ICE: Collaborating Memory and Process Management for User Experience on Resource-limited Mobile Devices

Changlong Li[1,2], Yu Liang*[2], Rachata Ausavarungnirun[3], Zongwei Zhu[4], Liang Shi[1], Chun Jason Xue[2]
[1] School of Computer Science and Technology, East China Normal University
[2] Department of Computer Science, City University of Hong Kong
[3] TGGS, King Mongkut's University of Technology North
[4] Department of Computer Science, University of Science and Technology of China

Abstract

Mobile devices with limited resources are prevalent as they have a relatively low price. Providing a good user experience with limited resources has been a big challenge. This paper found that foreground applications are often unexpectedly interfered by background applications' memory activities. Improving user experience on resource-limited mobile devices calls for a strong collaboration between memory and process management. This paper proposes a framework, Ice, to optimize the user experience on resource-limited mobile devices. With Ice, processes that will cause frequent refaults in the background are identified and frozen accordingly. The frozen application will be thawed when memory condition allows. Evaluation of resource-limited mobile devices demonstrates that the user experience is effectively improved with Ice. Specifically, Ice boosts the frame rate by 1.57x on average over the state-of-the-art.

CCS Concepts: • **Software and its engineering → Main memory**; **Scheduling**; Embedded software; • **Computer systems organization → Processors and memory architectures**; *Embedded software.*

Keywords: Memory Management, Process Freezing, User Experience, Mobile Device

1 Introduction

Mobile devices now play an important role in our daily lives. Among them, low-end and mid-range devices with relatively low prices and limited resources are prevalent in the market [3, 26]. More than one billion of these low/mid-range smartphones are in use today around the world [60]. For these resource-limited devices, providing a good user experience has been a big challenge for smartphone vendors. This paper takes a collaborating memory and process management approach to optimize the user experience for resource-limited mobile devices.

In mobile systems, when an application runs in the foreground, there are often applications alive in the background. Most mobile phone users do not clean up these background applications, which consume a significant amount of memory resources. When the memory is exhausted, mobile systems inherent the design principle of Linux that reclaims pages. Many prior works are proposed to optimize the memory reclaiming algorithm [28, 37, 39, 41, 42, 56]. For instance, in the Linux kernel, the LRU (least-recently-used [22]) algorithm is adopted, and page activity is defined based on the accessing history. SmartSwap [65] performs page reclaiming based on in-depth analysis of user behaviors. Acclaim [45] proposed a foreground-aware page eviction scheme so that pages belonging to the foreground application will not be reclaimed. With these solutions, inactive pages are efficiently identified and reclaimed. In this paper, we show the memory activities by background processes often negatively impact foreground process, which has not been fully considered by existing solutions [28, 37, 39, 41, 42, 45, 56, 65]. As a result, frame dropping is observed when there are background applications in resource-limited mobile devices.

Our study on four typical scenarios (e.g., video call, short-form video, screen scrolling, and mobile game) indicates that the frame rate often drops below 30 frame-per-second (fps) when there are multiple background applications. Such frame rate is not enough to provide a good user experience [50]. By collecting information from real-life smartphones, we provide four key observations. First, we observe that CPU contention is *not* the main reason. Second, while background applications do not use much CPU, they often run tasks that need to access memory pages [40][31]. As a result, we observe a large number of reclaimed pages are demanded soon in the background, which causes frequent refaults. Third, memory reclaiming affects the foreground application since

*Corresponding authors: Yu Liang, Email: yliang22-c@my.cityu.edu.hk.

EuroSys '23, May 9–12, 2023, Rome, Italy
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567497

priority inversion [30], and refault-induced memory thrashing amplifies such effects. Fourth, the problem in memory management is now actually caused by improper process scheduling. Based on these observations, we believe that a strong collaboration between memory management and process scheduling is a necessity for the user experience of resource-limited mobile systems.

In this paper, we propose Ice, which is designed to collaborate memory and process management in resource-limited mobile systems for an optimized user experience. The key idea of Ice is to selectively inhibit the background processes that are likely to cause page refaults. The process-inhibiting is driven by page refault. However, to realize the above design, there are several challenges. First, the cost of refault event tracking should be small. Second, the inhibiting target and intensity should be carefully controlled to ensure system stability and application robustness. Finally, Ice should be compatible with existing mechanisms in systems. To tackle these challenges, we propose two schemes: refault-driven process freezing (RPF) and memory-aware dynamic thawing (MDT). Specifically, process inhibiting is triggered by the detected refault event. In addition, Ice thaws the frozen applications periodically. The freezing and thawing cycles are dynamically tuned based on the memory pressure.

In summary, this paper makes the following contributions:

- Based on the data collected from real-life users, this paper reveals that the current user experience on mobile devices suffers from existing memory refaults with over-active background processes;
- We propose Ice, which is a collaborative memory and process management framework for mobile devices. To the best of our knowledge, this is the first effort to solve the ineffective memory reclaiming problem with the help of process management;
- Ice is implemented on real mobile devices. Experimental results show that the user experience is significantly improved with Ice. Specifically, Ice improves the frame rate by 1.57x on average over the state-of-the-art mechanisms [12, 22, 45, 59].

2 Background and Motivation

2.1 Memory Reclaim and Refault

Unlike server applications, mobile applications are often cached in the background for fast switching and state restoration [29]. These cached applications consume limited memory resources and the mobile system needs to perform page reclaiming when the memory is fully consumed by these applications. In Andriod, when the free memory is lower than a threshold (low-watermark [35]), a lightweight kernel thread, *kswapd*, will be woken up to reclaim memory. There are two types of pages that are allowed to be reclaimed: anonymous pages and file-backed pages. Anonymous pages hold the runtime data generated and used by processes. File-backed pages are correlated to segments of files at the secondary storage. When performing reclamation, the anonymous pages will be stored in a compressed memory area (ZRAM [23]), the dirty (modified) file-backed pages will be written back, and the clean (unmodified) file-backed pages will be discarded directly.

Based on the page reclaim algorithm (e.g., LRU), inactive pages are selected as candidates for reclaiming. Reclaimed pages are transferred to *bio* instance, which represents an in-flight block I/O request in the kernel [7]. The instances are delivered to the block layer and further handled by the driver of the target block device. For anonymous pages, the ZRAM driver will be woken up to compress the content of the *bio* and store the compressed data on a virtual RAM disk. For file-backed pages, the UFS/eMMC driver will be woken up to send a dispatching command to the flash device and store the data in the secondary storage. When the reclaimed page is demanded (so-called *page refault*), these demanded pages is moved back to the main memory by triggering a page fault interrupt. Earlier reclaimed pages will be decompressed or read from the storage. The time between a page being evicted out and faulted in, which is called $refault\ distance$ [20] in the Linux community, is hoped to be as long as possible.

2.2 Effect on User Experience

Page refaults can be classified into two types: foreground (FG) refaults and background (BG) refaults. The former occurs when earlier reclaimed pages are demanded by the FG application, while the latter occurs when demanded by BG processes. In this section, we will show that the user experience seriously deteriorated when refaults frequently occur in BG.

2.2.1 Workloads for Analysis.

To understand the negative impact, we conducted experiments on a HUAWEI P20 smartphone with four typical real-life scenarios.

Scenario A: Video Call. In this scenario, we use WhatsApp and initiate a video call with a remote user. The picture on the screen changes with the expression and action of the remote user.

Scenario B: Short-Form Video Switching. Nowadays, people are spending more and more time on short-form videos when using smartphones [11]. We evaluated this scenario with TikTok. Specifically, we play the videos recommended by the application, then switch to the next. We repeat the playing and switching operations.

Scenario C: Screen Scrolling. Screen scrolling is one of the most common actions in smartphone usage. We browse the *timeline* of Facebook to explore the user experience of screen scrolling.

Scenario D: Mobile Game. Users have a high demand for the fluency of mobile games. We use PUBG Mobile, a popular

Page refault is defined as the case where the page fault happens on the previously reclaimed page [46].

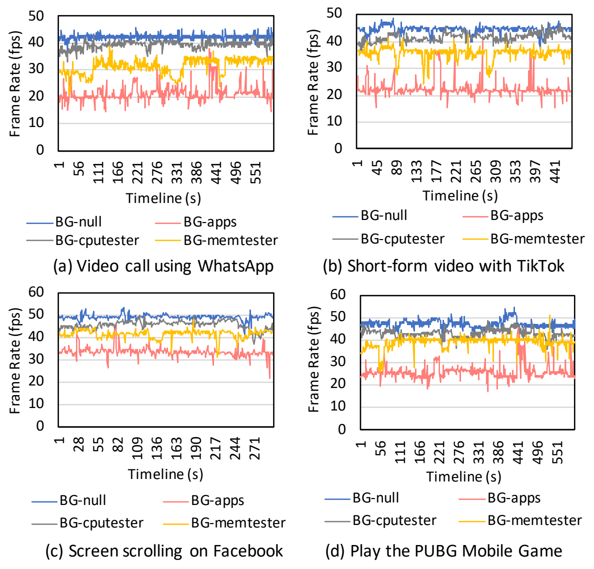


Figure 1. Frame rate statistic of the four evaluated scenarios. The frame rendering performance is significantly lower when multiple applications are cached in the BG.

real-time battle royale game in smartphones, to evaluate the user experience in this scenario.

2.2.2 Effect on Frame Rate. The deterioration of user experience such as a jerky screen, referred to as 'jank', often occurs when BG applications exhaust the memory. To quantify the user experience, we analyze the frame rate in the evaluation. It depicts whether a user can smoothly interact with the screen. Generally, more frames rendered per second provide a better user experience. FPS (frames-per-second) has been widely adopted as the metric of frame rate.

For each scenario, we configure BG applications with two cases: *BG-null* and *BG-apps*. "BG-null" refers to the case where the target applications (e.g., WhatsApp, TikTok, Facebook, and PUGB Mobile) run without applications cached in the BG. "BG-apps" refers to the case where eight applications are cached in the BG before running the target applications. We select a period from each of the four scenarios as a sample and the FPS of the foreground application is presented in Figure 1. X-axis represents the timeline and Y-axis represents the FPS per second. It shows that FPS is significantly degraded with applications in the BG. For example, when having a video call without caching applications (blue line in Figure 1(a)), the frame rate is 42.2fps on average. However, with eight applications cached in the BG, the video call becomes jerky. The cached applications are Twitter, Amazon, Angrybird, Youtube, Skype, Chrome, Uber, and Google Map. In this case, the FPS (red line in Figure 1(a)) is reduced by 51.7% on average. Even though differences depend on the application characteristics, noticeable FPS degradation is observed in all scenarios.

2.2.3 Root Cause Analysis. The FPS of the FG application could be affected by BG applications' CPU contention or memory contention. In this subsection, we examine both possibilities.

(1) CPU contention. BG applications introduce additional CPU consumption. To explore how much CPU resource is consumed by BG applications, we conducted the following evaluation: We monitor the CPU utilization with no application in the BG and FG. The CPU utilization is 43%, on average. The Linux kernel and Android framework's tasks take up the CPU resources. Then, we cache N applications in the BG. The BG applications stay in the BG for ten seconds with no FG application. To avoid bias, BG applications are selected randomly from 20 popular applications (detailed in Section 5.1). Each evaluation is conducted for ten rounds and the average is shown in Table 1. The average CPU utilization is increased to 55% when N equals 8, and the peak value is 69%. CPU information is collected by perfetto [61]. The above evaluation indicates that BG applications are not CPU intensive in general.

Table 1. CPU utilization with $N(0 \sim 8)$ apps in the BG.

Num. of BG apps	CPU Utilization	
(No FG app)	Average	Peak
0	43%	52%
2	46%	58%
4	47%	63%
6	51%	67%
8	55%	69%

To understand whether the FPS is affected by CPU, we configure the BG applications with an additional case: *BG-cputester.* In this case, BG applications are replaced by cputester, a self-developed tool that occupies CPU resources but does not introduce big memory pressure. The cputester occupies 20% CPU utilization in the BG, which is similar to the measured CPU consumption of BG applications. Also as shown in Figure 1, compared to the BG-null case, the FPS is only reduced by 6.3% on average with cputester in the BG. This indicates that the steep FPS reduction in the BG-apps case is not caused by CPU consumption of BG applications .

(2) Memory contention. In the BG-apps case, the memory is almost exhausted (more than 90% of the memory space is unavailable) by BG applications, so memory reclaiming occurs when running the target application in the FG. The FG application may wait for the completion of the reclaiming process which is *non-preemptive* [30], even though the former has a higher priority. To analyze the memory factor separately, we replace the BG applications with *memtester* [58], an open-source tool that occupies memory but does not

Note that Android is now differentiating the FG application from BG applications. For example, the FG application is always allowed to use CPU cores exclusively while BG applications must share the cores [14].

induce big CPU consumption. The size of occupied memory is similar to the BG-apps case.

Figure 1 shows that the FPS is reduced by 27.8% on average in the BG-memtester case, compared to the BG-null case. We traced the process of frame rendering (e.g., `animation`, `measure`, `layout`, and `draw`) using Systrace [19]. When the available memory is smaller than the watermark threshold, a high number of frame rendering tasks are blocked by memory reclaiming tasks. As a result, the FG application with a high priority is blocked by the BG processes with a lower priority. Such priority inversion is one of the main reasons for FPS degradation [30]. It demonstrates that memory contention can significantly affect the FG application.

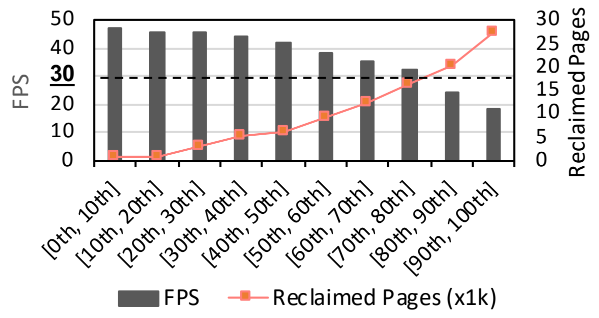
(3) BG refault. When the memory is filled by memtester (yellow line in the figure) instead of mobile applications (red line in the figure), the FPS degradation compared to BG-null is much smaller. Take Figure 1(a) as an example. We can see that the FPS in BG-memtester case stays at a low level (around 25fps) for a much shorter time. The frame rendering tasks still have an opportunity to run efficiently when enough memory space has been reclaimed. On the contrary, in the BG-apps case, the FG application constantly suffers interference.

One big difference between normal applications and the memtester is that a large number of reclaimed pages are demanded again in the former case. By checking the processes causing page refaults, it can be found that most of the refaults occur in the BG. The refaults result in many invalid memory-reclaiming operations. Hence, more memory reclaiming operations are detected in the BG-apps case. Specifically, compared to the BG-memtester case, 84.4% more memory reclaiming operations are detected in the BG-apps case (Figure 2(a)). BG applications are rarely CPU-intensive tasks, but their execution always needs to access memory pages [40][31]. As a result, BG applications *indirectly* but *constantly* interfere with the FG application by inducing memory-related activities.

To understand such effects, the BG-apps case is further analyzed. Specifically, we split the collected information from the four-scenario evaluations into time slices. Each time slice contains thirty seconds of frame rendering information. Then, these time slices are sorted according to the BG refault numbers. Based on that, the frame rates are compared from [0th, 10th]-percentile to [90th, 100th]-percentile. Figure 2(b) illustrates that the frame rate seriously deteriorated in the time window that BG refaults frequently occur. Specifically, the frame rate of the target applications is 47.2fps on average when refault rarely occurs ([0th, 10th]-percentile). It is reduced by 60.6% when suffering frequent BG refaults ([90th, 100th]-percentile). In addition, more pages are reclaimed in a fix-sized time window when more BG refaults occur. It illustrates that frequent BG refaults can induce more memory reclaims, thus interfering the FPS of FG application indirectly. Usually, frame dropping will typically occur with FPS lower than 30 [50]. So mobile users often suffer not-smooth interaction on low/mid-range devices, when suffering such steep FPS reduction.

Case	**Reclaim**	**Refault**
BG-null	76	3
BG-memtester	55,637	1,351
BG-apps	102,581	38,924

(a) The number of reclaimed and refaulted pages in total



(b) Correlation between frame rate and memory operations

Figure 2. Frame rate analysis. Frame rate degraded when suffering frequent refaults.

BG refaults amplify the interference to the FG application through three sources. First, more memory reclaiming operations are required to release enough space. These system activities could block the frame rendering task required by the FG application. Second, BG refault increases the I/O pressure. On one hand, file-backed pages may need to be read from the flash storage when suffering refault. On the other hand, the system could write back more file-backed pages to make room for the refaulted pages. Third, even though BG applications do not consume too much CPU resources, their activities in the BG still create memory interference to the FG applications.

3 BG Refault Analysis

In this section, we extend our analysis to illustrate how BG refault is a widespread problem in mobile systems. Then, we analyze the cause and show the opportunity to redesign the memory and process management in mobile systems.

3.1 BG Refault Study from User's Data

To understand page refaults in real usage scenarios, we provide eight Android smartphones to volunteers for daily usage. The details of the smartphones are shown in Table 2. We instrument the Android source codes on the smartphones to collect the required system information (with the user's authorization). Our instrumentation includes information on the number of refault pages, the number of reclaimed pages,

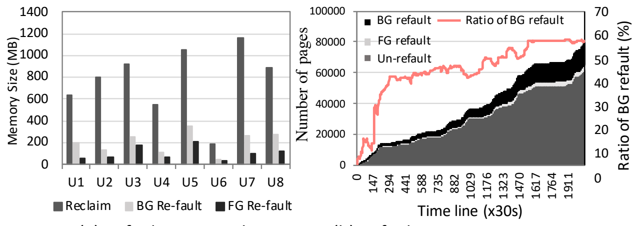
The volunteers are 18-60 years old.

the process each refaulted page belongs to, the process information of the FG application, and the timestamp of each page reclaiming/refaulting operation. Volunteers install and use applications following their habits. The information in the smartphones is collected over one month.

Table 2. Details of the tested smartphones.

Device	CPU	RAM	System	Volunteers
P20	Kirin 970	6GB	Android 9	User-1, User-2
P40	Kirin 990	8GB	Android 10	User-3, User-4
Pixel3	QSD 845	4GB	Android 10	User-5, User-6
Pixel4	QSD 855	6GB	Android 10	User-7, User-8

Figure 3(a) shows the number of reclaimed and refaulted pages per day for all eight smartphone users. It indicates that 39% of evicted pages are moved back, on average. In addition, more than 60% of refaults are caused by BG processes, instead of FG processes. The memory state over time in each device is also monitored. The cumulative numbers of evicted and refaulted pages are recorded. The information is collected every thirty seconds. Take HUAWEI P20 as an example. Figure 3(b) shows several observations. First, the number of refaulted pages increased when more pages were reclaimed. Second, the refault ratio is low at the beginning but finally fluctuated at a high level (38%). Third, the main contribution of the high refault ratio is BG refault. As presented in the figure, 65% refaults are caused by BG processes. The system wastes more available resources than necessary since many reclaiming operations are invalid. This phenomenon happens on all evaluated smartphones.



(a) Refault ratio analysis (b) Refault ratio on P20 over time

Figure 3. Page refault study: The refault ratio is high and most of the refaults are caused by BG processes.

3.2 Sources of BG Refault

A well-known source of BG refault is runtime garbage collection (GC) [29][40], which is responsible to collect the unused memory space in a managed language, such as Java. Android applications are running on Android runtime (Dalvik before Android 4.4 and ART on the later versions), which enables the GC feature. It may be triggered not only when allocating new objects on the heap but also when the application is in the BG. When GC runs, reclaimed pages may be moved back. In this section, we show that GC is not the only source of BG refault.

To explore the sources, we conduct a study with 40 popular applications. The basic method is to reclaim all pages of the application manually. Then we trace which pages are refaulted back and which processes cause the refaults. Specifically, we launch and run the tested application, then switch it to the BG. After that, we reclaim all file-backed and anonymous pages of the application by enabling the per-process reclaim feature in the Linux kernel [21]. The numbers and types of refaulted pages within thirty seconds are detected with the command `$cat /proc/{pid}/status`.

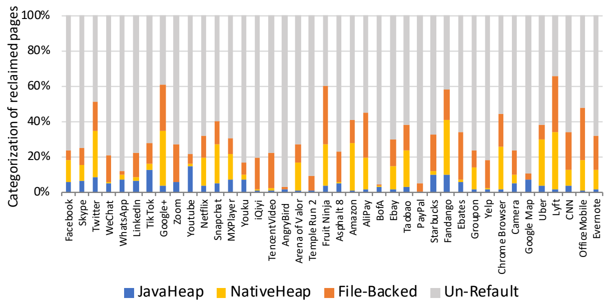


Figure 4. Categorization of reclaimed pages. Among the reclaimed pages of each application, both anonymous and file-backed pages suffer BG refault.

Figure 4 shows the categorization of refaulted pages. More than 30% of reclaimed pages are moved back in total. Among them, 48.6% are file-backed pages and 51.4% are anonymous pages. Furthermore, more than 56.6% of the refaulted anonymous pages are maintained in the native heap, and 43.4% are maintained in the Java heap. The native heap is allocated by using the underlying `malloc` and `free` mechanisms of the operating system. The Java heap contains the instances of Java objects. Since the tested application has not switched to the FG, all detected refaults occur in the BG. The proportion may be different when the application version or smartphone platform changes. But the phenomenon of frequent BG refault was widely observed.

Page refaults in the Java heap are mainly caused by the GC thread [53]. However, there are still 77% refaults observed, even when we disabled the "idle runtime GC" feature. We traced the tasks run on each core and found that *the BG applications are not as quiet as expected.* On each tested platform, we first obtain the process that runs on cores when no applications run in either BG or FG. In this case, more than 90% of the CPU usage time comes from system service tasks (e.g., *Binder*, *HeapTaskDaemon*, and *kworker*). Then we trace the processes that occupied CPU time after caching eight applications in the BG. No application runs in the FG. We conduct such an evaluation for ten rounds. The BG applications (detailed in Table 3) are selected randomly in each round. As evaluated, more than half (58%) of BG applications are observed to run on CPUs, including the main thread of Facebook, Twitter, WeChat, and TikTok. Further, more Android services are executed. For example, the location tracking (*gms.location.LocationListener*) thread. In addition, any application that has been granted location/microphone access could be running in the BG and making records. [1][8][2].

In addition, investigation illustrates that many applications were not developed in a system-friendly approach [31]. They may consume constrained resources in a wasteful manner. Some of them run in the BG to ensure its optimal performance and some even have bugs in their design. For example, Facebook had a buggy release [33] that left the application doing nothing but stay awake and running in the BG.

4 Ice Design

4.1 Overview

Ice is beneficial for reducing refault. As shown in Figure 5, Ice bridges the memory management and process management subsystem. The basic idea of Ice is to freeze the active process that will cause page refault and thaw them periodically. Two components are proposed to realize the features: refault-driven process freezing (**RPF**, detailed in Section 4.2) and memory-aware dynamic thawing (**MDT**, detailed in Section 4.3). In addition, a Ice daemon is maintained to communicate with existing system modules.

The control flow of process freezing and thawing are shown in (❶ - ❸) and (④ - ⑤), respectively. (1) Ice detects the refault event in kernel space. (2) The process ID (pid_i) that induces page refault is obtained and delivered to the RPF component. (3) Ice finally determines the freezing targets (pid_i, ..., pid_j). Then sending commands to the process management subsystem to inhibit the activity of selected processes. In addition, the MDT component monitors memory pressure (4) and intermittently thaws the processes (5). The thawing cycle is dynamically tuned based on the memory pressure.

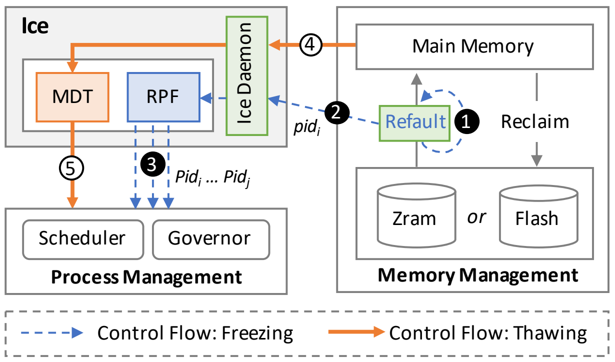


Figure 5. Ice overview. Ice detects the refault event and transfers the ID of the process causing refault to the RPF component for freezing (see the dashed lines). The MDT component monitors the memory pressure and thaws the frozen processes periodically (see the thick lines).

With RPF and MDT components, the following challenges are tackled in the design. First, the refault event can be monitored with low overhead and respond quickly. Second, the freezing target and intensity are intelligently controlled. Finally, the proposed solution is compatible with existing mechanisms in the system. Ice has been deployed on multiple Android smartphones without invasive modification in mobile applications or hardware infrastructure.

4.2 Refault-driven Process Freezing

RPF aims to freeze the active BG processes in case of sustained refault in the BG. The process freezing [36] is a mechanism by which processes are forced to hibernate. Once a process is frozen, it will never be executed before thawing, and thus will not induce refault. In our design, RPF selectively freezes the BG processes that cause page refault, instead of aggressively freezing all BG applications. This is because it always takes a longer time to switch a frozen application to the FG. To minimize the overhead, RPF only freezes the BG applications that may cause memory thrashing. Our experiment on real platforms (Section 6.2.1) will show that selective freezing is enough to inhibit the reclaim/refault behaviors in the BG.

Therefore, how identifying the BG process that will cause frequent refault is a critical issue. One approach is to predict based on historical information through machine learning, like Markov model [62, 64]. However, the overhead to maintain the machine learning model is high. A lightweight refault identification method is required. This paper finds that a process often demands multiple pages at a time. The source of multiple adjacent refaults is often the same application. Ice freezes the BG process once the first refault is identified. An event-driven freezing scheme is proposed.

4.2.1 Refault Event Handling. RPF follows the event-condition-active (ECA) rule [9], which can be described as: fired *events* that satisfy the *conditions* will trigger *actions* to execute. Specifically, RPF detects the refault event in the kernel space. It then determines whether the process causing the refault is a BG process and if it is allowed to be frozen. Processes that are not allowed to be frozen, like kernel or service processes, are sifted. Then, the ID of each 'freezable' process is delivered to an application-grain freezing module for further processing.

Refault identification. As illustrated in the right part of Figure 6, the refault event is detected based on the page table entry (PTE). If a page has been evicted to the storage, one flag bit in PTE will be modified. Specifically, bit-0 of the PTE is reserved for the `_PAGE_PRESENT` flag. When a page fault occurs, the system will access the PTE to obtain the page address. By checking this flag bit when suffering page fault, RPF can identify whether the required page was previously evicted or not. In this way, the refault event can be detected. The modern Linux kernel has already provided an interface to obtain the refault-related information (`shadow entry` [25]).

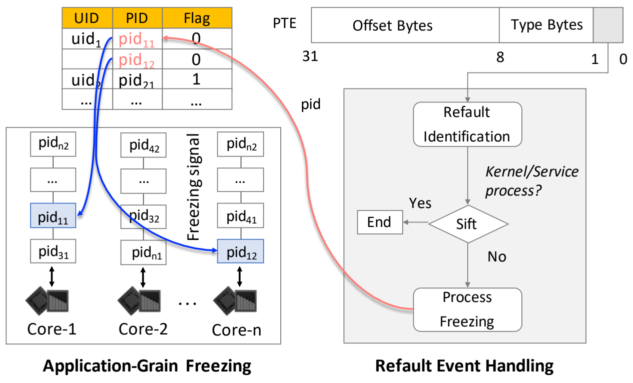


Figure 6. RPF overview. It consists of two core components: refault event handling and application-grain freezing.

For simplicity, we apply this interface to identify whether refault events occur.

Process selection. When a page refault event is detected, RPF identifies which process the page belongs to. Then it judges whether the process is freezable. Note that some processes are not allowed to be frozen. RPF first checks the process type. Kernel processes (e.g., `kswapd`, `kworker`) and Android services (e.g., UI thread, GC thread) are identified and will not be frozen. Here, a technical challenge is to accurately identify which process caused the refault. In the Linux kernel, page refaults start from the page fault interruption. It is easy to obtain the page table and the virtual address of this fault (by page fault handler) in the Linux kernel, and we can determine which process caused this page refault. In this way, the relationship between the memory page and the process is identified.

Response to the refault event. After process sifting, the process that causes refault will be handled. RPF tries to inhibit the process activity by freezing. Refault events can be detected in nearly real-time, and RPF can respond immediately as they occur. We realized that simply freezing a single process may affect the corresponding mobile application as a whole. Therefore, in this paper, freezing is conducted in the application granularity.

4.2.2 Application-Grain Freezing.

RPF conducts freezing in application granularity. It maintains a mapping table between application and process. The former is represented by the unique ID (UID). Once installed, the UID is fixed. As investigated, each application generates several processes when running. The mapping table is updated when an application is installed, deleted, or launched. Based on the mapping table, RPF can obtain the process information during the life cycle of mobile applications. When RPF tries to freeze a process, it checks the mapping table, identifies which application the process belongs to, and then sends freezing signals to the target processes. The processes react to the signals by calling function `try_to_freeze()` [24], which makes them hibernate until receiving the signal for thawing. In this way, the BG applications that cause page refault can be frozen.

In the design, the mapping table is maintained in the kernel space so that RPF can index the table quickly. There is no communication between the kernel and user space during the freezing process. The only cross-space communication happens when updating the mapping table. Specifically, we collect the application information from the Android framework and deliver it to the kernel through the proc file system. A function is predefined in this file system through `file_operations()`. When writing protocol string to the `/proc/{pid}/ice-mp` node, this function will be called. This function receives the application information (e.g., UID, PID, state) and updates the mapping table. In the life cycle of a mobile application, the above information is seldom changed, compared to the frequent process indexing operations of RPF, so such cross-space communication is acceptable.

RPF freezes processes in application granularity. It is because one application always maintains several processes at the same time. We observed these processes belonging to the same application always depend on or communicate with each other. By freezing the processes together, application robustness is enhanced. In this work, we freeze applications rather than kill them straightforwardly. Because the frozen application can still be fast switched (hot launched) and the history-state can restore.

4.3 Memory-aware Dynamic Thawing

This paper proposes MDT, a memory-aware dynamic thawing mechanism, to give frozen applications a chance to run. How long the applications should be frozen in the BG should be carefully calculated so that its impact on memory is minimized. The freezing intensity is tuned dynamically. Theoretically, the intensity should increase if the memory pressure is high and reduce when the pressure is alleviated. Here, freezing intensity is quantified with the ratio of *freezing duration* and *thawing duration*. The higher the freezing intensity is, the higher the probability is that applications are frozen in the BG.

As illustrated in Figure 7, MDT maintains a heartbeat in the system. Each heartbeat epoch is divided into two periods: the freezing period and the thawing period. At the beginning of each epoch, the selected applications are frozen for E_f seconds. Then thaw for E_t seconds. The epoch length is tuned based on the memory state. When memory pressure increases, the freezing period should be lengthened; Otherwise, the freezing period is shortened. In $epoch_i$, the target processes are frozen for $E_f(i)$ seconds and thawed for $E_t(i)$ seconds. After that, MDT detects the size of current available memory and update $E_f(i)$. Then enter the next

In general, several applications can reside in memory after execution. These applications are hot launched without needing to build a new process or generate application activities. [41].

epoch $E_f(i+1)$. MDT sends signals to the target processes at the beginning of each freezing and thawing period. MDT maintains only one heartbeat mechanism no matter how many applications are installed.

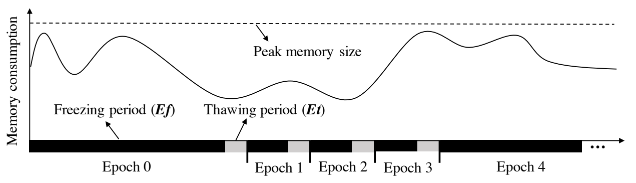


Figure 7. Memory-aware heartbeat.

Suppose that an application triggers a page refault and needs to be frozen at time T. If T occurs in the freezing period, where $0 < T \leq E_f$ in an epoch, this application will be frozen instantly and thawed at time E_f, then allowed to run in the following E_f seconds until the next epoch comes. If T occurs in the thawing period, this application will also be frozen instantly but will not be thawed until the thawing period of the next epoch. MDT dynamically tunes the freezing intensity by changing the ratio of E_f to E_t. This ratio (R) is tuned based on the memory watermark [35]. Specifically,

$$R = \frac{E_f}{E_t} = \delta \times 2^{\lceil \frac{H_{wm}}{S_{am}} \rceil} \tag{1}$$

Here, H_{wm} stands for the high watermark, and S_{am} stands for the size of available memory. δ is a weight coefficient. In a given system, the high watermark H_{wm} is determined at the system initialization phase. So R increases when the size of available memory S_{am} is small. As a result, the freezing intensity is enhanced with the increase of memory pressure. To easily control the freezing intensity, E_t is set as a unit time, one second, by default. So the length of an epoch can be tuned by simply changing the value of E_f.

Before adopting the freezing/thawing approach, this paper explored various schemes, such as priority reduction. However, our evaluation shows that even the process with the lowest priority can still run frequently. The reduction of page refaults is not significant. What is worse, the opportunity to suffer page refaults is unpredictable. By performing application freezing and thawing, the refaults in the BG can be strictly constrained and managed.

4.4 Implementation and Discussions

Whitelist for safety. To ensure that the freezing operation is user imperceptible, Ice exploits a whitelist approach. The basic idea is to add perceptible applications to the whitelist. These applications are not allowed to be frozen. In practice, Android now has a mechanism to identify which applications are user perceptible [13]. First, the application that is calling interaction activity (call the `onResume()` interface) is identified as FG application by the Android Framework. This application is critical and should not be frozen. In addition, some applications in the BG are user perceptible. For example, the application which is playing music or downloading files in the BG should not be frozen. Android identifies these applications by checking whether they are calling the `startService()` interface. Android provides a priority score mechanism for these applications. Ice can obtain the score (adj value in Android [13]) of each process by `#echo /proc/pid/oom_score_adj`. In the system, the score of FG processes is set to 0 and perceptible BG applications are set to 200 by default. Normal BG processes have a higher score. The applications with low process scores (≤200) are added to the whitelist.

We realize the whitelist by delivering the scores to the kernel space and recording them in the mapping table. The whitelist is updated when the scores are changed. For example, the downloading task in the BG is finished or the FG application is switched. In addition, the whitelist can be managed offline. If vendors wish to optimize some specific applications in the BG, for example, an antivirus malware tracker or the applications which need to receive calls or messages, they can put the UID of these applications into the whitelist in advance. When Ice determines the freezing target, it checks the whitelist to ensure that critical or perceptible applications will not be frozen.

Thawing on launch. The frozen applications will not respond to the user input even when it switches to the FG. To address this problem, Ice detects application switching behavior. When a frozen application is switched to the FG, Ice will thaw it before displaying the application on the screen. The thawing operation is asynchronous and will not be interrupted by the MDT scheme.

The above optimizations ensure that Ice's operation is user imperceptible. The source code of the Linux kernel is modified to enable Ice. Ice introduces new features in the system with no invasive modification in mobile applications. To enable Ice, people just need to recompile the Android project and replace the `system.img` and `boot.img` in *fastboot* mode. Ice can be implemented to other Android devices with diverse versions and ease of maintenance.

5 Experiment Methodology

5.1 Platform and Workloads

This paper evaluates Ice against the original Linux memory/process management schemes and the current solutions in mobile systems. The experiments are performed on two smartphones from different manufacturers: Google Pixel3 and HUAWEI P20, which represent low-end and mid-range devices, respectively. The former is equipped with Qualcomm Snapdragon845 core, 4GB DDR4 RAM, and 64GB eMMC5.1 Flash. Android 10.0 (r41) is deployed on the device. The latter is equipped with HiSilicon Kirin970 core, 6GB DDR4 RAM, and 64GB UFS2.1 Flash. Android 9.0 is deployed

Table 3. Applications that are used in the evaluation.

Category	Application
Social	Facebook, Skype, Twitter, WeChat, WhatsApp
Multi-Media	Youtube, Netflix, TikTok
Game	AngryBird, Arena of Valor, PUBG Mobile
E-Commerce	Amazon, PayPal, AliPay, eBay, Yelp
Utility	Chrome Browser, Camera, Uber, Google Map

on the device. A recent study shows that today's mobile users often run more than ten applications [45]. So we pre-install 20 top applications on the platforms to simulate daily usage. As shown in Table 3, the applications involved in all the evaluations are picked from the most popular ones of various categories on the Android market.

5.2 Evaluated schemes

Four schemes are evaluated for comparison: LRU [22] + CFS [12], UCSG [59], Acclaim [45], and the proposed Ice.

LRU+CFS. LRU is the default memory management scheme in the Linux kernel. With LRU, the least recently used pages are identified as *inactive* and advised to be reclaimed. CFS (completely fair scheduler) is the scheduler that is currently used in Android-based mobile devices. With this scheduler, FG and BG processes are treated in fair. LRU and CFS manage memory and process, respectively. They are implemented as the baseline for comparison.

UCSG. Different from the original process scheduling, UCSG declares that the FG application usually dominates the user's attention. It treats FG and BG processes differently and re-designs the priority scheme in process scheduling. With UCSG, processes belonging to the FG application are set with a higher priority.

Acclaim. Acclaim redesigns memory management in mobile systems. It is an FG-aware and size-sensitive reclaiming scheme. With Acclaim, pages belonging to the FG application are avoided to be reclaimed. Instead, pages belonging to the BG application prefer to be reclaimed even if their activity is higher than some of the FG pages. With Acclaim, FG refault was effectively reduced.

Ice. This is the proposed scheme. Different from previous approaches, Ice co-designs memory and process management. The goal of Ice is to reduce BG refaults.

5.3 Parameter Configurations

The parameters in the evaluation are listed in Table 4. S^g and S^h represent the size of ZRAM partition. These two parameters are set as 512MB and 1,024MB. They determine how many anonymous pages can be reclaimed at the maximum. The high-watermark (H^g_{wm} and H^h_{wm}) of the two smartphones are 256 and 1,024. The weight coefficient δ discussed

The low-watermark and min-watermark is 5/6 and 2/3 of the high-watermark, which follows the default configuration in the Linux kernel.

Table 4. Summary of parameters used by Ice.

Symbols	Semantics	Setting
S^g	Size of ZRAM partition in Pixel3	512MB
S^h	Size of ZRAM partition in P20	1,024MB
H^g_{wm}	High-watermark in Pixel3 system	256
H^h_{wm}	High-watermark in P20 system	1,024
δ	Weight coefficient of MDT strategy	8.0
Et	Epoch length to thaw an application	1 Second

in Section 4.3 is relative to the freezing intensity. This parameter is configured as 8.0 in the evaluation. E_t determines how long an application can run in each freezing epoch. For simplicity, we set this parameter as one second in default. The parameters are configurable.

6 Experimental Results and Analysis

6.1 Benefit on Frame Rate

As discussed in Section 2, it is better to draw more frames per second. In addition, if a frame failed to be rendered within 16.6ms (which is defined as an *interaction alert* by Systrace [19]), the display can become jerky or slow from the users' perspective. This is based on human eye sensitivity [49]. Therefore, we evaluate Ice's benefit on frame rate using the two metrics: FPS (frame per second) and RIA (ratio of interaction alert). The metrics are measured in the aforementioned scenarios: video call using WhatsApp, short-form video switching with TikTok, screen scrolling on Facebook, and game-playing of PUBG Mobile. Before running these applications, several applications are cached in the BG in advance. Then we run the test application in the FG and record the frame rate using Systrace. To avoid bias, each scenario is tested for ten rounds and the average is shown. For each evaluated application, we log in with the same user account and conduct a similar sequence of activities for ten rounds. In each round, we reboot the smartphone and re-select the BG applications from Table 3 randomly. All the evaluations are performed with a good Wi-Fi connection.

Figure 8 illustrates that Ice outperforms the other schemes when multiple applications are cached in the BG. For example, when having a video call (S-A) using WhatApp on Pixel3, the average frame rate with LRU+CFS, UCSG, and Acclaim is 25.4fps, 29.3fps, and 24.1fps, respectively. Usually, frame dropping will occur with FPS beneath 30 [50]. So the smooth user experience is affected. For a smoother user experience, especially for frame-sensitive scenarios, like AR/VR [47], higher FPS is required. We can see that the frame rate boosts to 37.2fps with Ice. The videos and games in the evaluation can be displayed more smoothly and fluently with Ice. In addition to FPS, Ice reduced the RIA. Taking PUBG Mobile

There are six BG applications cached in Pixel3 and eight BG applications cached in P20 to fully fill the memory. When caching more applications in the original system, LMK may be triggered.

as an example, when we play this battle royale game on P20, the RIA is up to 46%. It means almost half of the frames failed to be drawn within 16.6ms. This metric was reduced to 28% with Ice. The results prove that BG application induced frame rate degradation can be effectively alleviated.

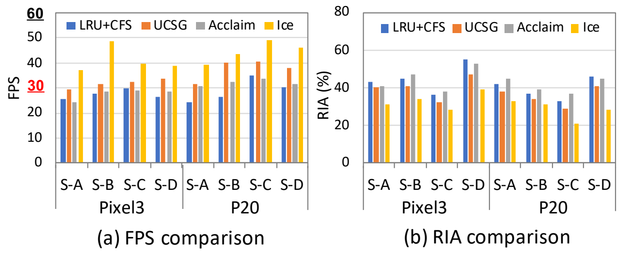


Figure 8. Frame rate comparison in four typical scenarios: video call (S-A), short-video switching (S-B), screen scrolling (S-C), and mobile game (S-D).

UCSG can also improve the frame rate. This is because the priority of BG processes is reduced. However, it does not focus on inhibiting the processes that cause page refault. Meanwhile, the processes with low priority can still be executed. Hence, the benefit is limited. As evaluated, the number of page refaults with UCSG reduced by 24.4% on average, compared to LRU+CFS. Acclaim aimed at reducing FG, instead of BG refaults. In some scenarios, like S-B on P20, Acclaim boosts the frame rate since FG refault is reduced. But in some scenarios, like S-C on Pixel3, the FPS degraded. This is because more pages belonging to BG applications were reclaimed with the foreground aware eviction (FAE) scheme of Acclaim. As a result, BG refaults have a higher possibility to occur in some scenarios with Acclaim. In summary, while existing solutions optimized the process and memory management, they cannot significantly reduce BG refault, which will be detailed in Section 6.2.1.

To understand the benefit of Ice, the four scenarios are further evaluated with the different number of applications cached in the BG. The average value of FPS and RIA in the four scenarios are calculated together. As the dashed lines show in Figure 9, the frame rendering performance degraded with more applications cached in the BG. In the "F" and "2B+F" cases, the FPS with and without Ice is similar (the small difference dues to measurement bias). Here, "F" means no application is cached in the BG and "2B+F" means two applications are cached. But the "6B+F" case on Pixel3 and "8B+F" case on P20 illustrates that the trend of the interference could be curbed when enabling Ice. Under the two cases, the FPS was enhanced by 1.57x and 1.44x, while the RIA was reduced by 32.7% and 34.6%, respectively. Evaluation results suggest that Ice can improve the user experience especially when memory space is almost exhausted by BG applications.

The game scenario data is obtained as the average of ten rounds.

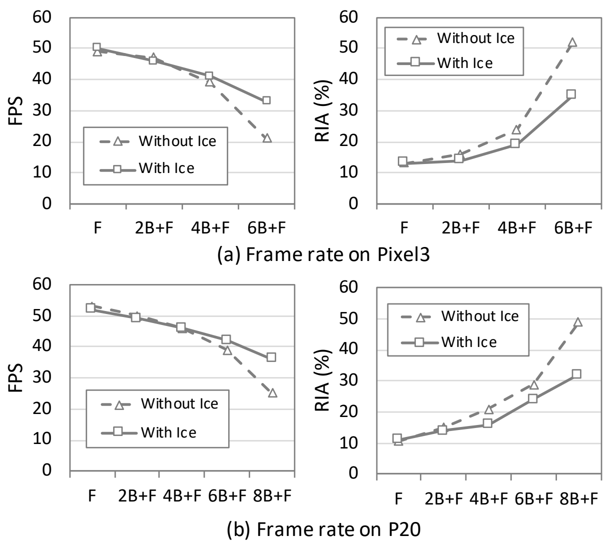


Figure 9. Frame rate comparison with various number of applications cached in the BG.

6.2 System Improvement Analysis

6.2.1 Reduction of refault and reclaim. The user experience benefit comes from the refault and reclaim reduction in the BG. To understand Ice's impact on the mobile system, we compare the number of reclaimed pages and the number of refaulted pages in the above evaluation. We only show the results on the P20 smartphone because of the space limitation. The number of refaults, as well as reclaims in the four scenarios, are evaluated separately. The comparison between LRU+CFS (L) and Ice (I) in Figure 10 indicates that fewer refaults occur with application freezing. As the dark histogram shows, the number of refaulted pages reduced by 42.1% in S-A, 44.4% in S-B, 57.6% in S-C, and 40.5% in S-D. In addition, the number of reclaimed pages was reduced. As the light histogram shows, the number of reclaimed pages with Ice is only 70.7% that of LRU+CFS, on average. Note that the reclaimed pages when using PUBG Mobile (S-D) are much more than the other three scenarios. This is because the mobile game is memory intensive. As measured, 100MB+ available memory is required to start a new round battle when playing this game. During the evaluation with Ice, only 4 BG applications on average are frozen. The inactive applications and the active applications that do not cause refault are not frozen by Ice. It demonstrates that Ice can effectively reduce reclaim and refault without needing to aggressively freeze all BG applications.

We can also see that the refault and reclaim reduction of Ice outperforms UCSG (U) and Acclaim (A) in the four scenarios. For UCSG, the refault and reclaim reduction are 51.7% and 53.9% of Ice. For Acclaim, the number of refaults even increased (by 4.3%) in some cases. Hence, we conclude that Ice is effective at reducing both reclaim and refault in the BG.

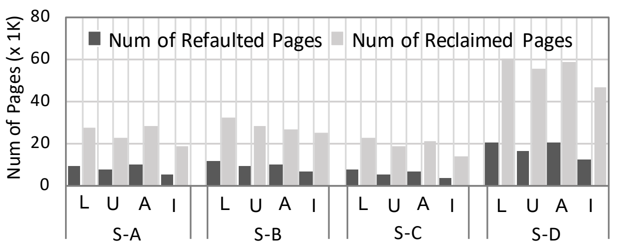


Figure 10. The number of refaulted and reclaimed pages with LRU+CFS (L), UCSG (U), Acclaim (A), and Ice (I).

Note that some commercial smartphones support the process freezing feature in power management, especially on modern high-end devices. We enabled the power-oriented freezing feature on the P20 smartphone and checked its effect on the refault and reclaim in memory management. As shown in Table 5, the power manager with freezing features has a positive impact on reducing refaults. Specifically, the total number of reclaimed and refaulted pages was reduced by 22.4% and 33.5% on average, compared to the LRU+CFS case. The inhibition of refault and reclaim is not as effective as Ice, because the power manager is not designed to dynamically and flexibly tune its freezing opportunity and target with *memory awareness*. For example, the power manager tries to freeze applications even when the memory pressure is low. Still, the freezing intensity is not changed when the memory pressure increases, even when suffering frequent refaults. Meanwhile, we observe that the power manager of some manufacturers' smartphones does not freeze when the device is charging. This design logic is reasonable in power management but is unfriendly to reducing BG refaults. The proposed Ice is orthogonal to the freezing feature of the power manager.

Table 5. The number of refaulted and reclaimed pages (x 1K) with process freezing (power manager vs. Ice).

Scenarios	Power manager		Ice	
	Refault	Reclaim	Refault	Reclaim
S-A	6.712	20.063	5.233	18.688
S-B	7.332	26.061	6.457	24.832
S-C	3.856	15.772	2.929	13.312
S-D	14.858	51.433	12.18	46.848

6.2.2 Reduction of I/O and CPU Pressure. The number of I/O requests is counted during the evaluation. It is easy to understand that the I/O size is reduced when active applications in the BG are frozen, in comparison to the unfrozen case. For fairness, we count the I/O amount over a long period (ten rounds of evaluation of the four scenarios, 5h+30min in total). During this period, the applications are not only frozen but also thawed with MDT many times. Evaluation results show that Ice did not introduce additional I/Os. Instead, the I/O size was reduced by 9.2% with Ice, compared to LRU+CFS. During the evaluation without Ice, we find that many file-backed pages are demanded by BG applications, then discarded during reclaim. Then demanded (refault) again. Such senseless I/Os are effectively avoided when enabling Ice.

In addition to the I/O pressure, Ice reduced the CPU pressure. The CPU utilization is 55.8% on average in the LRU+CFS case. This value was reduced to 47.3% with Ice. It shows that the waste of computing resources is effectively alleviated. On one hand, this is because some BG tasks are frozen, which reduces additional CPU consumption. On the other hand, the number of memory compression and decompression tasks decreased.

6.3 Impact on Application Launching

The impact of Ice on application launching is more complicated. On one hand, Ice has a negative impact on launching speed, as it takes time to thaw a frozen application before switching it to the FG. Moreover, when switching a frozen application to the FG, more pages may be demanded in the launching phase. On the other hand, Ice has a positive impact on launching speed since the interference of BG I/Os is alleviated and more CPU cycles can be allocated to the FG tasks. We measure the 20 applications to understand Ice's ultimate impact on launching speed.

We launch the applications for ten rounds repeatedly. Each application in the FG runs for 30 seconds. Then we switch it to the BG and startup the next one. In the ten-round evaluation, memory was quickly full filled, and page reclaim was triggered frequently. Note that some applications in the BG may be killed by Android LMK (low memory killer, [38]). This is close to real usage scenarios. During the evaluation, two tools are adopted: Android Debug Bridge (Adb [17]) and UI/Application Exerciser Monkey (Monkey [18]). Adb is a versatile command-line toolkit used to launch the applications, identify their launching styles, and record the launching latency. Monkey is adopted to simulate user behaviors. It generates pseudo-random streams of user events, such as clicking, touching, and scrolling. Based on the tools, applications are launched and executed automatically.

6.3.1 Impact on Launching Speed. During application launching, the time cost and launching style are obtained with the command `#adb am start`. Figure 11(a) shows that the average launching time with Ice decreased by 36.6%, in compared with LRU+CFS. The launching time is defined as the time span from the time point when a user taps on the shortcut icon of an application to the point the user is able to interact with the application. It indicates that the positive impact of Ice outperforms the negative impact. To further understand the reason, we analyze the time cost of cold launching and hot launching separately. Also as shown in the figure, it takes 4,237ms on average to cold launch an application with LRU+CFS, while the time cost is reduced by 28.8% when enabling Ice. Ice has a positive impact on cold launching, even though it does not accelerate the launching phase directly. With Ice, more resources can be allocated to the launching process and less interference occurs. In addition, the negative impact of application freezing does not apply to the cold launched application. So the cold launching speed improvement is significant.

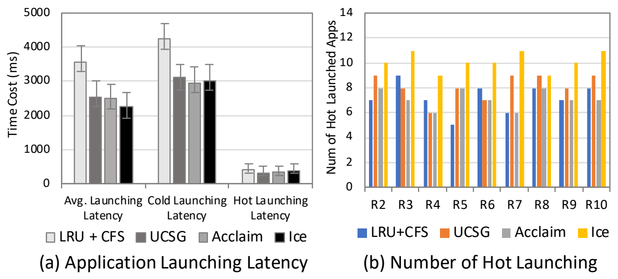


Figure 11. Impact of Ice on application launching. The results show that: (a) Ice's impact on hot launching speed is small and the cold launching speed can be improved. (b) More applications could be hot launched when enabling Ice.

The impact of hot launching varies among the applications. As evaluated, 47% of them slow down when enabling Ice. For others, the launching speed was enhanced. It depends on how serious are the applications affected by BG refaults. The latency increasing of hot launching are smaller than their standard deviations. It suggests that the benefit can offset the penalty in most cases, which is acceptable compared to the user experience benefit earned.

One potential penalty of Ice is that: if too many pages of a frozen application were reclaimed, its hot launching speed may be degraded seriously. So we further explore the worst case with Ice. Specifically, we reclaim all pages of an application and freeze this application in the BG manually. Then thawing this application and moving all required pages back during the launching phase. All the pre-installed apps are evaluated in this way. Evaluation results show that the hot launching latency in the worst case is 839ms on average, which is 1.98x of the hot launching latency with LRU+CFS. It demonstrates that Ice may slows down the hot launching speed. But we believe the worst case is acceptable. On one hand, such cases do not always occur in daily usage. On the other hand, the hot launching speed in the worst case is still much faster than cold launching. Note that this penalty can be further eliminated by using it in combination with application prediction [6, 52]. If a BG application is predicted as the next used application, Ice can thaw it ahead of time. Ice can also run together with launch-boosting schemes, like MARS[29] or ASAP [56], to further eliminate the penalty.

6.3.2 Ratio of Hot Launching.

The average launching speed is boosted not only due to the reduction of BG interference but also because more applications were hot launched. Figure 11(b) depicts the number of hot launched applications among the evaluated schemes. As introduced, the applications are repeatedly launched for ten rounds. In the first round, all applications are cold launched. In the following nine rounds, several applications are successfully cached and can be hot launched. This figure only shows the hot launching number from round 2 to round 10 because all applications are cold launched in the first round. The results illustrate that the application caching capability with Ice outperforms previous works. In general, only 7 or 8 applications are cached with LRU+CFS. When enabling Ice, 25% more applications could be hot launched. Ice is friendly to application caching because it alleviates the refault-induced memory pressure. As a result, the possibility to trigger LMK decreased. Meanwhile, the time point to trigger application killing has been delayed. More applications were switched to the FG before being killed.

6.4 Overhead Analysis

6.4.1 Memory Consumption.

Ice maintains a mapping table, incurring a space overhead in the memory. In the implementation, all applications installed by users are recorded in the mapping table. For a device with 20 installed applications, and each application consists of 3 processes, the memory consumption is 13.8KB at maximum, including 20×64B for UID, 20×3×64B for PID, 20×3×1B for freezing state, and 20×3×64B for priority score. Corresponding objects in the mapping table will be deleted if an application's life cycle ends. In this way, the scale of the mapping table is under control. In addition, we set an upper bound for the mapping table for safety. The upper bound is set to 32KB, which is enough to accommodate the information. In summary, with the optimizations in data structure management, the total memory consumption is at the ten-KB level. This is negligible compared to the total memory available on the smartphone.

6.4.2 Performance Overhead.

The performance overhead of Ice is broken down into three parts. First, it takes time to respond to the refault events. Specifically, Ice needs to obtain the process information, index the PID, and perform freezing. Ice performs these operations in an asynchronous approach so that the page accessing will not suspend. Second, the UID-PID mapping needs to be maintained. When the system is booted, Ice checks the configure file in Android to obtain the UIDs of the installed applications. The mapping table is initialized when the system is booted and is updated when installing or launching an application. When the life cycle of an application starts or ends, the process list in the mapping table needs to be updated. Meanwhile, when the freezing state changes, Ice needs to index the corresponding UID and update the corresponding state in the table. Since the table size is small, Ice maintain the table in memory. Thus, one table indexing can be completed at μs level. The time consumption is negligible in comparison with the launching latency. The third part is the freezing-induced performance overhead. As aforementioned, it takes only tens of milliseconds to thaw an application, which is acceptable compared with the launching time. In addition, Ice conducts a series of optimizations in the design to minimize the penalty. For example, only active applications are allowed to be frozen. The applications having no pages reclaimed, or the applications having pages reclaimed but no refaults, will not be frozen. The overhead is acceptable in comparison with the benefit.

7 Related Work

Memory management. Many efforts have been made to optimize the memory reclamation schemes [5, 28, 37, 39, 42, 43]. SmartSwap [65] predicts which applications are unlikely to be used and evicts pages from those applications ahead of time. FlashVM [55] focuses on changes to the virtual memory system to make effective use of available fast storage devices for memory reclaim. MARS [29] is designed to speed up application launching through flash-aware swapping. It employs a series of flash-aware techniques to speed up the launching speed. Based on that, SEAL [41] proposed a two-level swapping framework for user experience improvement. Modern mobile systems were effectively optimized by these schemes. However, the proposed BG refault issue has not been well concerned. Acclaim [45] focused on the FG application induced page refault problem. But based on our study, more than 65% page refault still occurs when applications are alive in the BG. The proposed scheme has a negative impact on BG refault. Marvin [40] found that runtime GC may cause BG refault and proposed a new memory management scheme in language runtime. With Marvin, the GC-induced page refault can be avoided. However, this paper has shown that more than 77% refaults cannot be effectively alleviated in this way. To address this issue, we propose a new freezing approach.

Process scheduling. There are a number of strategies to optimize the process [27, 44, 51, 54, 63]. Chang et al. [4] suggested that the FG process should be differentiated from BG processes because the former usually dominates the user's attention. In [57], the system identifies user-interactive processes at the framework level and then enables the kernel scheduler to selectively promote the priorities. Priority-based scheduling without aware memory insight cannot address the BG refault issue since there are great differences in the behavior characteristics of BG applications. These applications should be treated differently in terms of the refault factor. Energy managers that shipped with some smartphones support process freezing [15][34][48]. However, these features focused on energy saving. They tend to freeze the BG processes that consume a lot of energy, but cannot restrict refaults accurately and timely. There are also some efforts focused on maintaining the performance of interactive applications that are competing with BG tasks [32][16][10]. However, to the best of our knowledge, the memory-induced scheduling issue has not been fully addressed. By coordinating memory and process management, Ice significantly improves the user experience at the system level.

8 Conclusion

This paper aims to improve the user experience of resource-limited mobile devices and proposes Ice. Through selective background process freezing, the background page refault issue can be effectively alleviated in memory management. To realize Ice, two schemes are proposed: refault-driven process freezing (RPF) and memory-aware dynamic thawing (MDT). Ice can be implemented on smartphones with no invasive modification in mobile applications and hardware infrastructure. Experimental results show that the user experience can be significantly improved with Ice. Specifically, Ice boosts the frame rate per second by 1.57x on average compared to the state-of-the-art.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful and detailed comments, as well as the shepherd for guiding us through the revision process. This work is supported in part by NSFC 62141213, Shanghai Science and Technology Project (22692199000, 22QA1403300) and the Thailand MHESI grant RGNS 64-091.

References

[1] R. Bakker. 2022. Android 12: how to know if an app is using the microphone and the camera secretly. https://crast.net/90775/android-12-how-to-know-if-an-app-is-using-the-microphone-and-the-camera-secretly/.
[2] K. Bareckas. 2022. Is my phone listening to me? https://nordvpn.com/zh/blog/is-my-phone-listening-to-me/.
[3] J. Brumley. 2017. The Apple and Samsung smartphone market survey. https://seekingalpha.com/article/4101007-apple-samsung-continue-lose-smartphone-market-share-shift-toward-value.
[4] Y. M. Chang, P. C. Hsiu, Y. H. Chang, and C. W. Chang. 2013. A Resource-Driven DVFS Scheme for Smart Handheld Devices. In *ACM Transactions on Embedded Computer Systems.* 1–22.
[5] J. Choi, J. Ahn, J. Kim, S. Ryu, and H. Han. 2016. In-memory file system with efficient swap support for mobile smart devices. In *Transactions on Consumer Electronics (TCE).* 275–282.
[6] D. Chu, A. Kansal, and J. Liu. 2012. Fast app launching for mobile devices using predictive user context. In *International Conference on Mobile Systems, Applications and Services (MobiSys).* 113–126.
[7] J. Courville and F. Chen. 2016. Understanding storage I/O behaviors of mobile applications. In *32nd Symposium on Mass Storage Systems and Technologies (MSST).* 1–11.
[8] C. Crowder. 2022. Survey: Use Access Dots to Find Out If Apps Are Using Microphone and Camera in the Background. https://www.maketecheasier.com/prevent-android-apps-using-microphone-camera-background/.
[9] D. Dai, Y. Chen, D. Kimpe, and R. B. Ross. 2021. Trigger-Based Incremental Data Processing with Unified Sync and Async Model. In *IEEE Transactions on Cloud Computing (TCC).* 372–385.
[10] L. David, A. Chiu, and D. Yuan. 2021. M3: end-to-end memory management in elastic system software stacks. In *Proceedings of the Sixteenth*

European Conference on Computer Systems (EuroSys). 507–522.

[11] R. Davis. 2021. 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/.

[12] Engineers. 2009. Linux Completely Fair Scheduler. https://www.linuxjournal.com/node/10267.

[13] Engineers. 2020. Introduction of Android OOM Adjustments mechanism. https://developpaper.com/android-oom-adjustments/.

[14] Engineers. 2021. Process management in Android framework. https://developer.android.com/reference/android/os/Process.html.

[15] Hocuri et al. 2022. Source code of SuperFreezZ on GitLab. https://gitlab.com/SuperFreezZ/SuperFreezZ.

[16] Y. Feng, A. Riska, and E. Smirni. 2012. Busy bee: how to use traffic information for better scheduling of background tasks. In *ACM/SPEC International Conference on Performance Engineering*. 145–156.

[17] AOSP Foundation. 2021. Android Debug Bridge (ADB) Tool. https://androidmtk.com/download-minimal-adb-and-fastboot-tool.

[18] AOSP Foundation. 2021. Android ui/application exerciser monkey tool. https://developer.android.com/studio/test/monkey.

[19] AOSP Foundation. 2021. Performance tracing using Android Systrace. https://developer.android.com/topic/performance/tracing.

[20] Linux Foundation. 2012. Refault distance. https://lwn.net/Articles/495543/.

[21] Linux Foundation. 2017. Linux patch of per process reclaim. https://lwn.net/Articles/544319/.

[22] Linux Foundation. 2021. Least-recently-used (lru) algorithm in Linux kernel. https://www.kernel.org/.

[23] Linux Foundation. 2021. ZRAM in Linux kernel. https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[24] Linux Foundation. 2022. Process Freezing mechanism in Linux kernel. https://elixir.bootlin.com/linux/latest/source/include/linux/freezer.h.

[25] Linux Foundation. 2022. Shadow entry for refault awareness in the Linux kernel. https://android.googlesource.com/kernel/common/+/refs/heads/android-mainline/mm/workingset.c.

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

[27] R. Gouicem, D. Carver, J.P. Lozi, J. Sopena, B. Lepers, W. Zwaenepoel, N. Palix, J. Lawall, and G. Muller. 2020. Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance. In *USENIX Annual Technical Conference (USENIX ATC)*. 435–448.

[28] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 649–667.

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

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

[31] Y. Hu, S. Liu, and P Huang. 2019. A case for Lease-Based, Utilitarian Resource Management on Mobile Devices. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 301–315.

[32] G. Huang, M. Xu, F. X. Lin, Y. Liu, Y. Ma, S. Pushp, and X. Liu. 2017. Shuffledog: Characterizing and adapting user-perceived latency of android apps. In *IEEE Transactions on Mobile Computing*. 2913–2926.

[33] Facebook Inc. 2015. Report of Facebook application battery drain. https://www.facebook.com/arig/posts/10105815276466163.

[34] MeiZu Inc. 2019. The smart freeze 3.0 of Flyme. https://meizu-bg.eu/en/some-new-features-and-improvements-in-flyme-7-3/.

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

[36] The kernel development community. 2007. Freezing of tasks in Linux kernel. https://www.kernel.org/doc/html/v5.6/power/freezing-of-tasks.html.

[37] S.H. Kim, J. Jeong, and J. Kim. 2017. Application-aware swapping for mobile systems. In *ACM Transactions on Embedded Computing Systems*. 1–19.

[38] S.H. Kim, J. Jeong, J.S. Kim, and S. Maeng. 2016. SmartLMK: A memory reclamation scheme for improving user-perceived app launch time. In *ACM Transactions on Embedded Computing Systems*. 1–25.

[39] S. Kwon, S.H. Kim, J. Kim, and J. Jeong. 2015. Managing gpu buffers for caching more apps in mobile systems. In *International Conference on Embedded Software (EMSOFT)*. 207–216.

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

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

[42] C. Li, L. Shi, and Chun Jason Xue. 2021. MobileSwap: Cross-Device Memory Swapping for Mobile Devices. In *58th ACM/IEEE Design Automation Conference (DAC)*.

[43] C. Li, H. Zhuang, Q. Wang, C. Wang, and X. Zhou. 2018. LKSM: Light Weight Key-Value Store for Efficient Application Services on Local Distributed Mobile Devices. In *Transactions on Services Computing (TSC)*. 1026–1039.

[44] Z. Li and M. Ierapetritou. 2008. Process scheduling under uncertainty: Review and challenges. In *Computers and Chemical Engineering*. 715–727.

[45] Y. Liang, J. Li, R. Ausavarungnirun, R. Pan, L. Shi, T. W. Kuo, and C. J. Xue. 2020. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *USENIX Annual Technical Conference (USENIX ATC)*. 897–910.

[46] Y. Liang, Q. Li, and C. J. Xue. 2019. Mismatched memory management of android smartphones. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[47] X. Liu, C. Vlachou, F. Qian, C. Wang, and K.H. Kim. 2020. Firefly: Untethered Multi-user VR for Commodity Mobile Devices. In *USENIX Annual Technical Conference (USENIX ATC)*. 943–957.

[48] Nubia Technology Co Ltd. 2017. Patent of process freezing in mobile systems. https://patents.google.com/patent/CN107066320A/en.

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

[50] A. S. Miller. 2019. Study of the frame rate metric: frames-per-second. https://www.reneelab.com/frames-per-second.html.

[51] S. Nogd, G. Nelissen, M. Nasri, and B. Brandenburg. 2020. Response-Time Analysis for Non-Preemptive Global Scheduling with FIFO Spin Locks. In *IEEE Real-Time Systems Symposium (RTSS)*. 115–127.

[52] A. Parate, M. Bohmer, D. Chu, D. Ganesan, and B. M. Marlin. 2013. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 275–284.

[53] Android Open Source Project. 2012. Source code to manage the Java heap in Android. https://android.googlesource.com/platform/art/+/master/runtime/gc/heap.cc.

[54] J.P. Rodríguez and P.M. Yomsi. 2019. Thermal-Aware Schedulability Analysis for Fixed-Priority Non-Preemptive Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*. 154–166.

[55] M. Saxena and M. M. Swift. 2010. FlashVM: Virtual Memory Management on Flash. In *USENIX Annual Technical Conference (USENIX ATC)*.

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

[57] H. Sungju, J. Yoo, and S. Hong. 2015. Cross-layer resource control and scheduling for improving interactivity in Android. In *Software: Practice and Experience*. 1549–1570.

[58] Pyropus technology. 2017. Memory tester tool memtester. https://pyropus.ca/software/memtester/.

[59] P. Tseng, P. Hsiu, C. Pan, and T. W. Kuo. 2014. User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[60] Website. 2021. Number of Smartphone and Mobile Phone Users Worldwide in 2021/2022: Demographics, Statistics, Predictions. https://financesonline.com/number-of-smartphone-users-worldwide/.

[61] Website. 2022. Perfetto: system profiling, app tracing and trace analysis. https://perfetto.dev/.

[62] X. Zhang, R. Bashizade, Y. Wang, S. Mukherjee, and A. R. Lebeck. 2021. Statistical Robustness of Markov Chain Monte Carlo Accelerators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 959–974.

[63] S. Zhao, H. Gu, and A.J. Mashtizadeh. 2021. SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel. In *USENIX Annual Technical Conference (USENIX ATC)*. 759–772.

[64] S. Zhao, Z. Luo, Z. Jiang, H. Wang, F. Xu, S. Li, J. Yin, and G. Pan. 2019. AppUsage2Vec: Modeling Smartphone App Usage for Prediction. In *IEEE 35th International Conference on Data Engineering (ICDE)*. 1322–1333.

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