Sparrow: Flexible Memory Deduplication in Android Systems with Similar-Page Awareness

Guangyu Wei[1], Changlong Li*[1,2], Rui Xu[1,3], Qingfeng Zhuge[1], Edwin H.-M. Sha[1]
gywei@stu.ecnu.edu.cn, clli@cs.ecnu.edu.cn, ruixu55@cityu.edu.hk, {qfzhuge, edwinsha}@cs.ecnu.edu.cn
[1] School of Computer Science and Technology, East China Normal University
[2] MoE Engineering Research Center of Hardware/Software Co-Design Technology and Application
[3] Department of Computer Science, City University of Hong Kong

***Abstract*—Mobile devices have become ubiquitous in daily life. In contrast to traditional servers, mobile devices suffer from limited memory resources, leading to a significant degradation in the user experience. This paper demonstrates that the primary cause of memory consumption lies in anonymous pages associated with application heaps. Existing schemes are ineffective in deduplicating these pages due to the limited occurrence of the same anonymous pages. This paper presents Sparrow, a similar-page aware deduplication solution for mobile systems. Sparrow shows that memory pages still have the potential to deduplicate, even though the same pages are rare. An interesting observation inspires this, that is, a high number of pages having the partially-same contents. We have implemented Sparrow on real-life smartphones. Experimental results indicate that 30.45% more space can be saved with Sparrow.**

***Index Terms*—Memory deduplication, Mobile system, Android**

I. Introduction

Mobile devices (e.g., Smartphones, IoT, and Automotive systems) play an essential role in people's daily lives. As the most widely employed system in these devices, Android always suffers from memory exhaustion. Because mobile devices have limited space to store storage chips and Android encourages applications to be cached in memory to speed up application recovery [1]. The user experience is seriously affected when memory is exhausted [2].

There are many efforts on memory saving [3]–[6]. For example, the Linux kernel supports memory deduplication by KSM (Kernel Samepage Merging) [3]. The same pages can be identified and merged based on a red-black tree structure. The performance is further improved by MDedup++ [6]. Based on existing solutions, redundant pages are identified and effectively deduplicated. However, the memory space of mobile devices is still scarce. This paper will show that anonymous pages, which are generated by the process at runtime, gradually become the primary source of memory consumption during Android usage. As evaluated, the proportion of anonymous pages reaches up to 86% in daily usage. Unlike file-backed pages which are managed by the file system, anonymous pages are generated at runtime and few of them are the same.

This paper observes that *a lot of anonymous pages have similar content, even though they are not the same pages*. It suggests that mobile system still has the potential to further reduce memory waste. Different from traditional OS, Android runs all applications in a common language runtime (ART). This difference leads to a phenomenon that the spatial locality of mobile data is reflected in Java objects, rather than the pages (size of 4KB for each). As investigated, almost half of the objects are smaller than one page [1]. Improper coordination between the OS kernel and the language runtime induces a lot of deduplicated objects. Inspired by the observation, this paper proposes Sparrow, a Similar-page-aware memory deduplication strategy in Android. The basic idea is to identify the same part between two similar pages and duplicate them.

There are three challenges in the design. First, data in the memory should be identified and managed in a finer-, instead of in page-, granularity. Second, Sparrow tries to deduplicate the same contents in pages, such a process should be imperceptible to applications. Finally, the design should be lightweight so that the overhead is acceptable. This paper proposes two novel schemes to tackle the above challenges: fine-grained deduplication (***FG-Dedup***) and application-aware page sifting (**APS**). By sampling the blocks in pages, page similarity is identified in an effective and efficient approach. In addition, this paper will show that more than 92.3% similar pages are detected in the same application. APS sifts candidate pages with application awareness to further reduce the performance overhead. Sparrow is application-agnostic without requiring any change to application codes or the language runtime. We have implemented Sparrow on Android devices. Experimental results indicate that 30.2% space can be saved with Sparrow, which is 2.65x more than the state-of-the-art [3] [6].

The main contributions are summarised as follows:

- This paper observes that anonymous pages gradually become the primary source of memory consumption during Android usage and shows the reason. Furthermore, based on the page content analysis, this paper demonstrates that anonymous pages in modern Android systems still have the potential to be further deduplicated.
- This paper proposes Sparrow, a lightweight similar-page aware deduplication solution in mobile systems. To the best of our knowledge, this is the first work that considers page similarity.
- We have implemented Sparrow on real-life smartphones. Experimental results indicate that 30.2% space can be

Changlong Li* is the corresponding author. This work is supported by NSFC 62302169, OPPO Foundation (No. 2022310031000638) and Special Fund for Graduate International Conference of East China Normal University.

979-8-3503-4859-0/DATE24/© 2024 EDAA

saved with Sparrow. More importantly, Sparrow-induced performance cost is user-imperceptible based on the proposed FG-Dedup and APS schemes.

II. Motivation

A. Memory Consumption in Mobile Systems

In modern mobile systems, file-backed and anonymous pages are reclaimed separately to save memory space. Specifically, file-backed pages are written back or released, while anonymous pages are compressed by ZRAM [7]. However, the memory pressure is still high. Mobile users do not have the habit of positively closing applications, they rather than switch them to the background straightforwardly. One result is that the available memory is always less than 1GB in daily usage.

To understand the main sources of memory consumption, we conducted evaluations on smartphones from different vendors, including Google Pixel3, OPPO Reno7 Pro, and Redmi K50. Eight popular applications are installed on each device, including short-form video, mobile games, e-commerce, and social network applications. The details of the platforms and workloads are presented in Section IV. To simulate daily usage, we launched the applications one by one. Each application is switched to the foreground and runs for one minute. Then we collect the memory state. To avoid bias, we repeat the evaluation for ten rounds and take the average. Before each round, we reboot the device and drop the cache.

In Fig.1, the X-axis represents the timeline of each device during usage, and the Y-axis represents memory size. As the line chart shows, the percentage of anonymous pages is close to 50%. That is, the ratio of anonymous and file-backed pages is similar at the beginning. But more and more memory consumption is caused by anonymous pages during the following usage. Specifically, the percentage of anonymous pages increased by 27% on average. The line fluctuates in some ranges because LMK(Low Memory Killer [8]) is woken up for quick memory reclaim, but the increasing trend of the curves is observed in all of the devices.

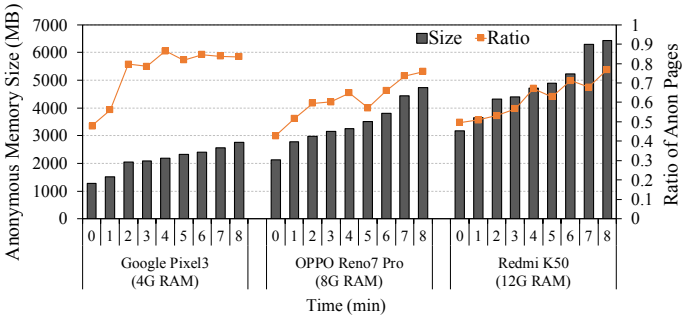


Fig. 1. Memory consumption of anonymous pages with the phone usage.

This phenomenon is caused by three reasons. First, the *kswapd* in the kernel space prefers to reclaim file-backed pages rather than anonymous pages, as most of the file-backed pages are not dirty and can be released directly. Second, a large number of anonymous pages are allocated in the application usage phase, instead of the launching phase. Third, the compressed pages still occupy memory space. As a result, anonymous pages gradually accumulate in the memory and become the main source of space consumption.

B. High Similarity between Anonymous Pages

Anonymous pages are not effectively deduplicated since few of them are the same. As evaluated, the same anonymous pages are no more than 6.8% on average. To explore the characteristics, pages are collected during the above evaluation and analyzed offline. We obtain page content by writing the data to a laptop through the Android debug bridge (ADB). Each collected page is stored as a file on the laptop. For each file, we calculate its hash value based on the TLSH (Tree Locality-Sensitive Hash) algorithm [9]. Specifically, the hash value (fingerprint) of each page is compared based on the TLSH algorithm and the similarity (range from 0% to 100%) is abstracted as a distance score.

Sixteen thousand pages are collected in total. For each page *X*, we scan the other pages and calculate all their similarity with *X*. Two pages having 100%-similarity means they are the same. We compare the similarity from [0%, 10%] to [90%, 100%] ranges. Fig. 2 shows the number of page pairs located in each similarity range. The X-axis represents the similarity and the Y-axis represents the number of page pairs in each range. We categorize these pages by application (around 2,000 pages for each).

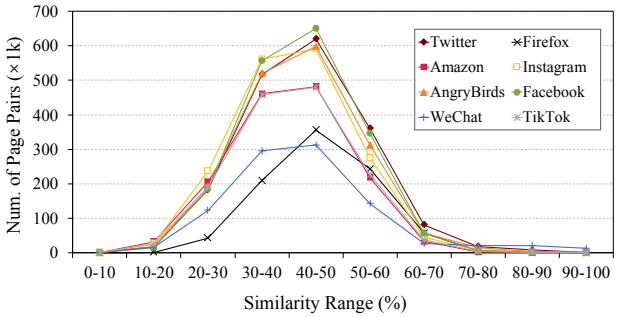


Fig. 2. Page similarity analysis

The page pair similarity has a Normal-like distribution. As counted, the same pages are less than 7%, while pages with huge differences (similarity from 0% to 20%) are few either. 86.1% of the tested pages have similar contents (similarity from 30% to 70%) on average. Taking Facebook as an example, around 650K page pairs's similarity lies in the range [40%, 50%]. This observation demonstrates a new opportunity to duplicate the memory in mobile systems.

III. Design

A. Sparrow Overview

Inspired by the observations, this paper proposes Sparrow, a flexible memory deduplication strategy in Android systems with similar-page awareness. As shown in Fig. 3, it consists of two critical components: application-aware page sifting (APS) and fine-grained deduplication (FG-Dedup).

The workflow is as follows: (1) Sparrow monitors the application state and identifies the inactive[1] ones. When an application is switched to the background and considered *inactive*,

[1]We identify the application activity based on the *adj* value [2] in Android.

Sparrow wakes up the ***Dedup-S*** daemon. (2) APS identifies the anonymous pages that belong to this application and calculates their fingerprint. Only anonymous pages with high similarity sifted through fingerprints will be further processed. (3) FG-Dedup splits the sifted pages into small blocks and performs block deduplication. (4) When a deduplicated application is used again, Sparrow restores the split pages immediately.

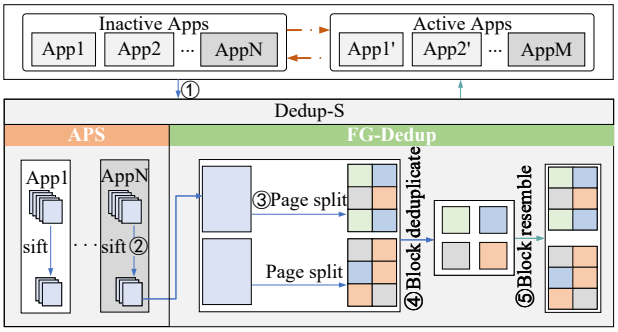


Fig. 3. Sparrow Overview

Based on the proposed APS and FG-Dedup, duplicate data can be identified with minimized page comparison scope. Now we introduce the two schemes in detail.

B. App-aware Page Sifting

1) Intra-App Deduplication: The computing overhead is high if performing similarity comparing indiscriminately on all pages. This paper finds that most of the highly similar pages appear within the same application.

We analyzed the similarity of anonymous pages across different applications to learn their characteristic. Fig. 4 illustrates that the size of duplicate data does not significantly increase when the pages of two applications are processed together, compared to one by one. It indicates that most similar pages are located in the same application. Therefore, it is proper to perform deduplication on pages belonging to the same application, instead of performing global hashing and similarity calculating within the entire memory scope.

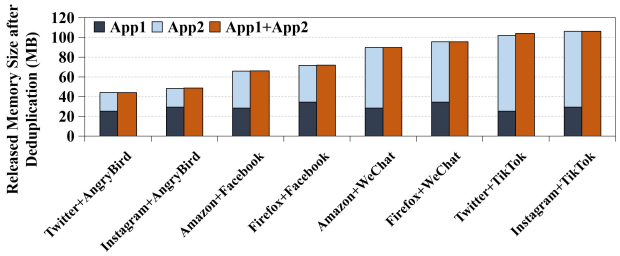


Fig. 4. Intra (App1, App2) and Inter (App1+App2) application deduplication.

In the design, only background applications that have not been used for a long time are allowed to perform deduplication. To simplify our design, we use LRU(Least Recently Used) lists [6] in the Linux kernel to identify page activity. Active pages may still exist even if the corresponding application is inactive. For each application, only pages append on the inactive LRU list are delivered to the sifter for further processing.

2) Page Sifting: In the memory, some pages only get a small space saving after investing CPU resources in splitting and deduplicating. These pages are called sparse pages [3]. To reduce CPU and memory overhead, we use page fingerprints to sift out non-sparse pages in advance. In the design, we adopt the fingerprint approach instead of fuzzy hashing, because the former consumes less computing resources and is more efficient (4.67x faster [9]).

Sparrow splits each page into multiple blocks. The block size is configured as parameter 2^{order}. Each page is split into $\frac{sizeof(page)}{2^{order}}$ blocks. Then, Sparrow samples these blocks, calculates their hash values, and obtains fingerprints. Suppose N=$\{S_1, S_2, ..., S_n\}$ samples are selected. The fingerprint of the page can be obtained by computing the hash of these N blocks using MD5, a lightweight hashing algorithm.

According to the spatial locality, if two pages have two same memory blocks, there is a high possibility that other same blocks exist. Since the offset of the same blocks on two pages may be different, we say two pages are similar as long as one fingerprint in the sample is the same. This approach results in missing a small number of the same memory blocks (4.83% as evaluated) but significantly reduces the computing overhead. To quickly compare the fingerprint (hash value), we adopt the hash table [10] as the data structure instead of comparing them in a loop. In this way, the time complexity is reduced to $O(n)$. The hashes are only used to detect similar pages, so the hash table can be released after sifting.

C. Fine-grained Deduplication

Different from traditional solutions, Sparrow tries to remove the repetitive part of two pages. The challenges are summarised as three folds: how to organize the data after deduplication; how to find the same data blocks; and how to quickly restore the split pages. To address these issues, three data structures are maintained.

- **Block Content Table (BC-Table)** is used to store the data of memory blocks. It consists of two segments: *Tag* and *Union*. *Tag* is a bitmap that identifies whether the content corresponding to the bit is original memory block data or metadata information (an integer indicating the address of the duplicate block). *Union* is a continuous virtual memory space, used to store block data or metadata.
- **Block Hash Table (BH-Table)** is responsible for storing the hash values of memory blocks and their addresses in the *Union* segment of BC-Table during deduplication. After completed, the BH-Table space can be released.
- **Page Address Table (PA-Table)** is responsible for locating the address of the first block of a page in the *Union* segment of BC-Table.

1) Block-Level Deduplication: For each block, the BH-Table is first searched to check whether the same hash values exist. If not, the block is written to the *Union* segment of BC-Table (abbreviated BC-Table_Union), and the BH-Table is updated. Otherwise, the block data corresponding to the existing hash value in the BH-Table is compared with the candidate block data byte-by-byte to prevent hash conflicts. If the data of the two blocks were not the same, candidate block data is directly written into the BC-Table_Union, and the BH-Table is not updated. If two same memory blocks are found, the address of

the existing duplicate block is written to the BC-Table_Union, and the corresponding BC-Table_Tag is set to 1, indicating that metadata is stored instead of the original block content.

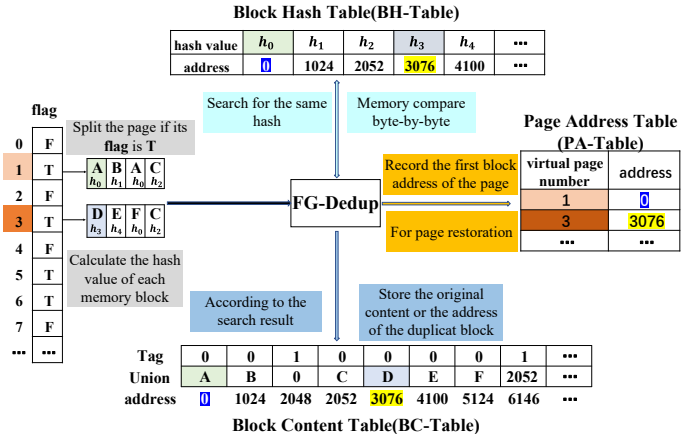


Fig. 5. Fine-grained deduplication (FG-Dedup). In this example, the redundancy identification granularity is set to 1KB.

Taking Fig. 5 as an example, when a block with content A is written for a second time, the block with address 0 in the BC-Table_Union is compared byte-by-byte with the candidate block because the same hash value exists in the BH-Table. Since the contents are the same, address 0 is written to the BC-Table_Union and the corresponding Tag is set to 1. For the block with content F, although the same hash value exists in the BH-Table, the contents are not the same, so the original content of the block is written to the BC-Table and no changes are made to the BH-Table. In this way, the comparison overhead caused by hash conflicts is effectively reduced.

2) Page Address Management: In the Linux kernel, the virtual address is first translated to the physical address based on the page table when accessing the data. Block deduplication conflicts with this paging mechanism. To solve this problem, Sparrow reassembles the split pages and restores them when demanded. If the candidate block is the first block of a page, obtain the page number of that page and the address of that block in the BC-Table_Union, then insert them into the PA-Table for page restoration in the future. Taking Fig. 5 as an example, the first blocks of pages 1 and 3 are addressed in the BC-Table as 0 and 3076, respectively. Therefore, Sparrow records (1, 0) and (3, 3076) in the PA-Table.

D. Page Restoration

1) Page restoring when application stays in the background: If the system needs to access a split page of an application that has already been deduplicated, the page can be restored according to the PA-Table and BC-Table. After restoring, the corresponding table entry in the PA-Table is eliminated, but no changes will be made to the BC-Table as the corresponding block may still be used by other pages. In addition, these restored pages are no longer split and deduplicated. Since applications in the background use only a small amount of CPU, resulting in few modified pages.

2) Page restoring when the application is switched to the foreground: If a deduplicated application is switched to the foreground, both fast restore and slow restore operations are performed on the application's memory pages. Fast restore means that if the page to be accessed has been split, the page is restored according to the PA-Table and BC-Table. Slow restore means that the restore operation is performed sequentially on the pages that have not been accessed nor restored, according to the order in the PA-Table. This slow restore operation can be performed when computing resources are not stretched to ensure the user experience.

IV. Evaluation

A. Experiment Setup

Sparrow is implemented on the Google Pixel3 smartphone, which is equipped with Qualcomm Snapdragon 845×8 cores, 4GB DRAM, and 64GB UFS storage memory. Android 12 and the Linux kernel version 4.9.270 are deployed on the device. Eight popular applications are preinstalled in the system, including Twitter, Firefox, Amazon, Instagram, AngryBirds, Facebook, WeChat, and TikTok. We compare Sparrow to KSM [3], a typical page-grained deduplication scheme. The redundancy recognition granularity is set to 64B ($order = 6$), and the number of samples per page N is set to 8 by default.

B. System Improvement

Memory space saving. To simulate the daily usage, we launch the preinstalled applications one by one and run them in the foreground. During the above operations, more than 91% of the memory is full-filled. To exactly understand how much space can be saved by Sparrow, we collect the anonymous pages as workloads. 216,825 pages are collected in total.

Sparrow and KSM run on the workloads to make a comparison. By subtracting the metadata size retained in memory from the size of the redundant data deduplicated, we know how much space is saved. Fig. 6 shows that 2.65x more space can be saved by Sparrow on average, compared to KSM. For Twitter, the improvement is even up to 4.28×. In addition, the total memory consumption is reduced by 30.45% on average when enabling Sparrow. It demonstrates that Sparrow can significantly alleviate memory waste of background applications, as more redundant objects are identified and removed.

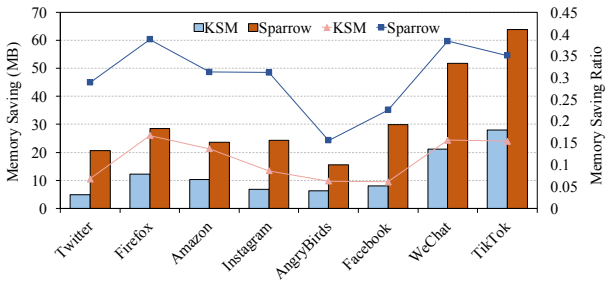


Fig. 6. Memory saving size. Histograms represent the size of memory saving and lines represent the ratio.

According to the statistics, there are 25,821 same pages. Table I shows the number of pages with different similarity

intervals and the space saved in each interval. It illustrates that Sparrow outperforms KSM since 61.19% of the similar, instead of the same pages[2] are deduplicated. What is more, it shows that more than 49.28% of the similar pages can be identified using the sampling approach.

TABLE I
SOURCES OF MEMORY SAVING WITH SPARROW AND KSM

Similarity	KSM		Sparrow	
	Deduplicated Pages	Memory Saving (MB)	Deduplicated Pages	Memory Saving (MB)
100	25821	97.49	25821	97.49
[90,100)	0	0	16097	24.61
[80,90)	0	0	14607	18.92
[70,80)	0	0	18215	25.42
[60,70)	0	0	18887	21.55
[50,60)	0	0	19035	29.26
[40,50)	0	0	13095	25.67
[30,40)	0	0	5425	13.09
[20,30)	0	0	1467	1.89
[10,20)	0	0	20	0.03
[0,10)	0	0	0	0
Total	**25821**	**97.49**	**132669**	**257.93**

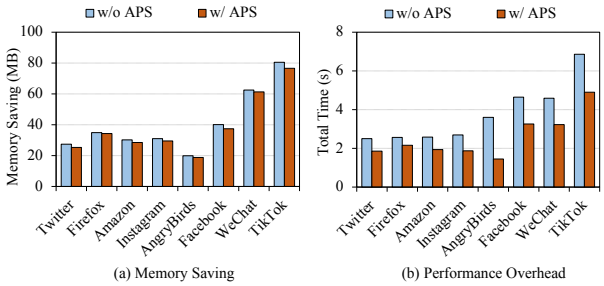
We further explored the effect of different redundancy identification granularities on memory saving. As shown in Table II, more space is saved when the block granularity is 64B, that is, parameter *order* equals 6. A too-small block will incur excessive metadata overhead. Therefore, we split one page into blocks size of 64B by default. Note that the most reasonable parameter may be different on various devices, so the parameters are maintained as a configurable `xml` file.

TABLE II
MEMORY SAVING AT DIFFERENT BLOCK GRANULARITIES

Granularity(B)	1024	512	256	128	**64**	32	16
Twitter	10.77	13.22	15.61	18.26	**20.68**	20.78	15.1
Firefox	20.37	23.39	25.95	27.82	**28.49**	26.67	18.04
Amazon	14.49	16.24	18.72	21.1	**23.52**	23.24	16.37
Instagram	14.2	16.77	19.16	22.11	**24.34**	23.57	16.34
AngryBirds	8.85	9.97	11.3	13.41	**15.47**	15.13	8.19
Facebook	16.21	19.53	22.37	25.81	**29.84**	29.95	20.77
WeChat	38.84	44.65	47.92	49.92	**51.77**	48.51	33.63
TikTok	38.94	46.43	53.46	59.46	**63.82**	60.82	42.47

Effect of page sifting. The effect of APS on the system is more complicated, since it may harm the deduplication. We further evaluate this proposed sifting strategy by comparing the deduplication size of the application with APS enabled and disabled. Fig. 7(a) shows that the size of memory saving decreased by 4.83% when enabling APS. This is because the sampling misses some pages that have a small number of the same blocks. On the other hand, the time cost of the above two cases is evaluated. Since similar pages can be sifted out, a lot of unnecessary page splits and comparisons are avoided. Hence, we can see the average time cost reduced by 30.6% on average, which is shown in Fig. 7(b). In summary, APS can significantly reduce the time cost and computing overhead by sacrificing a small amount of memory saving.

[2]The ratio of same pages is higher than 10%. This is because many unsimilar pages have already been sifted. If all memory pages are taken into account, this ratio is much lower.



(a) Memory Saving (b) Performance Overhead

Fig. 7. Memory saving and performance overhead of Sparrow with (w/) and without (w/o) APS.

C. Benefit on User Experience

To meet the memory demand, Android wakes up LMK when the watermark is lower than the min-threshold. By killing some applications, the memory space can be released quickly. However, application killing turns the launching style from *hot* to *cold*. It degrades the user experience as the launching time is prolonged and the interaction state is lost [2].

We evaluate the number of applications alive in the background under different schemes. First, we launched 16 applications for three rounds. In addition to the aforementioned applications, YouTube, Linkin, PUBG Mobile, eBay, Chrome, Camera, Google Earth, and Uber are used as workloads. Fig. 8 shows the number of hot-launched applications.

Fig. 8 shows the app caching capability under different schemes. Baseline refers to the system without any deduplication mechanisms. We evaluated ten rounds and took the average. The order of application startup in each round is randomly arranged. Sparrow (with the coordination of APS and FG-Dedup) improves the app caching capability by 30.51% on average, compared to the original system. Here, KSM also has a positive effect because it removes the same pages.

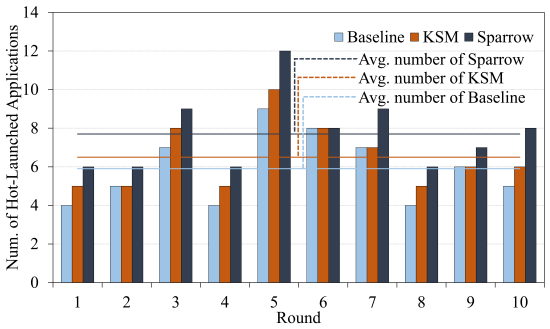


Fig. 8. Number of hot-launched applications in different rounds.

D. Overhead Analysis

Memory consumption. Sparrow maintains mapping tables, like PA-Table and BC-Table, incurring a space overhead in the memory. They are necessary to store the deduplicated data and perform page restoration operations. We made efforts to

reduce memory consumption. First, pages are effectively sifted before further processing, so many pages do not need to be recorded in the tables. Second, the structure of the mapping tables is designed with a simple approach. For each page, it consumes 8 Bytes in PA-Table for the virtual page number and first block address. For each removed block, 4 Bytes in BC-Table to record the address of the duplicate block. Such consumption is negligible compared to the total memory saved on the device.

Performance and energy overhead. The proposed Sparrow is designed in a lightweight approach to reduce the performance overhead. Based on our observation, most similar pages are located in the same application. Sparrow just needs to compare pages belonging to the same application, which significantly decreases the calculation scope. Furthermore, the comparison method is optimized. Sparrow sampling the pages rather than calculating the similarity with the traditional TLSH algorithm. This is because we find small-sized fingerprint calculating (μs-level) is 4.67× quicker than the TLSH's tree-based hashing and tree-distance calculating. Moreover, the restoration of deduplicated pages has little impact on the application access. As evaluated, each page restoration takes only 13μs on average.

When implementing Sparrow in the system, the *Dedup-S* daemon runs in the background. As evaluated, it increases power consumption by an average of 0.5W during running. In addition, this daemon is not always run. Experimental results show that it takes only several seconds (spending only 177μs per page on average) to compare and deduplicate all pages of an application. Most time, this daemon is hibernated. In summary, in comparison with the persistent energy consumption of touchscreen [11], *Dedup-S* induced consumption is negligible.

V. Related Work

Many efforts have been made to save space through deduplication [8], [12]. For example, UKSM [3] grades memory regions and allocates different amounts of CPU resources to different grades of regions for page merging. The hierarchy allows the deduplication process to improve responsiveness to newly generated memory pages, speeding up the deduplication process. Loc-K [5] scans pages with contiguous logical addresses and predicts the location of K potential same pages at a time as page scan targets, which improves deduplication efficiency. Medes [13] treats memory blocks in certain specific hot sandboxes in serverless computing as reusable sandbox chunks (RSCs). Based on the existing RSCs, all redundant blocks in the remaining sandboxes are removed. MDedup++ [6] identifies unstable pages in the system and hints to the memory deduplication scanners, which alleviates the deduplication-induced memory thrashing. Based on that, BCD Deduplication [10] proposes a combination of deduplication and compression. BCD uses partial data redundancy between blocks to increase the capacity. Special hardware is required to enable BCD Deduplication. In addition, there are also many deduplication schemes in external storage systems [14]–[16].

These solutions inspire the design of Sparrow. However, they cannot be ported to the mobile system straightforwardly. Because the data characteristics and access patterns of mobile applications are different from the traditional computer system. Sparrow is designed to identify the same parts in pages instead of just deduplicating the same pages.

VI. Conclusion

This paper presents Sparrow, a flexible memory deduplication scheme in Android with similar-page awareness. This paper first shows that the memory of modern Android systems still has the potential to be further saved. Furthermore, this paper proposes two novel schemes to identify similar pages and deduplicate the same parts flexibly and efficiently. Experimental results show that the memory consumption of mobile applications can be effectively reduced with Sparrow. Specifically, the memory consumption was reduced by 30.45% on average.

References

[1] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the senseless killing: Improving memory management for mobile operating systems," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020, pp. 873–887.

[2] C. Li, Y. Liang, R. Ausavarungnirun, Z. Zhu, L. Shi, and C. J. Xue, "Ice: Collaborating memory and process management for user experience on resource-limited mobile devices," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 79–93.

[3] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang, "Uksm: Swift memory deduplication via hierarchical and adaptive memory region distilling," in *Conference on File and Storage Technologies (FAST)*, 2018, pp. 325–340.

[4] M. Zhu, K. Zhang, and B. Tu, "Pca: Page correlation aggregation for memory deduplication in virtualized environments," in *Information and Communications Security (ICICS)*. Springer, 2018, pp. 566–583.

[5] S. Jia, C. Wu, and J. Li, "Loc-k: A spatial locality-based memory deduplication scheme with prediction on k-step locations," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 310–317.

[6] T. Veni and S. M. S. Bhanu, "Mdedup++: Exploiting temporal and spatial page-sharing behaviors for memory deduplication enhancement," *The Computer Journal*, vol. 59, no. 3, pp. 353–370, 2016.

[7] B. Lee, S. M. Kim, E. Park, and D. Han, "Memscope: Analyzing memory duplication on android systems," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015, pp. 1–7.

[8] S.-h. Kim, J. Jeong, and J. Lee, "Selective memory deduplication for cost efficiency in mobile smart devices," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 2, pp. 276–284, 2014.

[9] A. Lee and T. Atkison, "A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions," in *Proceedings of the SouthEast Conference*, 2017, pp. 18–25.

[10] S. Park, I. Kang, Y. Moon, J. H. Ahn, and G. E. Suh, "Bcd deduplication: Effective memory compression using partial cache-line deduplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 52–64.

[11] A. Carroll, "Understanding and reducing smartphone energy consumption," Ph.D. dissertation, UNSW Sydney, 2017.

[12] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "Xlh: More effective memory deduplication scanners through cross-layer hints," in *USENIX Annual Technical Conference (ATC)*, 2013, pp. 279–290.

[13] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella, "Memory deduplication for serverless computing with medes," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 714–729.

[14] S. Wu, C. Du, W. Zhu, J. Zhou, H. Jiang, B. Mao, and L. Zeng, "Ead: Ecc-assisted deduplication with high performance and low memory overhead for ultra-low latency flash storage," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 208–221, 2022.

[15] B. Mao, J. Zhou, S. Wu, H. Jiang, X. Chen, and W. Yang, "Improving flash memory performance and reliability for smartphones with i/o deduplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1017–1027, 2018.

[16] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Write deduplication and hash mode encryption for secure nonvolatile main memory," *IEEE Micro*, vol. 39, no. 1, pp. 44–51, 2018.