proceedings of the 2010 International Conference on Future Information Technology and Management Engineering, 2010. 430–433
23 Zhou D, Pan W, Wang W, et al. I/O characteristics of smartphone applications and their implications for eMMC design. In: Proceedings of 2015 IEEE International Symposium on Workload Characterization, 2015. 12–21
24 Jeong S, Lee K, Lee S, et al. I/O stack optimization for smartphones. In: Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), 2013. 309–320
25 Zhang P, Huang P, He X, et al. Resemblance and mergence based indexing for high performance data deduplication. J Syst Software, 2017, 128: 11–24
26 Zhu B, Li K, Patterson R H. Avoiding the disk bottleneck in the data domain deduplication file system. Proc Fast, 2008, 8: 1–14
27 Min J, Yoon D, Won Y. Efficient deduplication techniques for modern backup operation. IEEE Trans Comput, 2010, 60: 824–840
28 Lillibridge M, Eshghi K, Bhagwat D, et al. Sparse indexing: large scale, inline deduplication using sampling and locality. Proc Fast, 2009, 9: 111–123
29 Chen R, Wang Y, Hu J, et al. vFlash: virtualized flash for optimizing the I/O performance in mobile devices. IEEE Trans Comput-Aided Des Integr Circuits Syst, 2016, 36: 1203–1214
30 Kim H, Agrawal N, Ungureanu C. Revisiting storage for smartphones. ACM Trans Storage, 2012, 8: 1–25
31 Mao B, Wu S, Jiang H, et al. Content-aware trace collection and I/O deduplication for smartphones. In: Proceedings of the 33rd International Conference on Massive Storage Systems and Technologies (MSST), 2017. 1–8
32 Breiman L. Random forests. Mach learn, 2001, 45: 5–32
33 El-Shimi A, Kalach R, Kumar A, et al. Primary data deduplication—large scale study and system design. In: Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), 2012. 285–296
34 Jin K, Miller E L. The effectiveness of deduplication on virtual machine disk images. In: Proceedings of the SYSTOR 2009: the Israeli Experimental Systems Conference, 2009. 1–12
35 Meister D, Kaiser J, Brinkmann A, et al. A study on data deduplication in HPC storage systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012. 1–11
36 Park N, Lilja D J. Characterizing datasets for data deduplication in backup applications. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10), 2010. 1–10
37 Khatoon A, Corcoran P. Android permission system and user privacy—a review of concept and approaches. In: Proceedings of the 2017 IEEE 7th International Conference on Consumer Electronics-Berlin (ICCE-Berlin), 2017. 153–158
38 Bloom B H. Space/time trade-offs in hash coding with allowable errors. Commun ACM, 1970, 13: 422–426
39 Lu G, Nam Y J, Du D H C. BloomStore: bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In: Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012. 1–11
40 Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. Internet Math, 2004, 1: 485–509
41 Joo Y, Ryu J, Park S, et al. FAST: quick application launch on solid-state drives. In: Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 11), 2011
42 Kim S H, Jeong J, Kim J S, et al. SmartLMK: a memory reclamation scheme for improving user-perceived app launch time. ACM Trans Embed Comput Syst, 2016, 15: 1–25
43 Singh A, Agrawal A V, Kanukotla A. A method to improve application launch performance in Android devices. In: Proceedings of the 2016 International Conference on Internet of Things and Applications (IOTA), 2016. 112–115