ASAP: Fast Mobile Application Switch via Adaptive Prepaging

Sam Son, Seung Yul Lee, Yunho Jin, and Jonghyun Bae, *Seoul National University;* Jinkyu Jeong, *Sungkyunkwan University;* Tae Jun Ham and Jae W. Lee, *Seoul National University;* Hongil Yoon, *Google*

https://www.usenix.org/conference/atc21/presentation/son

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

ASAP: Fast Mobile Application Switch via Adaptive Prepaging

Sam Son[†] Seung Yul Lee[†] Yunho Jin[†] Jonghyun Bae[†] Jinkyu Jeong[‡] Tae Jun Ham[†]
Jae W. Lee[†] Hongil Yoon[††]
[†]*Seoul National University* [‡]*Sungkyunkwan University* [††]*Google*

Abstract

With mobile applications' ever-increasing demands for memory capacity, along with a steady increase in the number of applications running concurrently, memory capacity is becoming a scarce resource on mobile devices. When the memory pressure is high, current mobile OSes often kill application processes that have not been used recently to reclaim memory space. This leads to a long delay when a user relaunches the killed application, which degrades the user experience. Even if this mechanism is disabled to utilize a compression-based in-memory swap mechanism, relaunching the application still incurs a substantial latency penalty as it requires the decompression of compressed anonymous pages and a stream of I/O accesses to retrieve file-backed pages into memory. This paper identifies conventional demand paging as the primary source of this inefficiency and proposes ASAP, a mechanism for fast **a**pplication **s**witch via **a**daptive **p**repaging on mobile devices. ASAP performs prepaging by combining i) high-precision switch footprint estimators for both file-backed and anonymous pages, and ii) efficient implementation of the prepaging mechanism to minimize resource waste for CPU cycles and disk bandwidth during an application switch. Our evaluation using eight real-world applications on Google Pixel 4 and Pixel 3a demonstrates that ASAP can reduce the switch time by 22.2% and 28.3% on average, respectively (with a maximum of 33.3% and 35.7%, respectively), over the vanilla Android 10.

1 Introduction

With the broad capabilities and flexibility of mobile computing, mobile applications continue to tout rich features to meet users' diverse demands. This entails a continuous increase in both codes and data footprint [4, 31]. This trend has resulted in a constant demand for larger memory capacity on mobile devices to address memory pressure issues. However, the cost of the device and the power/area budget often limit its size.

Modern mobile OSes support virtual memory with compression-based swap [3, 13]. The virtual memory provides an illusion of physical memory space larger than the actual memory capacity, enabling multiple applications to run concurrently even under high memory pressure. However, the benefits come with additional overhead, degrading performance. Slow I/O accesses increase the latency of fetching non-resident file-backed pages from storage. To fetch anonymous pages in the compressed swap space, they first need to be decompressed by CPUs at a page fault. Allocating free pages also consumes system resources. Fetching pages on-demand via demand paging may not efficiently utilize available resources such as CPU cycles and I/O bandwidth.

Our empirical analysis shows that the application switch time can increase by a factor of 4× (in the order of hundreds of milliseconds) when the system is experiencing memory pressure, possibly when running many background applications. This slowdown is mainly attributed to the long blocking time introduced by demand paging for both file-backed and anonymous pages during the application switch rather than freeing allocated pages.

A recent study shows that today's smartphone users often run more than 10 applications [25], and thus it is likely that the system is often operating under memory pressure unless the phone has a large main memory capacity. It is also known that users switch between applications more than 100 times per day [9]. We speculate that such frequent, long-latency events can potentially affect smartphone user experience negatively. In this paper, we aim to reduce the latency of the application switch by minimizing the demand-paging related slowdown. To achieve this goal, we propose ASAP, a mechanism for fast **a**pplication **s**witch via **a**daptive **p**repaging. ASAP builds on the following key observations:

- Hardware resources for fetching non-resident pages are underutilized during the application switch when the system is under memory pressure. For eight popular Android applications, CPU utilization is 34.2% during the switch. Also, only 19.4% of the maximum disk bandwidth is used on average.
- File-backed pages and anonymous pages have differ-

ent characteristics in their *switch footprint*, a set of accessed pages during the application switch. In particular, the switch footprint for file-backed pages is much more invariant—about 75% of all accessed file-backed pages are invariant across switches, while only 44% of anonymous pages are invariant. This motivates us to use different prediction strategies for prepaging them.

We capitalize on these empirical observations to develop an effective prepaging approach. The first observation suggests that it is promising to utilize available resources to prepage pages likely to be accessed at the beginning of an application switch. The prepaging is helpful to maximize the effective CPU and disk bandwidth utilization, which can translate to performance gains (i.e., reduced switch time). The second observation suggests that the target pages to fetch need to be adapted at runtime to capture the applications' dynamically changing page access patterns. This improves the prediction accuracy for the switch footprint, hence making ASAP more effective.

At an application switch, ASAP wakes up multiple prepaging threads to start fetching both file-backed pages and anonymous pages. These threads run in parallel with application threads to overlap prepaging with application computation. To accurately predict switch footprint pages, ASAP employs an adaptive prediction mechanism. Specifically, a single predictor maintains two tables: a candidate table and a target table. The predictor promotes or demotes pages between the two tables based on the runtime information of their access patterns. The prepaging threads issue fetch requests only for the pages in the target table, while pages having a smaller chance of being accessed are maintained in the candidate table.

We have implemented ASAP in Android OS and evaluated it using a set of eight popular mobile applications on Google Pixel 4 and Pixel 3a. The evaluation results show that ASAP considerably reduces the application switch time under memory pressure. ASAP reduces the switching time by 22.2% and 28.3% on average (33.3% and 35.7% at maximum) on Pixel 4 and Pixel 3a, respectively, over the vanilla Android 10. This improvement is attributed to an average of 39.8% and 25.2% increase in CPU and disk bandwidth utilization, respectively, as well as 79.3% and 68.4% prediction accuracy for file-backed and anonymous pages, respectively.

In summary, this paper makes the following contributions:

- We empirically analyze the performance bottleneck of the application switch to identify opportunities for prepaging as a solution to the problem.
- We propose ASAP, an adaptive prepaging technique to reduce the switch time, which is a key user interaction on the mobile device. ASAP is application-agnostic without requiring any change to application codes.
- We integrate ASAP into Android OS and evaluate its performance by using eight popular mobile applications on high-end and mid-end devices (Google Pixel 4 and Pixel 3a). The results demonstrate the effectiveness of ASAP for reducing the application switch time by 22.2% and 28.3% on average, respectively, over the vanilla Android 10.

2 Background and Motivation

2.1 Android Application Memory Management

Application Lifecycle and Memory Management. In Android OS, an application (specifically the application activity) is either in the foreground (i.e., having focus) or in the background (e.g., not visible). In other words, the application that a user is actively using is considered to be in the foreground, while the applications that have been launched but are not currently being used are considered to be in the background. When the system has sufficient DRAM, all application data are kept in memory. However, a user often utilizes many different applications over time, and eventually, application data exceed the DRAM capacity. In such a case, the Android low memory killer daemon (*lmkd*) identifies the least essential application (e.g., one in the background) and kills it so that the memory space that it occupied is freed [21, 26]. Note that this does not necessarily result in the complete loss of the application state since Android applications often store a minimal set of its states when the application is moved to the background. With this mechanism, Android OS only stores a small set of essential application data in memory. For this reason, when a user starts an application that was moved to the background a long time ago and hence killed by *lmkd*, the application data is not resident in memory. Instead, the application needs to recreate all of its activities from scratch utilizing the saved state information. On the other hand, when a user starts an application that was moved to the background very recently, it is much more likely that this application's data still resides in memory, and the application will be ready-to-use in a much shorter period. The time Android OS requires for the former case is called *launch time* and the latter is called *switch time* (sometimes also called *hot launch time*).

Compression-based Swap. An alternative approach to secure the free memory space is the compression-based swap, which compresses the least essential memory pages and stores them in a separate memory region. Later, when the application accesses the compressed pages, they are decompressed back to memory. Compared to the traditional disk-based swap mechanism, the compression-based in-memory swap is faster since it can avoid long-latency disk accesses. This approach's drawback is that i) compressed memory pages still consume memory capacity, and ii) compression/decompression spends CPU cycles. This mechanism is enabled by

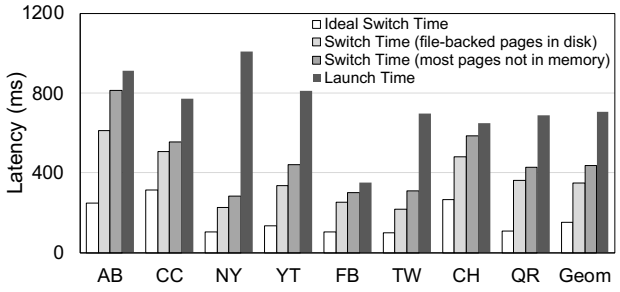

Figure 1: Application switch latency across different scenarios. *Ideal Switch Time* represents a case where all of the applications' anonymous and file-backed pages reside in memory. *Switch Time (file-backed pages in disk)* represents a case where all of the applications' anonymous pages are resident in memory, but almost all of file-backed pages do not reside in memory. *Switch Time (most pages not in memory)* represents a case where most of the applications' anonymous pages are already swapped out, so they are compressed and stored in the compressed memory pool based on the compression-based in-memory swap. Also, almost all of file-backed pages are evicted from memory due to memory pressure. This latency information is equivalent to the baseline switch time illustrated in Section 6.1. Finally, *Launch Time* indicates a case where an application needs to start from scratch. Refer to Section 6.1 for the detailed methodology.

default in many commercial mobile OSes such as Android OS and Apple iOS [3, 13]. However, in practice, Android OS by default does not actively utilize this mechanism since *lmkd* is often triggered first to reclaim memory space before a swap happens [21, 25, 26, 35].

2.2 Launch Time and Switch Time

When a user relaunches an application after a while since its last usage, the latency to reload may differ depending on the system's memory pressure. For example, if the system's memory pressure is low (e.g., the system has not used much memory since the application's last launch), the application's data will still reside in memory. Thus the application could reload quite quickly (i.e., *ideal switch time*). On the other hand, if the system's memory pressure is high (e.g., the user utilized many different apps during the time window), the application will be killed by *lmkd*. The relaunch is highly likely to require recreating the application's activities, incurring a much longer delay (i.e., *launch time*). Finally, if *lmkd* is disabled, the compression-based swap mechanism will come into play. The application's anonymous pages will be stored in memory in a compressed form, and the file-backed pages will be discarded. In this case, relaunching an application requires decompressing some of the application's anonymous pages and reloading file-backed pages from the disk.

Figure 1 presents the application launch/switch time of eight real-world applications (AB: Angry Bird, CC: Candy Crush Saga, NY: New York Times, YT: YouTube, FB: Facebook, TW: Twitter, CH: Google Chrome, QR: Quora) on Google Pixel 4. The figure shows that the switch time is lower than the launch time in all applications. This indicates that reconstructing activities of an application from scratch requires more time than retrieving the relevant anonymous pages and file-backed pages from memory and disks, respectively. This implies that an aggressive setting of Android *lmkd* increases the time to relaunch the application, which confirms the findings of the previous literature [18, 20, 21]. To avoid this unnecessary delay in relaunching the application, it is better to lower the *lmkd* threshold (or even disable it) so that the system can utilize compression-based swap more actively.

This figure also shows a significant gap between the ideal switch time and the switch times under memory pressure. The gap between the *ideal switch time* and the *switch time (file-backed pages in disk)* quantifies the overhead of retrieving file-backed pages from the disk. The gap between the *switch time (file-backed pages in disk)* and the *switch time (most pages not in memory)* indicates the overhead of decompressing anonymous pages from the compressed memory pool. In fact, this overhead increases the application switch time by a factor of 4× relative to the ideal switch time on average. Unfortunately, the real-world switch time is often closer to the switch time (most pages not in memory) than the ideal switch time when we consider recent trends: i) an increase in an application's memory capacity requirements and ii) an increase in the number of apps that a user runs concurrently.

To the best of our knowledge, there are no concrete studies on the threshold of user perception of the application switch. However, previous studies [7, 27] on related contexts imply that the delay of hundreds of milliseconds in the application switch may degrade the user experience. According to Card [7], users feel that a system is reacting instantaneously only when the response time is shorter than 100 ms. Olenski [27] reports that a 100 ms delay in web page loading can degrade the user experience, resulting in a 1% drop in a company's revenue. Thus, we are convinced that maintaining a lower switch time over a wide variety of usage scenarios is critical for user experience.

2.3 Opportunities for Prepaging

Limitations of Demand-Paging. Figure 1 shows that the switch time under memory pressure is substantially worse than the ideal switch time. We find that such a huge overhead resulting from i) decompressing anonymous pages and ii) retrieving file-backed pages from the disk is attributable to the inefficiencies of demand paging. Figure 2(a) shows the CPU utilization and Figure 2(b) shows the disk bandwidth utilization of Google Pixel 4 during switch time of eight applications under memory pressure. Overall, the CPU utilization

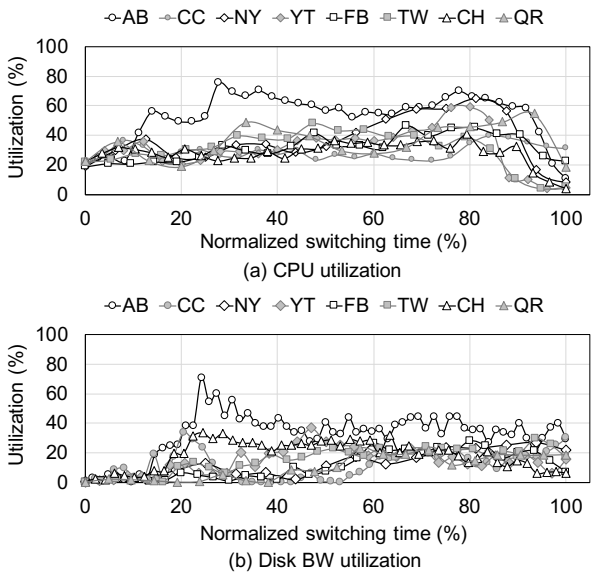

Figure 2: CPU and disk bandwidth utilization of a high-end device (Pixel 4) during the switch time.

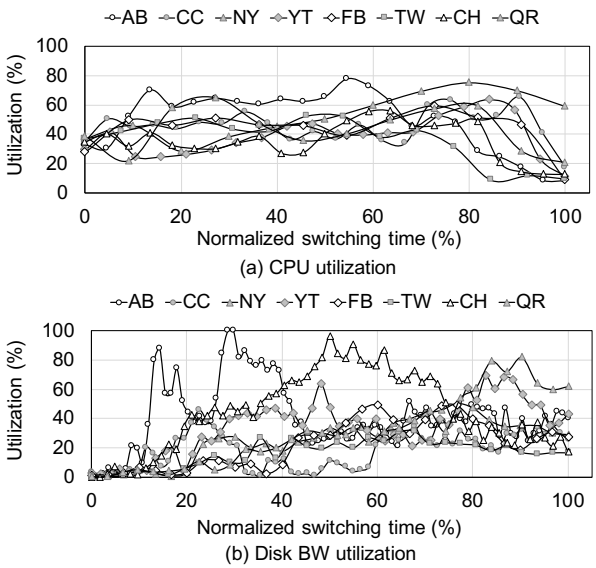

Figure 3: CPU and disk bandwidth utilization of a mid-end device (Pixel 3a) during the switch time.

remains relatively low (i.e., less than 50%) for all applications except AB. Similarly, disk bandwidth utilization is also much lower than the sustainable peak bandwidth most of the time. As shown in Figure 3 for Google Pixel 3a, its disk bandwidth utilization is higher than that of the high-end device (Pixel 4) because the disk bandwidth is relatively low. However, the empirical results show that the resources are not still fully utilized in both cases.

Ideally, the CPU should have been fully utilized to decompress compressed memory pages, and disk bandwidth should have been saturated to retrieve file-backed pages from the disk. However, since the default demand paging mechanism initiates the decompression of memory pages and I/O accesses only at a page fault, the system wastes available resources and spends more application switch time than necessary.

Opportunities and Challenges of Prepaging. The key idea behind ASAP is that we can significantly improve the switch time by letting prepaging threads aggressively decompress memory pages and perform I/O accesses before the application codes demand them. By doing so, ASAP can fully exploit the available system resources (i.e., CPU cycles and disk bandwidth), making the switch time under memory pressure much closer to the ideal switch time. There are two main challenges in this approach. First, the system should effectively identify the switch footprint, a set of pages to be accessed during the switch. Second, the prepaging threads should be efficiently implemented to fully exploit available resources while minimizing their interference with application threads. The following sections describe how ASAP addresses these two challenges.

3 ASAP Design Overview

The empirical observations in the previous section suggest that it is promising to design an adaptive prepaging. We first set two key requirements to design a practical prepaging mechanism:

1. The proposed design should be able to accurately predict a set of pages that are likely to be accessed during an application's switch time (i.e., switch footprint).

2. The proposed design should be able to maximize the efficiency of prepaging by achieving high system resource utilization (i.e., CPU cycles and disk bandwidth) without interfering with the execution of application threads.

ASAP satisfies these requirements with *Switch Footprint Estimator (SFE)* and *Prepaging Manager*. We have integrated them into the application switching process in Linux kernel. Thus, ASAP is application-agnostic without requiring any changes to application codes.

Figure 4 illustrates the overall structure of ASAP with key components shaded in gray. SFE consists of two estimators: one for anonymous pages and the other for file-backed pages. Based on the analysis on the switch footprint, SFE for file-backed pages utilizes offline profiling results as well as a lightweight runtime module to estimate the mostly invariant switch footprint of file-backed pages. On the other hand, SFE for anonymous pages is designed to track a dynamic switch footprint of anonymous pages by gradually promoting pages that are likely to be fetched again during the next switch.

These estimators generate a set of target pages for prepaging, which are retrieved at the beginning of an application

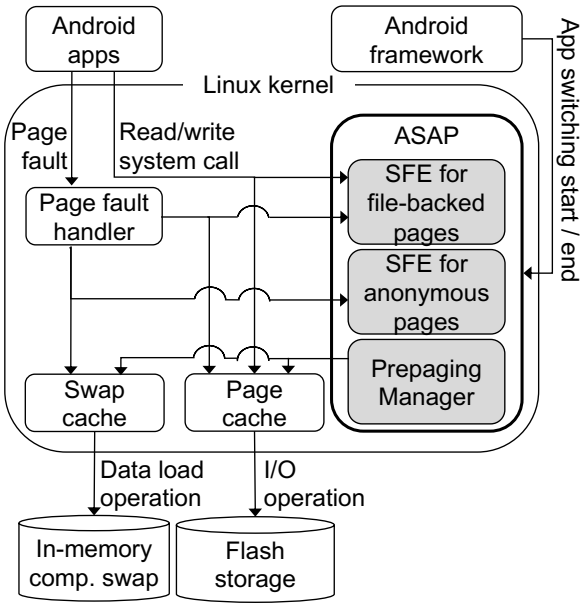

Figure 4: ASAP design overview.

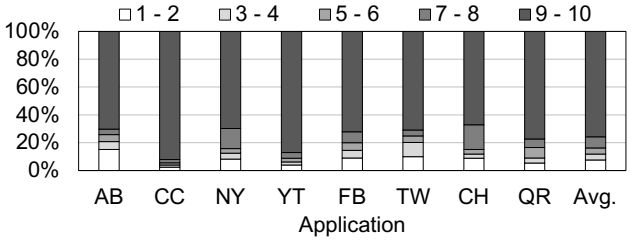

Figure 5: Switch locality analysis for file-backed pages.

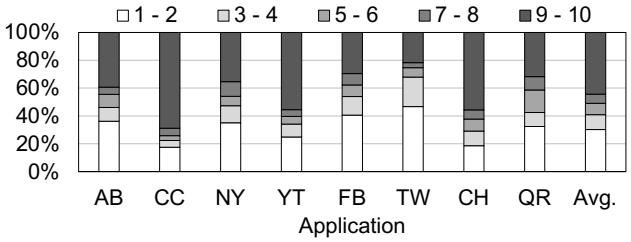

Figure 6: Switch locality analysis for anonymous pages.

switch. It is possible that page information in the prepaging target table becomes obsolete due to inconsistent memory reuse patterns of applications (e.g., application update, post-installation optimization with *dexopt*). Based on the hit/miss history of prepaging, the information is updated over time to ensure high prediction accuracy.

Prepaging Manager is responsible for prepaging threads that are used to fetch target pages from a prepaging target table. It monitors a timing signal that notifies the start and the end of the application switch event from the Android framework. Prepaging Manager promptly wakes up inactive prepaging threads for the switched application when it receives a start signal for application switch, and then it initiates prepaging. Multiple prepaging threads are created according to the number of available CPU cores and run in parallel with application threads to fully utilize the available system resources such as CPU cycles and disk bandwidth. Once they finish issuing fetch requests for all the pages from the prepaging target table, the prepaging manager makes them sleep again until the next switch.

4 Switch Footprint Estimator

4.1 Switch Footprint Analysis

To effectively estimate the targets for prepaging, it is important to understand the characteristics of the switch footprint: a set of pages that are accessed during the switch time. For this purpose, we perform an experiment that exhaustively records all pages accessed across 10 switches for each application (experimental details are available in Section 6.1). For this we cleared the access bit of all present PTE in the address space just before the switch and then checked them right after the switch is completed.

File-backed Pages. Figure 5 shows the switch footprint composition for file-backed pages. The stacked bar shows how many times pages are accessed over the 10 application switches. The switch footprint is largely invariant in this case. On average, about 75% of pages are accessed 9 or 10 times and only 10% are accessed fewer than five times. This highly invariant access pattern of the file-backed pages is due to the fact that a large part of codes and shared library files keep being loaded for the execution of an application.

Anonymous Pages. Figure 6 shows the switch footprint composition for anonymous pages. The access pattern is not as invariant as file-backed pages. About 44% of the anonymous pages are accessed 9 or 10 times across 10 switches. The portion of the invariant (i.e., always accessed) pages is much smaller as the set of accessed anonymous pages easily changes when the application context changes.

Implications. As the characteristics of the switch footprint for anonymous pages and file-backed pages differ, so should their switch footprint estimators. Estimation for file-backed pages can exploit the fact that file-backed pages are highly invariant to minimize the runtime overhead. On the other hand, estimation for anonymous pages needs to rely more on the runtime information so that it can correctly track dynamically changing switch footprints across switch events. Still, the runtime overhead of tracking the switch footprint for anonymous pages is relatively low as the number of anonymous pages in the switch footprint is much smaller than that of the file-backed pages, as shown in Figure 7. The rest of this section discusses the SFE design for both file-backed pages (Section 4.2) and anonymous pages (Section 4.3).

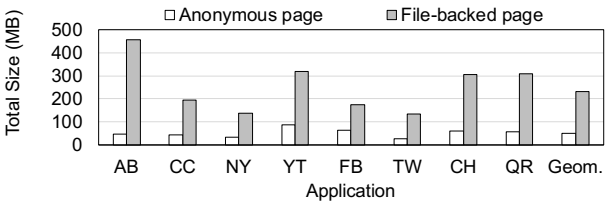

Figure 7: Switch footprint of anonymous and file-backed pages across different applications.

4.2 Estimator for File-Backed Pages

As shown in Figure 5, a major portion of file-backed pages accessed during the application switch are invariant across switches. To exploit this characteristic, SFE for file-backed pages first performs offline profiling to identify the set of potential candidates for prepaging, and then later utilizes minimal runtime information to maintain a concise set of prepaging targets, as shown in Figure 8.

Offline Profiling. The estimator performs offline profiling to obtain a set of prepaging candidates. For this purpose, we measure the file-backed pages that are accessed during ten switch events for each app, as in Figure 5. Then, pages accessed more than eight times (out of ten trials) are considered to be frequently accessed. The resulting set of pages is stored as a file (*Offline Candidate Table*). Specifically, as shown in Figure 8, the profiled result is stored as a map, where a filename is a key and a list of pairs (offset, len) is a value. Each pair represents [offset, offset + len) pages within a file that are accessed during an application switch (we call it an *extent* in the rest of this paper). Later, the profiled result is reloaded at the launch time of this application.

Fault Logging. Fault logging happens at every switch event. Specifically, SFE logs the inode and page indices of all faulted extents received from the kernel until the end of the switch time. This is stored in a *fault buffer*, which is later utilized by the estimator after the end of the switch time to update its prepaging targets.

Prepaging Target Management - Insertion. Once the switch finishes, a background thread performs prepaging target management, exploiting the information from the offline profiling and the fault logging. *Prepaging Target Table* stores information for extents that are to be fetched by the prepaging threads. Ideally, we should insert only those extents that are likely to be fetched in the near future. To identify such an extent, the estimator first inspects an extent in the fault buffer and checks if the extent is also found in the *Offline Candidate Table*. If so, the estimator inserts the corresponding entries into the *Prepaging Target Table*.

Prepaging Target Management - Extent Merging. The *Prepaging Target Table* may have multiple extents on the same file. In such a case, if two extents are close to each other (e.g., the end of one extent is less than 16 pages apart from the start of the other extent), we merge those two extents and create a larger extent that covers both. This is to avoid issuing multiple fragmented I/O requests and instead issue a single, sequential large I/O request, which is often handled much more efficiently.

Prepaging Target Management - Eviction. Eviction from a *Prepaging Target Table* happens when the fetched page turns out to be not utilized during a switch time. Specifically, the estimator checks the `mapcount` of each fetched page after the switch, and removes the page from the *Prepaging Target Table* if the `mapcount` is zero. When a page is part of an extent, the extent is divided into two smaller extents.

4.3 Estimator for Anonymous Pages

As shown in Figure 6, the set of anonymous pages accessed during the application switch changes much more frequently than files. Moreover, anonymous pages are allocated whenever the application is launched, and thus offline profiling is not helpful for identifying prepaging candidates. To effectively track the switch footprint for anonymous pages, we focus on runtime analysis, unlike the case of file-backed pages (Section 4.2). Policies of the estimator are depicted in Figure 9.

Fault Logging. During the application switch time, like the estimator for file-backed pages, the Switch Footprint estimator for anonymous pages logs all anonymous page faults. Fault information is logged at a fault buffer for later usage.

Access Logging. To track access information during switch time, this estimator clears the access bit of every PTE represented by each page identifier in both the *Prepaging Target Table* and the *Online Candidate Table* before every application switch time. Then, right before the end of the switch time, the access bits of all pages in both tables are again checked to identify a set of pages that are accessed during switch time.

Prepaging Target Management - Check & Insertion. After the application switch time, this estimator first checks if each page in the fault buffer is not already present in the *Online Candidate Table* nor the *Prepaging Target Table*. If there are pages that are not already present in the tables, they are inserted into the *Online Candidate Table*.

Prepaging Target Management - Promotion. Also, the estimator checks if each page in *Online Candidate Table* has been accessed during the switch time by inspecting the access log. If a page has been accessed during the last switch time, the page in the *Online Candidate Table* is then promoted to the *Prepaging Target Table*.

Prepaging Target Management - Eviction. Every page in both the *Online Candidate Table* and the *Prepaging Target Table* has its own timeout counter, which is the number of switch events a page can experience before getting evicted from a table. The timeout counter (e.g., 5) of a page is decremented after every switch time. If a specific page is not accessed until the timeout counter reaches zero, it is evicted from the

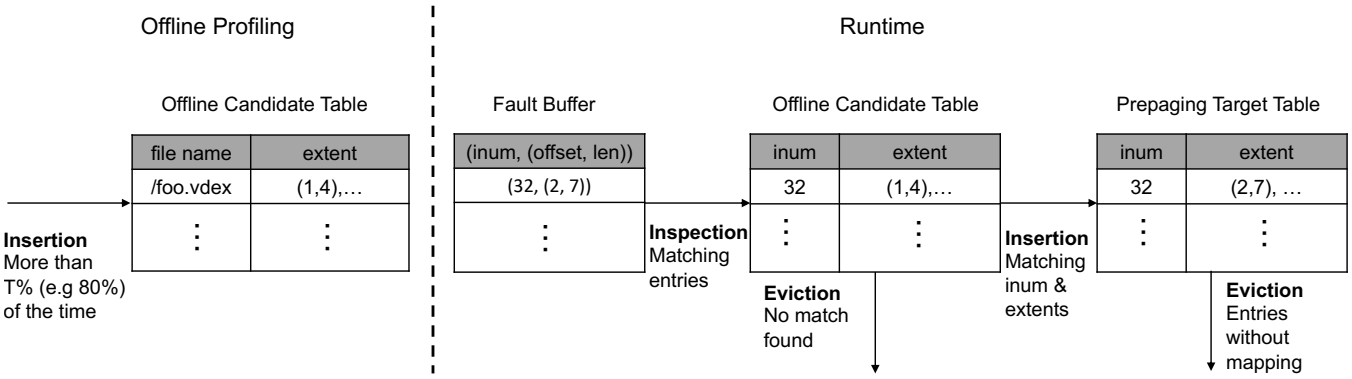

Figure 8: Switch footprint estimator for file-backed pages.

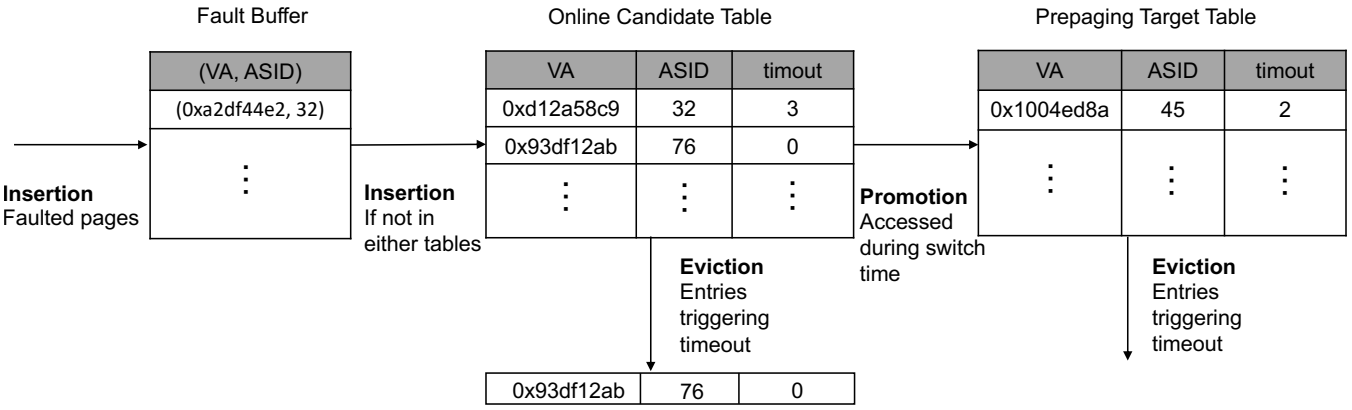

Figure 9: Switch footprint estimator for anonymous pages.

table that it belongs to. But, whenever a page is accessed, the timeout counter of an identifier is reset to the default timeout counter value (e.g., 5).

5 Prepaging Manager

Whenever an application switch event occurs, ASAP's Prepaging Manager wakes up threads. They prefetch pages in the *Prepaging Target Table*, which eventually constructs corresponding PTEs. We apply different prepaging policies to anonymous pages and file-backed pages as follows.

5.1 Prepaging Anonymous Pages

Prepaging of anonymous pages requires decompressing swapped out pages in the compressed in-memory swap space. Hence, the task is compute-intensive, and the task should be carefully scheduled not to incur CPU contention between application threads and prepaging threads. Although the CPU utilization is low, as we reported in Section 2.3, the application threads can demand more CPU resources due to reduced page fault events by prepaging operations.

To this end, the prepaging manager maintains a set of threads for anonymous page prepaging. We pinned a thread on each core, and we assigned the lowest priority (i.e., `SCHED_IDLE` [17]) to them. This allows us to opportunistically utilize the surplus CPU resources for the prepaging of anonymous pages while not incurring any CPU contention with the application threads.

The distribution of prepaging work is done in a work sharing manner. Each thread retrieves a batch (16 pages) from the *Prepaging Target Table*, and then conducts the prepaging operations for pages in the batch. Specifically, for each virtual page in the batch, each thread checks whether the virtual page is present in the process's address space. If non-present, it issues a swap-in operation for the virtual page to the swap subsystem (i.e., the swap cache). The swap-in operation eventually becomes the decompression operation in the in-memory compressed swap device. After the target page is decompressed, the thread finally makes the corresponding PTE point to the swapped-in page.

5.2 Prepaging File-backed Pages

The prepaging manager maintains another set of threads for prepaging file-backed pages. However, file-backed pages impose a higher miss penalty than anonymous pages due to long disk I/O time. Therefore, we take a different prepaging policy for file-backed pages as follows.

First, a file is a unit of prepaging work distribution. Each thread is assigned a file from the *Prepaging Target Table*, and then prefetches pages of the file. For the prepaging operation, each thread issues asynchronous page cache read operations for the corresponding extents in the *Prepaging Target Table*. Note that the *Prepaging Target Table* estimates pages are not only accessed through page faults but are also accessed via read/write system calls. Hence, not all prefetched pages need to be mapped in the process's virtual address space. Prepaging the pages in the *Prepaging Target Table* places fetched pages into the page cache, and thus page faults can still occur for those prefetched pages. However, their fault handling cost is light when the corresponding pages are in the page cache (i.e., minor faults in Linux). When a page fault occurs for a file-backed page, the page fault handler retrieves pages surrounding the missing page from the page cache and maps them all together in the virtual memory to reduce page fault frequency; hence performing prepaging.

Second, we dedicate at least one thread to continuously perform prepaging of large file-backed pages. Figure 10 shows the cumulative distribution of accessed pages of files in the switch footprint of the applications. As shown in the figure, about 90% of the total number of accessed pages is part of the top 15% of large files. Thus, we can expect spatial locality of such accesses to large files. This indicates that if those files are assigned to the prefetching threads with the `SCHED_IDLE` priority, pages of the files are not likely to be prefetched on time due to the lowest CPU scheduling priority. To avoid this problem, we designate one thread with `SCHED_NORMAL` priority running on the big core to be in charge of prefetching pages of large files. Considering the big-LITTLE heterogeneity of the CPU cores in mobile systems, the thread is assigned to run on a big core to maximize the prefetching performance. We have empirically found that this configuration is effective in reducing the miss ratio as well as the CPU contention with application threads.

Lastly, during the prefetching of file-backed pages, file metadata should be carefully handled. Unless file metadata (e.g., logical block addresses of file pages) is in memory, the metadata retrieval incurs additional delay, which in turn degrades the effectiveness of prefetching [23]. This problem exacerbates because of our extent-based prefetching. The metadata read I/O can stay behind a large prefetching I/O request, thereby blocking the prefetching threads as well as the application threads. To avoid this problem, our scheme attempts to read metadata blocks before prefetching pages of a corresponding file. The metadata block reads are done by accessing file pages with a large stride in file offset, 512 pages in our case, because a direct block contains LBAs of 512 data blocks in F2FS [22].

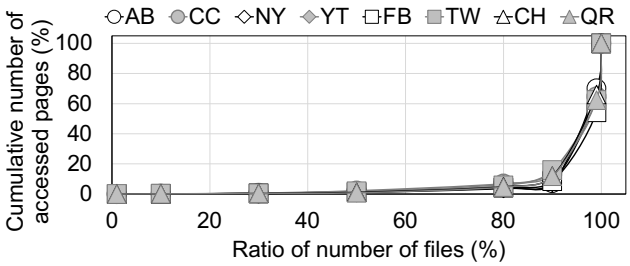

Figure 10: Cumulative number of accessed pages CDF of files across the various applications during switch from one application to another one. Files are sorted by size. 100% indicates the largest file.

6 Evaluation

In this section, we evaluate the effectiveness of ASAP. Section 6.1 describes the evaluation methodology and workloads. Then, we evaluate the latency benefits of our proposal in Section 6.2. Section 6.3 analyzes the accuracy of the switch footprint estimator. We evaluate the efficacy of the prepaging manager by considering improvement of the effective disk bandwidth and CPU utilization.

6.1 Methodology

Switching Latency Measurement. To measure the application switching latency, we used the *am* command in the Android debug bridge (*adb*) [2]. This command starts a selected application and reports two types of switching latency. One is latency from a user's touch to the first rendering, and the other one is latency from a user's touch to the full rendering [5]. The latter is reported only when the application developer implements the debug callback. The information is reported only by the YT application among eight benchmark applications. Thus, we use the time to the initial rendering as a metric. For the YT application, we observe that the additional latency overhead of the full rendering is less than 5% of the switch latency (10-20 ms). Users could also start to interact with applications in the middle of the rendering [16]. The actual latency overhead is expected to be insignificant when the performance benefits of ASAP are considered. This justifies our usage of the time to the initial rendering as the metric for evaluation.

System Configuration. For our evaluation, we use Google Pixel 4 and Pixel 3a, which represent high-end and mid-end devices, respectively. Table 1 describes their specifications. We implement ASAP in Android 10. When measuring the application switch overhead under memory pressure, we consider two aspects for our experimental methodology.

Table 1: Device Specifications

Device	Google Pixel 4	Google Pixel 3a
CPU	Octa-core Qualcomm Snapdragon 855	Octa-core Qualcomm Snapdragon 670
DRAM	6GB LPDDR4x (eff. 4GB)	4GB LPDDR4x
Storage	64GB UFS 2.1	64GB eMMC 5.1
OS	Android 10.0.0 (r41) with Linux kernel 4.14	Android 10.0.0 (r41) with Linux kernel 4.9
zram	2GB (default)	2GB (default)

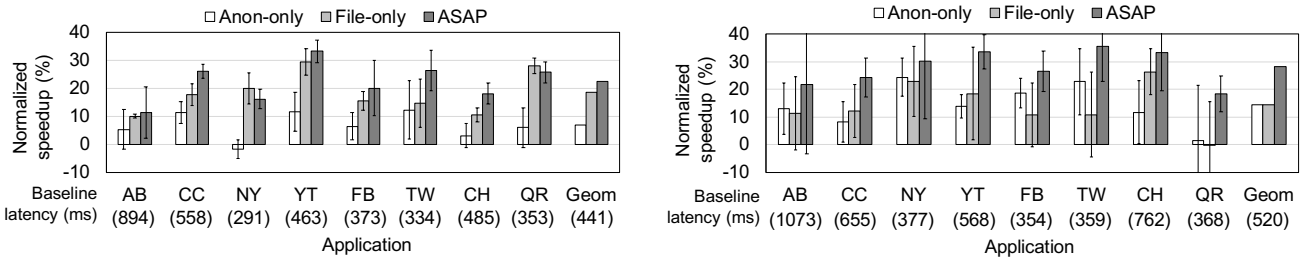

Figure 11: Normalized speedup of application switching latency on (a) Pixel 4 and (b) Pixel 3a. Numbers in parentheses indicate absolute switching latency of the baseline system in ms. Error bar shows standard deviation over different sequences.

Table 2: Applications and automated interactions to change contexts.

Application	Automated Usage Patterns
Angry Bird (AB)	Play a stage
Candy Crush (CC)	Play a stage
New York Times (NY)	Browse and read articles
Youtube (YT)	Watch videos
Facebook (FB)	Browse and read posts
Twitter (TW)	Browse and read posts
Chrome (CH)	Browse keywords
Quora (QR)	Browse questions and answers

Table 3: Chosen 3 application test sequences.

Sequence 1	YT-CH-CC-AB-NY-QR-FB-TW
Sequence 2	QR-NY-CH-CC-YT-TW-FB-AB
Sequence 3	AB-FB-QR-TW-CC-CH-YT-NY

First, we favor the compression based swap approach over the *lmkd*, which often acts first to secure free memory and prevents the system from being under memory pressure. Note that Android currently enables both features by default. We disable the *lmkd* for our evaluation to solely analyze the performance impact on application switch under memory pressure.

Second, users show different application usage patterns such as a spectrum of day-to-day use applications and the use of multitasking features. These lead to different memory usage patterns even among smartphone users who have the same devices.

In this work, thus, we focus on evaluating the memory pressure impact of the application switch for a fixed set of a wide spectrum of top rated applications (refer to Table 2). We enable memory ballooning by considering the entire footprint of the target applications instead of enabling numerous applications to cause memory pressure for the target devices. The effective memory size of both Pixel 4 and Pixel 3a is 4GB. Throughout our evaluation, we refer to the switch time measured on this configuration as the *baseline switch time*.

Workloads and Automation of Tests. In order to reduce the run-to-run variation in the experimental results, we carefully devise an automation program that closely mimics a set of pre-determined user interactions with *adb*. For example, the Facebook (FB) usage pattern contains scrolling down the main news feed, searching for user profiles, and watching their timelines. Another example would be YouTube (YT), where our program searches and watches different video clips. The details of the usage patterns are listed in Table 2.

After execution of a certain application, e.g., Candy Crush (CC), we switch to the next application, e.g., TW, by following a pre-determined sequence of applications. As there are 8! available application sequences for eight applications, we chose three random sequences to evaluate ASAP (Table 3). The start and end time of the application switching operation is informed by the Android activity manager [1, 12]. We iterate the selected sequence 10 times and measure the application switch time. With this user interaction automation program, we repetitively conduct the same evaluation process.

6.2 Application Switch Latency

Figure 11 presents the speedup of ASAP based on Pixel 4 and Pixel 3a, respectively, for 8 applications. We also evaluate the speedup by selectively enabling prepaging for either anonymous pages or file-backed pages. Compared to the baseline, ASAP shows an average of 22.2% and 28.3% performance improvement, and a maximum of 33.3% (YT) and 35.7% (TW) on Pixel 4 and Pixel 3a, respectively. We observe 6.8% and 14.6% performance improvement on each device on average when ASAP performs prepaging only for anonymous pages (*Anon-only*). Among the eight applications, YT and TW show the most noticeable latency reduction on Pixel 4 and Pixel 3a, respectively. With prepaging for file-backed page only (*File-only*), the latency is reduced by 18.3% and 14.4% on average. Here, YT and CH show a substantial latency reduction on each device.

When both SFEs are enabled (ASAP), we observe additional performance benefits for most cases as expected. However, in NY and QR on Pixel 4, integrating both SFEs does not further reduce their switch latency.

6.3 Estimator Efficiency

Figure 12 presents the efficiency of the proposed switch footprint estimators for both Anonymous SFE and File-backed SFE. Since we observe similar performance trends on both devices, we will only present the results on Pixel 4 in the rest of this section. Precision is defined as a fraction of correctly prepaged pages among entire prepaged pages. Recall is defined as a fraction of correctly prepaged pages among all faulting pages during the switch time when the baseline system is considered. Anonymous SFE shows an average of 68.4% precision and 60.4% recall. File-backed SFE shows an average of 79.3% precision and 52.2% recall. The Switch Footprint Estimator for file-backed pages shows better precision relative to that of the Switch Footprint Estimator for anonymous pages. The difference comes from the fact that the switch footprint of file-backed pages is more static, as described in Figure 5.

The gap between precision and recall comes from the coverage of the prepaging target tables. Note that both precision and recall have the same numerator value while the denominator of recall can cover more pages that have not been fetched by the proposed prepaging scheme. We see a larger gap between the precision and the recall of file-backed page prepaging relative to the gap of anonymous page prepaging. This could result from the limited coverage of the candidate pages that is based on the static profiling. The static profiling approach may not capture the entire set of pages that are likely to cause faults at runtime.

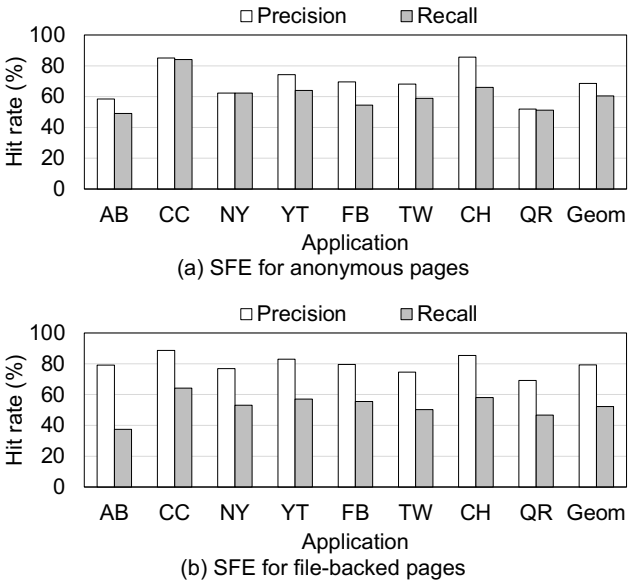

Figure 12: Switch footprint estimator performance.

6.4 Resource Utilization

To show the efficacy of ASAP on prepaging, we evaluated the changes in CPU and memory bandwidth utilization on Pixel 4. The bandwidth utilization is computed as the ratio of achieved file read throughput to the maximum sequential throughput measured in fio [10]. As depicted in Figure 13, ASAP eagerly allocates threads for decompression, which increases the CPU utilization to 1.18× on average over the total switch time, compared to the baseline switch. We also notice the maximum of 1.35× utilization increase. The CPU has been under utilized at the beginning of the application switch. In most cases, the anonymous prepaging threads have a large window of opportunity to fully exploit the available CPU resources. Therefore, when ASAP is enabled, the CPU utilization improves at the early stages of switching. Because the throughput of zram actually scales depending on the number of CPU cores, the anonymous prepaging threads can prepage anonymous pages at great speed. For most applications, prepaging threads finish at around the first 30% of normalized switching time. After that, the CPU utilization follows the CPU utilization pattern of the baseline. On the same page as the CPU, ASAP also improves the I/O bandwidth by 25.2% on average, as shown in Figure 14. In most cases, we observe a noticeable increase in the I/O bandwidth at the early stages of switching and the maximum achieved bandwidth is also higher than that of the baseline. ASAP does not induce significant improvement over the baseline in the case of AB. This is because AB is a highly parallel application with high I/O utilization. Therefore, the I/O bandwidth improvement from our asynchronous I/O threads is limited. The empirical analysis substantiates that ASAP efficaciously exploits the resources at the beginning of the switch to considerably reduce

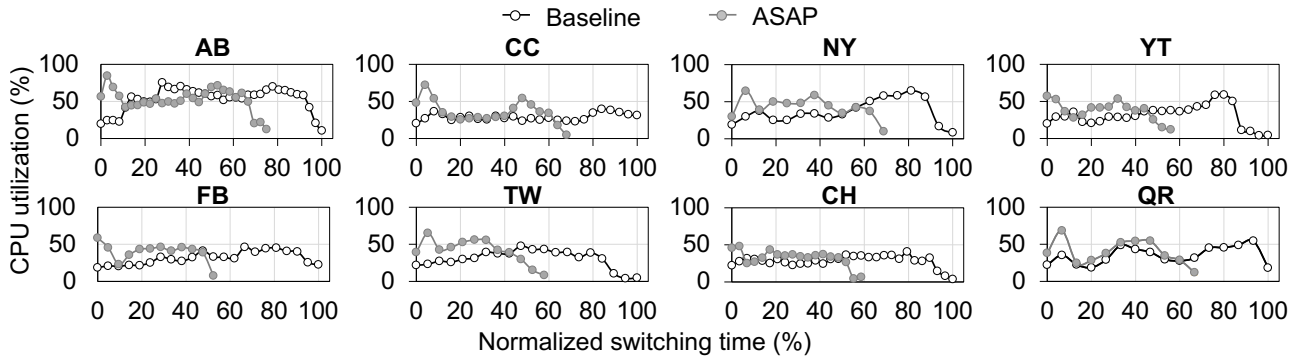

Figure 13: CPU utilization. X-axis is a timeline normalized to baseline's switch time.

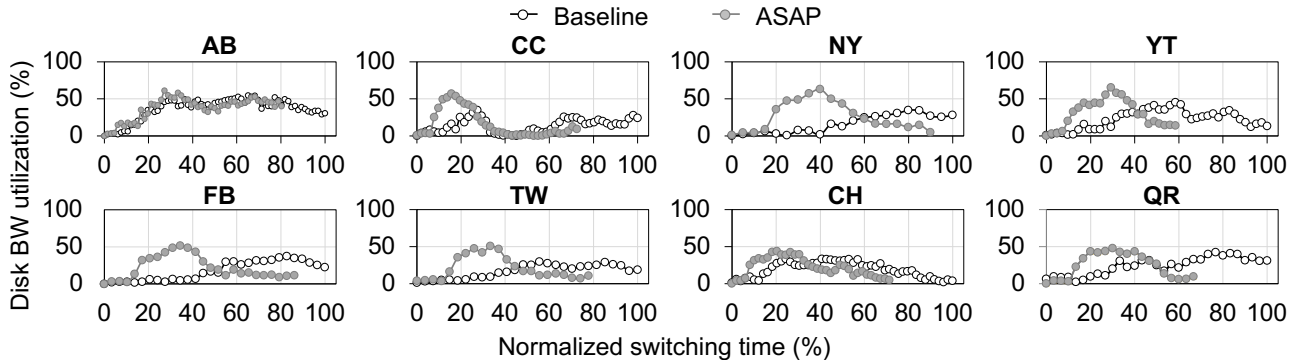

Figure 14: Disk bandwidth utilization. X-axis is a timeline normalized to baseline's switch time.

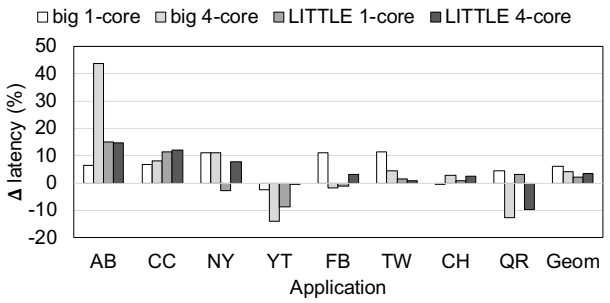

Figure 15: Switching latency changes depending on different core scheduling policy compared to ASAP's core scheduling policies. Positive latency change means that the static policy is worse than ASAP's policy.

the application switch latency.

6.5 Efficiency of Core Scheduling

To quantify the effect of core scheduling of the file prepaging threads (Section 5.2), we compare our policy on Pixel 4 with four other static policies: big 1-core, big 4-core, LITTLE 1-core, and LITTLE 4-core. For example, in LITTLE 4-core, four file prepaging threads are scheduled on the four LITTLE cores, and the threads are assigned the `SCHED_NORMAL` priority. And we enabled only file prepaging to reduce performance deviation. Figure 15 shows the delta of application switch latency (the latency of the four naive policies minus the latency of our scheme). Each naive policy shows 1.06×, 1.05×, 1.02×, 1.04× times slower than ours on average. Hence, our performance advantage comes from the fact that our policy is versatile to different situations. For example, the big 4-core policy showed 14% and 13% better performance than our policy on YT and QR, however its performance falls dramatically on AB since AB utilizes both CPU and disk bandwidth intensively, so file prepaging threads contended a lot with AB's application threads. On the other hand, the LITTLE 4-core policy is better than ours in QR and AB, but it is vulnerable to applications requiring heavy file I/O because of the slow prepaging speed.

6.6 Overhead

Anonymous SFE maintains an *Online Candidate Table*, *Prepaging Target Table*, and anonymous fault buffer. Their peak size for 8 applications is 1MB, 2.5MB, and 0.5MB, respectively. The size of the *Offline Candidate Table*, *Prepaging Target Table* and file fault buffer used by File-backed SFE is 1.5MB, 0.2MB, and 0.5MB, respectively, at their peak respec-

tively. On average ASAP uses about 800KB per application.

Access bit logging (clearing access bits at the beginning and inspecting them at the end of the switch time) extends the switch time by up to 14ms. Also, prepaging target management operations which opportunistically runs between the switch events takes 40ms CPU time in the worst case.

Finally, mis-prediction events result in extra fetch overhead, which could increase the energy consumption. On average, ASAP fetches an extra 10MB for anonymous pages and file-backed pages, respectively. Also the peak throughput of decompression and the disk bandwidth are 2GB/s and 600MB/s on Pixel 4, respectively. Therefore, each extra fetch takes tens of milliseconds. When the peak power of UFS 2.1 [33] and TDP of Snapdragon 855 [32] are considered, these extra fetches require negligible overhead. Actually, we expect ASAP to save the energy consumption of the entire device including other components (e.g., display) because ASAP reduces the total switch latency. Thus, this marginal energy overhead can be easily offset.

7 Related Work

Efficient Memory Management in Mobile Systems. Modern mobile systems reclaim free pages by killing the least essential applications (e.g., low memory killer in Android [26]). The traditional low memory killer selects a victim process by considering the priority and the number of pages of application only. SmartLMK [19] proposes to kill an application to minimize the expected user-perceived application performance by carefully considering application usage statistics and application launch times. However, killing an application process is the most aggressive policy in memory reclamation [6], and whenever a killed application is launched again, it takes a large amount of computation and I/O operations, which can increase the user-perceived launch latency and the energy consumption of mobile devices [21, 24]

To end this senseless killing, Marvin [21] swaps out predicted unlikely-to-be-used pages to disks using ahead-of-time swap by modifying Android runtime (ART). Similarly, SmartSwap [35] includes process-level early page swap based on the prediction result but by addressing kernel codes. A2S [18] combines the low memory killer and the compressed swap together by carefully selecting the victim pages for swap-out and the victim process to kill. Acclaim [25] prioritizes pages of foreground processes over those of background processes during swapping. Kwon et al. [20] propose to swap-out GPU buffers of background processes to relieve memory pressure on mobile devices. Chae et al. [8] propose to extend the swap space of mobile systems to the cloud.

Accelerating Application Launch. Numerous studies have been conducted to shorten the application launch time, and most have tried to prefetch data effectively [12, 15, 28, 30, 34]. FAST [15] profiles I/O sequences during application launches and uses the profiled sequences for data prefetching. FALCON [34] adopts machine learning to predict the users' application usage pattern. It predicts the next application a user is going to use and preloads the contents of the predicted applications. Nagarajan et al. [28] uses collaborative filtering to predict the impending applications while PREPP [30] uses prediction by the partial matching technique. IORap [12] in Android 11 profiles the required I/O during several cold-runs of an application and predicts which I/O will be required and does it in advance. These works only focus on predicting applications or I/O patterns during application launch events. However, our work predicts I/O patterns or memory access footprint during application switch events.

Efficiently Utilizing Disk I/O Bandwidth. The disk I/O performance is important to the user-perceived application performance. Accordingly, the efficient use of disk I/O is important. SmartIO [29] discovers that read I/O operations are penalized by write I/O operations and proposes to prioritize read I/O operations over write ones. Joo et al. [14] finds that swap I/O patterns for page faults are not efficient due to their small and random I/O request patterns. To overcome these inefficiencies, they insert pads to build large sequential I/O requests, which is more efficient in flash-based disks. FastTrack [11] prioritizes I/O requests from foreground applications over those from background ones throughout the entire I/O stack. These approaches are complementary to our work in terms of improving the I/O efficiency during disk access.

8 Conclusion

The goal of this paper is to improve user experience on mobile devices, focusing on the application switch, which is one of the most important user interactions. We proposed (ASAP), an adaptive prepaging scheme that accurately retrieves pages ahead of time that are expected to be accessed during application switch by fully exploiting the available system resources. Our experimental results based on real-world Android OS applications show that ASAP can reduce the application switch latency under memory pressure by 22.2% and 28.3% on representative high-end and mid-end smartphones, respectively. While ASAP was evaluated in the context of the application switch, we believe that it can easily be extended to reducing application launch time as well.

We open sourced Linux kernel we modified for ASAP. The code is available at `https://github.com/SNU-ARC/atc21-asap-kernel`.

Acknowledgments

We thank Lin Zhong for shepherding this paper. This work was supported by a research grant from Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1702-52. Hongil Yoon and Jae W. Lee are the corresponding authors.

References

[1] Activity manager. https://developer.android.com/reference/android/app/ActivityManager.

[2] Android Debug Bridge. https://developer.android.com/studio/command-line/adb.

[3] Memory allocation among processes. https://developer.android.com/topic/performance/memory-management.

[4] The Average Size of the U.S. App Store's Top Games Has Grown 76% in Five Years . https://sensortower.com/blog/ios-game-size-growth-2020.

[5] App Startup Time. https://developer.android.com/topic/performance/vitals/launch-time.

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

[7] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, page 181–186, New York, NY, USA, 1991. Association for Computing Machinery.

[8] D. Chae, J. Kim, Y. Kim, J. Kim, K. Chang, S. Suh, and H. Lee. CloudSwap: A cloud-assisted swap mechanism for mobile devices. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 462–472, 2016.

[9] Tao Deng, Shaheen Kanthawala, Jingbo Meng, Wei Peng, Anastasia Kononova, Qi Hao, Qinhao Zhang, and Prabu David. Measuring smartphone usage and task switching with log tracking and self-reports. *Mobile Media & Communication*, 7:205015791876149, 04 2018.

[10] fio - flexible I/O tester rev.325. https://fio.readthedocs.io/en/latest/fio_doc.html.

[11] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. FastTrack: Foreground app-aware I/O management for improving user experience of android smartphones. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 15–28, Boston, MA, July 2018. USENIX Association.

[12] Android I/O read ahead process. https://medium.com/androiddevelopers/improving-app-startup-with-i-o-prefetching-62fbdb9c9020.

[13] iOS Memory Deep Dive. https://developer.apple.com/videos/play/wwdc2018/416/.

[14] Y. Joo, D. Seo, D. Shin, and S. Lim. Enlarging I/O size for faster loading of mobile applications. *IEEE Embedded Systems Letters*, 12(2):50–53, 2020.

[15] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. Fast: Quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 19, USA, 2011. USENIX Association.

[16] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. Improving user perceived page load times using gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 545–559, Boston, MA, March 2017. USENIX Association.

[17] CFS Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html#scheduling-policies.

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

[19] Sang-Hoon Kim, Jinkyu Jeong, Jin-Soo Kim, and Seungryoul Maeng. SmartLMK: A memory reclamation scheme for improving user-perceived app launch time. *ACM Trans. Embed. Comput. Syst.*, 15(3), May 2016.

[20] S. Kwon, S. Kim, J. Kim, and J. Jeong. Managing GPU buffers for caching more apps in mobile systems. In *Proceedings of the 2015 International Conference on Embedded Software (EMSOFT)*, pages 207–216, 2015.

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

[22] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[23] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, Renton, WA, July 2019. USENIX Association.

[24] Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B. Shroff. Context-aware application scheduling in mobile systems: What will users do and not do

next? In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, page 1235–1246, New York, NY, USA, 2016. Association for Computing Machinery.

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

[26] Low Memory Killer Daemon. https://source.android.com/devices/tech/perf/lmkd.

[27] Why Brands Are Fighting Over Milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds/?sh=4f52e2f14ad3.

[28] Nagarajan Natarajan, Donghyuk Shin, and Inderjit S. Dhillon. Which app will you use next? collaborative filtering with interactional context. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, page 201–208, New York, NY, USA, 2013. Association for Computing Machinery.

[29] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 287–300, New York, NY, USA, 2015. Association for Computing Machinery.

[30] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, page 275–284, New York, NY, USA, 2013. Association for Computing Machinery.

[31] V. J. Reddi, H. Yoon, and A. Knies. Two billion devices and counting. *IEEE Micro*, 38(1):6–21, 2018.

[32] Qualcomm Snapdragon 855. https://www.notebookcheck.net/Qualcomm-Snapdragon-855-SoC-Benchmarks-and-Specs.375436.0.html.

[33] High Performance Universal Flash Storage (UFS) Solutions. https://www.samsung.com/semiconductor/global.semi.static/White_Paper_Samsung_UFS_Card_1806.pdf.

[34] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 113–126, New York, NY, USA, 2012. Association for Computing Machinery.

[35] X. Zhu, D. Liu, K. Zhong, Jinting Ren, and T. Li. Smartswap: High-performance and user experience friendly swapping in mobile systems. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.