Ariadne: A Hotness-Aware and Size-Adaptive Compressed Swap Technique for Fast Application Relaunch and Reduced CPU Usage on Mobile Devices

Yu Liang[§] Aofeng Shen[§] Chun Jason Xue[‡] Riwei Pan[†] Haiyu Mao[¶] Nika Mansouri Ghiasi[§]
Qingcai Jiang[𝒮] Rakesh Nadig[§] Lei Li[†] Rachata Ausavarungnirun[*⋆] Mohammad Sadrosadati[§] Onur Mutlu[§]

[§]ETH Zürich   [‡]MBZUAI   [†]City University of Hong Kong   [¶]King's College London
[𝒮]University of Science and Technology of China   [⋆]MangoBoost

As the memory demands of individual mobile applications continue to grow and the number of concurrently running applications increases, available memory on mobile devices is becoming increasingly scarce. When memory pressure is high, current mobile systems use a RAM-based compressed swap scheme (called ZRAM) to compress unused execution-related data (called anonymous data in Linux) in main memory. This approach avoids swapping data to secondary storage (NAND flash memory) or terminating applications, thereby achieving shorter application relaunch latency.

In this paper, we observe that the state-of-the-art ZRAM scheme prolongs relaunch latency and wastes CPU time because it does not differentiate between hot and cold data or leverage different compression chunk sizes and data locality. We make three new observations. First, anonymous data has different levels of hotness. Hot data, used during application relaunch, is usually similar between consecutive relaunches. Second, when compressing the same amount of anonymous data, small-size compression is very fast, while large-size compression achieves a better compression ratio. Third, there is locality in data access during application relaunch.

Based on these observations, we propose a hotness-aware and size-adaptive compressed swap scheme, Ariadne, for mobile devices to mitigate relaunch latency and reduce CPU usage. Ariadne incorporates three key techniques. First, a low-overhead hotness-aware data organization scheme aims to quickly identify the hotness of anonymous data without significant overhead. Second, a size-adaptive compression scheme uses different compression chunk sizes based on the data's hotness level to ensure fast decompression of hot and warm data. Third, a proactive decompression scheme predicts the next set of data to be used and decompresses it in advance, reducing the impact of data swapping back into main memory during application relaunch.

We implement and evaluate Ariadne on a commercial smartphone, Google Pixel 7 with the latest Android 14. Our experimental evaluation results show that, on average, Ariadne reduces application relaunch latency by 50% and decreases the CPU usage of compression and decompression procedures by 15% compared to the state-of-the-art compressed swap scheme for mobile devices.

1. Introduction

Mobile devices are integral to our daily lives, with users frequently relaunching and running various applications to meet their diverse needs [1–3]. To fulfill user expectations of seamless and rapid application relaunch, mobile systems preserve all execution-related data (called *anonymous data* in Linux [4]), such as stack and heap, in main memory. This practice, known as *keeping applications alive in the background* [1, 5–8], enables faster relaunches. However, it also results in significant main memory capacity requirements for each application.

As the demand for memory capacity in mobile applications grows and the number of applications running simultaneously increases, available memory is becoming an increasingly scarce resource on mobile devices [1, 7–9]. When memory capacity pressure is high, current mobile systems use a *RAM-based compressed swap scheme* (called *ZRAM* [10, 11]) to compress unused anonymous data into a specific memory region, called *zpool* [12], rather than directly swapping the data into secondary storage (i.e., NAND flash memory). This approach achieves shorter application relaunch latency because decompression is much faster than swapping data from secondary storage into main memory and relaunching terminated applications.

We observe that the state-of-the-art *ZRAM* scheme still prolongs relaunch latency and wastes CPU time due to two major reasons. First, it does not differentiate between hot and cold data, resulting in frequent and unnecessary compression and decompression. Systems might compress hot data when an application is in the background and decompress it when brought back to the foreground, even though enough memory space is available. The unnecessary compression and decompression not only prolongs application relaunch times but also wastes CPU resources. Second, it does not take advantage of different compression chunk sizes and data locality, leading to long compression and decompression times during application relaunch. As users typically switch between applications more than 100 times daily [3], these frequent long-latency relaunches can negatively impact the overall user experience on mobile devices [13–25].

The **goal** of this work is to minimize application relaunch latency and reduce wasted CPU usage while maximizing the number of live background applications for an enhanced user experience. To achieve this goal, we characterize the anonymous data of real mobile applications used on a real modern mobile phone (i.e., Google Pixel 7 [26]). Our experimental characterization yields three new observations. First, we classify anonymous data into three categories: i) Hot data, used during relaunch and impacting relaunch latency; ii) Warm data, potentially used during application execution after relaunch;

---

and iii) Cold data, usually not used again. We observe that hot data is usually similar between consecutive relaunches. Second, when compressing the same amount of anonymous data, small-size compression, which involves compressing data in smaller chunks, is very fast, while large-size compression achieves a better compression ratio. Third, there is locality in data access in *zpool* when swapping in anonymous data during application relaunch, meaning the data tends to be stored in contiguous or nearby memory locations in *zpool*. Thus, we can predict the next set of data to be used at the beginning of a relaunch.

Based on these observations, we propose a new hotness-aware and size-adaptive compressed swap scheme for mobile devices, called Ariadne, that incorporates **three key techniques**: First, a low-overhead hotness-aware data organization scheme aims to separate hot and cold data. Ariadne tries to maintain hot data and compressed warm data in main memory while swapping compressed cold data to secondary storage. Second, a size-adaptive compression scheme takes advantage of different compression chunk sizes. It uses small-size compression for identified hot and warm data to ensure fast relaunch and execution, while using large-size compression for cold data to achieve a high compression ratio. Third, a proactive decompression scheme predicts the next set of data to be used and performs decompression in advance for such data, further mitigating the negative impact of data swapping back into main memory and decompression latency on application relaunch.

We implement and evaluate Ariadne on a commercial smartphone, Google Pixel 7 [26] with the latest Android 14 operating system [27]. We test Ariadne with over 30 combinations of commonly-used concurrently-running mobile applications. Our experimental evaluation results show that, on average, Ariadne reduces application relaunch latency by 50% and decreases the CPU usage of compression and decompression procedures by 15% compared to the state-of-the-art compressed swap scheme for mobile devices.

This work makes the following key contributions:

- We are the first to quantitatively demonstrate the inefficiency of the state-of-the-art compressed swap scheme in mobile systems, highlighting its long application relaunch latency and high CPU usage, and identifying the root causes of these problems.
- We make three new observations from real mobile applications. First, data used during application relaunch is usually similar between consecutive relaunches. Second, when compressing the same amount of anonymous data, small-size compression is very fast, while large-size compression achieves a better compression ratio. Third, there is locality in data access when swapping in anonymous data during application relaunch.
- We propose a new hotness-aware and size-adaptive compressed swap scheme, Ariadne, for mobile devices. This scheme incorporates three key techniques: low-overhead hotness identification, size-adaptive compression, and proactive and predictive decompression.
- We evaluate Ariadne on a real smartphone with a cutting-edge

Android operating system. Our evaluation results show that our solution surpasses the state-of-the-art in terms of both application relaunch latency and CPU usage. To foster further research in the design and optimization of mobile compressed swap techniques, we open-source our implementations at https://github.com/CMU-SAFARI/Ariadne.

2. Background and Motivation

Mobile devices have unique features such as fewer foreground applications, smaller DRAM and flash memory capacity, and constrained power budget compared to general-purpose servers. However, mobile systems, especially Android [28], are built on the Linux kernel [29], originally designed for servers. As a result, many existing schemes in Android systems [1, 7, 9, 30] do not align well with mobile workloads, leading to suboptimal performance, reduced device lifespan [31–34], and increased energy consumption [35]. This section highlights the detrimental effects of the inefficient *ZRAM* scheme [10, 11] on Android systems, especially focusing on application relaunch latency and energy efficiency.

2.1. Application Launching and Execution

We first briefly describe the mobile application launching procedure and then explain how to keep an application alive in mobile systems.

Mobile application launching. Application launch latency is one of the critical metrics used to evaluate the user experience on mobile devices [7, 13–16]. It directly reflects the system's responsiveness and smoothness, as faster launch times contribute to a more immediate and seamless user experience. There are two types of application launching: *cold launch* and *hot launch*. Cold launching an application involves two main steps: i) creating one or more processes for the application, and ii) reading the application's data into main memory. In contrast, a hot launch means launching an application from the background, so it does not require process creation as the application's processes are already running in the background. Previous work [32] shows that process creation accounts for 94% of the total cold launch latency. Many studies [1, 7, 8] show that hot launch is much faster than cold launch, leading to an improved user experience. Keeping applications alive in the background enables hot launches for relaunching applications, thereby reducing relaunch latency.

Keeping mobile applications alive. To determine how to keep an application alive, we first analyze its execution. Mobile application execution typically generates two types of memory pages: *file-backed pages* and *anonymous pages*. *File-backed pages* directly correspond to files stored in secondary storage (e.g., NAND flash memory). When the system encounters insufficient available main memory, it frees up (i.e., reclaims) memory pages to accommodate new requests. The system writes data from a reclaimed *file-backed page* back to secondary storage. In contrast, *anonymous pages* do *not* correspond to any specific file in secondary storage but contain data associated with process execution, such as stack and heap information (called *anonymous data*). When the system reclaims an *anonymous*

page, it deletes the *anonymous data*, leading to the termination of the corresponding process. Therefore, to keep an application alive, it is essential to keep its *anonymous data* in main memory. To assess the feasibility of keeping a large number of applications alive, we measured the *anonymous data* volumes of five commonly-used applications on a Google Pixel 7 (see experimental setup details in Section 5). We gathered data for each application at two time points: 10 seconds and 5 minutes after launching. The results are presented in Table 1, which leads to two main observations.

Table 1: Anonymous data volume (in MB) of five applications, where 'GEarth' refers to Google Earth.

Time	Youtube	Twitter	Firefox	GEarth	BangDream
10s	177	182	560	273	326
5mins	358	273	716	429	821

First, each application generates substantial anonymous data, reaching up to 821 MB. Second, the volume of anonymous data increases as the application continues to run. We conclude that each application generates a significant amount of anonymous data during its execution. Consequently, mobile systems require a substantial amount of main memory to keep all applications alive. However, due to cost and power constraints, main memory capacity in mobile devices is typically limited, ranging from 1 GB to 8 GB in low/mid-end smartphones [36]. The available memory on these smartphones for applications is usually limited and allows only a moderate number of applications to run concurrently in the background [1, 8].

2.2. Android Memory Swap Schemes

To keep more applications alive on mobile devices with limited DRAM capacity, *flash memory-based swap schemes* are employed to expand available memory space by relocating inactive *anonymous pages* to a specific region in flash memory storage (i.e., flash memory swap space) [32, 33, 37–43]. These *flash memory-based swap schemes* have two main issues. First, they increase the number of writes to flash memory storage, accelerating the wear-out of flash cells [44] and consequently reducing the overall lifespan of the mobile device [31, 32, 45]. Second, compared to reading relaunch data directly from main memory, swapping data from flash memory storage into main memory can increase application relaunch latency [8, 33], negatively impacting user experience.

Cutting-edge compressed swap schemes. To address the issues of *flash memory-based swap schemes*, cutting-edge mobile devices often use a RAM-based compressed swap scheme (called *ZRAM*) [10, 11, 42, 43, 46–49]. Under *ZRAM*, the system compresses unused anonymous data and stores it in a dedicated region of DRAM called *zpool*. When the system relaunches an application, it decompresses the corresponding data from *zpool* [12] back into main memory to facilitate the application's relaunch. Figure 1 illustrates the data movement for compression and decompression when utilizing *ZRAM*.

Multiple applications (i.e., A, B, C, D, E, and F) concurrently run and continuously generate anonymous pages. When available memory becomes insufficient, the system identifies
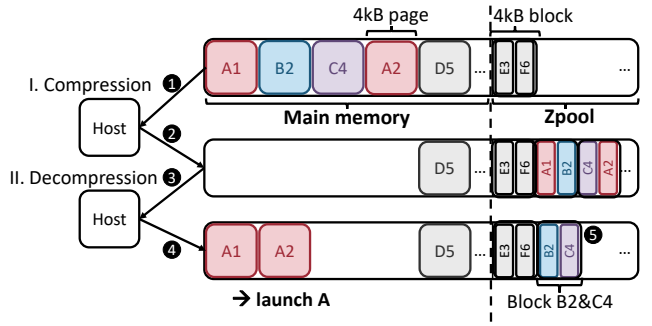

Figure 1: Data movement flow for compression and decompression when using *ZRAM* in Android systems. Ai (e.g., A1, A2) represents an uncompressed anonymous page of an application. Block B2&C4 refers to a compressed block that includes the compressed data of pages B2 and C4.

and moves a set of least-recently-used (LRU [50]) data pages (pages A1, B2, C4, and A2 in Figure 1, where different letters represent different applications) to the host CPU or accelerator for compression ❶. The system stores the compressed data in 4KB memory blocks [51]. Next, it writes the compressed data (blocks A1&B2 and C4&A2) back to *zpool* in DRAM ❷. When a user launches an application (e.g., A), the system reads the application A-related compressed blocks (blocks A1&B2 and C4&A2) from *zpool* to the host CPU ❸ and decompresses them. The system writes the decompressed pages A1 and A2 back to main memory to facilitate application A's relaunch ❹. Finally, the system merges the unused compressed data and writes block B2&C4 back to *zpool* ❺. As a result, the compression and decompression procedures due to *ZRAM* can incur high-cost data movement, as quantified in [35].

ZRAM with writeback (ZSWAP). With *ZRAM*, when *zpool* space is insufficient, the system deletes some inactive compressed data, potentially leading to application termination. *ZSWAP* extends *ZRAM* by using flash memory storage for additional swapping space. When *zpool* is full, the system writes some compressed data to the flash memory-based swap space. While *ZSWAP* increases the number of live background applications, it can also prolong application relaunch latency since some data needs to be swapped in from flash memory and decompressed during application relaunch. A major industrial vendor [52] reports that simply enabling *ZSWAP* could lead to a 6x increase in application relaunch latency. Due to such long relaunch latencies, multiple vendors (e.g., Google [26, 53] and Samsung [54]) may *not* enable *ZSWAP*. Therefore, the state-of-the-art compressed swap scheme is *ZRAM* shown in Figure 1.

2.3. Motivation

We describe the major issues with the state-of-the-art compressed swap scheme, *ZRAM*, used in modern mobile devices.

ZRAM's impact on performance. To quantitatively demonstrate the impact of *ZRAM* on the performance of commercial mobile devices, we evaluate the hot launch (i.e., relaunch) latency. We choose application relaunch latency for two reasons. First, *ZRAM* directly impacts application relaunch latency since

it needs to decompress the anonymous pages required for application relaunch. Second, relaunches occur frequently in users' daily lives (more than 100 times per day [3]), making it a crucial metric for evaluating performance [7]. Users typically perceive system responses as instantaneous if they occur within 100 ms [55]. Figure 2 shows the application relaunch latency of five commonly-used applications under three different swap schemes on Google Pixel 7: 1) *DRAM*, where the system reads all application data directly from DRAM (with the optimistic assumption that DRAM is large enough to host all such data), i.e., there is no swapping. 2) *ZRAM*, where i) some application data is read directly from DRAM as it is stored uncompressed; ii) most application data is read from DRAM after being decompressed as it is stored in compressed form in *zpool*. Decompression may also trigger on-demand compression operations when main memory is insufficient, as the system must first compress other data to free up space for the decompressed data in main memory. 3) *SWAP*, where the system reads data from the flash memory-based swap space into main memory during a relaunch. It does not involve compression or decompression.
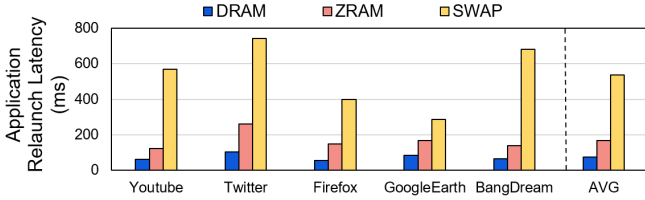

Figure 2: Application relaunch latency under different memory swap schemes.

Based on the results, we observe that *ZRAM* outperforms the flash memory-based *SWAP* scheme, but compression and decompression latencies still prolong application relaunch latency by an average of $2.1\times$ compared to reading data directly from DRAM without any compression or decompression.

Observation 1: *The state-of-the-art compressed swap scheme for mobile devices can lead to long application relaunch latencies due to long latencies of on-demand compression and decompression.*

To avoid these latencies, some vendors, such as Google [54], aggressively free up memory by proactively and periodically compressing data [26]. While this approach reduces the frequency of on-demand compression and decompression in *ZRAM*, as shown in Figure 2, it also increases CPU usage.

ZRAM's impact on CPU usage. To demonstrate the impact of *ZRAM* on CPU usage, we evaluate the CPU usage of the memory reclaim procedure under different memory swap schemes. We run the same application combinations as Figure 2 to trigger memory swapping for a total of 60 seconds under different swap schemes (details in Section 5) and use the system profiling tool, Perfetto [56], to collect the CPU usage of the memory reclamation thread. We test each swap scheme five times and calculate the average CPU time for each. Figure 3 shows the CPU usage of the reclamation thread (i.e., *kswapd* thread) across different swap schemes on Google Pixel 7: 1) DRAM, where there is no swap scheme for *anonymous data*, so the CPU usage includes CPU time used for writing *file-backed* pages back to

flash memory. 2) *ZRAM*, where the results also include the CPU usage for compressing *anonymous data*. However, the decompression procedure is not included because Perfetto can only track CPU usage for dedicated threads (e.g., reclaim thread), and decompression is not handled by such a thread. Therefore, ZRAM's CPU usage is actually higher than what we report here. 3) *SWAP*, where the CPU usage is collected while the system reclaims *anonymous data* by writing it to flash memory (Such usage can be low because when data is written to the storage device, CPU is usually yielded to other processes).
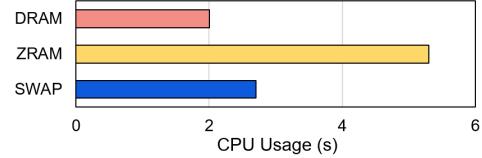

Figure 3: CPU usage of the memory reclamation procedure (i.e., *kswapd*) across different swap schemes.

Based on the results, we observe that *ZRAM* increases CPU usage of memory reclamation by an average of $2.6\times$ compared to *DRAM* and $2.0\times$ compared to *SWAP*. Although the CPU usage of memory reclamation accounts for only a small percentage of the total CPU usage in the test scenario, it could become severe under heavy workloads, making it critical to reduce CPU usage for mobile devices [57–60]. Notably, the memory footprint of applications and systems is expected to grow in the future, e.g., with the rise of emerging generative artificial intelligence (GenAI) models [61–63] and augmented reality (AR) games [64–67]. As a result, *ZRAM*'s CPU usage for compression and decompression is anticipated to increase, as memory capacity will remain constrained by cost and power limitations.

We also evaluate the total energy consumption of Google Pixel 7 under the above three swap schemes across two usage scenarios: light workloads (switching between ten applications (described in Section 5) with 1-second intermission time in between) and heavy workloads (launching ten applications sequentially without any intermission time). We collect energy consumption for 60 seconds using Power Rails [68]. The test is repeated five times, and we report the average results in Table 2. The results show that *ZRAM* increases energy consumption by 12.2% under light workloads and 19.5% under heavy workloads, compared to *DRAM*.

Table 2: Energy consumption under three swap schemes.

Workload		DRAM	ZRAM	SWAP
Light	Energy (J)	178.8	200.7	179.4
	Normalized	1.000	1.122	1.003
Heavy	Energy (J)	231.8	277.0	235.8
	Normalized	1.000	1.195	1.017

Observation 2: *The state-of-the-art compressed swap scheme for mobile devices consumes significant CPU time and energy.*

Analysis of data compressed by *ZRAM*. To determine the root causes of long relaunch latency and high CPU usage caused by *ZRAM*, we profile the swapped data. We analyze the swap process by sorting all compressed data in the order of compression
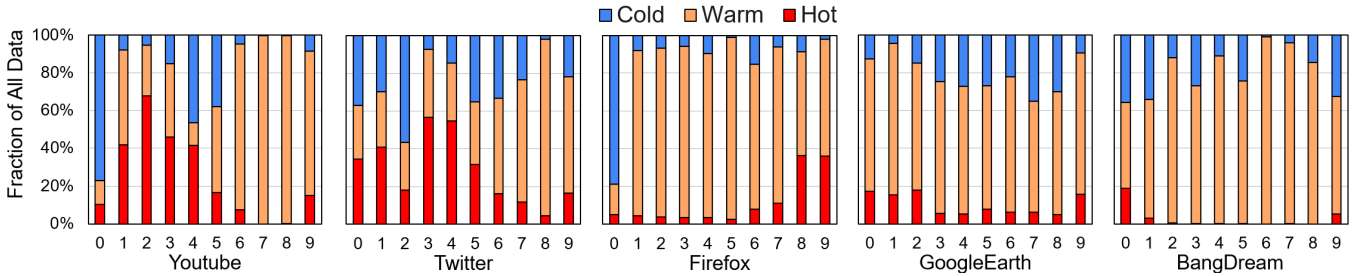
Figure 4: Proportion of hot, warm, and cold data in each part of compressed data. We sort all compressed data in the order of compression time and then divide it into ten equal parts (X-axis). The data in part 0 is the first to be compressed, that in part 8 is the last.

time and then dividing them into ten equal-sized parts. The data in part 0 is compressed first. To minimize swapping, cold data *should* be swapped out (i.e., compressed) earlier (e.g., in parts 0 and 1) and hot data later (e.g., in parts 8 and 9).

Figure 4 shows the proportion of hot, warm, and cold data in each part. The results indicate that mobile systems do *not* consider the hotness of the data when swapping/compressing data. For example, the first part (i.e., part 0) of the swapped data includes a significant amount of hot data for *all* applications. This is because the system still relies on the LRU scheme [50] for choosing which data to swap, even though the LRU scheme may *not* be useful for performance, especially when applications are switched often.

Observation 3: *Compressing data without distinguishing between hot and cold data is the primary cause of long relaunch latency and high CPU usage, as it leads to frequent compression and decompression.*

Summary. We empirically observe that the state-of-the-art ZRAM scheme prolongs application relaunch latency and consumes substantial CPU time because it does *not* consider hotness when compressing data, leading to unnecessary compression and decompression operations. There are numerous previous works [6–8, 69] that focus on optimizing flash memory-based swap schemes for mobile devices. However, most modern Android systems (e.g., [26, 53, 54]) adopt the ZRAM scheme [10, 11, 48] instead of flash memory-based swap schemes due to its better performance. No existing work specifically aims to reduce relaunch time and CPU usage by optimizing the compressed swap scheme on mobile devices.

2.4. Our Goal

Our goal in this work is to design a new compressed swap scheme for mobile devices that minimizes application relaunch latency and CPU usage while maximizing the number of live applications for enhanced user experience. Doing so requires reducing the frequency and latency of compression and decompression. We anticipate two **challenges** to achieving our goal. First, we would like to minimize compression and decompression frequency while maintaining efficient memory utilization. Second, we would like to reduce compression and decompression latency without negatively impacting the compression ratio.

To address these challenges, we profile mobile workloads with the goal of identifying new opportunities for designing a more efficient compressed swap scheme for mobile devices.

3. New Insights into Mobile Workloads

We profile the *anonymous data* of mobile workloads on the Google Pixel 7 (see Section 5 for an experimental setup and methods). Our profiling results reveal three major new insights.
Insight 1: *Hot data that is used during application relaunch is usually similar between consecutive relaunches.*

As discussed in Section 1, we categorize *anonymous data* into three levels of hotness. Separating hot and cold data and treating them differently for compression and decompression can reduce relaunch latency and CPU usage by minimizing unnecessary compression, decompression, and swapping. To better separate hot and cold data, our profiler collects all data during an application's relaunches. Each application is relaunched five times, and we collect hot, warm, and cold data for each relaunch. Figure 5 shows the percentage of identical hot data between two consecutive relaunches of an application (i.e., *Hot Data Similarity*) and the fraction of hot data from the prior relaunch that is reused in the later relaunch (i.e., *Reused Data*). Hot Data Similarity is calculated by dividing the amount of identical hot data between two relaunches by the total hot data used during the second relaunch. Reused Data represents the percentage of hot data from the first relaunch that is present in the hot and warm data sets of the second relaunch.
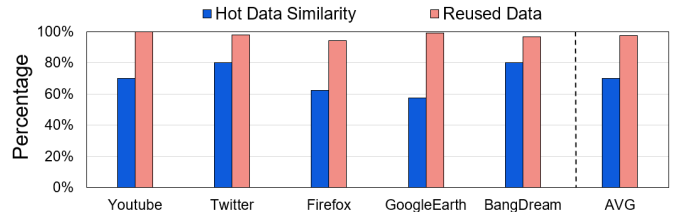

Figure 5: *Hot Data Similarity* and *Reused Data* between two consecutive relaunches of an application across different applications.

Based on the evaluation results, we make two observations. First, the average *Hot Data Similarity* between two consecutive relaunches of an application is 70%, indicating that hot data is generally similar between two consecutive relaunches of an application. Large *Hot Data Similarity* exists because consecutive relaunches of an application typically involve starting the same activities and loading the same interface (e.g., application's logo, user interface) and other status information (e.g., game's status and user's status). Second, the average *Reused Data* is 98%, indicating that the hot data from one relaunch is highly likely to become the hot or warm data in the subsequent relaunch.

Hence, **the first key idea** of our design is to identify hot data using only the information from the most recent relaunch and manage data based on its identified hotness to reduce unnecessary compression and decompression. There are two challenges to realizing this first key idea: 1) How can we identify data hotness dynamically with low overhead (e.g., CPU time and energy consumption)? and 2) How can we effectively handle data based on varying levels of hotness, specifically by either keeping it uncompressed in main memory, compressing it in DRAM, or swapping it into flash memory-based swap space? To effectively address these two challenges, we exploit two other new insights (Insights 2 and 3) that we explain next.

Insight 2: *When compressing the same amount of data, a small-size compression approach (i.e., compressing data in small chunks) is much faster than a large-size compression approach at the cost of lowering compression ratio.*

Compression algorithms typically divide the entire application data into multiple chunks and compress each chunk separately [70–72]. These chunks can be multiple kilobytes or larger (i.e., *large-size compression*) or multiple bytes to a few kilobytes of data (i.e., *small-size compression*).

When compressing the same amount of data, large-size compression achieves a higher compression ratio than small-size compression, as it leverages redundancy over a broader data range [73]. However, it is *not* obvious whether large-size compression causes longer or shorter execution time as there are multiple conflicting factors affecting compression algorithm latency. For example, large-size compression better utilizes the memory bandwidth (due to loading large chunks of data) but at the same time has a larger memory footprint that can negatively affect cache performance [74, 75]. To understand this trade-off better, we measure compression ratio and latency with varying chunk sizes (from 128B to 128KB) using mobile applications' *anonymous data*. We use the default compression algorithms in Android systems, LZO [71] and LZ4 [70], to compress 576 MB of *anonymous data* from real applications included in Section 2.

Figure 6 shows the compression latency (`CompTime`), decompression latency (`DecompTime`), and compression ratio (`CompRatio`) with various compression chunk sizes. Compression and decompression latencies indicate the time taken to compress and decompress a total of 576 MB. Compression ratio refers to the ratio of the original data size to the compressed data size and thus quantifies how much the data size is reduced using the compression algorithm.
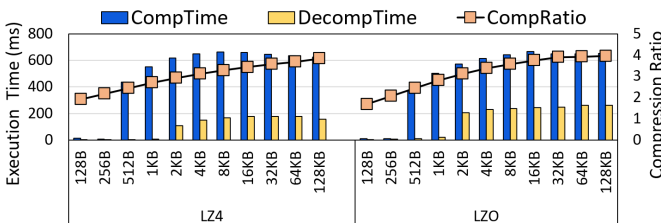

Figure 6: Compression latency, decompression latency, and compression ratio under various compression chunk sizes. X-axis represents the compression chunk size. "128B" means 128 bytes of data is compressed per operation (i.e., a 4KB page will be compressed via 32 operations).

We make two key observations. First, compression ratio increases from 1.7 to 3.9 as compression chunk size increases from 128B to 128KB. Second, small-size compression is significantly faster for the evaluated mobile *anonymous data* workloads. For example, compression latency using 128B compression chunk size is 59.2× and 41.8× faster compared to using 128KB compression chunk size for LZ4 and LZO compression algorithms, respectively. The primary reason for faster small-size compression is the finer data granularity in our evaluated mobile workloads, such as Twitter, YouTube, and Firefox (see Section 5). An *anonymous page* contains multiple types of data blocks, and similar types of data are gathered within a small region (e.g., 128B or 512B), which increases the efficiency of small-size compression.[1]

Hence, **the second key idea** of our design is to employ different compression chunk sizes depending on the hotness level of data. For example, we compress cold data using larger compression chunk sizes to achieve a better compression ratio without worrying about slow decompression latencies, as cold data is unlikely to be read again. For hot data, on the other hand, we use small compression chunk sizes to reduce decompression latency. The challenge in effectively realizing this idea lies in correctly identifying the hotness level of data. Inaccurate identification of the data hotness level can impose both latency and memory capacity overheads. To mitigate the penalty of inaccurate identification, we introduce a new insight (Insight 3), which we explain next.

Insight 3: *There is locality in the address space (i.e., sector numbers) in* zpool *when swapping anonymous data into main memory during application relaunch.*

To mitigate the penalty of inaccurately identifying data hotness levels, we aim to hide decompression latency and the latency of swapping data into main memory (called swap-in) by decompressing and swapping soon-to-be-used data in advance. To do this effectively, we need to predict the next set of data to be used. We assess the spatial locality in accesses to the compressed pages in *zpool*. To this end, we measure the probability of accessing *N consecutive pages* (i.e., pages that are physically adjacent in *zpool*). Table 3 reports the probability of accessing *two* or *four consecutive pages* in *zpool* for each evaluated application.

Table 3: The probability of accessing two or four *consecutive pages* in *zpool* for each evaluated application.

	Youtube	Twitter	Firefox	GoogleEarth	BangDream
2	0.86	0.81	0.69	0.77	0.61
4	0.72	0.61	0.43	0.54	0.33

We make two key observations. First, most applications exhibit locality in data access in *zpool* when swapping in *anonymous data* during application relaunch. For example, the probability of accessing two *consecutive pages* is 86% for YouTube. This means that if we pre-decompress and pre-swap the immediate next page of the currently-being-accessed page, the

---

[1]To foster further research in the design and optimization of mobile compressed swap techniques, we open-source our implementations at https://github.com/CMU-SAFARI/Ariadne.

pre-swapped page has an 86% chance of being used by the application soon. Second, the probability of accessing four *consecutive pages* is significantly lower than that of two *consecutive pages* (17%-46% lower across various applications). Hence, pre-decompressing the three immediate next pages can pollute main memory with pages that are *not* going to be used.

Hence, **the third key idea** of our design is to predict the next set of data to be used and pre-decompress it, reducing the impact of swapping data back into main memory and decompression latency on application relaunch. There are two design decisions associated with realizing this idea: 1) How much data should be pre-decompressed? and 2) When should we do pre-decompression? Our design addresses these decisions, as presented in Section 4.

Summary. We uncover three new insights by analyzing modern mobile workloads. First, hot data is usually similar between consecutive application relaunches. Second, small-size compression/decompression is fast, while large-size compression achieves a better compression ratio. Third, there is locality in data access in *zpool* when swapping in *anonymous data* during application relaunch. These new insights lead to three key ideas: hotness-awareness data organization, size-adaptive compression, and pre-decompression, as we discuss in Section 4.

4. Ariadne Design

4.1. Design Overview

We propose Ariadne, a new compressed swap scheme for mobile devices that reduces application relaunch latency and CPU usage while increasing the number of live applications for enhanced user experience. The key idea of Ariadne is to reduce the frequency and latency of compression, decompression, swap-in, and swap-out operations by leveraging different compression chunk sizes based on the hotness level of the data, while also performing speculative decompression based on data locality characteristics.
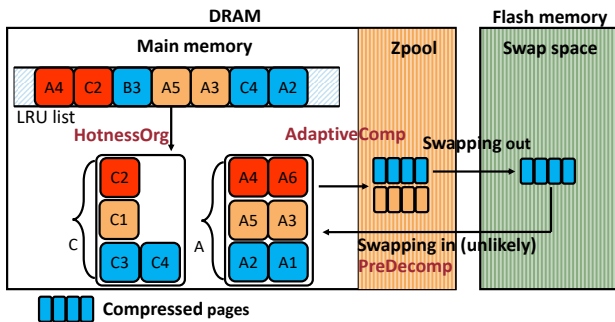

Figure 7: Design overview of Ariadne. Ariadne incorporates three key techniques: HotnessOrg, AdaptiveComp, and PreDecomp. It involves *zpool* and flash memory-based swap space management. Blocks refer to data pages in main memory. Colors represent the hotness of the data pages: red (hot), orange (warm), blue (cold).

Data storage architecture of Ariadne. Figure 7 presents an overview of Ariadne. We use colors to represent the hotness levels of data pages: red for hot, orange for warm, and blue for cold data. In Android systems, *anonymous data* of running applications can be stored in main memory, *zpool*, or flash

memory-based swap space. Main memory has the lowest access latency, while flash memory-based swap space has the highest. Therefore, systems usually prioritize storing *anonymous data* in main memory for best performance. When main memory capacity is limited, systems use the *ZRAM* scheme to compress the least recently used (LRU [50]) data into *zpool*. The flash memory-based swap space serves as main memory extension to store compressed data swapped out from *zpool* when there is insufficient main memory space. Ariadne chooses to swap out *compressed* data, which leads to smaller writes to flash memory and lower storage space consumption. However, this design choice may increase read latency due to decompression. We reduce the probability of incurring such latency by mainly writing cold data (that is unlikely to be read again) into the flash swap space.

Key mechanisms of Ariadne. Based on the above data storage architecture, Ariadne incorporates three techniques: First, Ariadne uses a low-overhead, hotness-aware data organization mechanism, called HotnessOrg, to determine data hotness and maintain data with different levels of hotness in separate memory page lists accordingly. The goal of HotnessOrg is to reduce the frequency of compression/decompression and swap-in/swap-out operations. To achieve this goal, Ariadne aims to maintain uncompressed hot data in main memory, compress warm data into *zpool*, and swap compressed cold data to the flash memory-based swap space (see Section 4.2). Second, Ariadne enables a size-adaptive compression mechanism, called AdaptiveComp, to leverage the benefits of different compression chunk sizes. The goal of AdaptiveComp is to achieve both short relaunch latency and a good compression ratio by using small-size compression chunks for identified warm data and large-size compression chunks for cold data (see Section 4.3). Third, rather than relying on on-demand decompression or data swapping-in operations during application relaunches, Ariadne employs a *proactive and predictive decompression* (i.e., *pre-decompression*) mechanism, called PreDecomp, that leverages data locality to proactively determine the best data and timing for compression and swapping. The goal of PreDecomp is to mitigate the negative impact of read latency on the user experience (see Section 4.4). We also consider the compatibility of Ariadne with different compression algorithms and memory management optimizations (See Section 4.5).

4.2. Low-overhead Hotness-Aware Data Organization

We propose a hotness-aware data organization mechanism, called HotnessOrg, that builds on LRU-based memory management. HotnessOrg aims to improve compression efficiency by separating hot, warm, and cold data efficiently. The challenge is how to identify data hotness dynamically and accurately with low memory capacity and CPU overhead. Specifically, HotnessOrg encompasses two aspects: data organization within an application and data organization among applications, as shown in Figure 8.

Within an application. Data organization within an application involves three components: hotness initialization, hotness update, and data eviction. HotnessOrg separates all the anony-
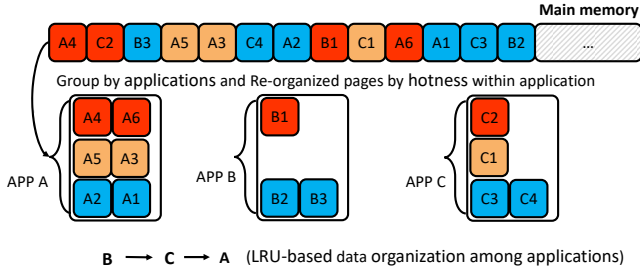
Figure 8: Hotness-aware data organization (HotnessOrg). Blocks refer to data pages in main memory. Colors represent the hotness of the data pages: red (hot), orange (warm), blue (cold).

mous data of the application into three LRU lists (hot, warm, and cold) rather than the typical two lists (active and inactive by default [29, 76]). First, for hotness initialization, when a system launches an application for the first time, the system adds a certain amount of data used during the launch to the hot list (i.e., the LRU list to store hot data in main memory). To reduce overhead, we profile data usage for each application during its relaunch to determine the *initial* size (i.e., data amount) of the hot list. The profiling procedure for this size is the same as the one used for the data shown in Figure 5. This profiling works effectively because: i) the amount of hot data remains similar for each relaunch of an application, as shown in our collected traces and the results in prior work [7]; ii) Ariadne adaptively updates the hot list during application relaunch and execution. Then, the system adds other data generated during application execution to the cold list. If the application accesses data in the cold list during execution, the system moves the data to the warm list. Moving data from the cold list to the warm list is similar to default Android systems, which move data from the inactive list to the active list. This initialization procedure does not incur additional overhead.

Second, for hotness update (i.e., moving data among hot, warm, and cold lists according to the access pattern), after re-launching an application, the system moves all old data in the hot list to the warm list and adds the data from this relaunch to the hot list. This ensures that the hot data from the most recent relaunch is in the hot list. Third, for data eviction, the system first chooses data from the cold list of an application for compression. If all cold data of all applications are compressed, it starts compressing data from the warm list, and finally (if absolutely necessary) the hot data. When the *zpool* space is insufficient to store all the compressed data, the system writes some compressed data to flash memory-based swap space following a policy that ensures cold data is swapped out first.

Across applications. For data organization across applications, we have two policies. First, we continue using the LRU policy to manage an LRU-based application list. Applications are added to the LRU list based on the access time of their most recently accessed page (eviction order of applications is A, C, and B, as shown in Figure 8). Second, we prioritize using the main memory (DRAM) capacity for foreground applications. This policy is compatible with the *mem_cgroup* function [77] that can be enabled in the Linux kernel.

All data organization tasks involve only LRU list operations, without physically moving data, similar to the baseline system. Only adding old hot data to the warm list is an additional LRU operation compared to the baseline system. Thus, HotnessOrg is a low-overhead data organization mechanism.

In summary, HotnessOrg efficiently identifies and exploits data hotness. By leveraging data access patterns during application relaunch and execution, HotnessOrg can efficiently identify data hotness and manage it with minimal overhead.

4.3. Efficient Size-Adaptive Compression

We propose an efficient size-adaptive memory compression mechanism, called AdaptiveComp, that allows for compressing data using different compression chunk sizes based on data hotness.

Adaptive size according to data hotness for data compression. Large-size chunks for compression (i.e., large-size compression) are not commonly used in current mobile systems for two reasons. First, large-size compression tends to increase data movement, computational overhead, and energy consumption, as large data chunks could involve more unused data, which could be redundantly transferred between host CPU and DRAM, as shown in Figure 1. Second, according to our Insight 2 in Section 3, the compression and decompression latency is longer when using large-size compression. AdaptiveComp addresses these issues by leveraging the hot and cold data separation supported by HotnessOrg (see Section 4.2). Ariadne utilizes large-size compression for *cold* data to achieve a good compression ratio. Compressing only cold data using large chunks mitigates performance penalties of using large chunks since cold data is unlikely to be reused. Conversely, AdaptiveComp uses small-size compression for hot and warm data to achieve better relaunch latency and execution performance. As a result, AdaptiveComp can take advantage of different compression chunk sizes without incurring their typical penalties. Figure 9 compares the decompression procedures for a given compressed page in *ZRAM* versus AdaptiveComp. We use decompression as an example to explain AdaptiveComp's workflow, as it better illustrates the penalties and benefits of our design compared to the baseline compression mechanism used in *ZRAM*.

For the baseline one-page (i.e., 4KB) compression chunk size (see Figure 9 (a)), when a user launches application A, the system reads the A-related compressed blocks (blocks A1&B2 and C4&A2) from *zpool* to the host CPU ❶ and decompresses them. The system writes the decompressed pages A1 and A2 back to main memory (DRAM) to facilitate application A's launch ❷. Finally, the system merges the unused compressed data and writes block B2&C4 back to *zpool* ❸. When decompressing block A1&B2, B2 is *not* decompressed because the system uses a one-page compression chunk size, meaning pages A1 and B2 are compressed individually.

In contrast, the decompression procedure in Ariadne differs from that in the default *ZRAM* in two major ways: First, Ariadne's HotnessOrg organizes data based on its hotness level, unlike the default *ZRAM* that uses LRU policy. As a result, the data layout in both the main memory and the *zpool* using
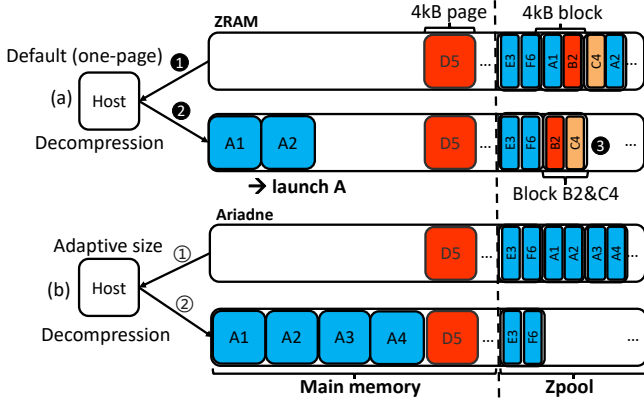
Figure 9: Decompression procedure of efficient size-adaptive compression (AdaptiveComp). Colors represent the hotness of the data pages: red (hot), orange (warm), blue (cold). The data layout in *zpool* differs between the baseline *ZRAM* scheme and Ariadne, as they employ different data organization policies.

Ariadne differs significantly from that in *ZRAM*. For example, when a user launches application A, the hot data required for the relaunch is in main memory.

Second, Ariadne performs compression operations based on data hotness levels. For example, large-size compression targets cold data that is unlikely to be accessed again. To illustrate both the benefits and potential drawbacks of large-size compression, we present a worst-case scenario (i.e., need to decompress the data that was compressed using a large size) of large-size decompression using Ariadne in Figure 9 (b). The worst case occurs when cold data is incorrectly predicted and later needs to be used after being compressed. In this case, when the system reads the A-related compressed blocks (A1&A2 and A3&A4) from *zpool* to the host CPU and decompresses them ①. By leveraging a large-size compression policy, the system decompresses pages A1 and A2 together, as well as A3 and A4 together. Following decompression, the system writes all four decompressed pages back to the main memory space in DRAM ② to facilitate application A's relaunch. Thus, when compressing data at large granularity, the system decompresses compressed blocks (e.g., A1&A2 and A3&A4) entirely, even if the application only requires a small portion of the blocks. This can result in wasted CPU time and memory capacity if decompressed pages are not accessed together.

By leveraging the hotness level-based data separation provided by HotnessOrg, Ariadne enables an efficient size-adaptive compression mechanism, AdaptiveComp. AdaptiveComp enables the processing of multiple cold data pages using a single compression operation while avoiding the drawbacks of compressing a large amount of data at once. Consequently, Ariadne significantly reduces the frequency of data compression and decompression operations by leveraging HotnessOrg and AdaptiveComp, thereby lowering application relaunch latency and CPU usage. To further mitigate the impact of decompression latency on application relaunches, we propose hiding this latency by decompressing soon-to-be-used data in advance, which we explain next.

4.4. Proactive and Predictive Decompression

We propose a proactive decompression mechanism, called Pre-Decomp (i.e., pre-decompression), to efficiently and proactively perform decompression operations ahead of reading, thereby reducing the negative impact of decompression latency on read latency. The challenge lies in accurately predicting the best data and timing for decompression while minimizing CPU and memory capacity overhead.

Fast prediction of data to be decompressed. According to Insight 3 in Section 3, there is locality in data access in *zpool* when swapping-in *anonymous data* during application relaunch. HotnessOrg organizes and maintains all hot and warm data with high locality. Thus, PreDecomp can accurately predict the next data for decompression using the data layout organized by HotnessOrg. For example, when a compressed page is required, its subsequent page will also be proactively pre-decompressed. Since the probability of accessing two *consecutive pages* is high (see Table 3), we pre-decompress only one compressed page at a time, ensuring high accuracy while minimizing the memory capacity overhead required to store the pre-decompressed data.

Lightweight pre-decompression method. To support pre-decompression, Ariadne maintains a buffer in the main memory to store the pre-decompressed data. When the buffer is full, it uses a first-in, first-out policy. The larger the buffer size, the longer compressed data can be stored before it is used in main memory. For example, if the buffer size is only one page, the system should use the pre-decompressed data immediately. Otherwise, the data will be compressed again.

In summary, PreDecomp exploits data access patterns to determine the best time and data for which to perform decompression and swapping. By doing so, PreDecomp efficiently performs pre-decompression operations to mitigate read latency, thereby enhancing user experience.

4.5. Compatibility of Ariadne with Other Techniques

We discuss the compatibility of Ariadne with: i) different compression algorithms and ii) other memory management schemes (e.g., memory allocation and memory reclamation schemes).

Compression algorithms. It is valuable for systems to support multiple compression algorithms to cater to different objectives. Therefore, ensuring compatibility with a range of compression algorithms is important. Ariadne is compatible with various compression algorithms, such as LZO [71], LZ4 [70], and base-delta compression [78, 79]. Ariadne naturally supports different compression algorithms, such as switching between LZO and LZ4, as it inherits *ZRAM*'s interface. Compression algorithms might require slight interface modifications to support AdaptiveComp by adjusting the compression chunk size.

Impact on other memory management schemes. Ariadne is compatible with various baseline memory management algorithms, such as default memory allocation and memory reclamation techniques used in modern systems. Ariadne does *not* impact the memory allocation procedure because memory allocation operates only on available memory space, while Ariadne focuses on organizing data. However, Ariadne affects the memory reclamation scheme. Specifically, in a mobile system, the

memory reclamation scheme selects data to reclaim (called victim data) based on the LRU policy. In contrast, the memory reclamation scheme of Ariadne uses our hotness-aware data organization scheme (HotnessOrg) to select victim data, enhancing reclamation efficiency by prioritizing the reclamation of cold data. Ariadne is also compatible with swap scheme optimizations, such as MARS [34], FlashVM [80], Fleet [8], and SmartSwap [33], as Ariadne is complementary to them.

5. Evaluation Methodology

Experimental platform. We implement and evaluate Ariadne on a real commercial smartphone, Google Pixel 7 [26] with the latest Android 14 operating system [27]. We list the detailed real system configuration in Table 4.

Table 4: Experimental Platform Configuration.

Name	System	Memory & Storage
Google Pixel 7	CPU: 8 cores 2x 2.85 GHz Cortex-X1 [81] & 2x 2.35 GHz Cortex-A78 [82] & 4x 1.8 GHz Cortex-A55 [83]	12GB DRAM 128GB flash
	Android: 14; Linux 5.10.157 [27]	UFS3.1

Workloads. We execute ten popular applications (Twitter, YouTube, TikTok, Edge, Firefox, Google Earth, Google Maps, BangDream, Angry Birds, and TwitchTV) via MonkeyRunner [84] to collect mobile workload traces.[2] Using mobile workload traces makes our methodology and results reproducible, as opposed to running real applications that execute differently for each different test. We use the collected traces for both insight analysis and final evaluation results. For example, we use the collected page data in traces as the input of compression and decompression algorithms for both the state-of-the-art *ZRAM* scheme and Ariadne. This allows us to reproducibly and consistently compare their compression latency, decompression latency, and compression ratio. A trace is composed of the page frame number (PFN), *ZRAM* sector, source application number (UID), and page data that needs to be compressed, swapped-in or swapped-out. We create *ten* traces. Our procedure for creating each trace is as follows. First, for each trace, we select a target application out of ten applications to launch and execute. Second, we put the target application in the background and launch the other nine applications. To capture more information across various usage scenarios, we launch the nine applications in different orders, creating several (e.g., three) distinct usage scenarios for each target application. Third, we relaunch the target application to collect its relaunch information.

To prevent interference during each trace creation and ensure the reproducibility of our methodology, we perform the following three actions for each trace collection: i) we close all applications and clear their cache files before rebooting the smartphone to eliminate the impact of old cache files, ii) after rebooting the smartphone, we clean the cache again to eliminate the impact of potentially buffered data in main memory, and iii) we use the same applications with the same user account and

---

perform the same sequence of activities using an auto-testing script via MonkeyRunner [84] to avoid human bias.

To foster further research in the design and optimization of mobile compressed swap techniques, we open-source all our source code, traces, and scripts at https://github.com/CMU-SAFARI/Ariadne.

Evaluated Schemes. We evaluate two compressed swap schemes: **1)** *ZRAM* [10, 11, 48, 49], which is the state-of-the-art compressed swap scheme used in modern Android systems. ZRAM employs Least Recently Used (LRU) [50] as the default policy for selecting data to compress. With LRU, the system selects the least recently used pages for compression. Modern Android systems optimize memory page organization by grouping data based on the associated application. This solution only supports single-page-size (i.e., 4KB) compression to avoid potential penalties (as discussed in Section 4.3) and does *not* allow data to be decompressed *before* it is required by the system, avoiding memory capacity waste and unnecessary CPU usage if the decompressed data will not be used.

2) Different versions of Ariadne, which is our proposed compressed swap scheme. We evaluate Ariadne under different configurations, whose parameters are shown in Table 5. *S* represents the size of *zpool*, which is set to 3GB. This parameter determines the maximum number of compressed pages that can be stored in *zpool* and consequently affects user experience in two ways: 1) The size of *zpool* impacts the number of writes to the NAND flash memory and its overall lifetime, and 2) it affects the relaunch-related data placement, thereby impacting application relaunch latency. *SmallSize*, *MediumSize*, and *LargeSize* represent the compression chunk sizes for hot list, warm list, and cold list, respectively. They serve as inputs for compression algorithms and affect both the compression ratio and compression latency. We denote these size configurations as *SmallSize-MediumSize-LargeSize* (e.g., 1K-2K-16K) for each version of Ariadne in Section 6.

Table 5: Summary of parameters used by Ariadne.

Parameter	Description	Setting (B)
S	Size of ZRAM partition	3G
SmallSize	Compression chunk size for hot list	256,512,1K
MediumSize	Compression chunk size for warm list	2K,4K
LargeSize	Compression chunk size for cold list	16K,32K

During an application relaunch, the system fetches all the launch-related data into main memory. In the optimal case, all data resides in main memory (DRAM). In other scenarios, we consider two situations: i) where data in the *hot* list is in main memory while other data is in either *ZRAM* or flash memory-based swap space, and ii) where all data needs to be read from either *ZRAM* or flash memory-based swap space. The first scenario excludes hot list data from compression and decompression operations, and the second scenario applies compression and decompression on data in all lists. We abbreviate these two scenarios as EHL (exclude hot list) and AL (all lists).

Evaluated Metrics. We evaluate Ariadne using three major sets of metrics. First, we assess the impact of Ariadne on factors that influence user experience, including application relaunch

latency and the CPU usage due to compression and decompression. Second, we analyze the auxiliary metrics, including compression/decompression latency, compression ratio, and the accuracy and coverage of data hotness level identification. Third, we analyze memory capacity and CPU usage associated with our full Ariadne implementation (not just compression/decompression).

6. Evaluation Results

We evaluate the effectiveness of Ariadne compared to the state-of-the-art *ZRAM*. Section 6.1 shows the overall effect of Ariadne on the user experience with modern mobile devices. Section 6.2 analyzes the effectiveness of key techniques of Ariadne by presenting auxiliary metrics. Section 6.3 provides a sensitivity study on compression chunk size configurations. Section 6.4 studies the memory capacity and CPU usage associated with our full Ariadne implementation.

6.1. Effect on User Experience

There are two metrics that significantly affect the user experience on mobile devices: i) application relaunch latency [1, 8] and ii) CPU usage, which directly impacts battery usage [85].
Application relaunch latency. Figure 10 shows the application relaunch latency of the evaluated applications under different compressed swap schemes (i.e., *ZRAM* and Ariadne with different configurations). We implement Ariadne on a real smartphone, the Google Pixel 7 [26], running the Android 14 operating system [27]. We execute our traces that are collected under different data organization policies (i.e., LRU in baseline *ZRAM* and HotnessOrg in Ariadne) on the smartphone using the *ZRAM* scheme and Ariadne, respectively. To demonstrate a lower bound for the best possible (optimal) latency for application relaunches, we also evaluate application relaunch latency under an ideal scenario, called *DRAM*, where the system reads *all* application data directly from DRAM (with the optimistic assumption that DRAM is large enough to host all such data), i.e., there is no swapping overhead. The x-axis represents the evaluated applications, and the y-axis indicates their relaunch latencies. We report results for five randomly selected applications (out of 10) for readability.[3]
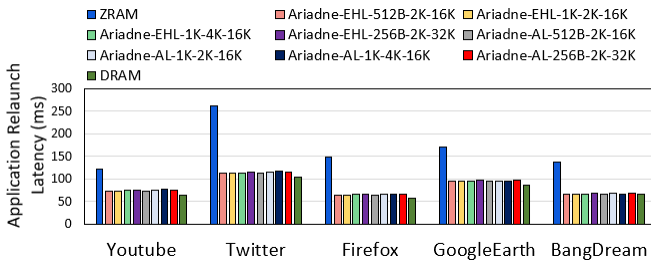

Figure 10: Application relaunch latency.

We make two key observations. First, all versions (i.e., different configurations) of Ariadne reduce the relaunch latency

by around 50%, on average, compared to *ZRAM*. We believe Ariadne enhances the user experience by reducing the relaunch latency as users switch among various applications with high frequency (e.g., >100 times a day [3]). Second, the relaunch latency of all Ariadne configurations is within 10% percent of that of the optimistic DRAM configuration. This demonstrates that Ariadne effectively hides most of the latency due to compressed swapping in main memory. Third, the performance difference between EHL and AL is negligible for a given same-size configuration. For instance, YouTube's relaunch latencies are 73 ms and 75 ms under *Ariadne-AL-1K-2K-16K* and *Ariadne-EHL-1K-2K-16K*, respectively. This is because Ariadne intelligently adapts to different compression chunk sizes based on the hotness level of the data. We conclude that Ariadne effectively reduces application relaunch latency, thereby significantly enhancing user experience.

CPU usage. Figure 11 illustrates the CPU usage for compression and decompression procedures across different versions of Ariadne, normalized to the CPU usage for these procedures using the baseline *ZRAM* scheme.

We make three key observations. First, all versions of Ariadne with EHL significantly reduce CPU usage during compression and decompression for applications that generate more hot data. For instance, Ariadne with EHL reduces CPU usage by 25% for YouTube and 30% for Twitter. Second, Ariadne with AL, using smaller-size compression (e.g., *256B-2K-32K*) achieves similar CPU usage to Ariadne with EHL. Third, for the applications that produce less hot data[4], such as BangDream, CPU usage increases by about 3% with EHL versus with AL. This is because more data is compressed using larger sizes. However, since warm and cold data are accessed less frequently than hot data, Ariadne can effectively offset the CPU overhead caused by large-size compression by reducing the frequency of compression and decompression operations.

Overall, compared to the state-of-the-art *ZRAM*, Ariadne achieves an average CPU usage reduction of approximately 15% across all configurations. We conclude that Ariadne significantly reduces CPU usage compared to the baseline *ZRAM* scheme that underscores the effectiveness of HotnessOrg and AdaptiveComp.
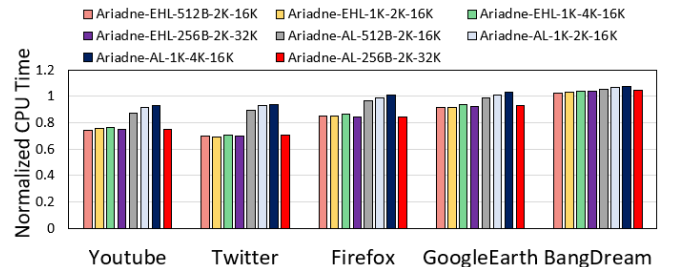

Figure 11: Normalized CPU usage of compression and decompression procedures across different versions of Ariadne, normalized to the CPU usage for these procedures under *ZRAM*.

---

[3]We present the same applications for Figures 10–13 and Figure 15. We release the results for all applications in our GitHub repository [86] and the appendices of the extended version of the paper [87].

[4]We report the proportion of data at different hotness levels from traces in our GitHub repository [86] and the appendices of the extended version of the paper [87].

6.2. Analysis of Ariadne

To study the effectiveness of the key techniques of Ariadne, we investigate how Ariadne influences relaunch latency and CPU usage through four key auxiliary metrics: i) compression and decompression latency, ii) compression ratio, iii) accuracy and coverage of hot data identification for an application relaunch.

Compression and decompression latency. Figure 12 shows the data compression and decompression latency in evaluated applications. We evaluate LZO [71] compression algorithm supported by Google Pixel 7. The x-axis represents the evaluated applications, and the y-axis shows the compression and decompression latency of data from their traces.
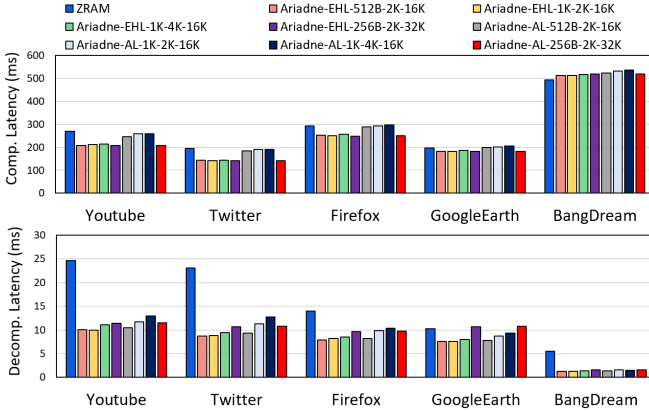

Figure 12: Compression and decompression latency using different compressed swap schemes (i.e., different versions of Ariadne and ZRAM).

We make two key observations. First, all versions of Ariadne significantly reduce the decompression latency. For example, Ariadne with the configuration *1K-2K-16K* reduces the decompression latency by approximately 60% for YouTube and Twitter, and by approximately 90% for BangDream, compared to the baseline *ZRAM* scheme. This is because Ariadne uses fast, small-size compression on frequently decompressed data, i.e., hot and warm data, which leads to fast decompression. Second, compression latency is also reduced for all applications except BangDream. For YouTube and Twitter, Ariadne-EHL with the configuration *1K-2K-16K* reduces compression latency by 20%. This reduction is primarily due to the use of large-size compression on cold data and reduced compression operations on hot data. We conclude that Ariadne significantly reduces decompression latency across various applications, which in turn reduces application relaunch latency.

Compression ratio. Figure 13 presents the compression ratio of data of different applications when using different compressed swap schemes. We make two observations. First, Ariadne-EHL with the size configuration *1K-4K-16K* consistently provides better compression ratio than ZRAM for every application. This is because larger compression chunk sizes result in better compression ratios across all hotness levels of data (as we have discussed in Section 3). Second, Ariadne-AL, using smaller compression chunk sizes (i.e., *512B-2k-16K*) achieves a similar compression ratio to that of *ZRAM*. This is because we select size configurations to balance the tradeoff between compres-

sion and decompression latency and the compression ratio. We conclude that Ariadne provides comparable or even better compression ratios, compared to the baseline ZRAM scheme, which can positively affect both application relaunch latency and flash memory lifetime.
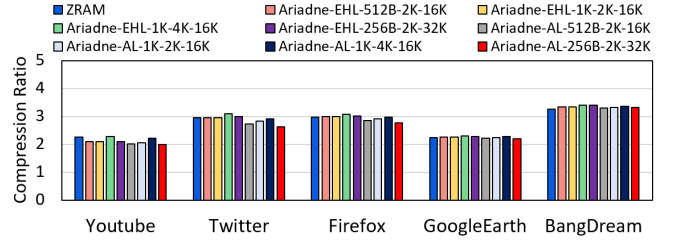

Figure 13: Compression ratios under different compressed swap schemes. Higher values are better.

Accuracy and coverage of hot data identification. Figure 14 shows the *Coverage* and *Accuracy* of hot data identification for all the evaluated applications. *Coverage* refers to the percentage of correctly predicted data of an application relaunch, and *Accuracy* denotes the percentage of data in the hot list that will be utilized next time, including the data used during both relaunch and execution. We make two key observations. First, Ariadne's *Coverage* for hot data is approximately 70% on average. When hot data is mistakenly categorized as warm or cold data, it is compressed in larger sizes, which can lead to longer decompression latencies (see Section 6.3 for more detail). Second, *Accuracy* of hot data identification is approximately 92%. This means that our prediction incurs a small penalty for storing all data in the hot list in main memory, as 92% of the stored hot data will be used in the next application relaunch or execution.
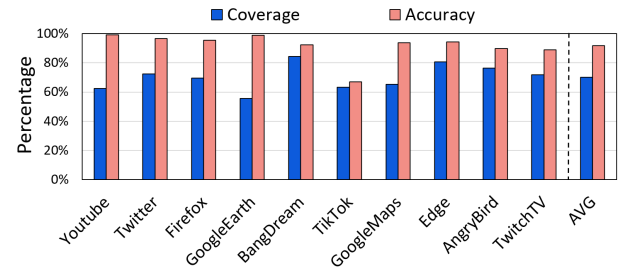

Figure 14: Coverage and accuracy of Ariadne's hot data identification method for different applications.

6.3. Sensitivity Study

We analyze the sensitivity of compression chunk size on compression/decompression latency and compression ratio in Ariadne. We evaluate two example configurations to illustrate the size configurations' impact on compression and decompression latency as well as compression ratio in Figure 15. The x-axis represents the targeted applications across all three figures. The y-axis, respectively, shows (a) compression latency, (b) decompression latency, and (c) compression ratio of the data from the targeted application traces.

We make two observations. First, selecting inappropriate compression chunk sizes for different hotness levels of data either increases the compression and decompression latencies or reduces the compression ratio. Second, using a very large
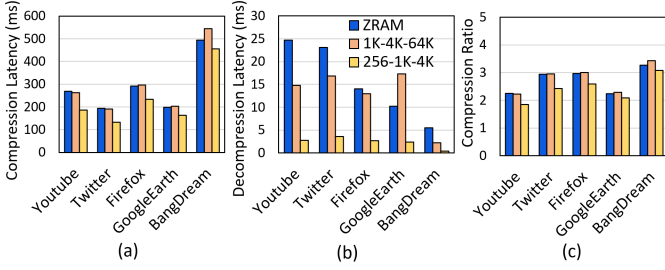
Figure 15: Sensitivity study: Compression latency (a), decompression latency (b), and compression ratio (c) under *ZRAM*, *Ariadne-AL-1K-4K-64K*, and *Ariadne-AL-256-1K-4K*.

compression chunk size for cold data increases the compression ratio without the penalty of long decompression latency. However, it also carries significant risks of potential performance loss if data profiling is inaccurate. If hot or warm data is misclassified as cold data, it gets compressed using a larger chunk size, resulting in longer decompression latencies and worse user experience during application relaunch. Thus, we avoid using excessively large chunk sizes (e.g., $\geq$ 64K) even for cold data.

6.4. Overhead Analysis

We analyze the memory capacity and CPU overhead for all three techniques: HotnessOrg, AdaptiveComp, and PreDecomp. First, HotnessOrg achieves hotness-aware data organization without physically moving data. Instead, it employs a new data selection policy during compression by operating on LRU lists. This policy does not affect application execution, relaunch latency, or energy consumption, as it only involves operations on the LRU lists, increasing them slightly over the baseline ZRAM system. Specifically, Ariadne increases LRU list operations to move part of the previous application's hot data into the warm list when relaunching a new application. Since an LRU list operation is much faster (e.g., 100× [88]) than swapping [5, 33], the overhead is negligible. Second, AdaptiveComp could introduce memory capacity and CPU overhead if it compresses data used at different times together, as discussed in Section 4.3. Thus, there is no overhead on hot and warm data, as we use small-size (i.e., smaller than one page) compression for them, ensuring that all the decompressed data will be used together. The overhead on cold data is negligible, as it is unlikely to be accessed again due to the high identification accuracy, as shown in Figure 14. Third, PreDecomp may result in memory capacity overhead and increased energy consumption if the predictions for pre-decompression are inaccurate. To minimize such overheads, we pre-decompress only one page, ensuring high prediction accuracy, as shown in Table 3. In summary, Ariadne has small overhead in terms of computation and memory space.

7. Related Work

To our knowledge, Ariadne is the first work that leverages different compression chunk sizes based on the hotness level of the data, while also performing speculative decompression based on data locality characteristics to improve the performance of compressed swap schemes on mobile devices. We have already compared Ariadne extensively with the state-of-the-art

ZRAM scheme [10] in Section 6. In this section, we discuss related work in two broad categories: flash memory-based swap schemes and emerging NVM-based swap schemes.

Flash memory-based swap schemes. Several prior works [6, 8, 34, 37, 40, 69, 80, 89–91] explore using flash memory-based storage as an extension of main memory. While doing so increases memory capacity, it reduces the lifetime of flash memory due to the increased number of writes. Some prior works [8, 34] aim to reduce writes to flash memory by reducing interference caused by the Android runtime garbage collector on page swapping. MARS [34] tracks pages that have undergone runtime garbage collection and avoids swap operations on these pages. MARS also employs several flash-aware techniques to accelerate swap operations. Fleet [8] performs runtime garbage collection only on soon-to-be-invalid data of background applications to reduce unnecessary swap operations on long-lifetime foreground application data. To further reduce swap latency, SmartSwap [33] predicts the most rarely used applications and dynamically swaps these applications' data to flash memory-based swap space ahead of time. FlashVM [80] modifies the paging system along code paths for allocating, reading, and writing back pages to optimize the use of storage devices for fast swapping. Flash memory-based swap schemes typically focus on minimizing flash writes or accelerating swap operations via data filtering or efficient page write-back mechanisms. In contrast, the key idea of Ariadne is to reduce the frequency and latency of compression, decompression, swap-in, and swap-out operations by leveraging different compression chunk sizes based on the hotness level of the data, while also performing speculative decompression based on data locality characteristics. Ariadne can be combined with these prior flash memory-based swap schemes.

Several *ZSWAP*-based works [5, 92, 93] aim to leverage both main memory compression schemes (e.g., *ZRAM*) [10, 11, 48, 49] and flash memory-based swapping space to reduce writes to flash memory. The key idea of *ZSWAP* [5, 92, 93] is to initially move pages to *zpool* and subsequently evict them to secondary storage to accommodate newly incoming pages. *ZSWAP* has already been incorporated into Ariadne (see Section 4).

An optimization of *ZSWAP*, *ezswap* [5], has two key features: 1) compressing both anonymous and file data, and 2) estimating compression ratios to selectively decide which page to compress. The first feature can be combined with Ariadne. While the second can improve *zpool* efficiency, it comes at the cost of additional compression latency and may impact application relaunch latency. Our evaluation shows *ezswap*'s compression ratio estimation overhead accounts for up to 16.7% of total compression latency. Ariadne avoids high-overhead compression ratio estimation by using different compression chunk sizes for hot and cold data.

Emerging NVM-based swap schemes. Several previous works [32, 33, 38, 39, 41–43, 94] investigate how to efficiently enable swap-based NVMs for mobile devices. These works aim to improve swap scheme performance by separating hot and cold data and efficiently exploiting hardware features. For example,

CAUSE [38] introduces a hybrid memory architecture for mobile devices that intelligently allocates DRAM or NVM based on the criticality of the data. Two other prior works [39, 41] utilize an NVM-based swap space for Android devices, leveraging hot and cold data management to efficiently handle swaps between DRAM and NVM. These works do not leverage the tradeoff between compression latency and compression ratio for mobile workloads with varying data hotness or criticality. Ariadne is complementary to these emerging NVM-based swap schemes.

8. Conclusion

State-of-the-art compressed swap schemes to manage limited main memory capacity on mobile devices lead to prolonged application relaunch latency and high CPU usage. To address this problem, we introduced a new compressed swap scheme for mobile devices, called Ariadne. Ariadne leverages different compression chunk sizes based on the hotness level of the data, while also performing speculative decompression based on data locality characteristics to improve the performance of compressed swap schemes. We implement and evaluate Ariadne on a real commercial smartphone with a cutting-edge Android operating system. Our experimental evaluation results show that Ariadne surpasses the state-of-the-art swap scheme in both application relaunch latency and CPU usage.

References

[1] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the Senseless Killing: Improving Memory Management for Mobile Operating Systems," in *USENIX ATC*, 2020.

[2] FinacesOnline, "Number of Smartphone and Mobile Phone Users Worldwide in 2024: Demographics, Statistics, Predictions," https://financesonline.com/number-of-smartphone-users-worldwide/, 2024.

[3] T. Deng, S. Kanthawala, A. Meng, W. Peng, A. Kononova, Q. Hao, Q. Zhang, and P. David, "Measuring Smartphone Usage and Task Switching with Log Tracking and Self-Reports," *Mobile Media & Communication*, 2019.

[4] Linux Foundation, "Concepts Overview of Linux," https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.htmlanonymous-memory, 2024.

[5] J. Kim, C. Kim, and E. Seo, "*ezswap*: Enhanced compressed swap scheme for mobile devices," *IEEE Access*, 2019.

[6] S. Bergman, N. Cassel, M. Bjørling, and M. Silberstein, "ZNSwap:un-Block your Swap," in *USENIX ATC*, 2022, pp. 1–18.

[7] S. Son, S. Y. Lee, Y. Jin, J. Bae, J. Jeong, T. J. Ham, J. W. Lee, and H. Yoon, "ASAP: Fast Mobile Application Switch via Adaptive Prepaging," in *USENIX ATC*, 2021.

[8] J. Huang, Y. Zhang, J. Qiu, Y. Liang, R. Ausavarungnirun, Q. Li, and C. J. Xue, "More Apps, Faster Hot-Launch on Mobile Devices via Fore/Background-Aware GC-Swap Co-design," in *ASPLOS*, 2024.

[9] Y. Liang, J. Li, R. Ausavarungnirun, R. Pan, L. Shi, T.-W. Kuo, and C. J. Xue, "Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems," in *USENIX ATC*, 2020.

[10] N. Gupta, "ZRAM: Compressed RAM Based Block Devices. Linux Foundation, San Francisco, CA, USA. [Online]," https://www.kernel.org/doc/Documentation/blockdev/zram.txt, 2021.

[11] M. K. Johnson, "An Introduction to Block Device Drivers. Linux Foundation, Linux Journal [Online]," https://www.linuxjournal.com/article/2890, 1995.

[12] Linux Patch, "Zoop Patch: ZRAM Use Common Zpool Interface. [online]," 2019, https://patchwork.kernel.org/project/linux-mm/patch/20191010232030.af6444879413e76a780cd27e@gmail.com/#22936275.

[13] C. Shin, J.-H. Hong, and A. K. Dey, "Understanding and Prediction of Mobile Application Usage for Smart Phones," in *UbiComp*, 2012.

[14] P. Leroux, K. Roobroeck, B. Dhoedt, P. Demeester, and F. De Turck, "Mobile Application Usage Prediction through Context-Based Learning," *JAISE*, 2013.

[15] Z. Shen, K. Yang, W. Du, X. Zhao, and J. Zou, "Deepapp: A Deep Reinforcement Learning Framework for Mobile Application Usage Prediction," in *SenSys*, 2019.

[16] M. Matsumoto, R. Kiyohara, H. Fukui, M. Numao, and S. Kurihara, "Proposition of the Context-Aware Interface for Cellular Phone Operations," in *INSS*, 2008.

[17] R. Baeza-Yates, D. Jiang, F. Silvestri, and B. Harrison, "Predicting The Next App That You Are Going To Use," in *WSDM*, 2015.

[18] N. Natarajan, D. Shin, and I. S. Dhillon, "Which APP will You Use Next? Collaborative Filtering with Interactional Context," in *RecSys*, 2013.

[19] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin, "Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones," in *UbiComp*, 2013.

[20] Y. Wang, X. Liu, D. Chu, and Y. Liu, "EarlyBird: Mobile Prefetching of Social Network Feeds via Content Preference Mining and Usage Pattern Analysis," in *MobiHoc*, 2015.

[21] J. Lee, K. Lee, E. Jeong, J. Jo, and N. B. Shroff, "CAS: Context-Aware Background Application Scheduling in Interactive Mobile Systems," *IEEE Journal on Selected Areas in Communications*, 2017.

[22] K. Zhu, X. He, B. Xiang, L. Zhang, A. Pattavina *et al.*, "How Dangerous Are Your Smartphones? App Usage Recommendation with Privacy Preserving," *MIS*, 2016.

[23] I. Shklovski, S. D. Mainwaring, H. H. Skúladóttir, and H. Borgthorsson, "Leakiness and Creepiness in App Space: Perceptions of Privacy and Mobile App Use," in *SIGCHI*, 2014.

[24] K. Martin and K. Shilton, "Putting Mobile Application Privacy in Context: An Empirical Study of User Privacy Expectations for Mobile Devices," *The Information Society*, 2016.

[25] D. Chu, A. Kansal, and J. Liu, "Fast App Launching for Mobile Devices Using Predictive User Context," in *ACM MobiSys*, 2012.

[26] Google, "Google Pixel 7 Introduction," https://en.wikipedia.org/wiki/Pixel_7, 2022.

[27] Android, "Android 14 Release Notes," https://developer.android.com/about/versions/14, 2023.

[28] C. Newsroom, "Google's Android Becomes the World's Leading Smartphone Platform." https://www.canalys.com/newsroom/google

[29] "The linux kernel archives." https://www.kernel.org/.

[30] Y. Liang, R. Pan, D. Yajuan, C. Fu, L. Shi, T.-W. Kuo, and C. Xue, "Read-Ahead Efficiency on Mobile Devices: Observation, Characterization, and Optimization," *IEEE TC*, 2020.

[31] Y. Liang, R. Pan, T. Ren, Y. Cui, R. Ausavarungnirun, X. Chen, C. Li, T.-W. Kuo, and C. J. Xue, "CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime," in *FAST*, 2022.

[32] D. Liu, K. Zhong, X. Zhu, Y. Li, L. Long, and Z. Shao, "Non-Volatile Memory based Page Swapping for Building High-Performance Mobile Devices," *IEEE TC*, 2017.

[33] X. Zhu, D. Liu, K. Zhong, J. Ren, and T. Li, "SmartSwap: High-Performance and User Experience Friendly Swapping in Mobile Systems," in *DAC*, 2017.

[34] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang, and W. Zheng, "MARS: Mobile Application Relaunching Speed-up through Flash-Aware Page Swapping," *IEEE TC*, 2015.

[35] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.

[36] 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 *EuroSys*, 2023.

[37] X. Zhu, D. Liu, L. Liang, K. Zhong, L. Long, M. Qiu, Z. Shao, and E. H.-M. Sha, "Revisiting Swapping in Mobile Systems with SwapBench," *FGCS*, 2017.

[38] Y. Kim, M. Imani, S. Patil, and T. S. Rosing, "CAUSE: Critical Application Usage-Aware Memory System Using Non-Volatile Memory for Mobile Devices," in *ICCAD*, 2015.

[39] K. Zhong, D. Liu, L. Long, J. Ren, Y. Li, and E. H.-M. Sha, "Building NVRAM-Aware Swapping through Code Migration in Mobile Devices," *TPDS*, 2017.

[40] J. Kim and H. Bahn, "Analysis of Smartphone I/O Characteristics—Toward Efficient Swap in a Smartphone," *IEEE Access*, 2019.

[41] J. Kim and H. Bahn, "Comparison of Hybrid and Hierarchical Swap Architectures in Android by Using NVM," *JSTS*, 2018.

[42] G. F. Oliveira, S. Ghose, J. Gómez-Luna, A. Boroumand, A. Savery, S. Rao, S. Qazi, G. Grignou, R. Thakur, E. Shiu *et al.*, "Extending Memory Capacity in Consumer Devices with Emerging Non-Volatile Memory: An Experimental Study," *arXiv:2111.02325*, 2021.

[43] G. F. Oliveira, S. Ghose, J. Gómez-Luna, A. Boroumand, A. Savery, S. Rao, S. Qazi, G. Grignou, R. Thakur, E. Shiu *et al.*, "Extending Memory Capacity in Consumer Devices with Emerging Non-Volatile Memory: An Experimental Study," *IEEE Access*, 2023.

[44] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash Memory-Based Solid-State Drives," *Proceedings of the IEEE*, 2017.

[45] Y. Luo, Y. Cai, S. Ghose, J. Choi, and O. Mutlu, "WARM: Improving NAND Flash Memory Lifetime with Write-Hotness Aware Retention Management," in *MSST*, 2015.

[46] Android, "Memory Allocation Procedure Among Processes," https://developer.android.com/topic/performance/memory-management, 2023.

[47] Apple, "iOS Memory Deep Dive," 2018, https://developer.apple.com/videos/play/wwdc2018/416//.

[48] Linux Foundation, "In-kernel Memory Compression," 2013, https://lwn.net/Articles/545244/.

[49] Linux Foundation, "3.14 Merge Window Part 3," 2014, https://lwn.net/Articles/583681/.

[50] Linux Foundation, "Least-Recently-Used (LRU) Algorithm in Linux Kernel," https://www.kernel.org/doc/Documentation/vm/zswap.txt, 2021.

[51] N. Gupta, "Zram Read and Write Unit," https://www.kernel.org/doc/Documentation/blockdev/zram.txt, 2024.

[52] "OPPO Smartphones," https://www.oppo.com/cn/smartphones/.

[53] Google, "Google Pixel 5 Introduction," https://en.wikipedia.org/wiki/Pixel_5, 2020.

[54] Samsung Community, "What is ZRAM in Smartphone Galaxy S," https://r2.community.samsung.com/t5/Galaxy-S/What-is-Z-RAM/td-p/10787922.

[55] N. Jakob, "Response Times: The Three Important Limits," https://www.nngroup.com/articles/response-times-3-important-limits/, 1993.

[56] Android, "Perfetto - System Profiling, App Tracing and Trace Analysis," https://perfetto.dev/, 2023.

[57] K. Zhong, D. Liu, L. Liang, X. Zhu, L. Long, Y. Wang, and E. H.-M. Sha, "Energy-Efficient in-Memory Paging for Smartphones," *IEEE TCAD*, 2015.

[58] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A Survey of Performance Optimization for Mobile Applications," *TSE*, 2021.

[59] Y.-C. Chang, W.-M. Chen, P.-C. Hsiu, Y.-Y. Lin, and T.-W. Kuo, "Lsim: Ultra Lightweight Similarity Measurement for Mobile Graphics Applications," in *DAC*, 2019.

[60] D. T. Nguyen, G. Zhou, X. Qi, G. Peng, J. Zhao, T. Nguyen, and D. Le, "Storage-Aware Smartphone Energy Savings," in *UbiComp*, 2013.

[61] W. Yin, M. Xu, Y. Li, and X. Liu, "LLM as a System Service on Mobile Devices," *arXiv preprint arXiv:2403.11805*, 2024.

[62] A. Karapantelakis, P. Alizadeh, A. Alabassi, K. Dey, and A. Nikou, "Generative AI in Mobile Networks: A Survey," *Annals of Telecommunications*, 2024.

[63] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Empowering LLM to Use Smartphone for Intelligent Task Automation," *arXiv preprint arXiv:2308.15272*, 2023.

[64] M. Yavuz, E. Çorbacıoğlu, A. N. Başoğlu, T. U. Daim, and A. Shaygan, "Augmented Reality Technology Adoption: Case of a Mobile Application in Turkey," *Technology in Society*, 2021.

[65] K. Perera, A. Gamage, M. Jawahir, G. Dias, and K. Sandaruwan, "Augmented Reality Supported Self-help Interventions for Psychological and Physiological Acute Stress," in *FTC*, 2021.

[66] S. Criollo-C, D. Abad-Vásquez, M. Martic-Nieto, F. A. Velásquez-G, J.-L. Pérez-Medina, and S. Luján-Mora, "Towards a New Learning Experience through a Mobile Application with Augmented Reality in Engineering Education," *Applied Sciences*, 2021.

[67] H. F. Hanafi, M. H. Abd Wahab, K.-T. Wong, A. Z. Selamat, M. H. M. Adnan, and F. H. Naning, "Mobile Augmented Reality Hand Wash (MARHw): Mobile Application to Guide Community to Ameliorate Handwashing Effectiveness to Oppose COVID-19 Disease," *IJIE*, 2020.

[68] Android, "Power Rails," https://developer.android.com/studio/profile/power-profiler, 2021.

[69] C. Li, L. Shi, Y. Liang, and C. J. Xue, "SEAL: User Experience-Aware Two-Level Swap for Mobile Devices," *IEEE TCAD*, 2020.

[70] K. Lee, ""LZ4 Compression and Improving Boot Time," https://events.static.linuxfound.org/sites/events/files/lcjpcojp13_klee.pdf, 2013.

[71] M. F. Oberhumer, "LZO Real-time Data Compression Library," https://www.oberhumer.com/opensource/lzo/, 2017.

[72] Y. Mao, Y. Cui, T.-W. Kuo, and C. J. Xue, "Trace: A Fast Transformer-Based General-Purpose Lossless Compressor," in *WWW*, 2022.

[73] M. Mahoney, "Large Text Compression Benchmark," https://www.mattmahoney.net/dc/text.html, 2011.

[74] V. Young, S. Kariyappa, and M. K. Qureshi, "CRAM: Efficient Hardware-Based Memory Compression for Bandwidth Enhancement," *arXiv preprint arXiv:1807.07685*, 2018.

[75] D. R. Carvalho and A. Seznec, "Understanding Cache Compression," *TACO*, 2021.

[76] Linux Documents, "Page Frame Reclamation." [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand013.html

[77] Linux Foundation, "Memory Cgroup Documentation," https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt, 2013.

[78] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate Compression: Practical Data Compression for On-Chip Caches," in *PACT*, 2012.

[79] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," in *MICRO*, 2013.

[80] M. Saxena and M. M. Swift, "FlashVM: Virtual Memory Management on Flash," in *USENIX ATC*, 2010.

[81] P. Dempsey, "Reviews-Consumer Technology. The Teardown: Xiaomi MiII smartphone," *Engineering & Technology*, 2021.

[82] A. Al-Shaikh, A. Shaheen, M. R. Al-Mousa, K. Alqawasmi, A. S. Al Sherideh, and H. Khattab, "A Comparative Study on the Performance of 64-bit ARM Processors." *JIMT*, 2023.

[83] H. Seo, P. Sanal, A. Jalali, and R. Azarderakhsh, "Optimized Implementation of SIKE Round 2 on 64-bit ARM Cortex-A Processors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[84] Android, "Monkeyrunner Tool," https://developer.android.com/studio/test/monkeyrunner, 2021.

[85] P. K. D. Pramanik, N. Sinhababu, B. Mukherjee, S. Padmanaban, A. Maity, B. K. Upadhyaya, J. B. Holm-Nielsen, and P. Choudhury, "Power Consumption Analysis, Measurement, Management, and Issues: A State-of-the-art Review of Smartphone Battery and Energy Usage," *IEEE Access*, 2019.

[86] "Ariadne GitHub Repository," https://github.com/CMU-SAFARI/Ariadne, 2025.

[87] Y. Liang, A. Shen, C. J. Xue, R. Pan, H. Mao, N. M. Ghiasi, Q. Jiang, R. Nadig, L. Li, R. Ausavarungnirun, M. Sadrosadati, and O. Mutlu, "Ariadne: A Hotness-Aware and Size-Adaptive Compressed Swap Technique for Fast Application Relaunch and Reduced CPU Usage on Mobile Devices," in *arXiv*, 2025.

[88] PHISON Blog, "Important Differences Between SSDs with and without DRAM," 2024. [Online]. Available: https://phisonblog.com/dram-or-not-the-difference-between-dram-and-dram-less-ssds-and-why-it-matters/

[89] S.-H. Kim, J. Jeong, and J.-S. Kim, "Application-Aware Swapping for Mobile Systems," *TECS*, 2017.

[90] "What is RAM Plus and How to Use It?" 2022, https://www.samsung.com/sg/support/mobile-devices/what-is-ram-plus-and-how-to-use-it/.

[91] "How to Use Xiaomi Virtual RAM to Speed Up Your Device?" 2022, https://xiaomiui.net/how-to-use-xiaomi-virtual-ram-to-speed-up-your-device-31416/.

[92] S.Jennings, "ZSWAP is a Lightweight Compressed Cache for Swap Pages," https://www.kernel.org/doc/Documentation/vm/zswap.txt.

[93] J. Han, S. Kim, S. Lee, J. Lee, and S. J. Kim, "A Hybrid Swapping Scheme Based On Per-Process Reclaim for Performance Improvement of Android Smartphones," *IEEE Access*, 2018.

[94] K. Zhong, T. Wang, X. Zhu, L. Long, D. Liu, W. Liu, Z. Shao, and E. H.-M. Sha, "Building High-Performance Smartphones Via Non-Volatile Memory: The Swap Approach," in *EMSOFT*, 2014.