Archer: Adaptive Memory Compression with Page-Association-Rule Awareness for High-Speed Response of Mobile Devices

Changlong Li[1,2,3], *Zongwei Zhu*[4,5]*, *Chao Wang*[4,5], *Fangming Liu*[6,7], *Fei Xu*[1], *Edwin H.-M. Sha*[1], *Xuehai Zhou*[4,5]
[1]*School of Computer Science and Technology, East China Normal University*
[2]*Jianghuai Advance Technology Center, Hefei 230026, China*
[3]*MoE Engineering Research Center of Hardware/Software Co-Design Technology and Application*
[4]*School of Software Engineering, University of Science and Technology of China, Hefei 230026, China*
[5]*Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou 215123, China*
[6]*Huazhong University of Science and Technology* [7]*Peng Cheng Laboratory*

Abstract

In mobile systems, memory can be compressed page-by-page to save space. This approach is widely adopted because memory data is accessed by page. However, this paper shows that the system response speed is significantly limited by page-grained compression. In this paper, we observe that approximately a quarter of anonymous memory pages are highly correlated, even though the association is implicit. Inspired by this, we propose Archer, an association-rule-aware memory compression framework in mobile systems. Archer demonstrates that memory in mobile devices should be compressed using flexible granularity, rather than relying solely on traditional page compression. To further integrate association-rule mining techniques into system design, we redesign the LRU mechanism and propose an adaptive memory compression region. Experimental results show that the average app launching speed is 1.55x faster when enabling Archer, and the average photographic speed and frame rate increase by 1.42x and 1.31x, respectively, compared to the state-of-the-art.

1 Introduction

With mobile applications' increasing demands for memory and a steady increase in the number of cached apps, memory resources are scarce on mobile devices. Such a trend will be more obvious as many memory-intensive tasks, like AI [1], AR/VR [2], and Transformer [15] are implemented. To address this issue, memory compression techniques are adopted in mobile systems (e.g., Android, iOS) [3–7, 9]. By compressing the least essential pages, the space can be saved for new memory demands.

There have been many studies on optimizing memory compression by improving the algorithm [7, 49, 50], minimizing its effect [5, 41], or making use of app characteristics [4, 6, 9]. For example, Liang et al. proposed a foreground-aware page reclaim scheme called Acclaim [5]. With Acclaim, pages belonging to the foreground application are denied to compress. ASAP [6] and SEAL [9] explored the characteristics of app launching required pages and boosted the launching speed by optimizing the memory (de)compression process. Existing solutions compress memory data at the page level, which is reasonable because memory is managed and accessed by *pages*. Page-granularity compression, as opposed to larger granularities, helps avoid read amplification. However, this paper reveals that such an approach wastes CPU bandwidth and introduces significant context-switching overhead. Our study indicates that the response latency can increase by up to 2.31x when suffering frequent compression.

This paper analyzes the page footprint of popular applications and finds a large number of highly correlated pages. These pages are either accessed together or rarely accessed. Such internal associations are implicit and difficult to be aware of, just like the relationship between diapers and beer in markets. If we can mine the association rule of memory pages and compress the highly correlated pages together, the system performance has the potential to be further improved with minimized read amplification.

Inspired by this observation, this paper proposes Archer, an **A**ssociation-**r**ule aware memory **c**ompression framework for the **h**igh sp**e**ed **r**esponse of mobile devices. Implementing such a system is not straightforward, as several fundamental challenges must be addressed. First, traditional association-rule mining techniques [38–40] cannot be directly ported to mobile systems, as there are too many dynamically changing memory pages and associations. Mobile operating systems use LRU-based page management [12]. While effective for general workloads, LRU is inefficient for data mining and poorly suited to identifying page associations. To leverage association-rule mining in system design, the underlying data management structure must be reengineered in a data-mining-friendly manner. Second, the number of associated pages varies. For instance, page *A* might be highly correlated with

*Corresponding authors: Zongwei Zhu, Email: zzw1988@ustc.edu.cn.
This work was supported in part by the National Key R&D Program of China (2024YFB4504400), Dreams Foundation of Jianghuai Advance Technology Center (No. 2023-ZM01Z011), NSFC under Grant No. 62302169 and No. 62372184, Major Key Project of PCL under Grant PCL2024A06 and PCL2022A05, and Shenzhen Science and Technology Program under Grant RCJC20231211085918010.

three other pages, while page *B* might be linked to five. Additionally, large-grain compression is not always ideal. For example, if an app requires 4KB of memory, but the system compresses 64 pages (each 4KB) together, rather than compressing a single page, it will increase the response time. Therefore, compression granularity should be adaptable based on page associations and memory demand. Traditional compression mechanisms, designed with fixed granularity, cannot accommodate this. To enable adaptive compression, the address space and compression region need to be redesigned. Third, given that mobile devices have limited resources, the proposed solution must be lightweight and not interfere with the performance of foreground applications.

To tackle the challenges, this paper further proposes three components: a footprint stream generator (***FSG***), a frequent-pattern tree list (***FT-List***) structure, and an adaptive compression region (***ACR***). The FT-List manages pages with app awareness. The combination of FSG and FT-List simplifies the indexing and management of associated pages. Using ACR, memory data in mobile systems can be compressed at any granularity. We have implemented Archer on real-world devices. Our evaluation shows that enabling Archer improves app launch speed by 1.55x. Additionally, photographic speed and frame rate increase by 1.42x and 1.31x, respectively, compared to the state-of-the-art solutions [3] [6]. More importantly, the tail latency is significantly reduced.

In summary, this paper makes the following contributions:

- We observe that there are internal associations among anonymous memory pages, a factor overlooked in previous operation system designs.
- This is the first work that proposes the possibility of large-grain compression in mobile systems. By leveraging internal page associations, memory compression can overcome the current performance ceiling without the need for hardware infrastructure changes.
- This paper demonstrates that LRU is inadequate for data mining on memory pages and introduces a novel structure, the FT-List. Additionally, we propose ACR to manage adaptively compressed data.
- Archer is app-agnostic without requiring any change to applications' codes, and compatible with existing mechanisms. Experimental results show a significant alleviation (up to 55%) of the poor response issue when memory is exhausted, compared to the state-of-the-art.

2 Background and Motivation

2.1 Memory Management of Mobile OS

Reclaim File-backed and Anonymous Pages. As in typical Linux-like operation systems (OS), memory-demanding requests of an application in mobile systems are delivered to the kernel through page fault. The kernel tries to assign 2^{order} pages from *free_list* [10]. In the buddy system, each $free\_list$ has an order, where the order is an integer ranging from 0 to 10. When enough pages are allocated, the application continues execution. Acquiring free pages is mostly done quickly. If no sufficient free pages are available, it takes a rather long time to create free pages by memory reclamation.

Two kinds of pages can be reclaimed: file-backed and anonymous pages. File-backed pages are indexed by file systems and anonymous pages are generated by processes at runtime [4]. These pages are reclaimed through compression. When enough space is saved, the buddy system reallocates pages and responds to the application. If the memory is not reclaimed on time, a low memory killer daemon (*lmkd*) will kill several background applications so that the occupied memory space can be released [11]. It results in the loss of the app state and slows down the app's launching speed [4].

Process of Memory Compression. Two LRU (least-recently-used [12]) lists are maintained to manage the anonymous pages: *active* and *inactive* list. When performing compression, pages on the inactive LRU list are selected as candidates. They are evicted from the list and delivered to the compression driver. The compression algorithm is registered as a driver in the Linux kernel and will be called to compress the pages one by one. The compressed pages are stored in the compression region, which is usually organized as a virtual RAM disk [3].

2.2 Limitations of Page Compression

Mobile systems perform compression in page granularity. This does not block the regular app execution when their demand for memory is not intensive. However, modern apps tend to demand a large amount of memory in a short time.

2.2.1 Workloads for Analysis

To understand how page-grained compression affects the user experience, we conducted experiments on a HUAWEI P20 smartphone with three typical scenarios.

- **Scenario A: App launching.** Fast app launching is crucial for the user experience on mobile devices. As investigated, people launch apps hundreds of times per day [13, 16]. To start up an app, the system does a lot of operations to make the launch activity visible to a user. Generally, decades of megabytes of memory space are required to launch an application.
- **Scenario B: Continuous shooting.** Users are increasingly engaging with the physical world through a digital lens, spending a large portion of their time with the integrated camera on phones. Modern smartphones consume a lot of memory to take pictures, especially when the camera is in continuous shooting mode, which allows

multiple photographs to be taken within a short time frame and in rapid succession.

- **Scenario C: Short-form video.** People nowadays are spending more and more time on short-form videos [8, 17]. Users watch these videos serially (e.g., based on a search or user-specific recommendations), with the ability to swipe from one to the next at any time. During the process, video data is buffered in the memory through the network [18], consuming a lot of space.

This paper describes the response time of the above scenarios with different metrics and methods. First, we adopt Adb (Android debug bridge [19]) to launch apps and record their launching time. Second, the speed of continuous shooting is quantified by calculating how many photos are taken per second. We deploy an additional device to record the photographic process via video. Then analyzes the video and calculates the performance using StagesepX [20]. Third, the frame rate is utilized to quantify the smooth experience of short-form video playing and switching. It depicts whether a user can smoothly interact with the screen. We employ Systrace [21] to monitor the FPS (frames-per-second).

2.2.2 Response Time Analysis

The response time of each scenario is evaluated in three situations: abundant, regular, and scarce. "Abundant" refers to the case where no app is cached in the background. The memory is enough to afford the tested application. "Regular" refers to the case when there is some memory used by background apps but enough free pages. In this case, the memory compression procedure is triggered but not frequently. "Scarce" is the case where background apps are almost exhausting the memory resources. Specifically, twenty apps run in advance and switch to the background, consuming more than 90% of the space. The evaluation method is detailed in Section 5.2.1.

Figure 1(a) shows the launching latency of mobile apps. Each app is tested for ten rounds and the average is taken. The tested apps are picked from the most popular ones of various categories on the Google application market, including Facebook, PUBG Mobile, Amazon, Twitter, YouTube, Chrome, WeChat, and Google Map. The average latency increased by 68.7% when the memory was scarce, compared to the regular usage case.

Figure 1(b) depicts the process when the camera takes photos continuously. The Y-axis represents the timeline. Each circular icon in the figure represents a photo taken at that moment. It illustrates that 16 photos can be taken within one second when the memory is abundant, while only 10 photos are taken in the Scarce case. The response time of the continuous shooting is prolonged by 1.6x when the memory is under pressure. In addition, Figure 1(c) shows that the frame rate of short-form video playing and swiping reduced from around 50fps to 27fps (see the grey curve) on average. It demonstrates

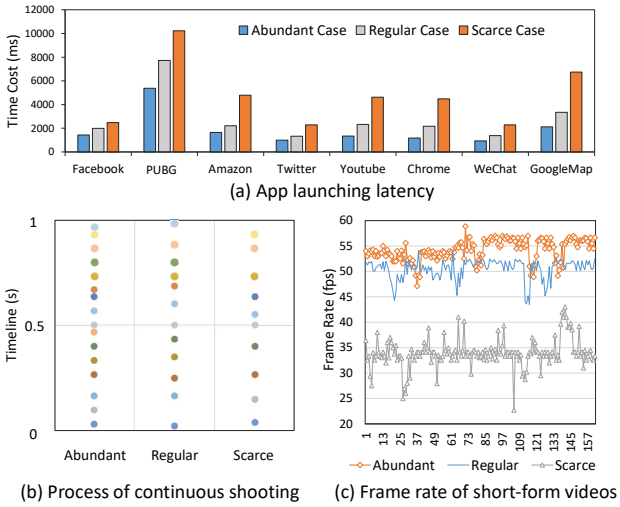


Figure 1: Response time of the mobile applications significantly increased when the memory is under pressure.

the response time increases when bursts demand a large space for an exhausted memory. Such a significant slowdown can potentially affect the user experience negatively.

2.2.3 Root Causes of User-perceived Delay

The degraded response speed could be caused by CPU or I/O issues. This paper examines both possibilities.

We modified the kernel into two versions: fileRec and anonRec, and compared to the original kernel (bothRec). FileRec refers to a kernel version that only allows file-backed page reclamation, say, no memory compression. Contrary, compression is allowed under anonRec, while reclamation toward file-backed pages is discarded. In this case, memory reclaim will not induce additional I/Os. In the bothRec version, both anonymous and file-backed pages are allowed to be reclaimed. These kernels are deployed on a P20 phone and measured separately. For each kernel version, we cold-launch the application and record the time cost in two cases: (i) launch the app when the available memory is enough; (ii) do that when the memory is almost exhausted.

Figure 2(a) shows that the launching speed is close between bothRec, fileRec, and anonRec when the available memory is enough. This is because the reclaim operations are not performed in this case. In case (ii), the average launching speed increased by 72.1% under bothRec. It illustrates that memory reclaim harms the system response speed[1]. From the evaluation results of fileRec and anonRec, we can see the latency is increased by 26.2% when enabling file-page reclaim, while increased by 58.7% when enabling memory compression. I/O is a main bottleneck in many situations and many solutions were proposed to improve it [16, 26].

[1]Memory reclaim is a necessity to cache more applications in the background. Hence, even though the introduced performance penalty, modern mobile OSes still widely support this feature.

However, the results illustrate that memory compression can also significantly affect the response speed. Its effect is high for two reasons. First, we observe that more than 93% of file-backed pages have not been modified. The OS releases them directly, without writing back. Instead, all anonymous pages are dirty (modified). All anonymous pages should be compressed in practice. Second, the ratio of anonymous pages keeps increasing during usage. As investigated, the average ratio of anonymous and file-backed pages increased from around 1:1 to 3:1 during usage [27].

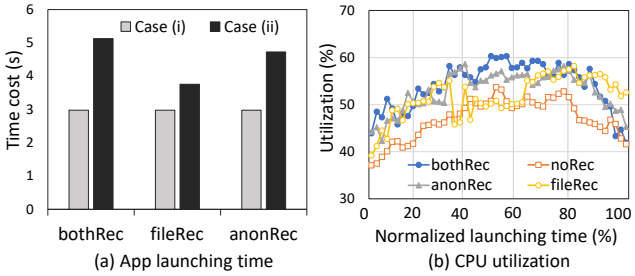


Figure 2: Root cause analysis.

A slow compression procedure can severely degrade the performance of the tested applications. One of the key reasons behind this performance degradation is the synchronous reclamation mechanism, called *direct reclaim*. Direct reclaim harms the performance because any page allocations must wait for the compression process to finish. As a result, the main task of the foreground app is suspended until its memory demand is met. This problem is known as priority inversion [28]. Even though the kernel thread *kswapd* can compress asynchronously, the buddy system may still get stuck in a loop before enough space is released.

2.3 Potential Benefit Discussion

Figure 2(b) depicts the CPU utilization during app launching. To easily show the relationship, we normalize the launching times and sort them from short to long. Based on that, CPU utilizations are compared from [0th, 20th]-percentile to [80, 100th]-percentile of the sorted launching time.

As evaluated, the average CPU utilization without memory compression is 46.9% (NoRec in case-i) and 51.6% (fileRec in case-ii). This value remains relatively low (i.e., 53.8% on average in the anonRec case) when memory compression is enabled during app launching. It demonstrates that the resources are not fully utilized. This is because the default compression mechanism deals with memory pages one by one, leading to a result that compression operations are frequently interrupted by the software stack and context switch. For example, the system needs to evict the pages going to compress from the LRU list and balance the *active* and *inactive* LRU lists. By tracing the corresponding kernel functions like `shrink_active_list()` in mm/vmscan.c, we observe that page compression operations are frequently mixed with the above software stack functions. These problems can be alleviated if compressing the data together.

3 Opportunities of Large-grain Compression

It is reasonable to compress pages individually. By doing so, read amplification is eliminated. Here, read amplification refers to the ratio of 'the total size of the decompressed data' to 'the size of the accessed data'.

Different from traditional computers, we observe that accessed pages on mobile devices have high associations, as the operation system and user behaviors difference. It allows us to conduct larger-grain compression and avoid read amplification simultaneously. To analyze the correlation, we collected the page access sequence and obtained each page's virtual address. The page access event can be detected by the original Linux kernel interface [30]. Then we convert virtual addresses to physical addresses using function `virt_to_phys()`. The physical address is adopted as page identification. The collected footprint is delivered to a laptop and analyzed offline.

This paper makes use of the association-rule mining technique to explore the internal relationship of the accessed pages. This technique was conceived as an unsupervised learning task for finding close relationships between items. The meaning of an association rule is: if antecedent X is satisfied, then it is highly likely that consequent Y will also be satisfied [31–33]. To explore the relation of memory pages, we explore the association *confidence* of all page combinations. The value of Confidence ($X \rightarrow Y$) represents the probability of accessing page Y when page X was accessed: P(Y|X)=P(XY)/P(X). Two pages are considered as highly associated if they are always accessed within N footprints. N is set as 32 in this evaluation. We calculate the *confidence* of all page combinations using an association-rule mining algorithm (Apriori with Python [34]).

Then we sort the page combinations according to their association *confidence* value. Based on that, the page combinations are compared from [0th, 10th)-percentile to [90th, 100th]-percentile. The higher this value is, the more correlated the pages are. Among all the combinations, 26.3% have a high possibility (>80%) to be accessed together. In some scenarios, the ratio is up to 34.1%. In addition, the number of page combinations in [0, 20) is 71.1% on average, while rarely combinations are located in the [20,80) range. This indicates that the correlation between the accessed pages is polarized: either having no relationship or being accessed simultaneously almost all the time.

This is because a large number of pages are only accessed in specific scenarios. For example, we observe that 36.2% of the anonymous pages are only demanded in the app launching phase on average. When running the PUBG Mobile, more than 58MB (59,392 pages) are only accessed when entering the Battle mode. These pages are not used except by the UNREAL engine [35] of the mobile game. Furthermore, many

applications call the camera component. To share the data between the CPU and the camera accelerator, like DSP [36], the camera demands a DMA buffer for ION memory (using `ion_alloc()`). We find 92.1% anonymous pages belonging to the camera have never been accessed except when calling the camera. Among them, 74.3% of pages are accessed more than 8 out of 10 times when the camera starts up.

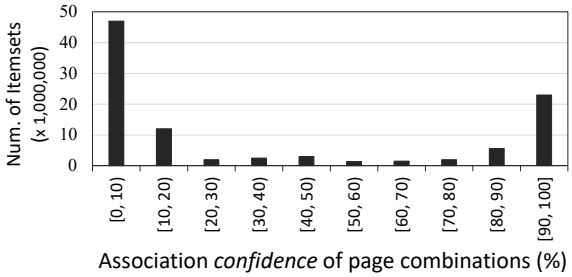


Figure 3: Association of accessed anonymous pages.

In normal cases, page-grained compression is enough to save memory space. When suffering burst memory demanding, like the scenarios discussed in Section 2.2.1, compressing the associated pages with higher granularity becomes important. As shown in Figure 3, more than a quarter of pages have high association. If we can identify these pages and make full use of them, the system response speed can be significantly improved in burst memory-demanding scenarios, with minimized read amplification. For clarity, we simplify the experiment and only show the correlation between page combinations. Note that associations do not just occur between two pages in practice. For example, eight or even more highly inter-associated pages are also observed in our evaluation.

4 Design

4.1 Archer Overview

This paper proposes Archer, an association-rule aware memory compression framework for mobile devices. The basic idea is to mine the internal association rules of memory pages and compress those that are highly correlated together. The design consists of three core components: a footprint stream generator (FSG), a frequent-pattern tree list (FT-List), and an adaptive compression region (ACR).

FSG is responsible for mining association rules based on the page access footprint, generating footprint streams, and sending them to the FT-List. FT-List differs from traditional LRU structures by efficiently depicting and indexing page associations. Together with FSG, it enables the compression of highly correlated pages. ACR allows flexible page compression and storage based on the discovered associations.

The workflow is as follows: (❶) During normal device usage, the footprint of accessed pages is collected by the FSG component. (❷) FSG encapsulates adjacent accessed pages into transactions using a sliding window mechanism.

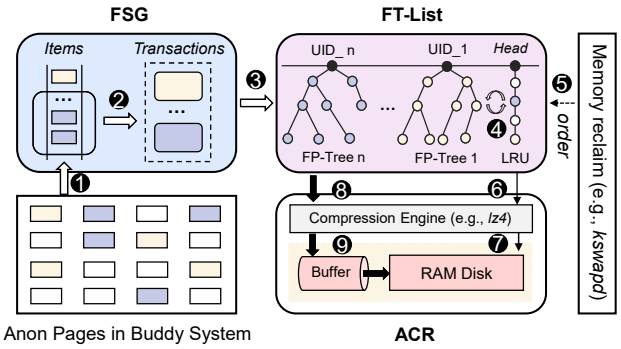


Figure 4: Archer Framework Architecture.

(❸) These transactions are sent to an association-rule mining model that identifies highly correlated pages. This model is integrated into the FT-List component. (❹) In FT-List, pages are appended on either a LRU list or frequent-pattern trees (FP-tree). Page associations can be updated during runtime. (❺) When performing memory reclaim, pages maintained by FT-List will be checked. (❻) If the memory demand is not urgent, Archer compresses pages on the LRU list one by one (narrow track). (❼) Compressed pages are then stored in the compression region. (❽) If memory demand becomes urgent, the compression switches to the wide track, where associated pages are compressed together. (❾) These compressed objects are stored in the compression region.

4.2 Footprint Stream Generator

In the traditional association-rule mining algorithms (e.g., Aporio [38], FP-Growth [39], CHARM [40]), a frequent pattern set will be built to contain all the elements whose frequency is greater than or equal to the minimum *support*[2] count. Following this rule, pages that are only accessed one time will be ignored. Such an approach is useful in mining a static transaction data set. However, in mobile systems, page access occurs continuously during runtime. Archer should be able to update the page association information as pages are accessed. Motivated by this, Archer collects page access patterns during runtime and generates a footprint stream. Now we introduce how the stream is generated.

In the original Linux kernel, page access events are monitored for page activity updating. Based on the existing interface, Archer can aware of page access without introducing additional computing overhead. We adopt the physical page address instead of the virtual address to identify pages because the virtual addresses of many pages belonging to different applications are duplicated.

Archer maintains a FIFO (first-in-first-out) circular queue to record the page access history. When an anonymous page is accessed, Archer obtains its physical address, which is treated as an item, and inserts it into the queue. The in-flight

[2]Support is a parameter of association-rule mining algorithms. It refers to the relative frequency of an item set in a dataset.

items constitute a data stream. Since page access is one of the most frequent operations in the system, aggressive data mining will negatively affect the system's performance. Hence, Archer processes the stream in a semi-offline way. It collects the items in real time but is allowed to delay mining their association, asynchronously.

A daemon is added as a separate control unit inside the kernel. It transfers the sequence items to transactions. We generate transactions using a fixed-size sliding window. For a window of width w, transactions are generated in three cases: (1) When X ($X \geq w$) items are in the queue, Archer caches the items as a transaction and evicts them from the queue. Then the head pointer of the queue moves w steps and the window slides to the new head position. (2) If $\lceil \frac{w}{2} \rceil \leq X < w$, the window will suspend for ΔT time long. When timeout, the X items are evicted from the queue and the sliding window moves X steps. (3) If $X < \lceil \frac{w}{2} \rceil$, the window suspends until one of the above two conditions is satisfied.

Example. Taking Figure 5 as an example, pages with the following addresses: `0x3E`, `0x29`, `0x313`, `0x32`, `0x10`, `0x2F`, `0x45`, `0x3E`, `0x29`, `0x32`, `0x10`, `0x2F`, `0x11`, `0x3E`, `0x29`, `0x13`, `0x28`, `0x29`, `0x13`, `0x37`, `0x2F`, `0x28`, `0x3E`, `0x51`, `0x29`, `0x10`, `0x10`, were accessed in that order. In this example, the window width w and queue length Lq are set as 6 and 10, with t_0 being the earliest and t_{26} being the latest access. At t_i, the head pointer of the queue stays at the t_i-th slot. Following the aforementioned strategy, Archer initially detects the number of items in the sliding window. At time t_5, 6 pages are inserted into the queue within ΔT, that is, $t_5 - t_0 < \Delta T$, the window evicts the 6 pages and delivers them to a transaction buffer. Then, the sliding window moves 6 steps and detects the coming items again. At time t_{11}, Archer generates a new transaction. After that, four new items come before time out. At time $t_{11} + \Delta T$, Archer converts the four items `0x11`, `0x3E`, `0x29`, and `0x13` to a transaction. Similarly, Archer generates two transactions at time $t_{15} + \Delta T$ and t_{26} separately.

People may use the phone busily in some periods while putting the phone down at other periods [37]. Page accessing is relatively frequent when the human-machine interaction is busy. As a result, page accessing induced newly generated items can quickly fill the queue. It means the coming items cannot be processed on time. In this case, Archer discards the earliest dispatched item in the queue without processing it. We believe it is worth ignoring some item processing when human-machine interaction in the foreground is busy.

The generated transactions constitute a footprint stream and are delivered to FT-List for further processing.

4.3 Frequent-pattern Tree List

4.3.1 Mining the Footprint Stream

FP Growth is a highly efficient method for association rule mining. By maintaining an FP-tree, page associations are

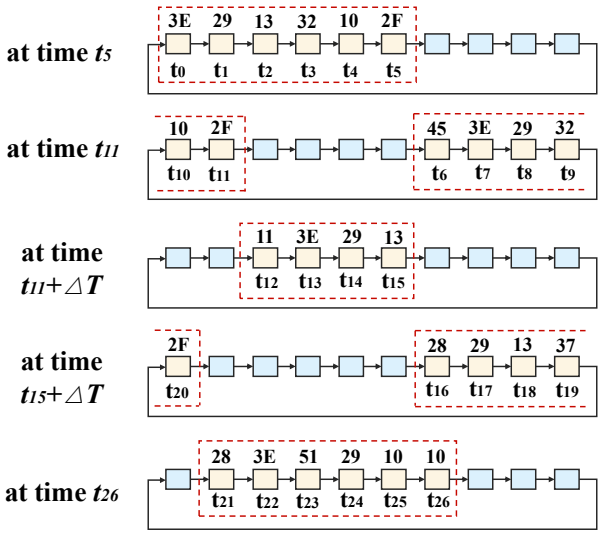


Figure 5: Example of transaction generation.

clearly represented. The tree's branches are built in descending order of frequency, with each node, except the root, representing a memory page. Pages are sorted based on the frequency of their occurrence, and the count is updated each time a page appears in a transaction. Each transaction is mapped to a path in the tree, with overlapping paths from different transactions keeping the tree compact. In summary, associated pages can easily be identified by traversing the FP-tree. This paper does not focus on optimizing the data mining algorithm, the algorithm detail can be seen at [22]. With the variant of the FP Growth [23], page associations can be updated based on the footprint stream.

Archer maintains a heartbeat with a cycle of three seconds by default to monitor the buffer and periodically mine the transactions inside. To avoid resource contention, we suspend the mining operation if the system is detected to be too busy. Specifically, in our implementation, we consider the system to be busy when CPU utilization exceeds 80%.

Note that mining continuously will not induce high overhead. The number of accessed pages will grow rapidly if the device is used frequently. In such cases, overflowed mining transactions are discarded, ensuring that Archer's mining cycles remain within an upbound. Conversely, if the device is seldom used, Archer's energy consumption is insignificant, as page access frequency is lower. When there are no sufficient transactions in the queue, Archer will defer triggering the mining task until after several heartbeats.

Despite the maturity of association-rule mining techniques, they cannot be directly applied to system management. The high number of memory pages makes global-scale mining costly and inefficient. Moreover, appending all pages to a single FP-tree would result in slow traversal and updates. Additionally, in the kernel, pages are indexed by LRU lists, which makes scanning and evicting highly correlated pages from the list inefficient. To address these challenges, this

paper co-designs LRU with FP-tree and proposes FT-List.

4.3.2 Codesign LRU with FP-Tree

The frequent-pattern tree list (FT-List) is motivated by a key observation: most highly correlated memory pages belong to the same application. In the preliminary study in Section 3, two sets of statistics were compared. First, we group pages and mine the associations within each group. The total number of associated pages was then calculated. Second, the total number of associated pages was counted without considering which application they belonged to. The results revealed that 93.1% of associated pages were from the same application. Thus, Archer focuses on app-aware association mining and management, rather than analyzing the entire memory scope.

FT-List Structure. As illustrated in Figure 4, the main branch of FT-List consists of several nodes (in dark color). The first node is a pointer to the inactive anonymous LRU list, while the remaining nodes serve as roots of FP-trees. Unlike traditional FP-trees where the root node is represented as *null*, in FT-List, the root node is used to distinguish applications by recording their unique IDs (UIDs). In Android, every app has a unique UID that remains fixed once the app is installed.

All associated pages belonging to a specific app are appended to the FP-tree under the corresponding UID. As the system is used, a footprint stream is generated, and the trees in FT-List are updated accordingly. The UIDs on the main branch of FT-List are sorted based on the activity of their respective apps. For simplicity, the Android priority mechanism (Adj [24]) is used to determine activity. The more inactive an app is, the closer its node is to the head of the FT-List.

Adaptive Compression with FT-List. During regular usage, Archer enters the code path of the original page reclaim. The system directly evicts the inactive page from the list and compresses it. When suffering burst memory demands, Archer selects the associated pages through scanning FT-List from head to tail. Then compressing them together. More than a quarter of pages in the memory can be processed in batch. It is enough to cope with sudden memory demanding.

In practice, the function `_alloc_pages_nodemask()` is called during memory reclaim. The parameter *order* is passed to this function, determining how many pages (2^{order}) need to be reclaimed in one round. If fewer pages are found, the next FP-tree will be checked. Note that not all pages in an FP-tree are pairwise associated; the tree uses a grouping mechanism to ensure that only pages with strong associations are compressed together. All the data mining-related structures are maintained in the memory, so the process can be done quickly.

4.4 ACR: Adaptive Compression Region

To compress pages together, all candidate pages are moved to the swap cache in batch. Archer merges them in the swap cache into a block, then delivers the block I/O (BIO) to the adaptive compression engine. BIO represents an in-flight block I/O request in the kernel [48]. Archer compresses the large-size BIO by calling compression algorithms. The compression engine parses this structure and obtains the pages. Since the compression algorithm requires contiguous memory as the input of compression in the implementation, Archer copies the physical pages to a buffer. After compression, the binary-format contents of the pages in the buffer are converted to a smaller-sized block. Since the number of pages in the buffer varies, the size of the compressed data is not fixed. Archer stores the compressed block to a compression region and manages the address dynamically.

The compression region should be redesigned to adapt to the new situation. Modern adaptive compression solutions [49, 50] are either not compatible with the existing page compression mechanism or bring too much overhead, as they ignore the redesigning of the compression region and software stack for memory reclamation. This paper proposes ACR to address this issue. In modern mobile systems, ZRAM [3] is the most widely adopted page compression solution. To simplify our design, we implement ACR based on the existing address indexing method and structure of ZRAM. Now we introduce how the compression region is managed in page granularity, then show our modification on it.

In the original system, compressed data are stored in a main memory region. Archer manages this compression region as a virtual block device (RAM disk). ZRAM manages the compression region based on the buddy system, it allocates memory space using the `__alloc_page()` interface. These pages are used to store the compressed objects. For easy distinction, pages in the compression regions are called *slots*. Since the data in main memory is compressed by page, the size of one page after compression is smaller than one slot (4KB). So multiple compressed pages can be stored in the same slot. ZRAM manages the content of each compressed page as an object and indexes it by a handle. The handles are indexed by the page table entry (PTE). In this way, the compressed page can be found when a page fault occurs.

We still use the *object* structure to manage the data when compressing the memory data with a bigger size. However, the compression block when enabling Archer is usually bigger than the 4KB slot. Archer divides the big block into multiple objects and stores them separately. Objects are also indexed by handles, so the content of one block is indexed by multiple handles. If pages are flagged, which means they are compressed in huge size, the pages' PTE will point to the same handle. The handle points to a metadata structure, instead of the corresponding object. The metadata consists of two elements: an array to store the page table information of all pages belonging to the corresponding block; and a pointer that helps indexing all handles of this block. Figure 6 shows that page 1 and page 2 are compressed in page granularity (narrow track), while page 3 to page 6 are compressed together (wide track). The compressed content of the later four

pages is stored as object 3 and object 4, which is indexed by two handles. For ease of differentiation, the second-layer handle is named *vhandle*.

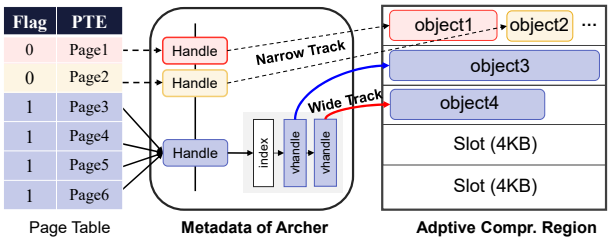


Figure 6: Address management of the compression region.

When accessing the compressed page in Archer region, the page fault is triggered. We modified the interrupt handling function of page fault so that it additionally checks the page flag. If the required page was compressed individually, the decompression method is the same as ZRAM. Otherwise, Archer decompresses the block that contains the demanded page. Based on the handle list (*vhandles* in the figure), all objects of the block can be found and decompressed. Since PTE information of all compressed pages is recorded as metadata, Archer knows which page each segment of the decompressed block belongs to. After decompression, Archer feeds back the page fault. The other decompressed pages are maintained in the swap cache.

This approach has two advantages. First, the highly correlated pages decompressed together, which helps enhance the CPU bandwidth when performing decompression. Second, the other decompressed pages are maintained in the swap cache instead of discarded, these prefetched pages will be accessed at high speed.

5 Evaluation

5.1 Evaluation Setup

This paper evaluates Archer against the original Linux memory compression scheme and the state-of-the-art solutions in mobile systems. The experiments are performed on Google Pixel6 Pro, Pixel3, and HUAWEI P20. The Pixel6 Pro is equipped with Google Tensor cores, 12GB DDR4 RAM, and 128GB UFS Flash, running Android 13.0. The Pixel3 is equipped with Qualcomm Snapdragon 845 core and 4GB memory. Android 10.0(r41) runs on it. The P20 is equipped with HiSilicon Kirin970 core and 6GB memory. Android 9.0 is deployed on the device.

The parameters in the evaluation are set as follows: The compression region sizes of Pixel6, Pixel3, and P20 are set as 2048MB, 512MB, and 1024MB. They determine how many pages are allowed to compress at maximum. The watermark threshold of the three is 1024, 512, and 512, respectively. The min- and high-watermark are 0.8x and 1.2x of the low-watermark, which follows the default configuration in the Linux kernel. The large-grain compression is woken up when the parameter *order* in one round reclamation is detected larger than 8. Regarding the association-rule mining model, the queue length (L_q) and ring buffer size are set to 256 and 128KB, respectively. For the FP-tree of FT-List, the confidence and support count are set to 70% and 0.2. The width w and the timeout threshold ΔT of the sliding window are set to 32 and 3s by default. In addition, we set the compression buffer as 128KB in the evaluation. We use *lz4* [53] to compress the data. This algorithm is widely adopted by commercial smartphones as the default for its efficiency in (de)compression[3].

5.2 Benefit on User Experience

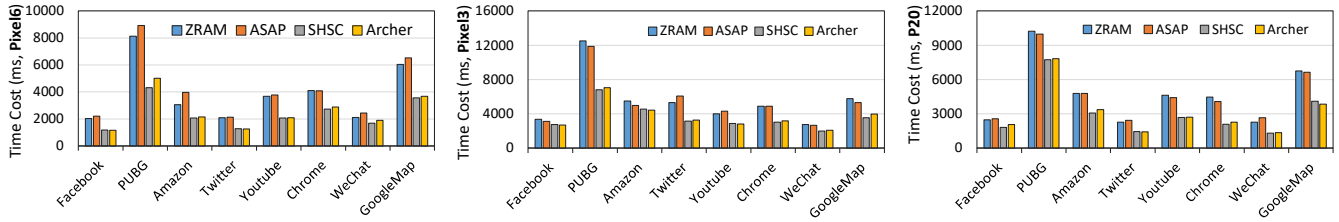
Now we explore Archer's overall effect on the user experience in three typical scenarios: app launching, continuous shooting, and short-form videos. Four schemes are evaluated for comparison: ZRAM [3], ASAP [6], Static Huge-Size Compression (SHSC) [54], and the proposed Archer. ZRAM is the default page compression scheme in modern mobile systems. In addition to ZRAM, this paper compares ASAP. Archer and ASAP improve the performance focuses on the compression and decompression phases, respectively. In addition, the huge page mechanism (SHSC) can also realize huge-grain compression. The impact of huge pages on the system is comprehensive and complicated. To make a fair comparison, we realize fix-sized (de)compression but retain the 4KB management mechanism of other modules.

5.2.1 App Launching Speed

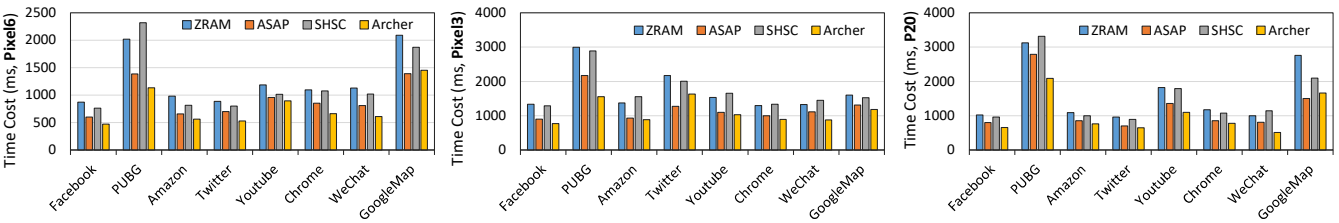
The effect of Archer on app launching is complicated since there are two launching styles. When first launching an app, the system needs to recreate all of the app's activities. On the contrary, when a user starts an app that was moved to the background recently, it is more likely that the life cycle of this app has not ended. The former is called *cold launch* and the latter is called *hot launch*. We evaluate the two launching styles separately. All the evaluations are performed with a good Wi-Fi connection and the battery is fully charged.

Before the evaluation, we ran 20 workload apps in advance (see Table 1). We launched an app and used it for 30 seconds. Then switch it to the background and start the next one. Normal operations like scrolling and clicking are included in the usage. During the process, memory was quickly filled, and page reclaim was triggered frequently. Note that some apps in the background may be killed by Android LMK (low memory killer [11]). This is close to real usage scenarios. After this, more than 87% of memory space is occupied.

[3]In addition to *lz4*, we also considered other compression algorithms commonly used in mobile systems, such as *lzo*, Snappy, QuickLZ, and zlib. Our preliminary evaluation indicates that their (de)compression throughput is lower than that of *lz4*.



(a). Time cost of cold launching on Pixel6, Pixel3, and P20 phones



(b). Time cost of hot launching on Pixel6, Pixel3, and P20 phones

Figure 7: App launching speed comparison.

Table 1: List of workload applications.

Category	Workload Application
Social Network	Skype, Zoom, WhatsApp, LinkedIn, Google+
Multimedia	MXPlayer, Netflix, Snapchat, Tencent Video
Mobile Game	Angry Birds, Asphalt 8, Fruit Ninja, Arena of Valor
Life	Alipay, BOA, eBay, Taobao, Uber, Lift, Evernote

We cold-launch the eight tested apps and record the launching speed. Each app is evaluated for ten rounds and the average is taken. Figure 7(a) shows that the average speed of cold launching with Archer increased by 37.2% on Pixel6, 30.6% on P20, and 32.9% on Pixel3, compared to ZRAM. With Archer, the memory is more efficiently allocated to satisfy the memory demand of app launching.

Here, Pixel6 shows more significant performance improvements than devices with smaller memory capacities. This is because applications on resource-rich smartphones tend to request more memory compared to those on low-end devices[4]. For example, we observe PUBG Mobile dynamically adjusts its graphics and textures based on the device's capabilities. This observation leads us to conclude that the burst memory demand problem will not be mitigated by increasing resource allocation. Instead, it is likely to worsen in the future as applications continue to scale their resource usage to match the growing hardware capabilities of high-end devices.

Also shown in the figure, taking Pixel6 as an example, it takes 3,902ms on average to cold launch an app with ZRAM, while the time cost is reduced by 39.5% with SHSC. It indicates that large-grain compression has benefits in improving the launching speed. SHSC's effect on cold launching is higher than Archer. This is because the former scheme conducts a more aggressive large-size compression. However, subsequent experiments will show that it causes severe read amplification in other usage scenarios. ASAP's impact on cold launching is not obvious. This is because no earlier reclaimed pages need to be swapped in during cold launching. Its advantages are not reflected in this case.

Furthermore, we evaluate their effect on hot launching. Figure 7(b) illustrates that the average launching speed on the three platforms is enhanced by 55.3%, 47.5%, and 29.6%, respectively. Archer boosts the speed from two aspects. On one hand, memory is reclaimed more efficiently. The buddy system can allocate memory space to support hot launching quickly. On the other hand, pages of the tested app may have been compressed. It takes time to decompress them when relaunching. We will analyze the detail in Section 5.3.

The impact of SHSC on hot launching varies among apps. About 43.75% of them experience slower launch times rather than improvements when this scheme is enabled. SHSC compresses data using a static, large granularity. However, not all compressed data is needed during a hot launch. SHSC fails to account for page associations, which means that some contents of a large compressed page may not be necessary during launch. As a result, despite SHSC increasing compression throughput, the negative effects of read amplification can outweigh the benefits in certain cases. These findings demonstrate that compression granularity should be dynamically adjusted to meet the complex demands of mobile systems.

ASAP outperforms ZRAM by 28.3% on the Pixel6, 28.1% on the Pixel3, and 25.4% on the P20. This improvement is due to the switch footprint estimator, which enhances anonymous page decompression. Archer is compatible with ASAP, these two schemes can boost both compression and decompression with coordination. The results illustrate that Archer ourperforms ASAP. For example, by 16.6% on the Pixel6. This is because Archer not only accelerates swap-in pages during hot launching (leveraging large-grain compression to maximize CPU bandwidth), but also improves the speed of swap-out pages: during hot launches, the system needs to reclaim memory to accommodate the necessary pages, and with Archer, the

[4]On a high-end smartphone, it might enable ultra-HD textures and detailed animations, consuming GB-level memory. In contrast, on a low-end device, it restricts graphics to low or medium settings and loads fewer assets.

page reclaim process is faster, further boosting performance.

5.2.2 Photographic Performance and Frame Rate

Figure 8 shows Archer's effect on photographic performance and frame rate. To understand the benefit, the two scenarios are evaluated under different memory pressures. Specifically, before testing the Camera and TikTok, *N* apps are launched and switched to the background (see the X-axis in the figure). For example, "TikTok+6Apps" in the figure means we evaluate TikTok when 6 apps were switched in the background in advance. It is expected that performance will worsen immediately after switching to the tested app and then improve over time, so we conducted the following evaluation after the system settled. Specifically, after switching the tested app to the foreground, we wait for 30 seconds before the measurement.

Continuous shooting using the camera. Figure 8(a) illustrates that the continuous shooting performance significantly degraded when the memory is under pressure. Pixel6, Pixel3, and P20 are evaluated separately. On each device, we evaluate ten rounds and take the average. Taking Pixel6 as an example, 24 photos are taken on average when no additional app consumes memory. This value decreased to 13 when memory was exhausted. Such reduction is alleviated when enabling Archer. It demonstrates that Archer's benefit appears when the memory is under pressure. As the red text in the figure shows, the photographic speed up by 1.42x on Pixel6, 1.34x on Pixel3, and 1.22x on P20, compared to the original system.

SHSC can also improve the camera's response speed. However, its negative effect is also obvious, especially when memory is not scarce. Note that the performance when prelaunching 6 and 8 applications in Pixel3 is similar. It is because this smartphone can only cache 6 apps on average. Additional apps in the background are killed.

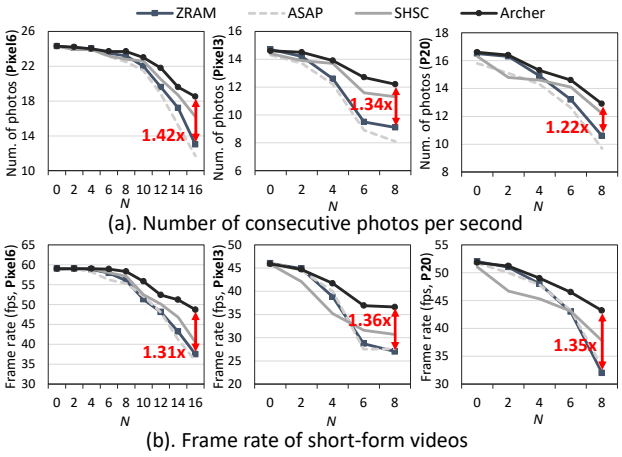


Figure 8: Effect on photographic performance and frame rate.

FPS of short-form videos using TikTok. If the frame rendering cannot be handled on time, the screen display may become jerky or slow from the user's perspective. This is based on human eye sensitivity [55]. As shown in Figure 8(b), the frame rate degraded when the memory is under pressure. Taking Pixel6 as an example, the frame rate is 37.5fps on average in the original system (see the ZRAM case). This metric is increased by 31% with Archer. Similar improvement is also observed on Pixel3 (by 36%) and P20 (by 39%).

5.2.3 Effect on Tail Latency

More importantly, in the above evaluations, we observe that the tail latency (i.e., the worst-case scenario across ten rounds of evaluation for each use case) is significantly reduced, which is critical for enhancing user experience. The summarized results are presented in Figure 9. For instance, on the Pixel6, the tail launching latency for eight apps is reduced by 44.9% for cold launches and 60.3% for hot launches (see Figure 9(a)). Furthermore, the worst-case performance in continuous shooting and frame rendering improves by 1.6x and 1.3x, respectively (see Figure 9(b)).

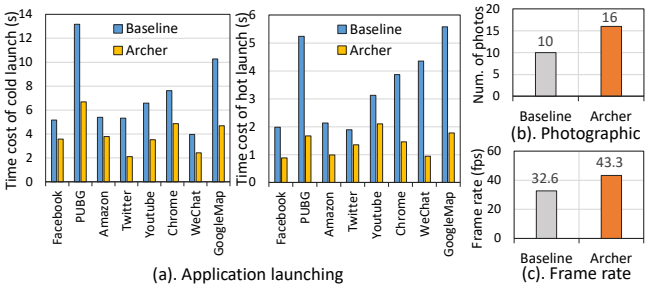


Figure 9: Tail latency comparison.

5.3 Performance Benefit Analysis

We collect the page access history and corresponding timestamps on Pixel6 during the above evaluation. To minimize the impact on the evaluation results, the collected information is analyzed offline. According to the statistics, 75.3% of the anonymous pages are identified as having no strong association with others, but the rest 20%+ correlated pages play an important role in the burst memory-demanding scenarios.

Compressed Pages during App Launching. In terms of app cold launching, an average of 61.7MB pages are compressed during each launching process. Among them, 80.5% pages are compressed in large grain on average. We record the task states on the cores using Systrace and breakdown the launching process. It illustrates that the task suspending time was effectively reduced. The hot launching process is more complicated because some launching required pages are located in the memory while some have already been compressed. As counted, 45.8MB pages are compressed during each hot launching process on average, and 83.2% of them are compressed in large grain. In addition, an average of 20.7MB pages are decompressed in this phase. This is because part of the launching required pages have been compressed earlier.

Thanks to the efficient FP-Growth technique, Archer can quickly degrade the association of 'drifted' pages. Most of the pages with constantly drifting associations are sifted out by

the miner. After the above evaluation, we further use the phone for two hours and compares the association. We observe that 96.3% pages that were identified as associated are still highly-correlated. It demonstrates that the associations identified as 'high' remain stable over time in real-world usage.

Compression Throughput. We collect the timestamp of the compressed pages and calculate the throughput. As evaluated, the throughput of 4KB-grained compression with Archer is 205.1MB/s. This is close to ZRAM. For '$\geq$512KB' compression, the throughput reaches 522.6MB/s. It demonstrates that the pages are compressed much more efficiently during app launching, compared to the original 4KB-grain compression. Similarly, large-grain compression is observed in continuous shooting and short-form video scenarios.

In addition to the above scenarios, Archer also evaluated transformer, an emerging memory-intensive application. It requires a large amount of memory during inference due to the quadratic scaling of self-attention and high-dimensional embeddings. As evaluated, the average latency of transformers is 7.3x higher than CNNs. When enabling Archer, latency is reduced by 39.2% on Pixel6. Archer allows memory to be allocated more efficiently to load the $N \times N$ attention matrix and intermediate representations.

Sensitive Study. The performance benefit is affected by parameter configurations of Archer. Figure 10 shows the evaluation results with different queue and transaction buffer sizes on Pixel6. All the queues and buffers are set as FIFO (cycle queue and ring buffer), except for the '$+\infty$' case.

When the structure size is not very large, the frame rate is optimized with the size increase. For example, the frame rate is 49.1fps on average in the <32, 16> case, while increased by 18.7% in the <256, 128> case. However, when the structure size kept increasing, the frame rate decreased instead. In the extreme case, where the size of the queue and buffer is unlimited (<$+\infty$, $+\infty$>), the frame rate is only 48.7fps on average. When the size of the queue and buffer is small, the smooth experience deteriorates because the read amplification is high. As a result, the average latency of page access increased. In addition, since many unnecessary pages are decompressed, the increased memory pressure induces more reclaim operations. Considering the memory and computing overhead, we do not encourage setting the size with a very high value.

The effect of Archer is also affected by other parameters, like the width of the sliding window (w), the timeout threshold ΔT, and the confidence value of the association-rule mining model, et al. Also, improper configuration may induce aggressive resource usage. Since the space is limited, this paper only shows the evaluation results of part parameters. Still, it is worth noting that a proper parameter set on one device should not port to the other devices straightforwardly. This is because the computing and memory capability between the two devices, as well as the OS settings on them, are different. We suggest to configure Archer on mobile devices customized.

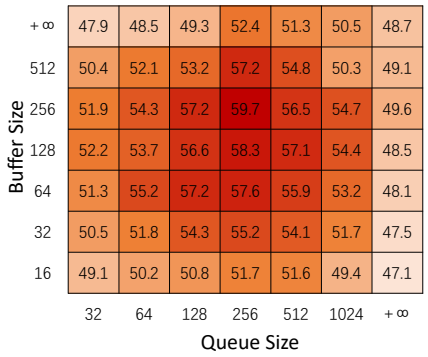


Figure 10: FPS under different <queue, buffer> pairs.

5.4 Potential Penalty Analysis

5.4.1 Read Amplification

One potential penalty is that Archer may increases the read amplification when accessing compressed pages. We analyzed the access footprints and calculated the read amplification. When a compressed page is demanded, we detect how many pre-decompressed pages are accessed shortly. We say a page-read request is amplified if other pre-compressed pages are not accessed in the following w footprints or within one minute. As evaluated, 92.6% of the batch-processed pages are accessed soon, the read amplification is no more than 1.08.

Of course, batch decompression prolongs the latency of the demanded page. It is hard to accurately detect the time cost of every page accessed online, so we compare the latency to decompress a 4KB and 256KB file with a micro-benchmark. Specifically, we obtain page content by writing the data to a laptop. Each collected page is stored as a file on the laptop[5]. We process them in two ways: separately and in batches. The comparison results illustrate that it takes 17.6% longer time to decompress 64 pages than one page. However, the whole process can be completed within sub-μs. More importantly, the latency to access the prefetched pages (the associated pages that are decompressed in advance) is reduced. In terms of Archer's overall performance benefit and high accuracy, batch decompression has small effect on the performance.

5.4.2 Energy Consumption

Archer's impact on energy consumption is multifaceted. On one hand, data mining operations result in additional energy usage. To minimize this, we employ FP-Growth for association-rule mining, as it is more energy-efficient compared to traditional methods like Apriori. On the other hand, Archer's efficient compression reduces context switches and retry operations, which helps conserve energy.

This paper evaluates Archer's overall effect on power consumption. The remaining battery percentage is used as the

[5]The laptop is used to collect memory pages and convert them into files. After the collection phase, these page files are transferred to the mobile device for further measurement.

metric. During the evaluation, the workload apps (see Table 1) were launched and used in the foreground for one hour on two Pixel6 phones. We evaluate them at the same time. Specifically, two devices are controlled by two hands. Two hands perform the same operations at any time. Among them, one device runs Archer and the other does not. Battery reduction is recorded. We repeat the evaluation for eight rounds. In case of possible differences between the two devices, we rotate the Archer running equipment during each round.

There is a significant difference in battery usage between different rounds. This is because the app and user behavior in each round varies. But in any round, the battery reduction of the two devices is close. Specifically, the energy consumption when enabling Archer is 0.69% more on average. That is, suppose the phone can be used for 12 hours without charging, it can still be used for more than 11.92 hours when enabling Archer. In comparison with the persistent energy consumption of touchscreen [57], the effect of Archer is negligible.

5.4.3 Memory Overhead

Archer maintains a mapping table, incurring a space overhead in the memory. In the implementation, all batch-compressed pages are recorded as metadata. There is a variable that points to the handle of the correlated pages. The other variables, like the handle structure, are maintained in the original kernel, so will not introduce additional memory consumption. Suppose there are 1,048,576 anonymous pages (4GB) in total, and 30% of them are highly correlated. Our evaluation shows the average size of frequency itemsets is 6.4. Taking this into account, 384KB additional space is required to store the mapping table. Corresponding variables will be deleted if corresponding pages are decompressed, so the metadata will not keep accumulating.

The other source of memory consumption is the item queue and transaction buffer. Archer uses cycle queue and ring buffer structure to avoid unlimited memory consumption. Each item (`unsigned long` type) in the queue is 64 bits. The width of the sliding window is set to 32 by default, so each transaction is 64×32 bits. Our evaluation illustrates that it is enough to cache 128 items and 512 transactions in most cases. Therefore, the queue and buffer consume 1KB (128 items $\times$ 64 bits) and 128KB (512 transactions $\times$ 32 items $\times$ 64 bits), respectively. In addition, as introduced in Section 4.4, Archer maintains a buffer to cache the candidate pages. Our statistic demonstrates that the number of pages in the same association group generally does not exceed 32. Hence, we set the upbound of the compression buffer as 128KB (32$\times$4KB). The hundred-KB level consumption is acceptable compared to the GB level available memory in the device.

Such small energy and memory consumption is due to our optimization in the design. Theoretically, if Archer brings all page access footprints in the model and performs association mining all the time, for example, disabling the low-power mode, the mining accuracy can be enhanced. However, in daily usage, the page access frequency can be very high. One cannot have an infinite memory space to cache the items and transactions. Besides, optimistically bringing footprints may create CPU contention, which degrades the overall performance. Archer trades off the overhead with comparatively lower page-association mining accuracy. We will discuss this in detail in the following section.

6 Related Work

Various schemes have been proposed to optimize memory compression in mobile systems [28, 29, 45, 64, 66, 67]. SEAL [9] and ASAP [6] illustrate that a high fraction of pages during app launching are accessed together. The former compresses launching required pages in prior, and the latter boosts launching speed by decompressing launching required pages in batch. In this paper, we showed that the page correlation characteristic does not only appear in the launching phase. Marvin [4] found that many compressed pages may be accessed again in a short time since the runtime garbage collection (GC). Acclaim [5] and Ice [41] help reduce the total number of compression operations. These solutions perform compression in page granularity and ignore the potential benefit of large-grain compression in mobile systems. Archer fills the gap of this field, and is compatible with existing solutions.

There are also many studies focused on improving data compression [58–62, 69]. BAC [49] adjusts the compression intensity dynamically and achieves a better tradeoff between compression speed and ratio. To get an outstanding compression ratio, EDC [50] applies different compression algorithms to the same data and selects the best one. EROFS [7] and RTO [63] are designed for file systems. Unlike traditional compression, they make the result of compression closer by considering the characteristics of file systems. Towards the association-rule mining technique, algorithms like Aporio [38], FP-growth [39], Eclat [65], and CHARM [40] have been proposed decades ago. There have been many optimization solutions [31–33, 46, 47] in recent years. Our work is inspired by these studies. Based on them, this paper further proposed PAM and ARC, which realize lightweight association-rule mining and flexible memory compression.

7 Conclusion

This paper proposed Archer, an association-rule aware memory compression framework for mobile devices. It first shows the opportunity for large-grain compression in mobile systems. Archer is the first work that introduces association-rule mining techniques on memory compression. Experimental results show that the average app launching speed is 1.55x faster when enabling Archer, and the average photographic speed and frame rate increase by 1.42x and 1.31x, respectively.

References

[1] TensorflowLite on Android. https://tensorflow.google.cn/lite.

[2] X. Liu, C. Vlachou, F. Qian, C. Wang, and K. H. Kim. Firefly: Untethered multi-user VR for commodity mobile devices. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, pp.943-657, 2020.

[3] The resource code of Zram in the Linux kernel, https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[4] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *USENIX Annual Technical Conference (USENIX ATC)*, pp.873-887, 2020.

[5] 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 Annual Technical Conference (USENIX ATC)*, pp.897-910, 2020.

[6] S. Son, S. Y. Lee, Y. Jin, J. Bae, J. Jeong, T. J. Ham, J. W. Lee, and Y. Hongil. ASAP: Fast Mobile Application Switch via Adaptive Prepaging. In *USENIX Annual Technical Conference (USENIX ATC)*, pp.365-380, 2021.

[7] X. Gao, M. Dong, X. Miao, W. Du, C. Yu, and H. Chen. EROFS: A Compression-Friendly Readonly File System for Resource-scarce Devices. In *USENIX Annual Technical Conference (USENIX ATC)*, pp.149-162, 2019.

[8] Qiang Chen and Changlong Li. Argus: Real-Time HQ Video Decoding with CPU Coordinating on Consumer Devices. In *IEEE Real-Time Systems Symposium (RTSS)*, 2025.

[9] C. Li, L. Shi, Y. Liang, and C. J. Xue. SEAL: User Experience-Aware Two-Level Swap for Mobile Devices. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp.4102-4114, 2020.

[10] Y. Liang, J. Li, X. Chen, R. Ausavarungnirun, R. Pan, T. W. Kuo, and C. J. Xue. Differentiating cache files for fine-grain management to improve mobile performance and lifetime. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, July 2020.

[11] AOSP Foundation, The low memory killer daemon, https://android.googlesource.com/platform/system/core/+/master/lmkd/README.md.

[12] Linux Foundation, Least-recently-used (lru) algorithm in Linux kernel, https://www.kernel.org/.

[13] Huang, J., Zhang, Y., Qiu, J., Liang, Y., Ausavarungnirun, R., Li, Q., and Xue, C. J. More Apps, Faster Hot-Launch on Mobile Devices via Fore/Background-aware GC-Swap Co-design. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Volume 3, pp. 654-670, 2024.

[14] H. H. Sung, J. A. Chen, W. Niu, J. Guan, B. Ren, and X. Shen. Decentralized Application-Level Adaptive Scheduling for Multi-Instance DNNs on Open Mobile Devices. In *USENIX Annual Technical Conference (USENIX ATC)*, pp.865-877, 2023.

[15] ChatGPT of OpenAI, https://openai.com/blog/chatgpt.

[16] W. Guo and K. Chen and H. Feng and Y. Wu and R. Zhang and W. Zheng. MARS: Mobile application relaunching speed-up through flash-aware page swapping. In *IEEE Transactions on Computers (TC)*, pp.916-928, 2016.

[17] R. Davis, Short-form video market soars: people are spending more and more time on short-form video, https://variety.com/2021/streaming/news/china-short-video-market-study-1235001776/.

[18] Z. Li, Y. Xie, R. Netravali, and K. Jamieson. Dashlet: Taming Swipe Uncertainty for Robust Short Video Streaming. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp.1583-1599, 2023.

[19] Android Debug Bridge, https://androidmtk.com/download-minimal-adb-and-fastboot-tool.

[20] Stagesepx: detect stages in video automatically, https://github.com/williamfzc/stagesepx.

[21] The Systrace tookit of Android, https://developer.android.com/topic/performance/tracing.

[22] FP-Growth Algorithm in Data Mining, https://medium.com/image-processing-with-python/fp-growth-algorithm-in-data-mining-e1064accf6a3

[23] Li, Q., and Peng, W. Research on Association Rules Mining for Data Stream. International Core Journal of Engineering, 8(2), 218-225, 2022.

[24] The Android Adj mechanism. https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-11.0.0_r45/services/core/java/com/android/server/am/OomAdjuster.md

[25] Brands are fighting over milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds/?sh=4f52e2f14ad3.

[26] C. Li, C. Wang, X. Zhou, and Edwin H.-M Sha. FlashDAM: Flexible I/O Throttling for the User Experience of Mobile Systems. In *IEEE 41st International Conference on Computer Design (ICCD)*, pp. 239-242, 2023.

[27] G. Wei, C. Li, R. Xu, Q. Zhuge, Edwin H.-M Sha. Sparrow: Flexible Memory Deduplication in Android Systems with Similar-Page Awareness, In *24th Design, Automation and Test in Europe Conference | The European Event for Electronic System Design and Test (DATE)*, 2024.

[28] S. Hahn and S. Lee and I. Yee and D. Ryu and J. Kim. Fasttrack: Foreground app-aware I/O management for improving user experience of Android smartphones. In *USENIX Annual Technical Conference (USENIX ATC)*, pp.15-28, 2018.

[29] Y. Q. Chou, L. W. Shen, and L. P. Chang. Rectifying Skewed Kernel Page Reclamation in Mobile Devices for Improving User-Perceivable Latency. ACM Transactions on Embedded Computing Systems, 22(5s), pp. 1-22, 2023.

[30] Page access mark in Linux, https://lkml.indiana.edu/hypermail/linux/kernel/1304.3/02605.html.

[31] A. Telikani, A. H. Gandomi, and A. Shahbahrami. A survey of evolutionary computation for association rule mining. In *Information Sciences*, pp.318-352, 2020.

[32] J. Dongre, G. L. Prajapati, and S. V. Tokekar. The role of Apriori algorithm for finding the association rules in Data mining. In *International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pp.657-660, 2014.

[33] Using the FP Growth algorithm in Python to for frequent itemset mining, https://towardsdatascience.com/the-fp-growth-algorithm-1ffa20e839b8.

[34] The Apriori algorithm development using Python language, *https://towardsdatascience.com/apriori-association-rule-mining-explanation-and-python-implementation-290b42afdfc6*.

[35] UNREAL engine for mobile games. https://www.unrealengine.com/en-US/.

[36] D. Xu, M. Xu, Q. Wang, S. Wang, Y. Ma, K. Huang, and X. Liu. Mandheling: Mixed-precision on-device dnn training with dsp offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pp.214-227, 2022.

[37] Y. Zheng, C. Li, Y. Xiong, W. Liu, C. Ji, Z. Zhu, and L. Yu. iAware: Interaction Aware Task Scheduling for Reducing Resource Contention in Mobile Systems. In *ACM Transactions on Embedded Computing Systems (TECS)*, pp.1-24, 2023.

[38] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*, pp.3-14, 1995.

[39] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. In *Data mining and knowledge discovery*, pp.53-87, 2004.

[40] M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proceedings of the SIAM international conference on data mining*, pp.457-473, Society for Industrial and Applied Mathematics, 2002.

[41] C. Li, Y. Liang, R. Ausavarungnirun, Z. Zhu, L. Shi, and C. J. Chun. ICE: Collaborating Memory and Process Management for User Experience on Resource-limited Mobile Devices. In *European Conference on Computer Systems*, 2023.

[42] K.Wang, L. Tang, J. Han, and J. Liu. Top down fp-growth for association rule mining. In *Advances in Knowledge Discovery and Data Mining: 6th Pacific-Asia Conference (PAKDD)*, pp. 334-340, 2002.

[43] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. In *Next generation data mining*, pp.191-212, 2003.

[44] M. H. Al and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX Annual Technical Conference (USENIX ATC)*, pp.843-857, 2020.

[45] X. Zhu, D. Liu, K. Zhong, J. Ren, and T. Li. Smartswap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*, pp.1-6, 2017.

[46] L. Bustio-Martínez, R. Cumplido, M. Letras, R. Hernández-Leon, C. Feregrino-Uribe, and J. Hernández-Palancar. FPGA/GPU-based acceleration for frequent itemsets mining: A comprehensive review. In *ACM Computing Surveys (CSUR)*, 54(9), pp.1-35, 2021.

[47] C. Fernandez-Basso, M. D. Ruiz, and M. J. Martin-Bautista. New Spark solutions for distributed frequent

itemset and association rule mining algorithms. In *Cluster Computing*, pp.1-18, 2023.

[48] J. Courville and F. Chen. Understanding storage I/O behaviors of mobile applications. In *32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pp.1-11, 2016.

[49] C. Li, D. Feng, Y. Hua, W. Xia, L. Qin, Y. Huang, and Y. Zhou. BAC: Bandwidth-aware compression for efficient live migration of virtual machines. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pp.1-9, 2017.

[50] B. Mao, S. Wu, H. Jiang, Y. Yang, and Z. Xi. EDC: Improving the performance and space efficiency of flash-based storage systems with elastic data compression. IEEE *Transactions on Parallel and Distributed Systems*, pp.1261-1274, 2018.

[51] Sysfs mechanism in the Linux kernel, https://docs.kernel.org/filesystems/sysfs.html.

[52] Sysctl mechanism in the Linux kernel. https://man7.org/linux/man-pages/man8/sysctl.8.html.

[53] The Lz4 in the Linux kernel. https://github.com/torvalds/linux/blob/master/include/linux/lz4.h.

[54] The huge page mechanism in the Linux kernel. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html.

[55] T. Masashi and U. Takeshi. Smartphone user interface. In *FUJITSU Science Technical Journal*, 2013.

[56] D. Wei, and G. Feng. Compression and Storage Algorithm of Key Information of Communication Data Based on Backpropagation Neural Network. In *Mathematical Problems in Engineering*, 2022.

[57] Y. Choi, S. Park, and H. Cha. Graphics-aware power governing for mobile devices. In *Proceedings of the 17th annual international conference on mobile systems, applications, and services*, pp.469-481, 2019.

[58] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.317-330, 2019.

[59] A. Ranjan, A. Raha, V. Raghunathan, and A. Raghunathan. Approximate memory compression. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4), pp. 980-991, 2020.

[60] Sardashti S. and Wood D. A. Could compression be of general use? evaluating memory compression across domains. In *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4), pp. 1-24, 2017.

[61] Shafiee, A., Taassori, M., Balasubramonian, R., and Davis, A. MemZip: Exploring unconventional benefits from memory compression. In IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 638-649, 2014.

[62] Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P. B., and Mowry, T. C. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 172-184, 2013.

[63] X. Zhang, J. Li, H. Wang, D. Xiong, J. Qu, H. Shin, and T. Zhang. Realizing transparent OS/Apps compression in mobile devices at zero latency overhead. In *IEEE Transactions on Computers*, pp.1188-1199, 2017.

[64] G. Lim, D. Kang, M. Ham, and Y. I. Eom. SWAM: Revisiting Swap and OOMK for Improving Application Responsiveness on Mobile Devices. In *The 29th Annual International Conference On Mobile Computing And Networking (MobiCom)*, 2023.

[65] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, Vol. 97, pp.283-286, 1997.

[66] C. Li, L. Shi, and C. J. Xue, MobileSwap: Cross-Device Memory Swapping for Mobile Devices, In *Design Automation Conference (DAC)*, 2021.

[67] C. Li, Y. Liang, L. Shi, C. Wang, C. J. Xue, and X. Zhou, Flexible and Efficient Memory Swapping Across Mobile Devices with LegoSwap, In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2023.

[68] M. Ju, H. Kim, M. Kang, and S. Kim. Efficient memory reclaiming for mitigating sluggish response in mobile devices. In *IEEE 5th International Conference on Consumer Electronics*, pp.232-236, 2015.

[69] E. Choukse, M. Erez, and A. R. Alameldeen. Compresso: Pragmatic main memory compression. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp.546-558, 2018.