SmartDedup: Optimizing Deduplication for Resource-constrained Devices

Qirui Yang, Runyu Jin, and Ming Zhao, ***Arizona State University***

https://www.usenix.org/conference/atc19/presentation/yang-qirui

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

SmartDedup: Optimizing Deduplication for Resource-constrained Devices

Qirui Yang
Arizona State University

Runyu Jin
Arizona State University

Ming Zhao
Arizona State University

Abstract

Storage on smart devices such as smartphones and the Internet of Things has limited performance, capacity, and endurance. Deduplication has the potential to address these limitations by eliminating redundant I/Os and data, but it must be considered under the various resource constraints of the devices. This paper presents SmartDedup, a deduplication solution optimized for resource-constrained devices. It proposes a novel architecture that supports symbiotic in-line and out-of-line deduplication to take advantage of their complementary strengths and allow them to be adapted according to a device's current resource availability. It also cohesively combines in-memory and on-disk fingerprint stores to minimize the memory overhead while achieving a good level of deduplication. SmartDedup is prototyped on EXT4 and F2FS and evaluated using benchmarks, workloads generated from real-world device images, and traces collected from real-world devices. The results show that SmartDedup substantially improves I/O performance (e.g., increases write and read throughput by 31.1% and 32%, respectively for an FIO experiment with 25% duplication ratio), reduces flash writes (e.g., by 70.9% in a trace replay experiment with 75.8% duplication ratio), and saves space usage (e.g., by 45% in a DEDISbench experiment with 46.1% duplication ratio) with low memory, storage, and battery overhead, compared to both native file systems and related deduplication solutions.

1 Introduction

Smart devices such as smartphones and the Internet of Things (IoT) are becoming pervasively used. The high volume and velocity of data produced by the growing applications and sensors on these devices present serious challenges to the on-device flash storage, which has limited performance, capacity, and endurance. Deduplication has the potential to address these limitations by reducing the I/Os and storage caused by duplicate data. But adopting deduplication on smart devices must address their unique resource constraints. In particular, the *limited memory* on the devices presents a difficult trade-off between the speed of deduplication and the amount of duplicates that it can find. The *limited storage performance and endurance* require the deduplication operations to incur minimal additional I/Os. For many devices with *limited power and energy capacity*, the design of deduplication also needs to be aware of its power and energy usage.

To address these challenges, this paper presents SmartDedup, a deduplication solution optimized for smart devices considering their various resource constraints. The architecture of SmartDedup is designed to support the symbiotic in-line and out-of-line deduplication. It employs in-line deduplication in the foreground to reduce redundant writes and runs out-of-line deduplication in the background to find duplicates missed by in-line deduplication. These two techniques work together cooperatively by sharing the fingerprint stores and information (e.g., fingerprints) about the I/O requests.

SmartDedup employs cohesively designed in-memory and on-disk fingerprint stores to minimize memory overhead while achieving a good level of deduplication. The small in-memory fingerprint store provides fast fingerprint lookup and effective write reduction; the on-disk fingerprint store supports complete data deduplication. Fingerprints migrate between the two stores dynamically based on data access patterns. SmartDedup also embodies several techniques to make efficient use of the two fingerprint stores. To further reduce the overhead, both fingerprint stores share the same indexing data structure to save memory usage; fingerprints are evicted from memory in groups to reduce the involved I/Os and wear-

out.

To support this study, we collected file system images and long-term (2-6 months) traces from real-world smartphones. The data confirms that there is a good level of duplication in real-world workloads: the average duplication ratio is 33% in the data from the images, and ranges from 22% to 48% among the writes in the traces. The specific applications and types of files that contribute the most duplicates differ by device, so a holistic system-level deduplication solution is needed to fully exploit these I/O and data reduction opportunities.

We prototyped SmartDedup on OSs (Android and Raspbian) and file systems (EXT4 and F2FS) commonly used by smart devices. We evaluated it on two representative devices, a Nexus 5X smartphone and a Raspberry Pi 3 device, using intensive benchmark (FIO [12]) and realistic workloads generated by DEDISbench [20] from sampling real-world smartphone images and by replaying real-world smartphone I/O traces. The results show that SmartDedup achieves substantial improvements in performance, storage utilization, and flash endurance compared to both the native file systems and related solutions.

For example, on Nexus, SmartDedup outperforms native EXT4 by 16.9% in throughput and 18.5% in 95th percentile latency for writes, and 38.2% in throughput and 18.7% in 95th percentile latency for reads, for intensive FIO workloads with only 25% duplicates; and it improves the write latency by 25.5% for a trace segment with a 47.9% duplication ratio. In terms of space saving, SmartDedup saves 45% of space (after factoring in its own space overhead) compared to EXT4 when the duplication ratio is 46.1% in a DEDISbench experiment. In terms of reducing the wear-out, SmartDedup reduces 70.9% of writes from EXT4 (after factoring in its own write overhead) in a trace replay with a duplication ratio of 75.8%. SmartDedup also outperforms the state-of-the-art related works Dmdedup [16, 23] by 1.5X in throughput and 57.1% in 95th percentile latency in an FIO write experiment with 25% duplicates, and CAFTL [9] by 37.7% in average write latency in a trace replay with 75.8% duplication ratio. All the improvements are achieved with low resource usage. In these experiments, SmartDedup uses less than 3.5MB of memory, and it actually reduces the battery usage by up to 7.7% and 49.2% for intensive workloads on Nexus and Pi, respectively.

2 Analysis of Real-world Device Data

To confirm the potential of deduplication for smart devices, we collected and analyzed I/O traces and file system images from smartphones used by different real users.

ID	Total I/Os (GB)	Read/ write ratio	# of days	Unique addresses (GB)	Unique data (GB)	Duplication ratio (%)
1	2096.8	7.5	173	148.2	192.7	21.9
2	773.6	3.4	85	36.3	96.2	45.3
3	426.7	4.8	73	34.6	56.5	23.2
4	2385.6	5.8	145	101.2	184.2	47.5
5	1119.7	3.4	92	126.7	148.6	41.6
6	765.9	3.4	72	23.9	124.8	28.3

Table 1: File system traces from real-world smartphones. For each trace, the table shows the total amount of data requests (4KB each), the ratio between reads and writes, the total number of days, the total amount of writes with unique addresses, the total amount of writes with unique data, and the total percentage of duplicate writes.

	Top 3 duplicate contributors (file type, % of writes, duplication ratio (%))		
Trace	1	2	3
1	(res, 12, 15)	(db, 10, 15)	(exe, 39, 13)
2	(tmp, 3, 73)	(exe, 36, 65)	(res, 27, 62)
3	(res, 17, 65)	(media, 3, 19)	(db, 11, 17)
4	(exe, 29, 78)	(tmp, 4, 77)	(media, 24, 49)
5	(media, 76, 48)	(res, 2, 34)	(exe, 12, 25)
6	(res, 19, 45)	(exe, 35, 36)	(media, 9, 31)

Table 2: The file types that contribute the most amount of duplicates in the file system traces collected from smartphones used by real users. For each top contributor, the table shows the file type, the percentage of writes that it contributes to the trace's total write volume, and the percentage of duplicate writes within only this type of files.

First, we studied the long-term file system I/O traces [4] collected from six smartphones (from VFS and EXT4 on Android) used by six users from four countries who are between 20 to 40 years old, which recorded the fingerprints of writes when they were flushed from the page cache to flash storage. Commonly used applications include chat (WhatsApp and WeChat), video (Youtube), and social network (Facebook and Weibo) applications. As summarized in Table 1, these traces confirm that: 1) real-world device workloads are indeed quite intensive, and do have a significant impact on the performance and endurance of devices. The average daily I/O volume ranges from 4.2GB to 17.6GB, and the amount of writes ranges from 0.7GB to 3.5GB; and 2) a good level of deduplication can be achieved on the writes captured in the traces. Considering the entire traces, the percentage of duplicate writes ranges from 21.9% to 47.5%; and on a daily basis, on average between 16.6% and 37.4% of writes are also duplicates.

To understand where the duplicates came from, we further analyzed the effectiveness of deduplication within

the writes to each type of files. We classified the files into several categories (*resource* files, *database* files, *executables*, *temporary* files, and *multimedia*), following the methodology in [14]. [1] Table 2 shows that the file types that contribute the most duplicates vary across the traces collected from different users' devices. This observation suggests that applying deduplication only to specific types of files, or even more narrowly, only to specific applications [17], is insufficient. Although not shown in the figure, our results also reveal that whole-file-based deduplication [3] is also insufficient as over 80% of the duplicates are from files that are not completely duplicate.

To complement the above trace analysis, we also studied file system images collected from 19 real-world smartphones, which on average have 10.4GB of data stored on the device and 33% duplicates in the data. The analysis also confirms that there is a good amount of duplicate data stored on the devices. We also analyzed the effectiveness of deduplication on different types of files, and as in the trace analysis, we did not find any pattern—the file types that contribute the most to deduplication differ across the devices. For example, on one image, a large percentage of duplicates is found within database files (69.1%) and apk files (74.8%), but this percentage is low on the other images; on another image, thumbnail files have a duplication ratio of 99.0% whereas on another image this ratio is only 0.9%.

Overall the above analysis of real-world device traces and images shows strong evidence for the potentials of deduplication on devices. They also suggest that a holistic, system-level solution is necessary to fully exploit the deduplication opportunities.

3 Design and Implementation

3.1 Design Overview

Overall the design of SmartDedup is based on the following key principles:

I The storage on smart devices has limited bandwidth, capacity, and endurance, so deduplication should be applied as much as possible to improve its performance, utilization, and lifetime.

II The available memory on devices is often limited, so the use of in-memory data structures should be kept as low as possible. To complement the low memory footprint, disk space should also be leveraged to keep additional data structures.

III Many smart devices are power or energy constrained (e.g., limited battery life), and deduplication should work adaptively according to the current power or energy availability.

While following these general principles, we also cautiously design the data structures and operations used by deduplication so that its overhead is as low as possible. The rest of this section presents first an overview and then details of this design.

Deduplication can be performed at different layers (file system or block layer) of the storage stack. SmartDedup chooses the design of file system level deduplication, which allows it to exploit useful semantics and improve efficiency (e.g., avoid deduplicating unallocated blocks or processing files that have not been modified). Although hints can be passed from the file system to the block layer [16], they may not be sufficient (e.g., for providing the above semantics), and the file system's unawareness of deduplication also leads to inefficiencies. For example, the file system either cannot exploit the space saved by deduplication or has to assume a fixed deduplication ratio which does not always hold for the actual workload.

According to Design Principle I, SmartDedup considers both *in-line* and *out-of-line* deduplication to maximize the effectiveness of deduplication. In-line deduplication removes duplicate writes before they reach the disk, and can thereby quickly reduce the data and avoid the wear-out caused by duplicates. But it needs to run in the I/O path at all times; otherwise, it may miss many deduplication opportunities. Out-of-line deduplication works in the background to remove duplicates already stored on disk, and can use fingerprints stored both in memory and on disk to identify duplicates. Although out-of-line deduplication can be integrated with garbage collection to reduce wear-out [10], it is not as effective as the in-line method which removes duplicates before they reach the disk. Therefore, SmartDedup combines in-line and out-of-line deduplication to take advantage of their complementary strengths, and optimizes their uses for resource-constrained devices. In particular, these two deduplication procedures share the same fingerprint store to reduce the resource overhead (per Design Principle II); and both procedures can be dynamically enabled or disabled and dynamically change the processing rate based on a device's current power or energy status (per Design Principle III).

According to Design Principle II, to address the memory limitations of smart devices, SmartDedup adopts cohesively designed two-level in-memory and on-disk fin-

[1] Our results show that the writes to database-related files account for 21.9% of the total amount of writes, which is much lower than the 90% observed by [14]. We believe that this discrepancy is because the related work considered only writes from Facebook and Twitter, whereas we analyzed system-wide writes.

gerprint stores. The fingerprint store is the core data structure of a deduplication solution: it maintains the fingerprints of existing data so that it can determine whether new data is duplicate or not by comparing the fingerprint to existing ones. In SmartDedup, the *in-memory fingerprint store* supports fast deduplication of commonly used data with low memory cost; the *on-disk fingerprint store* keeps fingerprints that are not in memory and supports more thorough data deduplication; and fingerprints can dynamically migrate between these two stores. Together, these two fingerprint stores support the efficient operation of both in-line and out-of-line deduplication.

The rest of this section explains the various components of SmartDedup and how they function together. Our core designs, including cooperative in-line and out-of-line deduplication and tiered fingerprint stores are applicable to different types of file systems. We use our prototypes for EXT4 [18], which is the *de facto* file system on Android devices, and F2FS [15], a new flash-optimized file system increasingly used by smart devices, to explain SmartDedup.

3.2 Two-level Fingerprint Stores

In-memory Fingerprint Store. SmartDedup uses only a small amount of memory to store important fingerprints (and the corresponding PBNs) and support fast deduplication, while the other less important fingerprints are kept on disk. When the in-memory fingerprint store gets full, some of the fingerprints are demoted to the on-disk fingerprint store to make room for new ones.

To support fast fingerprint search at low memory cost, the store uses a prefix tree (implemented using a Linux radix tree [8]) as the *fingerprint index* (Figure 1). To conserve memory, different from typical indexes which provide direct locations of individual fingerprints, our index provides the locations of fingerprint groups—the fingerprints in each group shares the same prefix. For example, with an 18-bit index, all the fingerprints that share the same 18-bit prefix are grouped together. This design also facilitates the group-based fingerprint eviction and on-disk fingerprint lookup discussed later in this section. Within each group, the fingerprints are indexed by a linked list. The list is sorted by the fingerprints' remaining bits, which allows misses to be determined sooner than using an unsorted list. Moreover, the length of such a list is generally short because 1) the in-memory fingerprint store is typically small, and 2) the cryptographic hash function used for fingerprinting tends to distribute the fingerprints evenly across the different groups. Experiments from replaying our traces confirm that the average length of these lists is 9 (maximum length is 63).

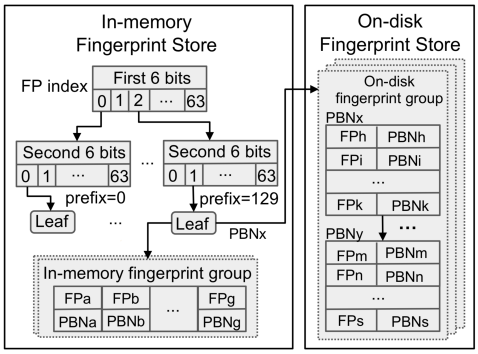


Figure 1: SmartDedup's two-level fingerprint stores.

With the above design, the space overhead of the in-memory fingerprint store is kept low. If we use 1% of the device's memory (40MB of a 4GB memory) to store MD5-based fingerprints, we can take the first 18 bits of each fingerprint as the prefix index and limit the height of the tree to three. Under this setting, the fingerprint index uses 2.03MB of memory. Considering the data structure overhead, the in-memory store can keep 1.3 million fingerprints for 5GB of unique data. For SHA1-based fingerprints, the number of fingerprints that the store can hold is 1.12 million.

The in-memory fingerprint store is used by both in-line and out-of-line deduplication as explained later. By allowing them to share this store, SmartDedup further reduces its memory usage on resource-constrained devices.

On-disk Fingerprint Store. The on-disk fingerprint store maintains the fingerprints that are evicted from memory due to the limited space of the in-memory fingerprint store. It allows SmartDedup to make full use of the existing fingerprints for providing thorough deduplication, and supports the promotion of fingerprints from disk to memory when they become important to the current workload. It is implemented as a sparse file on disk where the fingerprints are stored sequentially along with the PBNs. The size of the sparse file grows and shrinks, block by block, on demand with the number of fingerprints in the store for space efficiency. For 256GB of device storage, the total number of fingerprints that need to be stored on disk is 2^{26} in the worst case, assuming all the data is unique, which requires 2GB of disk space. In comparison, deduplicating merely 1% of 256GB of data saves 2.6GB of space and can already compensate the overhead of the on-disk store.

To enable fast search of on-disk fingerprints, SmartDedup also needs an index in memory, but it reuses the same *fingerprint index*—the prefix tree—described above for the in-memory store to reduce its memory usage (Figure 1). In fact, it adds only an address to each leaf node

of the index, which is the starting PBN of the on-disk group of fingerprints with the same prefix as the fingerprints of the in-memory fingerprint group. Each group of fingerprints is stored in an array on disk, which is sorted by its remaining fingerprint bits and stored sequentially from this PBN address in one or multiple disk blocks. In this way, the same fingerprint index is shared by both in-memory and on-disk fingerprint stores, and each leaf node can point to both an in-memory fingerprint group and an on-disk fingerprint group that share the same prefix. For a three-level tree that indexes an 18-bit prefix, the addition of the PBN in each leaf node adds at most 1MB of memory usage.

This scheme allows efficient operations on on-disk fingerprints. To search for a fingerprint on disk, SmartDedup looks for the corresponding leaf node in the fingerprint index. If the node does not exist in the prefix tree, SmartDedup knows immediately that the fingerprint does not exist on disk. If the node exists and contains a valid PBN, SmartDedup loads the whole group from that address into memory and searches for the given fingerprint in memory using binary search. The I/O overhead for accessing the on-disk groups is small because the size of each group is generally small. Assuming each group shares the 18-bit prefix, for 256GB of device storage with no duplicate data blocks, there are about 256 fingerprints per group, requiring only one to two 4KB blocks to store them. For even larger disks, we can increase the length of the prefix to bound the group size.

As discussed above, our proposed fingerprint index provides the functionality of a Bloom filter; in comparison, employing a separate Bloom filter incurs additional time and space overhead. For example, using a Bloom filter to determine whether a group of fingerprints exists or not would require 0.92MB of memory and applying five hash functions. In addition, it needs to deal with the difficulty of fingerprint deletions [7, 11].

Fingerprint Migration. Fingerprints evicted from the in-memory store are moved to the on-disk store; conversely, when an on-disk fingerprint is matched by new data, it is promoted from disk to memory. When deciding which fingerprints to evict, the in-memory fingerprint store tries to keep the fingerprints that are important to the current workload. Our evaluation results show that a simple policy such as least recently used (LRU) achieves good deduplication ratios (Section 4.2).

But the I/O overhead of fingerprint migrations is an important consideration for devices. Evicting a fingerprint from memory to disk requires two I/Os for loading the corresponding fingerprint group from disk and storing the updated group back to disk. To reduce disk I/O overhead, instead of evicting one fingerprint at a time, SmartDedup evicts a group of fingerprints at a time, so that a number of slots are freed up at once in the in-memory fingerprint store and can be used to store a number of fingerprints from the future requests. With the design of a prefix-tree-based index, the fingerprints linked to the same leaf node share the same prefix and automatically form an eviction group. Note that when migrating fingerprints from disk to memory, SmartDedup still promotes one fingerprint, instead of a whole group, at a time, since there is limited locality within each group. Moreover, with a small group size (9 on average), the deduplication ratio is also not compromised much by evicting the whole group together.

To implement a group-based eviction policy, SmartDedup keeps an LRU list for all the groups in the fingerprint index. Whenever a fingerprint is matched to a new request, its group is brought to the head of the LRU list. When eviction is needed, the entire group of fingerprints that is at the tail of the LRU list is evicted. Since both the in-memory and on-disk fingerprint stores share the same index, fingerprints that are evicted together from the in-memory fingerprint store also belong to the group that shares the same prefix in the on-disk fingerprint store, so they can be inserted into the on-disk group using a single read-merge-write operation.

3.3 Hybrid Deduplication

In-line Deduplication happens when the file system handles a write request, and it removes a duplicate write by modifying the file system's logical block to physical blocks mappings. Specifically, in our prototypes, the write paths of EXT4 and F2FS are modified in the following manner to make sure that the deduplication procedure does not violate the basic design principles of modern file systems. First, SmartDedup achieves deduplication by changing the one-to-one mappings that the file system maintains from logical blocks to physical blocks to many-to-one. Second, SmartDedup performs in-line deduplication when the file system writes back buffered data to disk; by doing so, it saves itself from processing repeated writes to buffered data, which does not hurt either performance or endurance. SmartDedup also handles direct I/Os, but the discussion here focuses on buffered I/Os since they dominate common device workloads.

In-line deduplication may not be able to find a match for a request even if there is a duplicate block on the file system, because the in-memory fingerprint store is not large enough to hold all the existing fingerprints. For such requests, in-line deduplication hands them over to out-of-line deduplication, which searches the on-disk fingerprint store in the background without slowing down the foreground application.

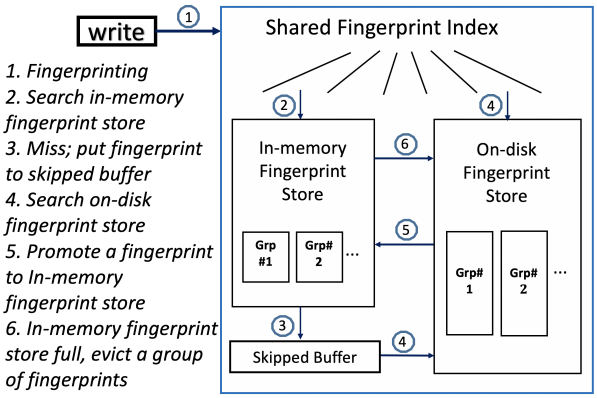


Figure 2: SmartDedup operations.

Out-of-line Deduplication works in the background on data that is not processed by in-line deduplication, which may be still in the page cache waiting to be written back or be already stored on disk. When processing a block of data, it looks for the fingerprint in both the in-memory and on-disk stores. When a match is found for a logical block, it changes the mappings between the logical block and physical block and, if needed, deallocates the redundant physical block to perform deduplication.

For efficiency, SmartDedup avoids processing data blocks that have not been modified since the last time they were processed by either in-line or out-of-line deduplication. It uses an in-memory buffer, called *skipped buffer*, which stores the list of blocks skipped by in-line deduplication, either because the latter is disabled or is enabled but cannot find fingerprint matches in the in-memory store. Each entry in this buffer stores a skipped block's information (inode number, LBN, and fingerprint (if available)) for the out-of-line deduplication to process the block quickly. It is implemented using an array-based hash table, indexed by inode and LBN. When the fingerprint of a data block is updated, the previous content of the hash table entry is replaced by the new one.

The size of the skipped buffer is kept small and is also adjustable depending on the device's current memory availability. For example, with 0.5MB of memory, the skipped buffer can store the information of 22K requests and 65K requests with and without their fingerprints, respectively. If the buffer does get full, SmartDedup converts it to store only the inode numbers of modified files so that out-of-line deduplication processes only these files. As the inode number requires only 4 bytes, the buffer rarely overflows in practice.

3.4 Putting Everything Together

Read and Write Path. When handling a write, SmartDedup fingerprints the request, searches for it in the fingerprint store(s), and deduplicates it if a match is found, as discussed above and illustrated in Figure 2. The read path in the file system is also modified to make use of deduplication to improve read performance. Because the page cache is indexed by each file's LBNs, a read that cannot find its LBN in the page cache cannot be avoided even if the requested data duplicates another logical block that is already in the cache.

To address this limitation, SmartDedup employs a *page cache index* which maps from PBNs to their corresponding pages in the page cache. For a read that cannot find a match in the page cache by its LBN, SmartDedup searches the index using its PBN, before considering it a miss as the native file system does. If a match is found, it means that the requested data already exists in the page cache, and there is no need to perform the actual read I/O. SmartDedup directly copies the data from the duplicate page in the cache. The size of this index is bounded by the size of page cache, and it can be further restricted when SmartDedup indexes only the important set of pages (e.g., the most recently used ones) in the cache.

Handling Data Updates and Deletions. A complication to the above write process is that when handling an update to or deallocation of an existing block, the fingerprint stores need to be consistently updated. SmartDedup needs to find the original fingerprint of the block so that it can update the reference count (the number of logical blocks deduplicated by this fingerprint) and delete the fingerprint if the reference count drops to zero. But SmartDedup does not have the original data's fingerprint; it has only the new request's LBN (and from there the PBN). To address this problem, SmartDedup maintains a *reverse index* (a sparse file) on disk for the fingerprint stores, which maps from a fingerprint's PBN to its corresponding leaf node in the *fingerprint index* of the fingerprint store using an array where the PBN is the index and the entries store the leaf node addresses. The leaf node represents a group of fingerprints either in memory or on disk, and SmartDedup can search this group to quickly locate the fingerprint. Compared to mapping from PBNs directly to the locations of the fingerprints, this design reduces the overhead of the reverse index because when a fingerprint migrates between the in-memory and on-disk stores, the leaf node that the fingerprint belongs to does not change and the index does not have to be updated.

Adaptive Deduplication. To further reduce the overhead of deduplication, SmartDedup can adapt its processing rate based on the current resource availability. For out-of-line deduplication, SmartDedup adapts the number of blocks that it processes per period of time (e.g., every minute). For in-line deduplication, it adapts the process-

ing rate by selectively processing n out of the N write requests that it receives—n/N defines the *selectivity*. Considering CPU and I/O load, SmartDedup automatically reduces its processing rate whenever it detects that the CPU or disk is fully utilized. Considering battery usage, SmartDedup reduces its processing rate proportionally to the remaining battery life, and completely disables deduplication when the device enters low-power mode. Similar policies for other resource constraints can also be easily specified and carried out by SmartDedup using this adaptive deduplication mechanism.

To further reduce resource usage, SmartDedup can also adapt its processing rate (by adjusting the selectivity) based on the level of data duplication observed in the workload. When the observed duplication level is low in the previous time window, SmartDedup gradually reduces its processing rate, but it quickly restores its processing rate when it detects an increasing duplication level in the current workload.

File System Consistency. Because the LBN-to-PBN mapping is already kept consistent by the native file system, the only metadata that SmartDedup needs to safekeep is the reference counts of the fingerprints—it relies on the reference counts to decide when to free a data block and when to perform copy-on-write. SmartDedup stores the reference counts persistently as part of the on-disk reverse index (together with the leaf node addresses of the corresponding fingerprints as described above).

To ensure consistency, on EXT4, SmartDedup journals the modifications to the reverse index as part of the file system journaling. The design of the reverse index helps reduce the overhead from its journaling. The entries in the index are sorted by the PBNs, so consecutive updates to the reference counts of adjacent physical blocks can be aggregated into much fewer updates to the reverse index blocks—a 4KB block stores 512 entries. Experiments using our traces confirm that the amount of additional writes to the reverse index is less than 0.5% of the total write volume. After a crash, the file system can be brought back to a consistent state by replaying the latest valid journal transaction. Similarly, on F2FS, SmartDedup ensures that modifications to the reverse index and on-disk fingerprint store are captured by the file system checkpoint so that they can always be brought to a valid state after the file system's crash recovery. The overhead of recovery is also small as it requires only updating the affected reference counts using the journal or checkpoint.

All other data structures that SmartDedup maintains can be safely discarded without affecting file system consistency. The in-memory fingerprint store will be warmed up again after the system recovers. The page cache index will be reconstructed as the page cache warms up again. The loss of the skipped buffer will make SmartDedup miss the requests that have not been processed by out-of-line deduplication. For a 0.5MB skipped buffer, at most 254MB of data will be missed (assuming that the buffer stores only the inode and LBN of each request). To reduce this impact, SmartDedup periodically checkpoints the inodes and LBNs from the skipped buffer.

	Nexus 5X	*Raspberry Pi 3*
CPU	Qualcomm Snapdragon 808	Broadcom BCM2837
RAM	2 GB	1 GB
Storage	32GB eMMC	16GB SDHC UHS-1
Operating System	Android Nougat	Raspbian Stretch Lite
Kernel Version	Linux 3.10	Linux 4.4
File System	EXT4	F2FS

Table 3: Specifications of the testing devices.

4 Evaluation

We evaluated SmartDedup based on prototypes implemented on EXT4 and F2FS. Testing devices include a Nexus 5X phone and a Raspberry Pi 3 device (Table 3). We considered the following workloads to provide a comprehensive evaluation:

- *FIO* [12]: We used FIO to create intensive I/O workloads with different access patterns and levels of duplication.
- *Trace Replay*: We replayed our collected real-world smartphone traces (Table 1), which helps us understand the performance of SmartDedup for real-world workloads of smart devices.
- *DEDISbench* [20]: We used DEDISbench to scan our collected real-world Android images and then generate workloads that reflect the data duplication characteristics (such as the distribution of reference counts) of these images.

We compared SmartDedup to two related solutions: Dmdedup [16, 23] and CAFTL [9]. Dmdedup is a block-level in-line deduplication solution that supports flexible metadata management policies. It can use a copy-on-write B-tree to store metadata and provide a consistency guarantee by flushing metadata periodically. To provide a fair comparison, we further enhanced Dmdedup by passing hints from the file system and allowing it to flush metadata only at journal commit times.

CAFTL implements both in-line and out-of-line deduplication at the flash translation layer (FTL), with several techniques designed for the resource constraints at this layer. Sampling hashing fingerprints only one data

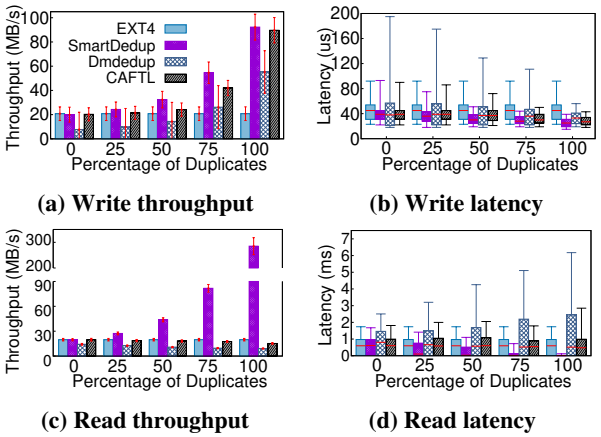
block in a request that has multiple blocks to reduce overhead. Lightweight pre-hashing applies CRC32 to pre-hash data blocks and filter out the unmatched ones to save fingerprinting overhead. Dynamic switches dynamically enable and disable deduplication based on the available cache space. For a fair comparison, we implemented sampling hashing and lightweight pre-hashing in the EXT4 writeback path, but dynamic switches are not necessary because the page cache supports rate limiting.

In all the experiments, the memory usage was capped at 3.5MB for all the evaluated solutions (unless otherwise noted). SmartDedup used 3MB for in-memory fingerprint store, which used a 14-bit prefix, and 0.5MB for the skipped buffer, which stored full information and never overflowed during the experiments. All the experiments were started with empty in-memory and on-disk fingerprint stores. Both in-line and out-of-line deduplication were used in the experiments (unless otherwise noted); adaptive deduplication was enabled and evaluated only in Section 4.4. SmartDedup uses fingerprints generated by cryptographic hash functions [19, 21] to find duplicates. The overhead for fingerprinting one 4KB block of data on Nexus 5X is about 9μs if using SHA1 and 16μs if using MD5, and on Pi is about 30μs for both. Due to limited space, we present only the MD5 results here. Each experiment was repeated at least five times.

4.1 FIO

We ran FIO with three threads, each issuing random 4KB reads or writes, using buffered I/Os (which is what real-world applications typically use). For all FIO experiments, the total read or write size was set to 2GB on Nexus and 1GB on Pi. We varied the percentage of duplicates in the workloads; at 0%, SmartDedup's in-memory fingerprint store can hold 20% and 10% of total fingerprints for the 1GB and 2GB experiments, respectively. The read experiments were performed using random reads on the data written by FIO in the write experiments (after the page cache was dropped).

Nexus 5X Results. Figure 3a and 3b show the write performance on Nexus 5X. The worst case for SmartDedup is when there is no duplicate, where SmartDedup has only 3.8% overhead in throughput and 1.1% overhead in 95th percentile latency compared to EXT4, including all the overhead from fingerprinting and operations on in-memory and on-disk fingerprint stores. In comparison, Dmdedup has a much higher overhead, 62.8% in throughput and 1.1X in 95th percentile latency, which we believe is due to 1) deduplication at the block layer adds another level of LBN-to-PBN mapping and additional overhead; 2) to guarantee consistency, the copy-on-write



(a) Write throughput (b) Write latency

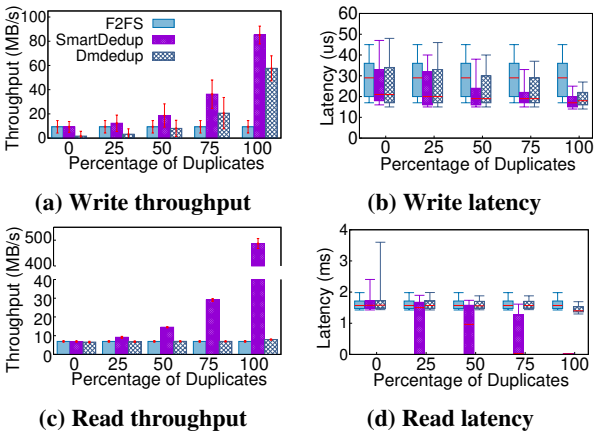
(c) Read throughput (d) Read latency

Figure 3: FIO on EXT4 using Nexus 5X. Figures 3a and 3c show the average write and read throughput, respectively, with the error bars showing the standard deviations. Figures 3b and 3d illustrate the write and read latency results, respectively, using box and whisker plots, where the whiskers indicate the 5th and 95th percentiles.

B-tree requires more metadata I/Os. For this experiment, Dmdedup introduces 75.6% more metadata writes than EXT4, whereas SmartDedup introduces only 9% more. CAFTL has less overhead than SmartDedup (1.4% less in throughput and 3.2% less in 95th percentile latency) because, with no duplicates in the workload, its pre-hashing can save substantial fingerprinting.

As the percentage of duplicates in the workload grows, the performance of SmartDedup quickly improves and exceeds EXT4. With 25% duplicates, SmartDedup already outperforms EXT4 by 16.9% in throughput and 18.5% in 95th percentile latency. Dmdedup has an overhead of 52.8% in throughput and 90.2% in 95th percentile latency. CAFTL outperforms EXT4 by only 3.1% in throughput and 6.5% in 95th percentile latency. We found out that using pre-hashing hurts deduplication performance—since requests are filtered out by pre-hashing, CAFTL does not have their fingerprints and cannot deduplicate future requests that have the same data. To verify this observation, we tried removing pre-hashing from CAFTL and the deduplication ratio indeed increases (by 14% for FIO with 25% duplicates). Without pre-hashing, CAFTL is still slower than SmartDedup (6.7% in throughput and 9.3% in 95th percentile latency), because 1) its out-of-line deduplication works only when the system is idle and cannot help much; 2) its reference-count-based eviction policy cannot exploit temporal locality in fingerprint accesses (further discussed in Section 4.2).

Figure 3c and 3d compare the read performance. Even in the worst case with no duplicates, the overhead of SmartDedup is small, merely 0.6% in throughput and 2.7% in median read latency, which is mainly from main-



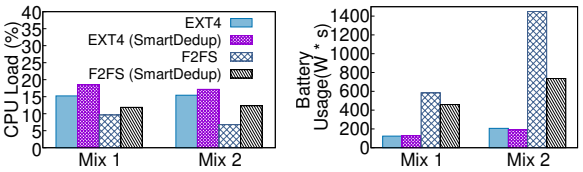
(a) Write throughput **(b) Write latency**

(c) Read throughput **(d) Read latency**

Figure 4: FIO on F2FS using Raspberry Pi. Figure 4a and 4c represent the average write and read throughput, respectively, and the error bars represent the standard deviations. Figure 4b and 4d illustrate the write and read latency results, respectively, using box and whisker plots, where the whiskers indicate the 5th and 95th percentiles.

taining the page cache index. As the percentage of duplicates increases, the performance of SmartDedup rises accordingly, as expected, and SmartDedup substantially outperforms EXT4, improving the throughput by up to 13.4X and reducing the 95th percentile latency by up to 93%, owing to the page cache index which allows SmartDedup to use cached data to satisfy read requests for different logical blocks but with the same content (as discussed in Section 3.4). In comparison, both Dmdedup and CAFTL are slower than EXT4. CAFTL has up to 23% and 64.1% overhead in throughput and 95th percentile latency, respectively, which we believe is due to the fragmentation induced by deduplication [13]. SmartDedup's use of page cache index helps compensate for this overhead; additional techniques [22] can also be adopted to address this problem as discussed in Section 4.5.

Raspberry Pi Results. Figure 4 compares the FIO performance on Raspberry Pi which has even fewer resources than Nexus 5X. The results show that SmartDedup also achieves substantial improvements compared to the native file system (F2FS) and the related solution (Dmdedup). For example, SmartDedup achieves a speedup of 31.1% and 32% in throughput for the write and read workloads with 25% duplicates, respectively compared to F2FS. Compared to Dmdedup, the throughput improvement is 2.8X and 34.5% for write and read, respectively. Although the improvement of 95th percentile latency is not as significant as in the Nexus results, SmartDedup still improves the 99.5th percentile latency substantially. Compared to F2FS, SmartDedup reduces the write tail latency by 30.4% and the read tail latency by



(a) CPU load **(b) Battery usage**

Figure 5: Resource usage of FIO.

	Write size (MB)	Duplication ratio (%)	Read/Write ratio	Source Trace #
Segment 1	17612.8	75.8	1.5	4
Segment 2	12697.6	47.9	2.2	6
Segment 3	9318.4	26.4	6.8	2
Segment 4	65.8	46.1	51.3	4
Segment 5	78.5	19.4	69.8	4

Table 4: Trace segment statistics.

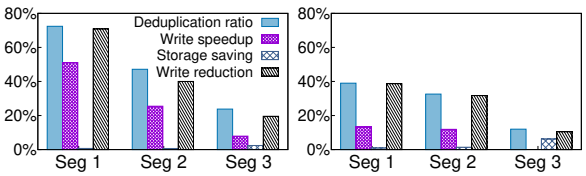
15.7%; and it outperforms Dmdedup by 85.4% in write and 18.7% in read.

Resource Usage. To understand the resource usage under realistic settings, we used FIO with different mixes of reads and writes to mimic the composition of real-world workloads. In our traces, the percentage of reads varies from 77.4% to 88% and the percentage of duplicates varies from 21.9% to 47.5%. For power measurement, we used the Trepn profiler [24] (for Nexus) and Watts Up Pro [5] (for Pi). Figure 5 shows the results. In Mix 1, the workload has 4GB of I/Os with 50% reads and 25% duplicates. SmartDedup's CPU overhead on EXT4 and F2FS is 3.3% and 2.2%, respectively, which are both reasonably small. For battery usage, SmartDedup has 4% overhead on EXT4 and uses 21.2% less battery on F2FS. In Mix 2, the workload consists of 6GB of I/Os with 66% reads and 25% duplicates. In this setting, SmartDedup's CPU overhead is merely 1.7% on EXT4 and 5.5% on F2FS; but it actually saves 7.7% of battery usage on EXT4 and 49.2% on F2FS because its saving on FIO runtime outweighs its overhead in power consumption. SmartDedup achieves these results while using only 0.2% (3.5MB out of 2GB) of the device's memory. Therefore, it is reasonable to believe that for typical device workloads, SmartDedup does not incur much resource overhead, and can in fact save the battery usage of the devices.

4.2 Trace Replay

The above FIO results give us insight into SmartDedup's performance and overhead under highly intensive settings. In the following two sections, we consider more realistic workloads using traces and images collected from real-world smartphones.

Replay on Real Devices. We replayed several represen-



(a) SmartDedup **(b) CAFTL**

Figure 6: Trace replay on EXT4 using Nexus 5X.

tative segments of the traces as summarized in Table 4 (Segments 1 to 3) using the real implementation of SmartDedup and CAFTL on the Nexus 5X device. These segments have different levels of duplication and all have a substantial amount of writes. Therefore, they can well represent the characteristics of the entire traces.

Figure 6 shows the results of the trace replay, including the achieved deduplication ratio and the speedup and the space and write savings compared to EXT4. The deduplication ratio reported here as well as the rest of the evaluation is computed using only the duplicates discovered by in-line deduplication. SmartDedup delivers a good speedup, up to 51.1%. It also achieves a high level of write reduction, up to 70.9%, after factoring in its own overhead in journaling and managing the on-disk data structures, e.g., 24.8 MB and 127.9 MB data were written to the reverse index and on-disk fingerprint store, respectively, during the replay of Segment 1. But there is not much space saving here, mainly because these trace segments are dominated by updates to existing data on the file system. During the replay of Segment 2, 34.2 MB and 12.5 MB disk space were used by the on-disk fingerprint store and reverse index, respectively. In comparison, CAFTL achieves less improvement (up to 13.4% and 38.7% in speedup and write reduction, respectively).

Since navigation is one of the typical applications on IoT devices [1, 2], we extracted the Google Map I/Os from our smartphone traces and replayed them (Segments 4 and 5 in Table 4) on the Raspberry Pi. As shown in Figure 7, SmartDedup also achieves good write speedup (up to 30.9%) and reduction (up to 47%) on the Pi.

Replay on Simulator. We also replayed three entire traces listed in Table 1 on a simulator of SmartDedup. The simulator implements SmartDedup's data structures and operations in user space and allows us to replay these months-long traces within a reasonable amount of time. Even though it does not model real-time performance, it allows the study of several important aspects of SmartDedup by replaying the entire traces.

First, we used the simulator to study the impact of our group-based fingerprint eviction (*G-LRU*) versus the standard, individual fingerprint eviction (*LRU*). Figure 8 shows that for different traces, *G-LRU* achieves a deduplication ratio that is at most 3% lower than *LRU*, which confirms that group-based eviction does not compromise the effectiveness of deduplication while saving substantial I/O overhead (on average 87%).

Next, we compared our recency-based fingerprint replacement, which replaces the least recently used fingerprint, to CAFTL's reference-count-based replacement, which replaces the fingerprint with the smallest reference count. As discussed in Section 4.1, pre-hashing is detrimental to deduplication ratio; here we considered the modified CAFTL that does not use pre-hashing. The results confirm the importance of exploiting temporal locality which allows SmartDedup to achieve 41.8% higher deduplication ratio than CAFTL.

We also studied the effectiveness of SmartDedup's in-memory fingerprint store design, which uses a prefix tree to index the fingerprints, by comparing its results to Dm-dedup, which uses a copy-on-write B-tree as the index. We varied the amount of memory that each solution is allowed to use from 1MB to 40MB. The results show that SmartDedup's memory-conserving designs allow it to achieve higher deduplication ratios (by up to 35.7%), especially when the available memory is limited.

Finally, we evaluated the effectiveness of our two-level fingerprint store design by comparing the deduplication ratio of G-LRU with (*G-LRU*) and without (*G-LRU (in-line only)*) the on-disk fingerprint store. As expected, when the in-memory fingerprint store is small (1MB), the availability of an on-disk store and out-of-line deduplication improves the deduplication ratio from 6.7% to 12.1% (Trace 1). With a larger in-memory fingerprint store, the use of an on-disk fingerprint store still increases the deduplication ratio from 36.7% to 43.1% (Trace 2). These results prove that our designs for synergistic in-line and out-of-line deduplication with two-level fingerprint stores work well for real-world workloads, and they are particularly important for devices with limited memory capacity.

4.3 DEDISbench

In addition to using real traces, we also created additional workloads by sampling real-world smartphone images using DEDISbench [20]. We chose two of the smartphone images that we collected, with duplication ratios of 46.1% and 19.4%, and used DEDISbench to generate workloads that represent the data duplication characteristics of these images. All experiments were done on Nexus 5X using four threads and a total of 2GB of random 4KB reads or writes (SmartDedup's in-memory fingerprint store can

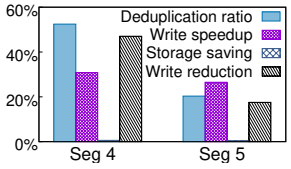
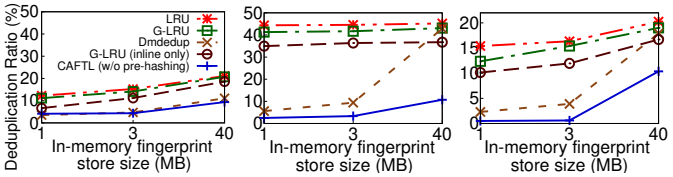


Figure 7: Trace replay on F2FS using Pi



(a) Trace1 (b) Trace2 (c) Trace3

Figure 8: Deduplication ratio from different migration policies.

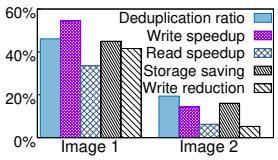
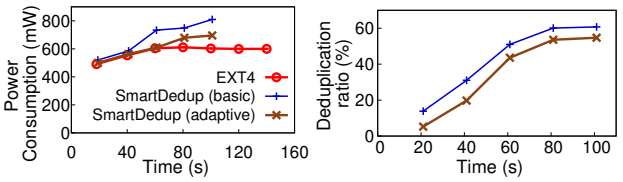
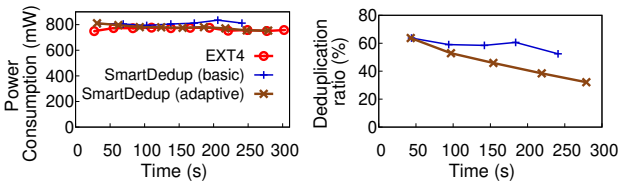


Figure 9: DEDISbench



(a) Power consumption (b) Deduplication ratio

Figure 10: Adaptive deduplication based on duplication level



(a) Power consumption (b) Deduplication ratio

Figure 11: Adaptive deduplication based on available battery.

hold 20% of all the fingerprints during these experiments). The write experiment was done in the peak mode (the more intensive mode of DEDISbench) with the hotspot I/O distribution (which DEDISbench uses to model real-world workloads with hotspot regions in their requests). The read experiment was done by reading from what DEDISbench generated in the write experiment (after the page cache was dropped).

Figure 9 shows the deduplication ratio achieved by SmartDedup and its I/O speedups, storage savings, and write reduction compared to EXT4. The write and read speedups are both significant, up to 54.4% and 33.6%, respectively, and largely follow the deduplication ratio of the workload. The read speedup is lower than the write speedup, because not all duplicate data can be found in the page cache due to cache evictions.

The space and write savings are also substantial, up to 45.0% and 41.6%, respectively. Note that these savings are computed after SmartDedup's overhead—the space and writes used for its on-disk data structures (including the on-disk fingerprint store and the reverse index and its journal)—is factored in. These results confirm the effectiveness of our techniques (Sections 3.2 and 3.4) for reducing the I/O overhead of deduplication.

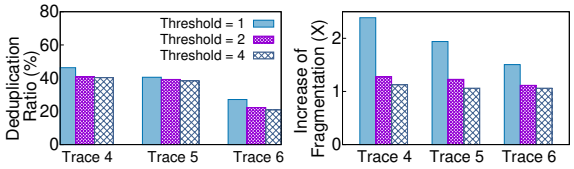
4.4 Adaptive Deduplication

Next, we evaluated the effectiveness of adaptive deduplication described in Section 3.4 by replaying trace segments on Nexus 5X. The first experiment studied the effectiveness of adapting deduplication selectivity based on the level of duplication observed in the workload. Following the general strategy described in Section 3.4, the specific algorithm used by SmartDedup is as follows. It computes the deduplication ratio of the last window—the last 150 write requests—and compares it to the average ratio from the past 30 windows. If the former is lower, it indicates that the current workload has fewer duplicates, and SmartDedup slowly reduces the percentage of requests that it fingerprints in-line in the next window (by 10% until it reaches a lower bound of 30%). But if the deduplication ratio of the last window is higher, SmartDedup quickly increases the percentage of requests that it fingerprints in-line in the next window (by 30% until it is back to 100%). How quickly SmartDedup adjusts its selectivity offers a tradeoff between performance and battery usage. We omit the sensitivity study's results due to lack of space. With the setting mentioned above, SmartDedup reduces its power consumption overhead (compared to EXT4) by up to 14% at the cost of 8% loss in deduplication ratio (as shown in Figure 10).

The second experiment evaluated adaptive deduplication based on the available battery level (Figure 11). We replayed a 12-hour long trace segment and assumed that the device's battery level was 100% (when it was fully charged) at the start of the replay and dropped to 20% (when it entered low-power mode) at the end. With adaptive deduplication, SmartDedup automatically increased the selectivity of fingerprinting as the available battery reduced. The power consumption overhead (compared to EXT4) dropped from 8%, when the battery level is 100%, to 0.3% when the battery level is 20%, at the cost of reducing the deduplication ratio from 51% to 32%.

4.5 Fragmentation Resistance

Deduplication usually brings fragmentation to disk and can hurt I/O performance. Even though flash storage is



(a) Deduplication ratio result **(b) Increase of fragmentation w.r.t. EXT4**

Figure 12: Fragmentation-resistant deduplication.

much less affected by fragmentation than HDDs, Hahn et al. [13] showed that flash devices still suffer from fragmentation due to increased I/O stack overhead. To address fragmentation, we leveraged the filtering technique from iDedup [22], which applies deduplication only to a physically contiguous sequence of writes that are duplicates. It sets a threshold on the length of a duplicate sequence, and filters out all sequences shorter than this threshold.

To evaluate the effectiveness of this filtering technique in SmartDedup, we replayed three complete traces listed in Table 1 on our simulator while varying the value of the threshold from one to four (when the threshold is one, the filtering is essentially disabled). We evaluated the impact on fragmentation, by measuring the total number of extents created by the workload. The results in Figure 12 confirm that by integrating the filtering technique, SmartDedup can reduce fragmentation without hurting the effectiveness of deduplication. For example, as the threshold increases from one to four, the deduplication ratio of SmartDedup drops by 6.0% while the increase in fragmentation (compared to native EXT4) reduces from 1.5X to 1.06X for Trace 6.

5 Related Work

There are several related deduplication solutions designed for resource-constrained systems. As mentioned in Section 4.1, CAFTL includes several techniques designed for deduplication on flash device controllers. The key differences of SmartDedup are its symbiotic use of in-line and out-of-line deduplication and the synergistic combination of in-memory and on-disk fingerprint stores for low-overhead and effective deduplication. In comparison, CAFTL relies mainly on in-line deduplication, while its out-of-line deduplication plays only a minor role and is completely separate from the former.

A recent study [17] proposed per-application, in-line deduplication for smartphones. It groups the fingerprints by applications, loads only the group for the foreground application, and swaps it out to disk when the application is switched to the background. As discussed in Section 2, per-application deduplication can miss many duplicates that exist across different applications. Moreover, migrating applications' entire fingerprint sets between memory and disk can be expensive when they become large. For example, our traces show that commonly used applications such as Gmail and Youtube have over 20MB of fingerprints and Weibo has 40MB. In comparison, SmartDedup supports system-wide deduplication with fine-grained fingerprint migration, and performs well with much lower memory usage.

Hybrid use of in-line and out-of-line deduplication has been studied in other related works. For example, DDFS [6, 25] employs both in-line and out-of-line deduplication for backup systems, and like CAFTL, they are not well integrated as in SmartDedup. DDFS caches fingerprints in memory to reduce on-disk fingerprint lookups, but unlike SmartDedup's in-memory fingerprint store, it is not designed for memory-constrained scenarios. For example, DDFS requires complex data structures to organize fingerprints that are grouped by their spatial locality. This design is important for deduplication on high-performance backup systems, but is unnecessary and costly for deduplicating the primary storage of low-end devices.

6 Conclusions and Future Work

This paper presents a deduplication solution optimized for smart devices. The novelties of this work lie in a new architectural design that synergistically integrates in-line with out-of-line deduplication and in-memory with on-disk fingerprint stores. The entire solution is cautiously designed and optimized considering the various resource constraints of smart devices. An extensive experimental evaluation based on intensive workloads and smartphone images and I/O traces confirms that SmartDedup can achieve substantial improvement in performance, endurance, and storage utilization with low memory, disk, and battery overhead. In our future work, we will further study the effectiveness of SmartDedup in other types of resource-constrained environments such as various Internet of Things and embedded storage controllers.

7 Acknowledgements

We thank the anonymous reviewers and our shepherd, Geoff Kuenning, for their thorough reviews and insightful suggestions. We also acknowledge Wenji Li and other colleagues at the ASU VISA research lab for their help in collecting the traces. This research is sponsored by the National Science Foundation CAREER award CNS-1619653 and awards CNS-1562837, CNS-1629888, IIS-1633381, and CMMI-1610282.

References

[1] Android auto. https://www.android.com/auto/.

[2] Apple car play. https://www.apple.com/ios/carplay/.

[3] Apple file system (APFS). https://developer.apple.com/wwdc/.

[4] VISA lab traces. http://visa.lab.asu.edu/traces.

[5] Watts Up Pro power meter. https://www.vernier.com/products/sensors/wu-pro/.

[6] Yamini Allu, Fred Douglis, Mahesh Kamat, Philip Shilane, Hugo Patterson, and Ben Zhu. Backup to the future: How workload and hardware changes continually redefine Data Domain file systems. *Computer*, 50(7):64–72, 2017.

[7] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: From I/O ports to process management*. O'Reilly Media, Inc., 2005.

[9] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[10] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 29–44, Santa Clara, CA, 2017.

[11] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[12] FIO — Flexible I/O tester synthetic benchmark. http://git.kernel.dk/?p=fio.git.

[13] 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 *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 759–771, Santa Clara, CA, 2017.

[14] 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.

[15] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, 2015.

[16] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 315–322, 2016.

[17] Bo Mao, Suzhen Wu, Hong Jiang, Xiao Chen, and Weijian Yang. Content-aware trace collection and I/O deduplication for smartphones. In *Proceedings of 33rd International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[18] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new EXT4 filesystem: Current status and future plans. In *Proceedings of the Linux symposium*, pages 21–33, 2007.

[19] FIPS PUB NIST. 180-1: Secure hash standard, 1995.

[20] Joao Paulo, Pedro Reis, Jose Pereira, and Antonio Sousa. DEDISbench: A benchmark for deduplicated storage systems. In *Proceedings of Confederated International Conferences On the Move to Meaningful Internet Systems*, pages 584–601. Springer, 2012.

[21] Ronald Rivest. The MD5 message-digest algorithm. RFC 1321, Internet Request For Comments, 1992.

[22] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2012.

[23] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *Proceedings of Linux Symposium*, 2014.

[24] Trepn power profiler. https://developer.qualcomm.com/software/trepn-power-profiler/.

[25] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2008.