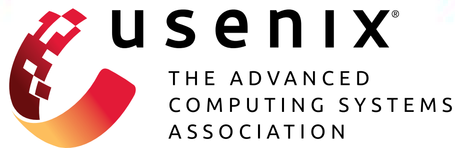


CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime

Yu Liang, *Department of Computer Science, City University of Hong Kong and School of Cyber Science and Technology, Zhejiang University;* Riwei Pan, Tianyu Ren, and Yufei Cui, *Department of Computer Science, City University of Hong Kong;* Rachata Ausavarungnirun, *TGGS, King Mongkut's University of Technology North Bangkok;* Xianzhang Chen, *College of Computer Science, Chongqing University;* Changlong Li, *School of Computer Science and Technology, East China Normal University;* Tei-Wei Kuo, *Department of Computer Science, City University of Hong Kong, Department of Computer Science and Information Engineering, National Taiwan University, and NTU High Performance and Scientific Computing Center, National Taiwan Univesity;* Chun Jason Xue, *Department of Computer Science, City University of Hong Kong*

https://www.usenix.org/conference/fast22/presentation/liang

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings of the 20th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime

Yu Liang[1,2], *Riwei Pan*[1], *Tianyu Ren*[1], *Yufei Cui*[1], *Rachata Ausavarungnirun*[3], *Xianzhang Chen*[4*],
Changlong Li[5*], *Tei-Wei Kuo*[1,6,7], *and Chun Jason Xue*[1]
[1] *Department of Computer Science, City University of Hong Kong*
[2] *School of Cyber Science and Technology, Zhejiang University*
[3] *TGGS, King Mongkut's University of Technology North Bangkok*
[4] *College of Computer Science, Chongqing University*
[5] *School of Computer Science and Technology, East China Normal University*
[6] *Department of Computer Science and Information Engineering, National Taiwan University*
[7] *NTU High Performance and Scientific Computing Center, National Taiwan University*

Abstract

Mobile applications often maintain downloaded data as cache files in local storage for a better user experience. These cache files occupy a large portion of writes to mobile flash storage and have a significant impact on the performance and lifetime of mobile devices. Different from current practice, this paper proposes a novel framework, named CacheSifter, to differentiate cache files and treat cache files based on their reuse behaviors and main-memory/storage usages. Specifically, CacheSifter classifies cache files into three categories online and greatly reduces the number of writebacks on flash by dropping cache files that most likely will not be reused. We implement CacheSifter on real Android devices and evaluate it over representative applications. Experimental results show that CacheSifter reduces the writebacks of cache files by an average of 62% and 59.5% depending on the ML models, and the I/O intensive write performance of mobile devices could be improved by an average of 18.4% and 25.5%, compared to treating cache files equally.

1 Introduction

Mobile devices are now dominant in people's daily lives [23, 39]. Almost all mobile applications need to download files or data from networks because of the dynamic nature of applications and overall system optimization. Even with the high bandwidth of modern communication networks (e.g., WiFi and 5G), many applications still rely heavily on data cached on mobile devices to avoid re-downloading data through the network and meet their execution latency demands. Current mobile devices store cache files in the main memory first and then write them back to flash storage. These applications' cached data are usually managed as cache files and can be re-accessed quickly [9, 41]. However, the number and the size of applications' cache files have grown exponentially in recent years, as applications demand increasing amounts of data. For example, Facebook can generate 1.2GB of cache files on a mobile device in two hours [27]. In addition to performance degradation, most cache files are eventually written to the flash storage of a mobile device, increasing writes and thus decreasing the lifetime of flash devices [2, 45].

A number of research studies on mobile systems have been performed in recent years [11, 12, 16, 18, 20, 23, 26, 27, 32, 37, 38]. These techniques include optimization of memory management [23, 26], defragmentation [11], storing cache files in memory [32, 38], re-designing the directory cache of mobile systems [37], an application-aware swapping mechanism [20], and I/O management [12, 16, 18]. Unfortunately, little work exists that has differentiated cache files in management. Although Liang et al. [27] elucidate differences among cache files, a solution was not proposed. Since the total size of cache files has increased dramatically, improper writebacks of cache files to flash storage will markedly reduce the lifetime of the flash storage of a mobile device. It is also worth noting that some cache files are used only once throughout their lifetime while others may be re-accessed multiple times before deletion. In current practice, however, cache files are treated equally.

Android operating systems store cache files in local storage in consideration of performance and latency [9]. However, cached data on a mobile device can significantly reduce the lifetime of its flash storage, as the replacement cycle of smartphones increases [40]. Recent works propose to store cached data in DRAM to reduce writebacks, and thus can improve both system performance and lifetime [32, 38]. These techniques suffer from two major problems as applications increase their demand for cached files. First, cached data vary greatly in frequency of access, lifetime, and size. Treating them equally leads to inefficiency. Second, the available memory is insufficient in mobile devices, maintaining useless cache files could degrade the overall system performance because of memory competition. *The goal* in this work is to improve *both* system performance and the lifetime of flash storage by managing cache files according to their reuse behaviors.

The proposed novel cache file management framework,

*Corresponding authors: Changlong Li, Email: clli@cs.ecnu.edu.cn; Xianzhang Chen, Email: xzchen@cqu.edu.cn.

named CacheSifter, dynamically categorizes cache files into different categories using a light-weighted machine learning (ML) algorithm and dynamically places the cache files of different categories in DRAM or flash storage based on their data access patterns. Three cache-file categories are proposed based on their revisiting possibility: Burn-After-Reading (BAR) files, Transient files, and Long-living files. A quasi-in-memory file system is proposed for better management of Transient files in DRAM and to avoid the operating system from accidentally evicting them out of DRAM. A cache-file eviction mechanism is also developed to utilize DRAM more effectively in keeping cache files.

CacheSifter adheres to the semantics of the Android cache file management and does not produce new safety vulnerability. Experimental results over popular applications show that CacheSifter reduces the writebacks of cache files by an average of 62% and 59.5% depending on the ML models, and the I/O intensive write performance of mobile devices could be improved by an average of 18.4% and 25.5%. [1]

2 Cache Files in Mobile Systems

Unlike servers' applications, most mobile applications frequently download fresh data such as news and videos from networks. Mobile systems generally store the downloaded data as cache files in local storage temporarily to reduce redundant data downloads. For example, Android systems maintain temporary cache files in the main memory for a period of time (30 seconds by default) and then write them back to flash storage [9, 41]. This is similar in spirit to how Linux manages its files, which treats all files equally. Writing *all* of the cache files back into flash storage will significantly degrade system performance [8], reduce the lifetime of flash storage [2], and occupy large storage space, which is markedly limited in mobile devices. With the exponential growth of mobile applications' cache files induced by high-speed networks in recent years, optimization of their management has become urgent.

While previous works [32, 38] aim to maintain all cache files of targeted applications in the main memory to accelerate cache files' accesses, the total size of cache files for an application can occupy a significant portion of the main memory, which could substantially degrade the performance of the other running applications via memory contention. This paper, however, aims to manage cache files according to their access patterns and thus only necessary cache files will be stored in main memory or storage.

2.1 Required Space and Writes of Cache Files

This section aims to quantify cache file usages in current Android systems via both static and dynamic methods.

[1]Note that the reduction of writebacks can substantially improve the lifetime of flash-memory storage and notably benefit I/O-intensive phases in application execution, application launch, and application installation, which are crucial to the mobile user experience [3].

Required space for cache files. The storage occupation of cache files is determined by taking snapshots of cache files in storage. We survey 60 volunteers[2] with real mobile device usage, including 42 models of 5 vendors, for one week. We collect the snapshots of cache files for the commonly-used applications (4-15 applications according to volunteers' usage behaviors) once per day on 50 of the mobile devices. On the other ten mobile devices, data are collected three times per day. Table 1 presents the total size of cache files of each mobile device and different applications. We choose the most-commonly-used application for each type on each smartphone. Some smartphones might be missing certain types of applications due to different user behaviors.

Table 1: Cache files' sizes of different devices or applications.

Group by	Code name	No. of devices	Average (GB)	Max (GB)
Vendors	Huawei	30	0.4	1.79
	Oppo	6	4.11	8.82
	Vivo	6	1.58	1.85
	Xiaomi	17	2.17	4.55
	Meizu	1	1.68	1.7
Apps	Social media	60	0.35	2.73
	Video	57	0.22	2.23
	Website	60	0.26	5.25
	Game	11	0.18	1.33

The collected data shows that cached file size varies greatly between vendors and applications. Based on the data collected from these mobile devices, it is found that some mobile systems or third-party software delete cache files. Moreover, some users habitually delete cache files to alleviate the shortage of storage space. However, even though these cache data are deleted after writes, their damage to the performance and lifetime of flash storage has already occurred. As a consequence, it is critical to evaluate the actual writes of cache files during run-time.

Write behaviors. The writes of cache files are now profiled from two perspectives: user behaviors and representative applications. We first collect the write size of cache files under volunteers' usage behaviors by instrumenting the source code of Linux in the experimental mobile devices to collect every write of cache files to flash storage. Five volunteers used the experimental mobile devices for three days.

The collected data reveals that the writes of cache files can reach 500MB per day, even for users that spend less than three hours per day on their mobile devices. Data from a mobile device vendor (top-five) shows that the total writes of their testing users is about 10GB on average and up to 30GB per day. Cache file writes is approximately 6.4GB on average per day and up to 19.2GB because cache file writes represent an average of 64% of total writes to storage for mobile devices based on experimental results, as shown in Figure 1.

The writes of cache files of the top-20 representative applications are collected, including social media, map, game, video, and browser. In this experiment, the volunteers used each application continuously for two hours. The ratio of

[2]The volunteers are 18-60 years old.

cache file writes to total writes for the twenty applications is presented in Figure 1. The write count ratio is the ratio of the number of writes of cache files to the number of writes of total files. It is found that most applications write a large amount of cache files. For example, the write count ratio of Facebook is up to 92.6%, and the write size ratio of YouTube is as high as 95.7%. In contrast, CandyCrush is a stand-alone game that does not need to download much data from the network.

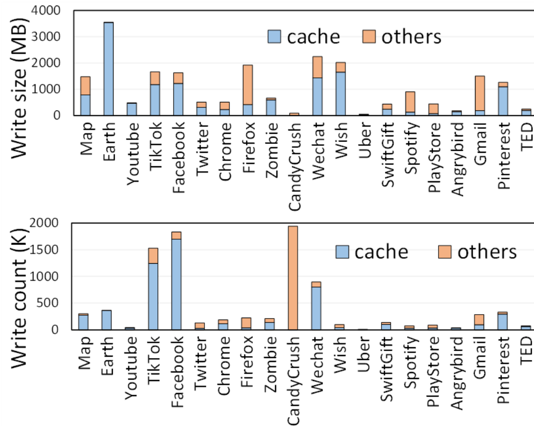


Figure 1: Write count/size of representative applications in two hours.

Existing applications write many cache files into flash storage during run-time. Even though many cache files are deleted by the system or applications, the cache files still occupy large storage space. Notably, the write and delete operations of cache files not only increase I/O contention, which could degrade I/O performance, but also shorten the lifetime of flash devices [10, 21]. In addition, the problem will become increasingly severe with the continual increase of network speed, growing usage of applications, and use of newer flash chips (e.g., TLC, QLC, and 3D-NAND flash) with shorter endurance [13, 17, 44].

2.2 Differences among Cache Files

As mentioned above, existing Android systems treat all cache files as normal files that always require persistent storage. Cache files are time-sensitive data, however, and it is often unnecessary for them to be persistently stored. Based on observations at the block layer, Liang et al. [27] proposed to classify all cache files into three categories, i.e., Burn-After-Reading (BAR), Transient, and Long-living, according to the distinctly varied access patterns of cache files in flash storage.

After defining the categories of files, numerous questions arise regarding how files are categorized and how categories are managed in a practical system, none of which offer straightforward answers. However, all of these questions are addressed in this paper. Furthermore, paper [27] demonstrates differences between cache files at the block layer. This paper finds that access information at the VFS layer is more suitable for categorizing cache files because the cache files should be handled prior to the block level. Therefore, this paper reuses the name of three types of files in paper [27] but with different definitions.

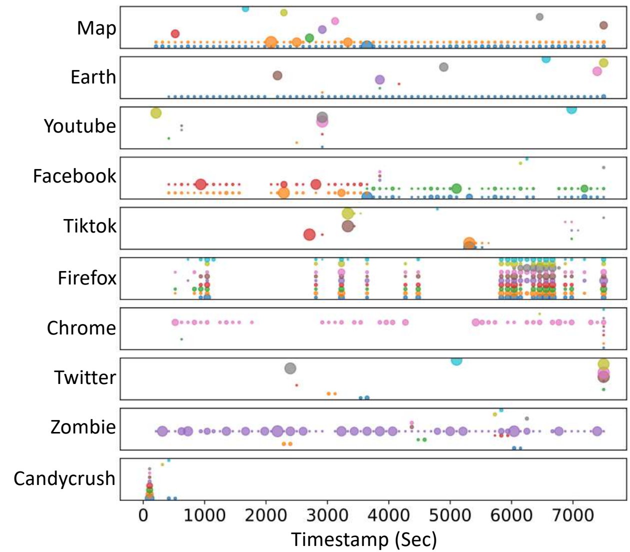


Figure 2: Access patterns of the top-10 accessed cache files of ten representative applications.

Burn-After-Reading (BAR) represents cache files that only have tiny re-accesses that take place at the beginning of their lifetime. For many typical mobile applications, most cache files are rarely re-accessed, which is similar to the conclusion reached in a previous study [4]. In particular, some cache files in the flash storage were deleted without being re-accessed at all. Consequently, there is no need to write such cache files to storage.

Transient refers to cache files that have a large re-access count as well as a short active period. Figure 2 shows the access patterns of the top-10 accessed cache files (denoted by ten colors) of ten representative applications. The size of a circle indicates the access count of the corresponding cache file within 100 seconds. The access patterns of the cache files vary greatly. Moreover, some of these files have a short active period and a large access count. For example, the third cache file (labeled in green) of Map is only re-accessed within a short time after it is created. Accordingly, we deem such cache files as Transient files, since the applications only re-access them in the near future.

Long-living represents the rest of cache files, especially the files that are re-accessed frequently over a long period of time.

2.3 Challenges in Cache File Management

Even though Liang et al. [27] proposed to manage cache files following their access patterns, they did not explore classification or management methods of cache files. Two major challenges exist in the management of cache files. First, cache files' behaviors change over time. For this reason, it is im-

portant that management should be adaptive to the run-time behavior of cache files. Second, existing systems store cache files according to the same routine. It is necessary, however, to manage dissimilar types of cache files by different policies. The main goal of this paper is to improve both system performance and storage lifetime. We will explore the access patterns of applications' cache files and consider the characteristics of DRAM-based main memory and flash-based storage of mobile devices in terms of performance and endurance.

3 CacheSifter Design

We propose CacheSifter to categorize cache files and manage them by exploiting their access patterns.

3.1 Overview of CacheSifter

3.1.1 Design Principles

We discuss five design principles for categorizing and managing cache files in mobile systems.

User application transparency. CacheSifter should have an insignificant impact on user experience. CacheSifter should also be compatible with the semantics of existing mobile systems requiring zero changes in existing user applications.

Online Categorization. While offline profiling simplifies the categorization process, an offline classifier cannot adapt to the dynamic system status and the configuration of users during usage of the mobile device. As a result, CacheSifter needs to be able to categorize cache files online while avoiding the extra overhead of storing BAR files and Transient files.

Adaptive memory management. CacheSifter always attempts to maintain the cache files that will be used in the main memory to achieve high file access performance. However, using too much memory for the cache files may degrade system performance. In this case, CacheSifter should adaptively adjust its memory usage along with different active applications.

Adapt to changes in user behavior. CacheSifter should adapt to changes in user behaviors. A categorized file may need to be re-categorized. For example, a file is categorized as a BAR file because it is only used once immediately after it is downloaded. When user behavior changes, and it is used many time repeatedly, it should be re-categorized into a Transient or Long-living file to avoid frequent re-download.

Ensure safety when deleting cache files. CacheSifter should not produce any new vulnerabilities as compared to existing mobile systems. Since CacheSifter may discard BAR files and Transient files during execution of applications, it is critical that discarding data by CacheSifter will not cause an application crash or user data loss.

3.1.2 CacheSifter Framework

Following these principles, we design CacheSifter to categorize cache files and manage them in DRAM/flash storage according to their reuse patterns to avoid unnecessary writing back. Figure 3 shows the framework of CacheSifter. CacheSifter lives in the kernel rather than in an intermediate or less-privileged layer. CacheSifter can directly categorize cache files on the page cache without additional memory consumption and data copy. CacheSifter also does not require any changes in existing user applications, i.e., it is transparent to user applications.

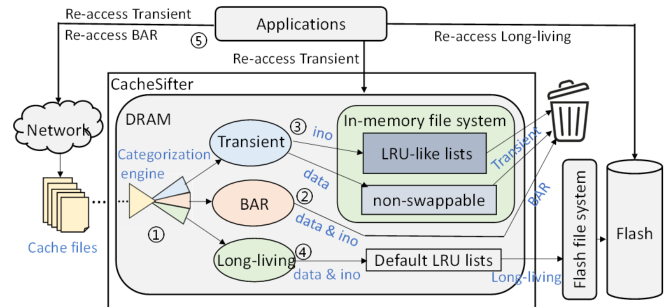


Figure 3: Framework of CacheSifter.

All newly downloaded cache files are first maintained in the main memory and wait for categorization. ① CacheSifter adopts a lightweight machine-learning-based categorization engine (See Section 3.2.1) to divide the newly downloaded cache files into three types, i.e., BAR, Transient, and Long-living in two stages online. ② To better utilize memory and storage, CacheSifter discards all BAR files because they are typically not reused. ③ For Transient files, CacheSifter keeps them in DRAM by using a quasi-in-memory file system, which is designed to avoid an accidental swap-out of Transient files. CacheSifter discards Transient files when there is insufficient DRAM space using an LRU-like file eviction policy (See Section 3.2.2). ④ For Long-living files, CacheSifter writes them to the flash storage by exploiting the default LRU-based eviction scheme of the Android system. Moreover, CacheSifter deletes the cache files from their corresponding storage when they are invalidated by applications. ⑤ The deleted files will be re-downloaded from the network when they are accessed in the future, which provides an opportunity to change the categorization of cache files according to changes in user behavior (See Section 3.3). Finally, CacheSifter exploits a safe list mechanism to maintain known potential paths of cache files that are important to users, or in cases in which their deletion could threaten system stability (See Section 3.4).

CacheSifter provides three key benefits. First, CacheSifter avoids pushing-out the BAR and Transient files to flash storage, which reduces write contention, extends the lifetime of the flash storage, improves overall system performance, and conserves storage space. Second, Transient files are accessed directly from DRAM, improving the access latency of this type of cache files. Third, CacheSifter significantly optimizes the management of cache files with a lightweight machine-learning-based engine in the kernel, which not only has negligible overhead but is also transparent to user applications.

3.2 Feature-based Cache Files Management

The effectiveness of CacheSifter relies highly on the accuracy of the categorization engine. The overhead of the categorization engine and the cache file management mechanism of cache files should be as small as possible to minimize the impact on system performance. In this section, we describe the design of these two key components.

3.2.1 Lightweight Categorization of Cache Files

Machine learning based categorization. According to the design principles, categorization should be both lightweight and conducted online. Based on our observations, we know the categorization of each cache file by observing its reuse patterns. However, to avoid writebacks of BAR and Transient files, this method requires storing all cache files and their access information in main memory for a long time period for categorization, which imposes a high cost. Heuristic-based methods, such as suffix based methods [14, 24], can categorize files with a small cost. However, they do not consider the access patterns of cached files, and thus cannot be used to recognize BAR, Transient, and Long-living files. For example, a video (.exo) file could be any type of cache file according to user behaviors. Moreover, since the users' behavior and access pattern of cache files are different across different applications, we expect non-ML approaches to be less flexible and generalized. Thus, the categorization engine in CacheSifter utilizes machine-learning-based schemes to automatically perform categorization based on features within a short period of time and observation-based labels.

This paper chooses a lightweight neural network method (MLP [5]) in the experiments because of its performance and low cost.[3] To further reduce cost, categorization is divided into two phases (BAR/non-BAR and Transient/Long-living) by exploiting two MLP models because we find that Long-living cache files cannot be recognized by short-time information. We train these two MLP models offline and use them for online categorization, and thus this method needs to retrain the models after a period of time to adapt to applications' changes. Certainly, one can also choose a lightweight reinforcement learning method [35] to avoid retraining, which is beyond the scope of this paper.

Metrics for analyzing prediction models. Three metrics are used to evaluate our categorization models. First, we use *Accuracy* to reflect how correctly the model predicts the categories of files, as shown in Equation 1:

$$Accuracy = (TP + TN)/Total_Instances \quad (1)$$

Where TP is an outcome in which the model correctly predicts the positive class; a true negative TN is an outcome in which the model correctly predicts the negative class. Considering the penalty of misclassification, positive class is non-BAR in the first phase of categorization in which the negative class is BAR. In the second phase of categorization, Long-living is denoted as the positive class, while Transient is the negative class. Based on our observation, the data of each category is highly unbalanced. Therefore, the above *Accuracy* cannot represent the accuracy of each type of file. Accordingly, we introduce another metric, *Recall*, in Equation 2:

$$Recall = TP/(TP + TN). \quad (2)$$

In the high-recall model, we care more about the predicted accuracy of files with high mis-predicted overhead, such as long-living files. If a long-living file is incorrectly predicted as a BAR or Transient file, it could induce redownload overhead. Finally, to visualize the results, we also use the third metric, *PR curve,* which is simply a graph with Precision values on the y-axis and Recall values on the x-axis. A good PR curve has a large area under curve (AUC).

Based on these three metrics, we train high-recall and high-accuracy models with a high PR curve. The high-recall model aims to reduce writebacks of cache files and minimize re-download overhead; whereas, the high-accuracy model aims to reduce writebacks of cache files with minimum mis-categorization.

3.2.2 Cache File Management Mechanism

To better utilize memory/storage to reduce writebacks of cache files and minimize re-download penalty, cache files are processed according to their categorization.

BAR file. BAR files are deleted immediately after categorization because these files are not likely to be reused.

Transient file. Since the usage of Transient files in mobile applications exhibits both strong locality and time sensitivity in a certain period of time, CacheSifter always attempts to maintain the Transient files in the main memory during their active period to achieve higher file access performance. At first, we try to exploit an existing file system, such as tmpfs or ramfs, to manage Transient files. However, to avoid writeback operations prior to the categorization of cache files, each cache file will have two inodes, i.e., one in F2FS and tmpfs/ramfs each, which complicates the implementation and brings additional overhead. As a consequence, a quasi-in-memory file system (QMFS) is proposed to manage Transient files in the main memory during their active period.

QMFS is implemented by two LRU-like lists (an active list and an inactive list), as shown in Figure 4. The active list is designed to ensure that files will not be deleted within their active period. The inactive list is used to balance memory pressure and file performance. Specifically, when memory is sufficient, files will be maintained in memory for a longer time to reduce the penalty of mis-classifications. In the default memory management, the LRU list of page cache is page-granularity since the pages of files cached in the main memory will be written back to storage. If a page of a Transient cache file is deleted, however, this means that the whole cache file

[3] We compare the performance of MLP, Random Forest, Linear Regression and Logistic Regression and find MLP to be the most effective.

in QMFS is invalid. Therefore, the LRU-like lists of QMFS in CacheSifter are maintained in file-granularity.

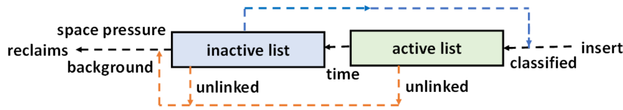


Figure 4: Eviction scheme of LRU-like lists in QMFS.

In QMFS, a cache file is put into the active list after categorization. Subsequently, the file in the *active list* may be moved into the *inactive list*, depending on the size of free memory and its existing time in the *active list*. If the existing time of a file is longer than its active period (a threshold), it will be moved to the *inactive list* to wait for deletion. Insufficient memory also triggers the movement action. If a file in the *inactive list* is referenced before it is deleted, it will be moved back to the *active list*. If a file is deleted, it will be deleted from the corresponding lists. If a file is truncated, CacheSifter works in the same manner as the default Android system. Specifically, the file's pages will be deallocated whereas the inode number will be still maintained in the LRU-like lists.

When the active period of Transient files ends, the Transient files generally will not be used again. For this reason, the longest-lived Transient files should have the highest priority to be evicted. Furthermore, to improve the performance of foreground applications, the cache files generated by a background application should also have a higher priority for eviction. We use UID to identify the files of background applications, as previously described [12]. After file eviction, memory space will be reclaimed. The parameters for reclaim are established in Section 5.2.

Long-living file management. Unlike the eviction schemes of BAR and Transient files, Long-living files are managed by the default page-based eviction scheme of the page cache in Android systems. Long-living files are maintained in the default LRU-based lists of the page cache. When a page of a Long-living file is unused for a long period of time, it will be evicted from the page cache and written back into the flash storage if it is dirty. Consequently, Long-living files will be stored in storage infinitely unless the applications delete them.

3.3 User Behavior Adaptation

Even if the feature-based cache file management worked well, user behaviors could change. Therefore, CacheSifter should be able to re-categorize cache files when user behavior changes to avoid frequent re-downloads. There are four types of state changes, as listed in Table 2. **BR**, **TR**, and **LL** represent the BAR category, the Transient category, and the Long-living category, respectively. Thus, "BR-> TR, LL" means that a BAR file shifts to a Transient file or a Long-living file.

Table 2 shows actions that trigger state changes of cache files, and the corresponding benefit or cost. When user behavior changes, CacheSifter only updates the category of cache files after re-download since CacheSifter performs categorization only when a file is newly-downloaded from the network.

Table 2: Actions based on state changes.

Types	State changes	Action	Benefit/Cost
(1)	BR-> TR, LL	Re-categorize files after re-download	None
(2)	TR -> BR	Do nothing and wait for discard	Memory space
(3)	TR -> LL	Re-categorize files after re-download	High performance
(4)	LL -> BR, TR	Do nothing	Flash space

CacheSifter treats and re-categorizes the re-downloaded file as a new file, and thus CacheSifter can adapt to stage change types (1) and (3) in Table 2. When type (2) stage change occurs, CacheSifter does not need to do anything, since Transient files will be discarded just like BAR files. Compared with BAR files, Transient files will remain in the main memory for a longer period of time and consume memory space. Type (4) is similar to type (2). Therefore, CacheSifter also does nothing, which consumes flash storage for a short time.

3.4 Safety Mechanism

CacheSifter discards the BAR and Transient files eventually. To make these operations safe for user data and applications, CacheSifter exploits a *safe_list* approach for cache file directories. It is not difficult to track and manage *safe_list* paths. In fact, Android now exploits these paths, which can be seen through the cache-delete button in the Android setting [6]. CacheSifter uses the same paths of the cache-delete button as the *safe_list* paths. Moreover, the *safe_list* can be managed offline. If vendors wish to optimize certain specific application, such as YouTube, they can obtain the cache paths of YouTube in advance and put them into the *safe_list*.

4 CacheSifter in Android

We implement CacheSifter in the Android system as a case study for mobile systems. In our implementation, CacheSifter categorizes cache files by using a dedicated thread. In this section, we first present the details of MLP-based categorization. We then show how the categorized cache files are managed by the flash file system, F2FS [24], and the proposed QMFS. Finally, we discuss implementation considerations of CacheSifter.

4.1 MLP-based Cache File Categorization

Categorization features and labels. In order to avoid unnecessary writes of cache files, categorization should be completed as rapidly as possible by using as few features as possible. The challenge here is that the Long-living files cannot be recognized by short-time features easily. To accurately categorize cache files, we first perform a fast categorization to detect BAR files and then dedicate additional time for the second pass to further separate Transient and Long-living files. Importantly, the memory space overhead is not large because there are not many Long-living files. In addition, categorization should also adapt to changes in user behaviors. Therefore, the objective of feature design is to characterize the access patterns and attributes of each file with a low memory cost.

To achieve this goal, the access patterns (read, write, and I/O size) and attributes of files (file size and active period) are selected as the features for machine-learning methods. The designed features are shown in Figure 5.

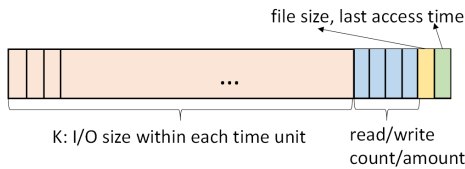


Figure 5: Features for learning.

In general, the system maintains K+6 features for each data point. The first K dimensions of a data point are sequential data that correspond to access information within the first K time units after creating a file, where each value represents the sum of access I/O size within one time unit. The following 4 features are read amount, read count, write amount, and write count within K time units, while the last 2 features are file size and active period. File size is the maximum size of each file within the K time units. The active period is calculated by the last access time minus the first access time within the K time units. If a file is accessed only once within this period, the last access time is set to be K time units. To achieve high accuracy and low memory cost, the value of K of the first phase of categorization (BAR or other) is smaller than the second phase of categorization (Transient or Long-living). When a new cache file is created, its access information during time K (e.g., 30s) will be recorded.

In addition to the features, the labels of cache files can be used to train the categorization model. Although we cannot use the observation-based categorization online due to its high overhead, the labels for the cache files in the training dataset can be obtained based on the observations of the reuse patterns throughout their lifetime. Specifically, we label a file as a BAR file ("1"), or a Transient file ("2") or a Long-living file ("3") according to their active period.

Dataset Collection. We instrument the source code of the Android kernel and use the Android Debug Bridge (**adb**) tool [7] to collect the access information and file size of cache files at the VFS layer in fs/read_write.c. Based on the collected data[4], the labels and features are obtained to train our machine learning models.

To make the model as general as possible, many data are collected. Our collected data includes four parts: (1) information from ten representative applications gathered over 20 hours; (2) information of the same applications by different users, in order to include more user behaviors; (3) information of the same applications after three months for checking the retraining period; and (4) information of different applications in order to assess the prediction accuracy of untrained applications. This case study aims to optimize these ten applications.

Categorization methods. With sufficient data with features $\mathbf{x}$ and labels y collected, the subsequent step is to find a proper machine learning model that learns the mapping $f(\cdot): \mathbf{x} \rightarrow y$. In this work, we compare some simple machine learning methods and choose to use the popular Multi-Layer Perceptron (MLP) as it theoretically approximates any function if given sufficient capacity, according to the universal approximation theorem [5]. We choose to use a lightweight MLP layer that takes the features as input and outputs the categorization results. A large network capacity (size) causes great CPU and memory consumption while reducing the network capacity might decrease the performance. We use a grid search to find the best network capacity. We start from an over-parameterized neural network and evaluate its classification accuracy on the validation set. The network size is gradually decreased by re-training the network until its performance on accuracy starts decreasing. The same strategy is applied to other network hyper-parameters, which will be elaborated in Section 5.1.

We first train the categorization models and evaluate them offline (on a PC), which can assist tuning the parameters to identify the best models for cache file categorization. Then, the trained models will be used in the Linux kernel for dynamic categorization. When the optimized applications are upgraded, the models could need to be retrained. Based on our dataset (3), the model can still accurately predict the new data that are generated after at least three months. Therefore, the period of retraining could be longer than three months in our case. In this case, we also provide a fuse mechanism, CheckStop, to stop CacheSifter once the prediction accuracy is lower than a threshold. To avoid retraining, one can choose a lightweight reinforcement learning model for the categorization.

4.2 Management of Categorized Cache files

The management of categorized cache files mainly includes two parts: handling data pages and managing inodes. All of these pages and inodes are managed and maintained by several lists.

There are three lists in CacheSifter for inode management: *temp_list*, *category_list1*, *category_list2*. The categorization engine, which is a dedicated thread, wakes up periodically to scan these lists and control the migration of inodes among them. prior to categorization, the inodes of all cache files are maintained in *temp_list* after creation and their data pages are managed in the *unevictable_list* in the page cache layer to avoid accidental eviction caused by the Android system.

Two-phase Categorization. For categorization, the inodes in *temp_list* are moved to *category_list1* periodically to improve concurrency. In the first phase, the categorization engine scans *category_list1* and determines whether an inode is BAR. Then, the BAR inodes are deleted, and the remaining inodes in *category_list1* are migrated to *category_list2*. After the second phase of categorization, the data pages of Transient files remain in the *unevictable_list*, while the inodes of Transient

[4]Released in https://github.com/yliang323/CacheSifter.

files are stored in our LRU-like lists of the QMFS. The data pages and the inodes of Long-living files are moved to the default LRU lists (inactive file list) of the page cache layer, and they are set as dirty. They are then written back into flash storage by the default Android system.

4.3 Implementation Discussions

Adaptation of CacheSifter. Vendors train models by using the dataset of targeted applications. When CacheSifter is deployed on different mobile devices, the machine learning model does not need to be retrained because it is based on the behaviors of applications. A large training dataset can cover extensive user behavior with a small implementation overhead under the selected machine learning method.

Stop CacheSifter in unforeseen cases. To handle some rare cases, we design a lightweight prediction checking mechanism, named CheckStop, to determine if CacheSifter should be stopped. The main idea here is to calculate the re-download rate by recording the hash values of downloaded files and deleted files in a time window. If the rate is larger than a threshold, CacheSifter is suspended. To minimize overhead, CheckStop only works when CacheSifter detects abnormal signals such as a significant change in the number of write-backs or file creations with the same hash value.

CacheSifter in the future. CacheSifter could be more useful for future generations of mobile devices for the following three reasons. First, with a faster network, more data could be accessed and cached per time unit, and thus the amount of cache files could be increased. Second, with the usage of new flash chips(e.g., TLC, QLC), storage lifetime is becoming increasingly crucial since the endurance of many new flash devices have become smaller. Third, the memory capacity of mobile devices is growing, which can support more in-memory cache files and better machine-learning-based categorization methods. Additionally, CacheSifter can be used in other Internet of Things (IoT) systems or automotive systems.

5 Evaluation Methodology

We implement and evaluate CacheSifter on real mobile devices with two different categorization models.

5.1 Categorization Models

In current Android systems, cache files are maintained in the main memory for 30 seconds by default and then written back to the flash storage. For this reason, in the evaluation, we label a file as a BAR file ("1") if its active period is shorter than 30 seconds to avoid extra memory usage. Rather than writing BAR files back to flash storage, CacheSifter deletes them after their categorization. If the active period of a file is longer than 30 seconds but smaller than 90 seconds, it will be labeled as a Transient file ("2"). Otherwise, it is labeled as a Long-living file ("3"). According to the active period of a set of cache shown in Figure 6, the majority of cache files (93%) in this dataset are BAR files. Consequently, a large number of writebacks of cache files can be avoid.

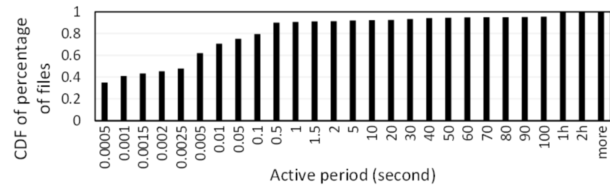


Figure 6: Active period of cache files.

To avoid using too much main memory, CacheSifter categorizes all of the cache files within 30 seconds. We utilize MLP as we found the mechanism to be the most accurate out of all other simple machine learning methods (random forest, linear regression and logistic regression). We test different parameters and present the results in Table 3.

Table 3: Categorization results by using different features.

Total time	Time unit	K	High Recall		High Accuracy	
			Recall	Accuracy	Recall	Accuracy
1s	0.01s	100	0.92	0.56	0.45	0.89
1s	0.05s	20	0.82	0.56	0	0.90
5s	0.25s	20	0.89	0.56	0	0.90
10s	0.5s	20	0.88	0.59	0	0.90
20s	1s	20	0.80	0.65	0.56	0.86
40s	1s	40	0.94	0.85	0.94	0.85
60s	1s	60	0.95	0.88	0.95	0.88
80s	1s	80	0.94	0.89	0.94	0.89

Our goal is to achieve enough accuracy or recall with small memory usage (smaller K). Therefore, we choose 20 and 60 as K_1 and K_2 for the first and second phases, respectively. Based on the feature within 20s, we can only choose high accuracy or high recall, and thus we use two models for different purposes.

Training models. The collected data are grouped by applications for training. We divide our datasets (1) and (2) (See Section 4.1) into 80% training and 20% testing instances. We use the training dataset to train a MLP network and exploit its "Accuracy" and "Recall" by evaluating the trained model on the testing dataset. We gradually decrease the neural network size by re-training the network until its accuracy performance starts to decrease. The same strategy is applied to other network hyper-parameters. All of the hyper-parameters are listed in Table 4.

The trained/re-trained model can be deployed to users' mobile devices as a system update. It is used for online categorization, which includes three parts: online feature collection, implementation of the trained MLP models, and the model-based categorization. First, we collect features of every new file for 20 seconds and maintain them in the main memory. A dedicated thread periodically wakes up to check the features and categorize the files. CacheSifter deletes BAR files after categorization and continues collecting features for files in other categories for an additional 40 seconds.

Categorization results of MLP models. We evaluate the

Table 4: Summary of hyper-parameters.

Hyper-parameters	For the first phase	For the second phase
number of hidden layers	4	4
hidden layer size	[512, 200, 2]	[512, 200, 2]
activation function 1	Tanh	Tanh
activation function 2	ReLU	ReLU
The function of output layer	Softmax	Softmax
loss_function	Focal loss function	Focal loss function
learning rate	0.1+MultiStepLR	0.1+MultiStepLR
optimizer	SGD + momentum = 0.5	SGD + momentum = 0.5
weight decay	1.00E-04	1.00E-05
sampler	WeightedRandomSampler	WeightedRandomSampler
batch size	200	200

trained models by ten representative applications and their random combinations. For the combinations, we use the first two letters to identify the application's name, and the results of which are shown in Figure 7. Some values are missed because the testing dataset may have just one type of cache files. For example, there are no non-BAR files in the testing dataset of Earth, and all of its data are predicted as the BAR class. Thus, it *AUC* is N/A, *Recall* is N/A, and *Accuracy* is 1. The results show that as long as an application has been trained, the model can classify its files well, irrespective of with what applications it is combined.

5.2 Evaluation Setup

We evaluate all of the experiments on two smartphones: (1) P9 equipped with an ARM Cortex-A72 CPU, 32GB internal flash memory and 3GB DRAM running Android 7.0 with Linux kernel version 4.1.18, and (2) Mate30 equipped with an ARM Cortex-A76 CPU, 128GB internal flash memory and 8GB DRAM running Android 10 with Linux kernel version 4.14.116. Ten representative applications, including social media, map, game, video, and browser, are used to collect features of cache files and evaluate CacheSifter. Their workload profiles (i.e., cache file ratio and data access patterns of their cache files) are presented in Figure 1 and Figure 2. We revise the kernel to print the access information of each file and the file attribute in functions new_sync_read() and new_sync_write() in fs/read_write.c. We filter the cache files by using the specific cache path of applications (/data/<packagename>/cache/).

We compare CacheSifter with the management scheme of cache files in default Android systems. The parameters of CacheSifter applied in the evaluation are listed in Table 5. To make a fair comparison, both the user and activities are the same for each comparison. For each testing, we follow the same sequence of actions: 1) we close all apps and clean their cache files prior to reboot to eliminate the impact of old cache files; 2) after reboot, we clean the cache to eliminate the impact of potentially buffered data; 3) we use the same application, login with the same user account, and conduct the same sequence of activities; and 4) We attempt our best to make each test the same, and we also conduct each test more than five times to eliminate possible nuances.

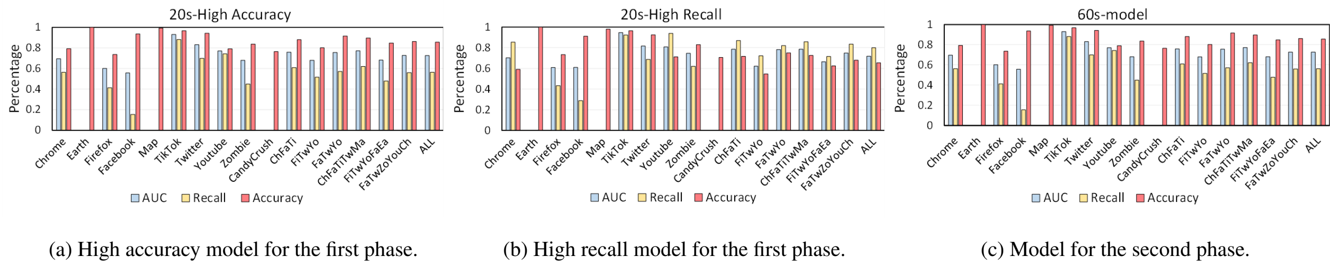
The parameters are selected based only on the targeted applications and independently of the experimental platform. The two smartphones run the same version of applications and use the same parameters.

Table 5: Summary of parameters used by CacheSifter.

Symbols	Semantics	Setting
K_1	The time for the first phase of categorization	20 seconds
K_2	The time for the second phase of categorization	60 seconds
T_1	The period of time for waking up the thread	10 seconds
E_1	Period of time to inactive	20 seconds
S_1	Size of each background reclaim	To W_1
T_2	The period of time for background reclaim	20 seconds
MS	Maximum RAM size for Transient files	20MB
W_1	Low watermark for background reclaim	50%*20MB
W_2	High watermark for foreground reclaim	90%*20MB
S_2	Size of each foreground reclaim	10%*20MB

Parameter configurations. K_1 and K_2 are the time to collect features of cache files for corresponding phases of categorization. Their values are determined as discussed in Section 5.1. T_1 is relative to CPU and memory consumption. If it is too small, the dedicated thread would run frequently and thus consume CPU time. On the other hand, if it is too large, the cache files will stay in the main memory for a long period of time to wait for categorization even if they already have enough features. Since we find the features within 20 seconds to be sufficient for the first phase of categorization, we choose 10 seconds to make sure the first phase can be finished within the default 30 seconds to avoid extra memory usage and frequent wake up. E_1 is the time that Transient files can be deleted. Since the active period of Transient files is 90 seconds in our evaluation, E_1 is 20 seconds (90-K_2-$T1$). S_1 is the reclaim space that to prepare for future usage, and it is related to W_1. T_2 does not need to be frequent because the reclaimed memory is enough for the next usage of Transient file within K_2. Therefore, we set it as 20 seconds according to our experience to reduce the CPU consumption. It is also not sensitive to the performance. These parameters do not need to be modified for different models of mobile devices if they use the same version of applications. If the versions of targeted applications are updated, the parameters MS, W_1, W_2, and S_2 may need to be changed due to workload changes. To show how to select these three parameters, we first present the cache file's size that was produced within 60 seconds in Figure 8.

Based on the data from Figure 8, we find that the maximum size of cache files of targeted applications is 21MB. Because not all files are transient, we configure MS, which is the upper bound of memory usage of the Transient files of targeted applications within K_2, to 20MB. This allows more memory to be used for other purposes. W_1 and W_2 are the watermarks of reclaims. Overall, the larger are their values, the more space will be used by Transient files; Thus, the re-access performance of Transient files is better but the performance of other applications could be worse because of memory contention. W_1 should be the maximum value of memory usage of the



(a) High accuracy model for the first phase. (b) High recall model for the first phase. (c) Model for the second phase.

Figure 7: Predict results of two MLP models for different applications and their combinations. For the combinations, the first two letters are used to identify the application's name. "ChFaTi" represents the combination of Chrome, Facebook, and TikTok.

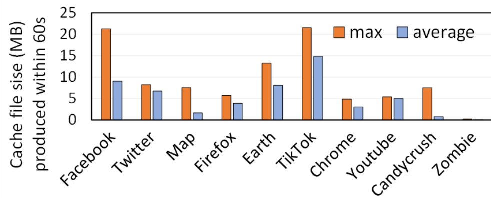


Figure 8: Cache files produced by applications within 60s.

Transient files of targeted applications within K_2, and 10MB (50% of MS) is enough for our case. W_2 and S_2 should be the minimum value that is just enough for the Transient files of targeted applications within K_2, and 2MB (10% of MS) is enough for our case. These parameters can be changed later for different platforms and manufacturers.

6 Evaluations

We evaluate CacheSifter's performance using two key metrics: (1) the reduction in writebacks of cache files and extension in lifetime of mobile flash storage; and (2) the improvement in read and write performance under intensive I/Os. We show the writeback reduction on two platforms while only show the other results on one platform since they are similar on different platforms and we do not have enough space for them.

6.1 Lifetime Improvement

Reduction in writebacks of cache files. We compare the writebacks of cache files and the number of block I/Os of CacheSifter against the default system. Since the results can vary under different user behaviors, each test is conducted ten times, the average results of which are shown in Figure 9. We evaluate both the high-recall model and the high-accuracy model.

The results reveal that writebacks of cache files vary for different applications. The reduction in writebacks when using the high-recall model is similar to that of the high-accuracy model in this experiment. Theoretically, the high-recall model constitutes a conservative-delete scheme that tends to keep cache files in the mobile device to reduce the penalty of re-download. In contrast, the high-accuracy model is a radical-delete scheme to pursue higher overall predict accuracy.

On P9, the writebacks of cache files are reduced by the high-recall model and the high-accuracy model by an average of 62% and 59.5%, respectively. The number of total I/Os is also significantly decreased by both models, i.e., an average of 29.7% and 31.2%, respectively. The high-accuracy model treats all of the three classes with the same priority. The high-recall model attempts to minimize incorrect predictions in the two cases (LL->BAR and LL->Transient) to reduce the re-download penalty. Since the long-living files are a small part of all cache files (less than 5%), as shown in Figure 6, the write reduction is similar under these two models.

On Mate30, the writebacks of cache files are reduced even more by both models, i.e., an average of 88.3% and 85.5%, respectively. The number of I/Os is also decreased more by both models, i.e., an average of 47.7% and 46.6%, respectively. The results on Mate30 show that the models trained by the data collected from P9 also work well on Mate30 because CacheSifter is platform-independent. There are two main reasons for the difference between the results on P9 and Mate30: different user behaviors, and the default system management schemes.

Based on Figure 6, 93% of the cache files are BAR in that dataset, but writebacks are not reduced as much in this case primarily because 1) the directory of cache files must be written back to flash storage to maintain consistency because there are some Long-living files that uses the directory information; 2) different user behaviors; and 3) the predict accuracy is not 100%.

Sensitivity Study. To evaluate the sensitivity of CacheSifter, we use the same parameters and the same models on P9 and Mate 30. The write reduction shown in Figure 9 indicates that both P9 and Mate 30 achieve similar benefits from CacheSifter. Moreover, we conduct a sensitivity study for the parameters in Table 5. The write reduction results on Mate30 with different MS are presented in Table 6. The sensitivity results show that the total writes could be affected by the value of MS due to different memory usage.

Boosted lifetime of mobile flash storage. Cai et al. [2] present the following method to compute the lifetime im-

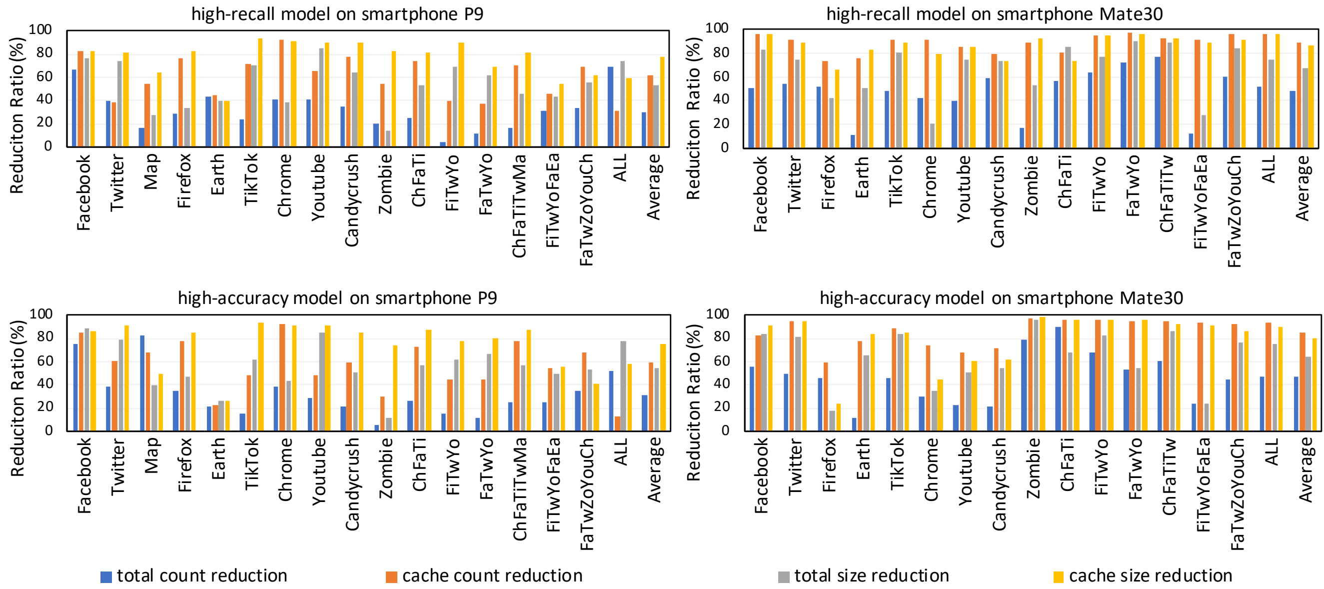


Figure 9: Normalized reduction ratio of cache files' writebacks and total I/Os. We evaluate the trained models with ten representative applications and their combinations. For the combination, we use the first two letters to identify the application's name. "ChFaTi" represents the combination of Chrome, Facebook, and TikTok.

Table 6: Sensitivity study with parameter *MS*.

MS	Total count	Cache count	Total size	Cache size
20MB	55%	83%	83%	91%
15MB	53%	77%	76%	94%
10MB	39%	72%	73%	84%
5MB	34%	76%	57%	91%

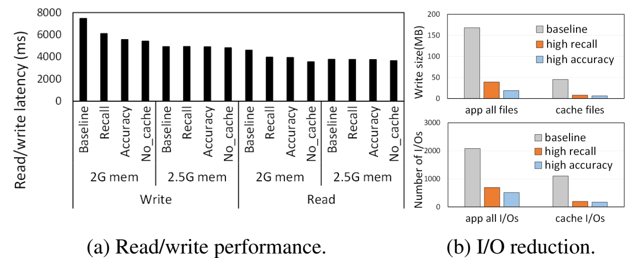
provement:

$$lifetime = \sum_{i=1}^{n} \frac{PEC_i \times (1 + OP_i)}{365 \times DWPD \times WA_i \times R_{Compress}} \quad (3)$$

In Equation 3, WA_i and OP_i are the write amplification and over provisioning factor for ECC_i, respectively, and PEC_i is the number of P/E cycles for which ECC_i is used. In our case, other parameters are constants, and thus the lifetime is inversely proportional to the number of full disk writes per day (DWPD) which depends on the amount of data written. Taking P9 as an example, we can reduce the amount of I/O by an average of 53.2% and 54.7%, respectively by the two models. Therefore, the lifetime can be improved by an average of 113.7% (1/(1-53.2%)-1) and 120.8% (1/(1-54.7%)-1), respectively.

6.2 Performance Improvement

Read/write performance improvement. Reduction in writebacks of cache files could improve read and write performance because of the reduction in I/O contention. To quantify the impact of writebacks of cache files on read and write performance, especially under intensive I/O, we assess the latency of running read/write micro-benchmarks when using a cache-intensive application, i.e., Facebook. Since most I/O sizes on mobile devices are in the size of 4KB [4], we sequentially write with fsync or read 512MB in size of 4KB by using the micro benchmarks to evaluate read/write performance. We scroll news on Facebook for five minutes and collect the latency of read and write in the default system (`Baseline`) and in the system with CacheSifter (`Recall` and `Accuracy`). `No_cache` represents the latency of read and write without using Facebook so that there is no interference of cache files generated by Facebook. We use memtester [34] to occupy physical memory, so that cache files will be written back quickly (to general the situation that memory is insufficient). To reduce bias, we conduct the experiment five times, and the average latency of the entire 512MB operation is presented in Figure 10a. To show more breakdown information, the I/O and writebacks produced by Facebook are presented in Figure 10b.



(a) Read/write performance. (b) I/O reduction.

Figure 10: The impact of CacheSifter on read and write performance under different memory pressure. The system could occupy approximately 2GB memory in this device.

In the baseline system, writebacks of cache files generated by Facebook degrade read and write performance by an average of 29.6% and 38%, respectively under memory pressure (with 2GB memory). Compared to `baseline`, the read and

write latency are reduced by an average of 13.9% and 18.4%, respectively by the high-recall model, while the numbers are 14.4% and 25.5%, respectively, when using the high-accuracy model. When there is sufficient memory (at least 2.5GB), the impact of cache files is marginal on read and write performance because few cache files will be written back to flash storage to generate I/O contentions with the read/write of the micro-benchmark. The performance improvement of CacheSifter derives from the write reduction of cache files. The benefit is significant under I/O intensive workloads or when memory is insufficient. According to paper [26], eight background applications are common. Memory pressure occurs frequently even in mobile devices with relatively large memory (8GB) as shown in Table 7.

Table 7: Free memory in mobile devices when running various numbers of applications.

Devices	Total memory	1 App	3 Apps	5 Apps	8 Apps	10 Apps
P9	3G	88M	80MB	90MB	82MB	80MB
Mate30	8G	1.8GB	1GB	680MB	167MB	95MB
Pixel6	8G	1.5GB	177MB	166MB	172MB	106MB

Impact of CacheSifter on framerate. Even though CacheSifter can improve read and write performance, re-accessing discarded cache files from networks can negatively impact user experience. We measure the possible loss in user experience with Frame Per Second (FPS) by PerfDog, a popular gaming benchmark [43]. Figure 11 shows the average FPS of Twitter. We choose Twitter as a foreground application, that is denoted as "F" because Twitter is another one of the most cache-intensive applications that could be relatively more affected by CacheSifter. There are various numbers of background applications, and "3B" means that there are three background applications. Background applications are randomly selected from the optimized ten applications.

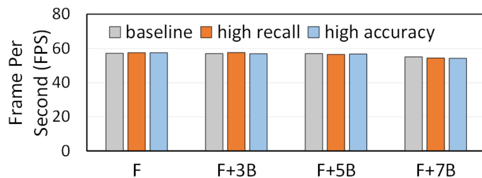


Figure 11: Impact of CacheSifter on application execution.

For the average FPS, the results in Figure 11 show that neither the high-recall model nor the high-accuracy model has a noticeable impact on FPS of the application execution. We also obtain the two important factors that impact FPS: CPU and peak memory. Table 8 lists the details of CPU usage and peak memory of Twitter. The results reveal that cache files being re-accessed by CacheSifter has a minimal impact on CPU usage and peak memory.

6.3 Overhead Analysis

Network overhead. Similar to the state changes shown in Table 2, there are six types of miscalssifications : "BR->TR,LL", "TR->BR,LL", and "LL->BR,TR". "BR-> TR, LL" means

Table 8: Information of the foreground application.

Factors	Methods	F	F+3B	F+5B	F+7B
Peak memory	baseline	334MB	323MB	302	304MB
	high recall	333MB	337MB	308MB	301MB
	high accuracy	343MB	328MB	315MB	323MB
CPU	baseline	9.9%	10%	8.9%	9%
	high recall	10%	10.1%	10.3%	10.7%
	high accuracy	10.7%	10.9%	10.5%	11.9%

that a BAR file is misclassified as a Transient file or a Long-living file. Notably, only three misclassifications, "TR->BR" and "LL->BR,TR", could induce re-download. Amount of these three misclassifications, the "LL->TR" case has small a possibility to be re-downloaded while other two cases have a large possibility to be re-downloaded. Based on this, we show the re-download upper bound and lower bound in Figure 12.

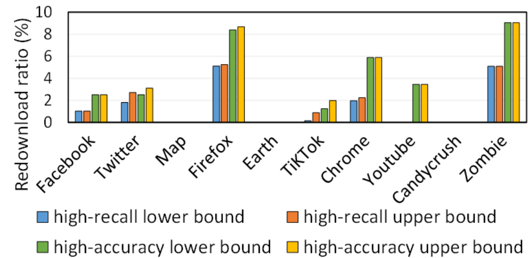


Figure 12: Re-download ratios of optimized applications.

The upper bound is equal to the total number (accurate and misclassified cases) divided by the sum of the number of these three cases. The lower bound is equal to the total number divided by the sum of the number of "LL,TR->BR" cases. The results show that the re-download penalty of high-recall mode is smaller than high-accuracy mode because of its goal (reducing re-download). This allows the operating system to deploy either high-recall or high-accuracy mode based on the users' network and data plan.

Memory overhead. Three components of CacheSifter introduce extra memory overhead: categorization, maintaining Transient files, and the ML inference. For categorization, cache files are maintained in the main memory until they are categorized. The extra memory usage depends on the size of Long-living files that are generated within 60 seconds because they are usually written back to the flash storage in default systems. The average memory overhead of this part is 492KB in our evaluations. The Transient files are stored in our QMFS with a maximum size of 20MB. When more than 10MB is occupied, a reclaim thread wakes up to free the memory, which ensures that the overhead stays below 10MB. Memory is used to run the machine learning method, specifically the inference step. Ten matrices for each model remain continually in the main memory for inference, which occupies approximately 2MB (0.89MB for the first model and 1.13MB for the second model). In summary, memory overhead is usually smaller than 12.5MB.

CPU time overhead. Training/retraining is conducted offline, and the overhead on smartphones is only the cost of categorization and eviction. We train a model for 20 applications by

using the data of 20 hours on a PC, and the training time is approximately one day. This training cost occurs only once over three months in our study. A dedicated thread wakes up periodically to conduct categorization and eviction. Categorization takes an average of 82ms out of 10s in our evaluation, as the matrices are relatively small. Moreover, the eviction scheme is used to shrink the in-memory file system. For this part, only the list move/insert operations are needed, and the overhead (1.9 ms out of 10s on average) is negligible. In summary, CPU time overhead is an average of 84ms for each iteration (10 seconds in our evaluation).

7 Related Works

Cache file optimization. User experience could be degraded due to too many cache files. Establish guidelines [10, 21] indicate that deleting the cache files of browsers can improve the overall performance of mobile devices. However, the deleted data must be re-downloaded from the network when users re-access them. This could degrade the performance, especially for the frequently-used data. Previous works [32, 38] show that keeping all of the cache files in the main memory can improve the performance because of their fast re-accessing. The benefits only occur when the cached files are accessed frequently. Otherwise, additional memory consumption may degrade the overall performance. Currently, the Android system and the existing works treat all cache files equally. Liang et al. [27] show that cache file vary greatly and should be managed differently but do not provide a corresponding solution.
Categorization of cache files. Caching files in memory is widely used to improve system performance. Korner et al. [22] firstly studied a knowledge-based remote file caching model and used multiple LRU lists to manage cache files on a server platform. Madhyastha et al. [29] employed a hidden Markov model to automatically classify file access patterns and tune the policies of the file system to improve global performance based on the observed patterns. In addition to a server platform, researchers introduced some cache file categorization schemes on mobile platforms. For example, Immanuel et al. [15] proposed a cache taxonomy that can decode several Android cache formats and display the contents in an accessible manner.
Eviction scheme. To our best knowledge, the eviction scheme used in CacheSifter is the first file-based eviction scheme to do so from within the kernel. Numerous page-based eviction schemes exist, which are usually designed based on the access locality of pages. The Linux firstly began to work on a page eviction mechanism from Kernel 2.4 [42], also termed page aging, which attempts to perform background scanning of the pages and use inactive lists to manage pages which are already idle. Liang et al. [25, 28] proposed a size-tuning scheme which can reduce pre-fetched pages in order to avoid a high page cache eviction ratio.
Server caching. The cache not only exists in mobile devices but also on servers. As the information provider, the caching mechanism on the servers is different from that on mobile devices, which are contents consumer for the most of time. The consistency of the cache [33] on distributed servers is the main concern. The cache used on the servers is also designed to reduce latency of responding to clients [1, 36]. Mital et al. [31] proposed a framework to store files across multiple SBSs. Jiang et al. [19] introduced a new DRAM caching techniques based on filter caches, and also presented two filter caching techniques and specified when they should be employed. Meng et al. [30] designed a dynamic, self-adaptive framework, called vCacheShare, which automate server flash space for the cache in virtual environments.

8 Conclusion

Current mobile systems treat cache files equally, storing them in the main memory first and then writing them back into flash storage. Mobile device performance depends heavily on cache utilization, with the challenges of tackling variable file patterns and flash durability. This paper proposes a cache file management scheme, named CacheSifter, to sift cache files by a lightweight machine-learning-based categorization engine and manage them by a set of eviction schemes to shield flash from ephemeral cache data writes. CacheSifter is evaluated on two Android devices and over a collection of representative applications. Evaluation results demonstrate that CacheSifter can reduce writebacks of cache files by an average of 62% and 59.5%, by using different models, and the I/O intensive write performance of mobile devices is improved by an average of 18.4% and 25.5%. We conclude that CacheSifter provides significant benefits to both I/O performance and storage lifetime with marginal overhead.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Prof. YouJip Won for their feedbacks and guidance. This paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (No.11204718) and National Natural Science Foundation of China (No. 61772092 and 61802038).

References

[1] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl. Reducing internet latency: A survey of techniques and their merits. *IEEE Communications Surveys Tutorials*, 18(3):2149–2196, 2016.

[2] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu. Error characterization, mitigation, and recovery in flash-

memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[3] David Chu, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *ACM MobiSys*. ACM, June 2012.

[4] J. Courville and F. Chen. Understanding storage i/o behaviors of mobile applications. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, May 2016.

[5] Balázs Csanád Csáji et al. Approximation with artificial neural networks. 2001.

[6] Android Developers. Android systems delete cache files. https://developer.android.com/training/data-storage/app-specificjava, 2021.

[7] Engineers. Android debug bridge (adb) tool. https://androidmtk.com/download-minimal-adb-and-fastboot-tool, 2019.

[8] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H.-M. Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2014.

[9] Google. Android source tree. https://source.android.com/setup/build/downloading, 2020.

[10] Michelle Greenlee. How to clear the cache on your android phone to make it run faster. https://www.businessinsider.com/how-to-clear-cache-on-android-phone, 2019.

[11] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, Santa Clara, CA, July 2017.

[12] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. Fasttrack: Foreground app-aware i/o management for improving user experience of android smartphones. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 15–28, 2018.

[13] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 147–162, July 2021.

[14] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. *ACM Transactions on Computer Systems (TOCS)*, 30(3):1–39, 2012.

[15] Felix Immanuel, Ben Martini, and Kim-Kwang Raymond Choo. Android cache taxonomy and forensic process. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 1094–1101. IEEE, 2015.

[16] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting quasi-asynchronous i/o for better responsiveness in mobile devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 191–202, 2015.

[17] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 61–74, Santa Clara, CA, February 2014.

[18] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/o stack optimization for smartphones. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 309–320, 2013.

[19] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[20] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. Application-aware swapping for mobile systems. *ACM Trans. Embed. Comput. Syst.*, 16(5s):182:1–182:19, September 2017.

[21] Adrian Kingsley-Hughes. Hidden android tricks to speed up your smartphone. https://www.lifehacker.com.au/2019/11/android-smartphone-running-slow-try-deleting-the-app-cache/, 2019.

[22] Kim Korner. Intelligent caching for remote file service. In *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 220–221. IEEE Computer Society, 1990.

[23] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887, July 2020.

[24] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[25] Yu Liang, Yajuan Du, Chenchen Fu, Riwei Pan, Liang Shi, and Chun Jason Xue. Boosting read-ahead efficiency for improved user experience on mobile devices. *ACM SIGBED Review*, 16(3):75–80, 2019.

[26] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 897–910, July 2020.

[27] Yu Liang, Jinheng Li, Xianzhang Chen, Rachata Ausavarungnirun, Riwei Pan, Tei-Wei Kuo, and Chun Jason Xue. Differentiating cache files for fine-grain management to improve mobile performance and lifetime. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, July 2020.

[28] Yu Liang, Riwei Pan, Yajuan Du, Chenchen Fu, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Read-ahead efficiency on mobile devices: Observation, characterization, and optimization. *IEEE Transactions on Computers*, 2020.

[29] Tara M Madhyastha and Daniel A Reed. Exploiting global input output access pattern classification. In *SC'97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 9–9. IEEE, 1997.

[30] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vcacheshare: Automated server flash cache space management in a virtualization environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 133–144, Philadelphia, PA, June 2014.

[31] N. Mital, D. Gündüz, and C. Ling. Coded caching in a multi-server system with random topology. *IEEE Transactions on Communications*, 68(8):4620–4631, 2020.

[32] Ngoan Nguyn. Ram disk: an app to mount a folder directly into the ram. https://apkpure.com/ramdisk/com.yz.ramdisk, 2019.

[33] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[34] pyropus technology. Memory test tool memtester. http://pyropus.ca/software/memtester/, 2017.

[35] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. USENIX Association, February 2021.

[36] S. P. Shariatpanahi, S. A. Motahari, and B. H. Khalaj. Multi-server coded caching. *IEEE Transactions on Information Theory*, 62(12):7253–7271, 2016.

[37] Z. Shen, L. Han, R. Chen, C. Ma, Z. Jia, and Z. Shao. An efficient directory entry lookup cache with prefix-awareness for mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2020.

[38] SoftPerfect. How to improve your computer performance and ssd life span with a ram disk. https://www.softperfect.com/articles/how-to-boost-computer-performance-with-ramdisk/, 2020.

[39] Statista. Number of smartphone users worldwide from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2020.

[40] Statista. Replacement cycle length of smartphones worldwide. https://www.statista.com/statistics/786876/replacement-cycle-length-of-smartphones-worldwide/, 2020.

[41] Linus Torvalds and thousands of collaborators. The linux kernel archives. https://www.kernel.org/, 2020.

[42] Rik Van Riel. Page replacement in linux 2.4 memory management. 2001.

[43] Wetest. Fps test tool perfdog. https://perfdog.wetest.net/, 2020.

[44] Sangjin Yoo and Dongkun Shin. Reinforcement learning-based SLC cache technique for enhancing SSD write performance. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, July 2020.

[45] Tao Zhang, Aviad Zuck, Donald E. Porter, and Dan Tsafrir. Apps can quickly destroy your mobile's flash - why they don't, and how to keep it that way (poster). *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, page 207–221, 2019.