Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems

Yu Liang[1], *Jinheng Li*[1], *Rachata Ausavarungnirun*[2], *Riwei Pan*[1], *Liang Shi*[3], *Tei-Wei Kuo*[14], *Chun Jason Xue*[1]

[1] *Department of Computer Science, City University of Hong Kong*

[2] *TGGS, King Mongkut's University of Technology North Bangkok*

[3] *School of Computer Science and Technology, East China Normal University*

[4] *Department of Computer Science and Information Engineering, National Taiwan University*

Abstract

While the Linux memory reclaim scheme is designed to deliver high throughput in server workloads, the scheme becomes inefficient on mobile device workloads. Through carefully designed experiments, this paper shows that the current memory reclaim scheme cannot deliver its desired performance due to two key reasons: page re-fault, which occurs when an evicted page is demanded again soon after, and direct reclaim, which occurs when the system needs to free up pages upon request time. Unlike the server workload where the direct reclaim happens infrequently, multiple direct reclaims can happen in many common Android use cases. We provide further analysis that identifies the major sources of the high number of page re-faults and direct reclaims and propose Acclaim, a foreground aware and size-sensitive reclaim scheme. Acclaim consists of two parts: foreground aware eviction (FAE) and lightweight prediction-based reclaim scheme (LWP). FAE is used to relocate free pages from background applications to foreground applications. While LWP dynamically tunes the size and the amount of background reclaims according to the predicted allocation workloads. Experimental results show Acclaim can significantly reduce the number of page re-faults and direct reclaims with low overheads and delivers better user experiences for mobile devices.

1 Introduction

With many optimizations to Linux's memory reclaim scheme, the existing Linux memory reclaim scheme can efficiently manage pages inside the main memory in desktops and servers [6, 11, 14, 17]. Android mobile devices, which have seen remarkable growth, inherit the same Linux kernel designed for desktops and servers. As a result, these mobile devices utilize the same memory reclaim scheme inherited from Linux desktop and server distributions.

In this work, we show that the nature of mobile devices workloads is fundamentally different from those of desktop and server workloads. Hence, policies designed to improve the efficiency of the memory reclaim scheme in desktops and servers fail to deliver similar efficiency on mobile devices. Specifically, we observe that a slow page reclaim procedure and severe page thrashing can severely degrade the performance of Android applications. One of the key reason behind this performance degradation is page re-fault, which is a page fault that happens on a previously evicted page. This page re-fault can become a bottleneck and lower workloads' read performance because the system now needs to read the page from the storage instead of the much faster main memory, leading to 100x or more increase in page read latency. Another key reason behind performance degradation is direct reclaim. Direct reclaim can negatively impact performance because any page allocations *must wait* for the direct reclaim process to finish. During a direct reclaim operation, many dirty pages may need to be flushed and thus greatly prolong the allocation procedure. To avoid costly direct reclaim operation, an alternative reclaim scheme using the lightweight background reclaim (***kswapd***), is used by the Linux system. ***Kswapd*** is a kernel thread that is wakened up periodically or by page allocation to reclaim free pages. However, in the current Android kernel, we observe that ***kswapd*** takes too long to free up the necessary number of pages and thus direct reclaim has to be triggered.

Through experiments using popular mobile applications, we show that the current Android memory reclaim scheme does not adapt to the characteristics of Android applications. Experiments show that even when launching one small-size application, page re-faults could happen regularly. Under a set of common use cases, 31% of all the evicted pages are page re-faults. This high ratio of page re-faults to normal page evictions means that page thrashing is very severe in modern Android mobile devices. Aside from the high rate of page re-faults, our experiments also show that the percentage of direct reclaims in all reclaims is 0.8% on average under common use cases. Once direct reclaim happens, up to 1024 dirty pages will be flushed to flash storage, which dramatically extends the latency of the page allocation. Thus, direct reclaim should be avoided.

Prior researches focused on reducing the number of page

faults by optimizing page eviction algorithms on mobile devices [32, 40]. These previously-proposed eviction algorithms treat the pages of background and foreground applications with the same priority and choose victim pages according to their access time and the frequency of accesses. The optimized LRU is known as a good eviction algorithm and is applied in Android [2]. To avoid direct reclaim, the Android operating system reserves some free pages by setting watermarks for free memory. This additional free pages can prevent direct reclaim from being triggered if there is a sudden and urgent heavy allocation. However, our experimental results show that the number of direct reclaims is still surprisingly high (could be triggered 96 times in five minutes in one common use case) on Android mobile devices. To reduce the long latency of memory allocation caused by poor insight of mobile OSes, Marvin [28] implements most memory managements in the language runtime, which has more insight into an app's memory usage. However, Marvin misses the opportunities at the OS level, e.g. taking into account the foreground/background states of applications to predict applications' allocation.

In this paper, we observe that under certain user behaviors, page re-fault depends on the amount of available memory, while the direct reclaim depends on both the amount of available memory *and* the latency of background reclaims. Based on these observations, we uncover two main causes that lead to a high rate of page re-faults and direct reclaims on mobile devices. First, background applications are not truly inactive but their reduced activities and unevicted pages still create high memory pressure, penalizing the foreground application in an unfair way. Furthermore, low memory killer (LMK) [1] does not help much. Second, we found that the large-size reclaim, which is suitable for desktop and servers as the large-size reclaim amortizes the long latency of each direct reclaim process, is overly aggressive and coarse-grained for Android because 1) it prolongs the latency of the background reclaim and negatively impacts the user experience and 2) Android workloads typically issue page allocation requests that are much smaller compared to desktop and server requests.

Based on these two main causes, we propose Acclaim, a foreground aware and size-sensitive reclaim scheme. Acclaim consists of two major runtime components: foreground aware eviction (FAE) and a lightweight, prediction-based reclaim scheme (LWP). FAE relocates free pages from background applications to foreground applications; it does so by lowering the priorities of the pages belonging to background applications during page eviction. LWP tunes the sizes and amounts of background reclaims based on its prediction of allocation workloads. Evaluation results show that Acclaim benefits I/O-intensive phases in application execution, notably application launch and application installation, which are known crucial to mobile user experience [8].

The contribution of this paper is listed as follows.

- This work reveals that the current memory reclaim scheme fails to deliver a good page re-fault ratio (of up to 31%) and frequency of direct reclaims (of up to 96 times when using only one foreground application for five minutes) on Android mobile devices.
- We analyze the root causes of the inefficient memory reclaim scheme on mobile devices and propose Acclaim, a foreground aware and size-sensitive reclaim scheme, to improve the performance.
- We conduct a survey to collect the usage information of applications through deploying our monitoring application on fifty-two real mobile devices. We evaluate Acclaim according to our survey. The experimental results on a real mobile device show that the performance improves in most use cases.

2 Background

To analyze the latency bottleneck of Android mobile devices, we first look at how Android read a page of data.

2.1 Android I/O Latency

Android is a Linux-based lightweight operating system designed for mobile devices. Figure 1 shows the architecture of Android I/O stack that including the userspace, the Linux kernel, and the I/O devices.

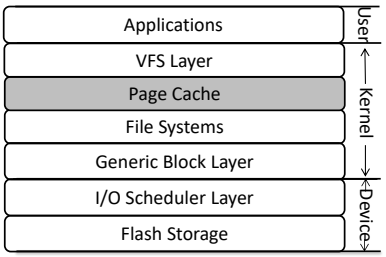


Figure 1: An overview of the Android I/O stack.

We use a read operation as an example to show the latency bottleneck. When an application reads a page in the I/O stack, the application sends a read request to the kernel. The kernel then searches the page cache to see whether the requested page is in the page cache or not. If the requested page is in the page cache, the page cache returns the page to the application. Because the page cache resides in the main memory, the access latency of accessing the page cache takes about one hundred nanoseconds to complete [39]. If the requested page is not in the page cache, a page fault is generated. In this case, the page allocation operation will be triggered to allocate a new page. When the memory is full, the Linux's reclaim scheme is triggered to free pages within the main memory. There are mainly two reclaim schemes: asynchronous background reclaim and synchronous direct reclaim. Background reclaim frees unmapped pages while direct reclaim frees mapped pages or dirty pages, and thus direct reclaim has a heavy cost, especially when writing back

dirty pages. After page allocation, the request is delivered to the file system layer, which finds the logical address of the requested page. Then, a read request goes through generic block and I/O scheduler to access the requested page from flash storage through I/O operations. Going through each of these layers contributes to additional microsecond scale latencies to this read request, which can include addressing latency of the file system, queuing latency of the I/O operation, and reading latency of the flash storage. After these operations, the fetched page is finally stored in the main memory and future accesses can be fetched directly from the page cache. Due to these reasons, a page fault can take microseconds to finish, leading to much longer read latency especially when a direct reclaim is triggered.

To quantitatively show the influence of page fault on Android mobile devices, Yu et al. [21] measure the latency of launching Twitter and Facebook applications in three different situations. Figure 2 shows the three scenarios across two setups based on F2FS [19] and EXT4 [23] file systems, which are commonly used in Android. "Cached" refers to the case where most requested pages can be found in the page cache. This case is implemented by re-launching the application right after it is closed, and thus its data is still in memory. "Read" is a case when there are some page faults but there are enough free pages, [1] and thus the reclaim procedure will not be triggered. This case is implemented by launching the application after cleaning the page cache. "Reclaim-first" is the case where there are some page faults and there are not enough free pages, triggering the reclaim procedure. This case is implemented by launching the applications after sequentially launching twenty other applications (to ensure that the page cache is full prior to the launch of Twitter or Facebook.)

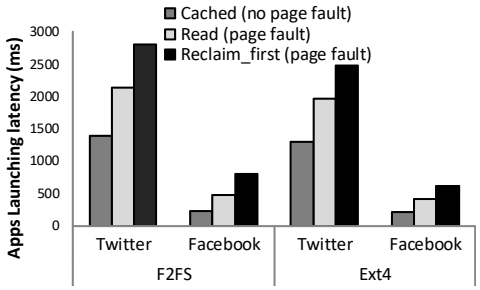


Figure 2: Influence of page fault and reclaim on application launch latency on Android mobile devices.

The results in Figure 2 show that the latency of launching an application is the shortest in the "Cached" case. Compared to the "Cached" case, the "Read" and the "Reclaim_first" cases take longer to launch for both applications. The extended latency is caused by page faults. The launch latency is the longest in "Reclaim_first" case because the reclaim procedure is triggered. Especially, when direct reclaim is triggered, the latency increases significantly.

[1] Cache status is checked by the command **dumpsys meminfo**.

2.2 Key Factors that Affect Performance

Page Re-fault. Page fault can happen in three scenarios. First, a page fault occurs because physical memory has not yet been allocated for the requested page. This occurs, for example, when the page is read for the first time. Second, a page fault occurs because the application wants to read an already evicted page. We define this case as a page re-fault. Third, a page fault occurs because a process wants to illegally access invalid memory. In this case, the operating system will kill the process. Out of these three cases, the system can be designed to minimize page re-fault because the requested page had been in memory but was evicted by system's page reclaim scheme. Page re-fault can be used to measure the page thrashing and thus evaluate the efficiency of the memory reclaim scheme.

Direct Reclaim. Direct reclaim is a heavy-weight synchronous reclaim scheme that is triggered during the page allocation procedure when there is not enough free space for the system's demands. Once direct reclaim is triggered, Android system needs to pause the allocation process, resulting in additional performance degradation. An alternative solution is to use background reclaim. When the number of free pages is lower than a threshold ($watermark_{low}$), background reclaim threads are woken up to reclaim and free unmapped pages asynchronously. During the background reclaim, the Android system does not pause the allocation process. Hence, background reclaim is lightweight. However, if the background reclaim is unable to reclaim enough free pages in time and there are not enough free pages for the current page allocation, direct reclaim is triggered to reclaim the mapped pages or dirty pages. When the memory is extremely scarce, direct reclaim cannot help and some background applications will be killed by the Android low memory killer (LMK) [1] to reclaim memory. Because the overhead of LMK is larger than direct reclaim [28], LMK cannot be used to replace direct reclaim. Thus, LMK complements direct claim and only handles extreme cases.

3 Analysis of Android Memory Reclaim

In this section, we measure the Android memory reclaim scheme by counting page re-faults and direct reclaims while running popular applications.

3.1 Survey of Application Usage Patterns

We survey the distribution of the numbers of background applications from real phones, then using that numbers to conduct controlled experiments and study launch latencies. We develop a monitoring application [2] and deploy it on the phones of sixty Android users. Out of the 60 users, we verified the data invalidation and selected 52 users (90% 18-35 years old and 10% 35-50 years old) for our analysis. Our

[2] https://github.com/MIoTLab/Accliam.

monitoring application collected data on more than twenty mobile device models over a two-month period. During this time, the monitoring application generates an hourly report on other applications' activity, RAM usage, and device information. Our monitoring application runs without root permission. Hence, the application only checks the applications' activity conducted by users but not by systems. With this data, we can estimate the distribution of background applications' usage information as shown in Table 1.

Table 1: Collected data from 52 real phones.

# of phones	# of background applications	Workloads
0	$N < 2$	light
8	$2 \leq N < 5$	light
39	$5 \leq N < 10$	moderate
5	$N \geq 10$	heavy

Based on the survey, we reproduce different realistic usage scenarios by running several popular Android applications. These applications include Facebook, Twitter, Instagram, WhatsApp, Pinterest, Wish, Chrome, Firefox, Google Earth, Google Map, Uber, Angrybird, CnadyCrush, NewsBreak Youtube, and Spotity. We evaluate both launching and execution of applications with a different number of background applications as shown in Table 2.

Table 2: Application combinations used in experiments. A represents a foreground application and B represents a background application. 3B+A means launching a foreground application when there are three background applications.

Applications	Operations	Memory	Workloads
A	Launch and use an application for 5 minutes	Avail.	Light
3B+A	3 background applications	Avail.	Moderate
8B+A	8 background applications	Full	Moderate
15B+A	15 background applications	Full	Heavy

All our following experiments are performed on a Huawei P9 smartphone with an ARM's Cortex-A72 CPU, 32GB internal memory and 3GB RAM, running Android 7.0 on Linux kernel version 4.1.18. We also conduct experiments on 2.5GB RAM by using **memtester** [34] to occupy memory. There is no external SD card in order to force all the I/O requests to the internal eMMC flash storage (/data partition) of Android. We instrument the Android kernel source code and use the **adb** (Android Debug Bridge) tool [37] to obtain information on memory allocations and the reclaim process of our evaluated smartphone. Our instrumentation framework includes information on the number of re-fault pages, the number of evicted pages, the size of each allocation, the size of each reclaim, the number of direct reclaims, and the number of all reclaim operations. To reproduce the real usage scenarios, after system start, background applications will be launched and wait for a while. And then we start to collect the information while we launch and use the foreground application for five minutes. To avoid bias, each experiment is conducted ten times with the same subset of background applications and the average is shown. In Sections 3.2 and 3.3, we show that page re-fault and direct reclaim happen on Android mobile devices unexpectedly frequently even when a small-size foreground application running.

3.2 Page Re-fault on Mobile Devices

The ratio and the number of page re-faults when launching and running popular applications are shown in Figure 3. We define the page re-fault ratio as the proportion of re-faulted pages on all evicted pages. It can be used to evaluate page thrashing. The results show that the page re-fault ratio could be up to 31% when running popular applications. This means the Android memory reclaim scheme often reclaims pages that will be used soon. Although the ratio of page re-fault depends on users' behaviors, when using only one application in a system with 3GB of memory, page re-faults should not occur because the working set of one application does not exceed 3GB [3]. The existence of page re-faults in Figure 3 indicates that the pre-loaded data and processes' data occupy the memory space causing many page re-faults.

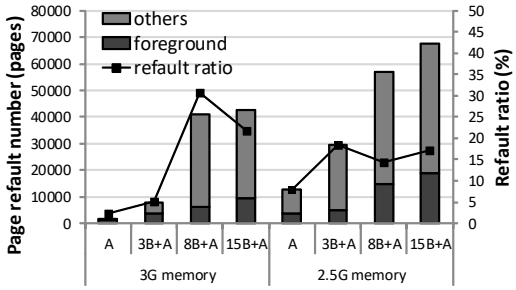


Figure 3: Ratio and number of page re-faults when using one foreground application for five minutes. For each case, there are different number of background applications. "Others" includes background applications and system services.

Moreover, we find that the increase in the number of background applications has a great impact on the number of page re-faults. The ratio of page re-faults is up to 31% when there are eight background applications with 3GB memory. This means almost one-third of the evicted pages will be reused. We further find that a major fraction (37% on average) of page re-faults happens on the foreground app. Because foreground applications directly interact with users, it is important to minimize the number of page re-faults of foreground applications.

3.3 Direct Reclaim on Mobile Devices

Compared to page re-fault, direct reclaim can cause more severe performance degradation and fluctuations because it could flush many dirty pages during a page allocation routine. We show the ratio and number of direct reclaims when running

[3] It is checked by the command **dumpsys meminfo**.

popular applications in Figure 4. We define the direct reclaim ratio as the proportion of the number of direct reclaims on total of reclaims. Even if the direct reclaim ratio is small (up to 2%), it could induce a large latency because page allocations need to wait for the direct reclaim to finish. The latency taken by the direct reclaim can be thousands of times the latency of the background reclaim. Thus, the direct reclaim is supposed to be triggered in memory of emergency cases.

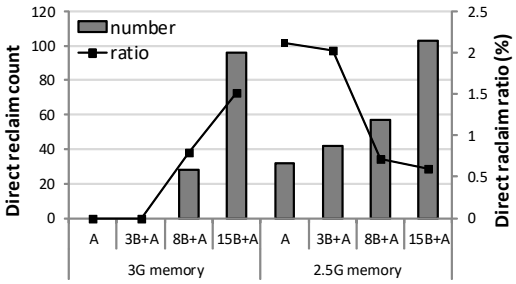


Figure 4: Ratio and number of direct reclaims when using one foreground application for five minutes. For each case, there is different number of background applications.

We find that the increase in the number of background applications and the reduction of physical memory have major effect on the number of direct reclaims. However, the direct reclaim ratio trend is different between on 3GB (default) and on 2.5GB (using **memtester** to occupy memory). When memory is relatively large (3GB), direct reclaim triggers only when there are some background applications and increases as the number of background applications increases. However, when memory is extremely scarce and direct reclaim become ineffective, OS will kill some applications to reclaim memory space. Thus, direct reclaim ratio is decreasing with a larger number of background applications on 2.5 GB of memory. The number of direct reclaims could be up to 96 in five minutes when there are fifteen background applications with the default 3 GB memory. An efficient memory reclaim scheme should minimize the number of direct reclaims.

4 The Cause of Page Re-fault and Direct Reclaim on Mobile Devices

Substantial re-fault rates were also seen on servers, e.g. as high as 14% reported by Google engineers [9]. Compared to servers, mobile devices have vastly different characteristics: much smaller page allocation request size, limited memory, and highly-interactive foreground applications [10, 13]. According to these characteristics, we conduct another set of experiments to analyze the Android memory reclaim scheme to find the main causes of a high number of page re-faults and direct reclaims.

Observation 1: Page re-fault depends on the available memory. Figure 5 shows that the number of page re-fault and evict pages under different available memory. To further eliminate the impact of user behavior, in this experiment, we used different usage scenarios from that in Figure 3. "A" means to launch one foreground app. "A4A" means to launch one foreground application and then launch four background applications, and finally re-launch the foreground application. Relaunching a foreground application is a typical scenario to produce page re-faults.

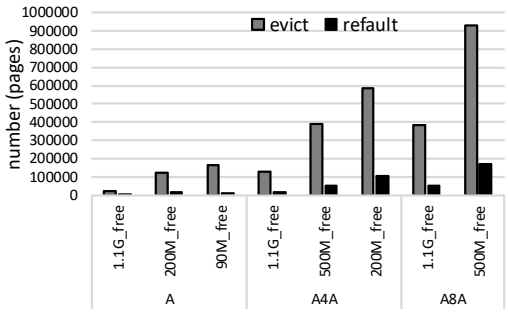


Figure 5: Number of page re-faults and evicted pages under different available memory.

The results show that the number of background applications has a major impact on the number of re-fault and evict pages. Moreover, reducing physical memory can increase number of both re-fault and evict pages. In a word, the number of page re-faults depends on the available memory.

Observation 2: Direct reclaim depends on both available memory and the latency of the background reclaim. The factors which affect the frequency of the direct reclaim can be found in its flow chart which is shown in Figure 6.

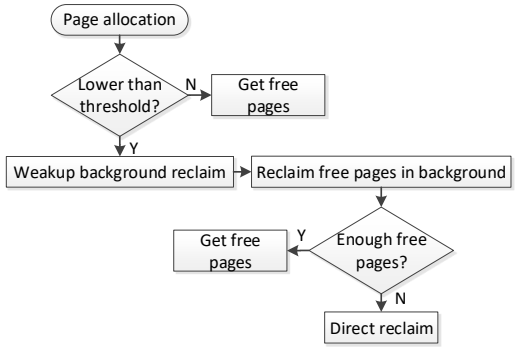


Figure 6: The flow chart of reclaim scheme.

During page allocation, if the number of free pages is lower than a threshold, the background reclaim starts asynchronously. If there are not enough free pages for this allocation or the launched background reclaim does not reclaim enough pages in time, direct reclaim will kick in synchronously. The flow chart shows that direct reclaim depends not only on available memory but also on the latency of the background reclaim. Notably, a larger reclaim size also significantly increases the latency of the background reclaim.

Based on the above two observations and the specific characteristics of mobile devices, we found that there are two additional factors that can increase the number of page re-faults and direct reclaims.

Observation 3: Background applications keep consuming free pages even though they do not have the same impact on user experience compared to the foreground applications. For mobile devices or other highly-interactive systems, foreground applications have significant impact on user experience. However, we found that background applications keep consuming free space under the current memory reclaim scheme due to two main reasons.

First, we find that anonymous pages from background applications thrash pages from the foreground application. In the android system, all the pages are in one of five LRU lists: *Active_anonymous*, *inactive_anonymous*, *active_file*, *inactive_file*, and *unevictable*. The pages in the *unevictable* list will not be evicted. Since anonymous pages contain the heap information associated with a process, these anonymous pages are considered to be more important than file pages to the process. In most cases, the pages in *active_anonymous* list will not be evicted, even if they belong to a background app. Thus, many anonymous pages of background applications stay in memory, while the file pages from the *foreground application* are evicted. These evicted file pages from foreground applications may be accessed again soon, leading to a high number of page re-faults. Moreover, the anonymous pages of background applications occupy free space and thus affect the frequency of the direct reclaim.

Second, we found that background applications are still active even after they are in the background for thirty minutes. The details are shown in Table 3. The results are collected when there are seven user background applications and one foreground application. Notice that system services are treated as applications here.

Table 3: Applications are still active in the background.

Time	Evict apps	Refault apps	Foreground	Background	System
5 mins	53	31	6.2%	34.4%	59.4%
30 mins	52	19	18.3%	33.3%	48.4%

"Evict apps" represents the number of applications that have pages being evicted while "Refault apps" represents the number of applications that have page re-faults. "System" includes system services and private applications that are installed on the mobile device at the factory. The percentages in the Table 3 means the percentage of re-faults. The results show that background applications still request free pages and thus induce page re-faults. Moreover, background applications still consume free space and thus affect direct reclaim frequency.

In summary, the reclaim scheme keeps the pages of background applications in memory and could induce a high number of page re-faults and direct reclaims. For servers, the foreground and background applications usually have the same priority. However, for mobile devices, only the foreground application has a major impact on the user experience.

Observation 4: Large-size reclaim prolongs the latency of the background reclaim. In the buddy system, every memory block has an order, where the order is an integer ranging from 0 to 11. The size of a block of order n is 2^n. The distribution of allocation order when running popular applications is shown in Figure 7. These results are collected from the allocation function *_alloc_pages_nodemask()*. The results show that on the Android mobile device, 99% of allocation orders are 0 (1 page), and more than 99.9% of orders are smaller than 4 (16 pages). This is because the requests on Android mobile devices are mostly in small size. One of the main reasons is that most Android applications use SQLite as the database. SQLite and its temporary files are mostly accessed in 4KB (1 page) units [20, 29].

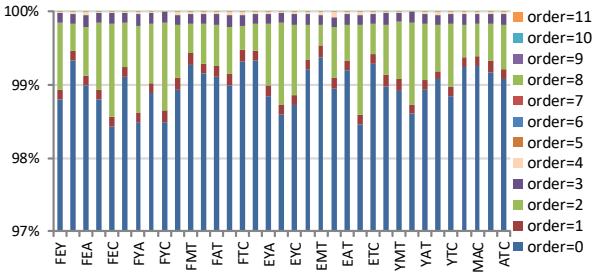


Figure 7: The distribution of allocation orders. The corresponding allocation size equals to 2^{order} [21].

The distribution of reclaim sizes is shown in Figure 8. The results show 80% of reclaim sizes are larger than 32 pages (order=5). Android inherits much of the reclaim scheme from its server counterpart. The latter often reclaims memory as much as possible to fulfill large allocations and avoid expensive direct reclaim or killing processes under memory pressure. As Android applications tend to allocate multiple small blocks of data (shown in Figure 7), the reclaim process becomes overly aggressive and ends up reclaiming excessive pages for each of these small allocations.

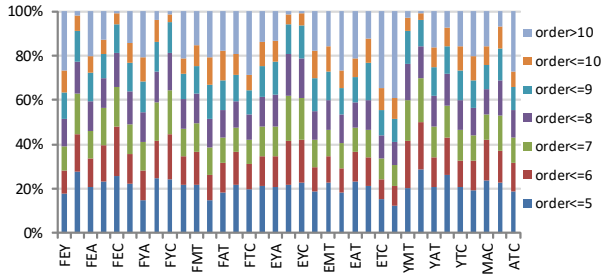


Figure 8: The distribution of reclaim sizes. These results show the reclaims from LRU lists, and they are collected in the functions *shrik_lruvec()* [21].

Large reclaim size prolongs the latency of the background reclaim. According to observation 2, the latency of the background reclaim will affect the frequency of the direct reclaim. Moreover, a large-size reclaim scheme could induce more page re-faults than necessary [21].

Based on our data in Section 3, Android does not efficiently manage its memory. This observation is in-line with multiple technical news: Google Pixel 3 has memory management

issues, such as killing background applications [35] and is unable to shuffle between a few applications at a time [33]. Moreover, its memory management issue seemingly gets worse when users use the camera [16]. Hardware vendors tend to address the issue by putting a large-capacity DRAM on devices, which alleviates the problem in a short term but leaves the issue in the future. Moreover, the brute-force solution has many problems, such as cost efficiency, power consumption, the growing trend of application size, etc. and these problems cannot be addressed solely by dropping in more DRAM. Instead, our work aims to improve the efficiency of Android's memory management.

5 Our Solution: Acclaim

With the understanding of the four observations leading to a high number of page re-faults and direct reclaims, we proposed Acclaim, foreground aware and size-sensitive reclaim scheme, which includes two parts. The first part, foreground aware eviction (FAE), is used to solve the problem that background applications keep consuming free pages. FAE takes space from background applications and allocates it to the foreground application. The second part, a lightweight prediction-based reclaim scheme (LWP), is used to reduce the reclaim size of the background reclaim and thus minimize its latency. LWP tunes the size and amount of the background reclaims according to the predicted allocation workloads. In summary, FAE decides from where to reclaim, while LWP decides how much to reclaim.

5.1 Foreground Aware Eviction (FAE)

The memory is always not large enough to eliminate all page re-faults and direct reclaims. Both the number and size of applications increase with memory capacity [38]. Moreover, when the memory capacity increases, mobile device manufacturers often make optimizations, such as locking commonly-used files [3] and pre-loading predicted applications [31] in the memory. All these optimizations consume free memory.

Since the memory size is limited, the total page re-faults can be hardly reduced. The reduction of page re-faults of the foreground application can have a major impact on the user experience of mobile devices or other highly-interactive systems. Thus, we propose to reduce the page re-faults of foreground applications by sacrificing space from background applications. Foreground aware eviction (FAE) is proposed to lower the priority of background pages in LRU lists, causing them to be evicted faster and thus freeing more memory space.

5.1.1 Framework of FAE

The framework of FAE is shown in Figure 9. FAE needs to know whether a page belongs to background applications. Each application has a unique ID (UID). Once an application is installed, its UID is fixed. The UIDs of user applications are added to Page Table Entry (PTE). PTE is only accessible during the page walk process through the page walker. User applications will not be able to access these UID bits as this is handled by OS or hardware. The page's UID is used by FAE during the eviction procedure. Currently, only 8 unused bits of each PTE (the 56th to the 63rd) can be used to store UIDs. Thus, Acclaim only supports 256 unique UIDs at a time, which is an implementation limitation.

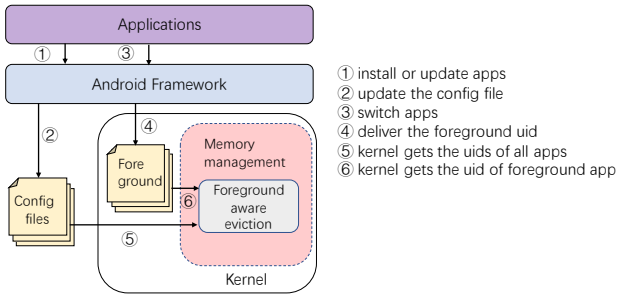


Figure 9: Framework of foreground aware evict scheme.

There is only one foreground application at a moment but there could be several background applications. This list of all background applications can be obtained by subtracting the foreground application from an application list. The main task of FAE is to create a list of background applications and lower their priority compare to the foreground task. To do this, FAE stores the UIDs of applications in the application list in a configuration file. This file will be updated when installing or deleting applications. To identify the current foreground application. FAE notifies the UID of the foreground application to the kernel when users switch applications. Based on the UIDs of applications in the application list and the UID of the foreground application, the system can then lower the priorities of all other applications' pages in LRU lists.

By default, Acclaim deprioritizes all background user applications by assigning them lower priorities in page eviction. To accommodate a small number of applications that keep serving the users in the background, e.g. music or video players, Acclaim can treat them as exceptions without degrading their priorities by excluding them in the application list.

5.1.2 Lower Priority of Background Applications

Initially, we tried to raise the priorities of foreground applications' pages or system's pages. However, this method maintains too many useless pages of foreground applications and the system in memory because of their higher priorities. These useless pages may lead to OS crashes when free memory is used up. Thus, FAE chooses to lower the priority of pages of background applications. Under this scheme, the priority of foreground applications' pages and the system's pages will not be changed. Their useless pages will be evicted from memory and thus free memory will not be used up to crush OS. The details are shown in Figure 10.

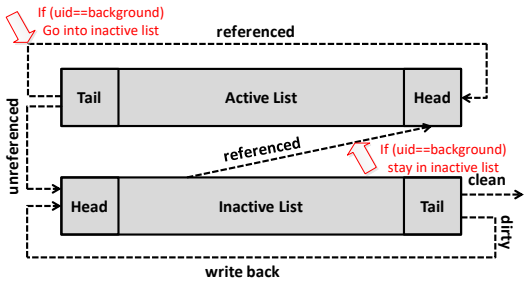


Figure 10: Foreground aware evict scheme of LRU lists.

Page movement between the "active" and the "inactive" LRU lists is driven by memory pressure. Unused pages in the active list go to the inactive list. Pages are taken from the tail of the inactive list to be freed. If the page has the reference bit set, it is moved to the head of the active list and the reference bit is cleared. If the page is dirty, writeback is commenced and the page is moved to the head of the inactive list. FAE lowers the priorities of background pages (See Figure 10) and moves them out of LRU lists quickly. Thus FAE can extract space from background applications for foreground applications and thus to reduce the foreground page re-faults.

Sharing pages have the same priority as their creators. To reduce dynamic-conversion overhead, a sharing page maintains the UID of the application that created it. Thus, the sharing page gets the priority according to the status of its creator. For example, let us assume page A is shared by application D and E. D creates page A first and D is in background while E is in foreground. If page A is not used for a long time, it will be likely to be evicted because its owner is in the background list and thus has low priority. However, if page A is used frequently, A will remains in the memory. Acclaim does not move the application's pages when it is changed from background to foreground and vice versa. Acclaim only checks page's UID and moves a page when it needs to be moved under the default eviction scheme (See Figure 10).

With FAE, the re-launch time of background applications can increase because their pages are out of LRU lists. However, this penalty is much smaller compared to the baseline scenario where these background applications are killed by the Android low memory killer (LMK) [1, 28]. Moreover, the penalty of FAE can be minimized by combining it with application prediction [8, 27, 31]. If the system predicts a background application will be used soon, FAE removes these applications from the background list, and thus does not decrease the priorities of its pages in the LRU lists.

Additionally, FAE is compatible with LMK. For example, FAE can further categorize background applications. Background applications that may be used in the near future can donate some of their memory space with Acclaim while background applications may not be used again can be killed by LMK when memory is getting full.

We expect FAE to benefit impromptu, short interactions (checking and replying instant messages, or switching among applications within a short time span). FAE recognizes them as foreground applications and optimizes them accordingly.

5.2 Lightweight Prediction-Based Reclaim Scheme (LWP)

The original reclaim size of each background reclaim is the maximum number of requested pages until the time of the reclaim. From Section 4, we find that the current reclaim size is too large for the allocation requests of mobile devices. Large-size reclaim induces a high number of page re-faults and direct reclaims. However, there is a trade-off between the reclaim size and performance. If the reclaim size of the background reclaim is too small while the application workloads are heavy, the free pages will be consumed quickly and the heavy-weight direct reclaim will be triggered, lowering performance. On the other hand, if the reclaim size of the background reclaim commands is too large, page re-faults and direct reclaim will happen frequently and, again, degrades overall performance. Thus, we incorporate a workload-prediction based reclaim scheme into our design by incorporating the historical information obtained through a lightweight predictor (See Section 5.2.1). Based on the predicted results, the system can tune the reclaim size of the background reclaim. Another challenge can occur if we reduce the reclaim size according to each workload. In this case, the number of reclaim operations can increase while the amount of reclaimed pages remain unchanged, and thus reducing reclaim size will waste CPU time. To solve this problem, we incorporate the amount of reclaiming pages to dynamically tune the size according to the predicted workloads.

5.2.1 Framework of LWP

LWP consists of two parts, a lightweight predictor and a moderator. Its framework is shown in Figure 11. The lightweight predictor is run during the page allocation procedure. To reduce memory overhead, the sampled allocation requests as inputs are stored in the lightweight predictor. The outputs of the lightweight predictor are the predicted reclaim size and the trend of reclaim amount. The moderator modifies the reclaim size and the amount of the background reclaim according to the predicted reclaim size and the amount trend.

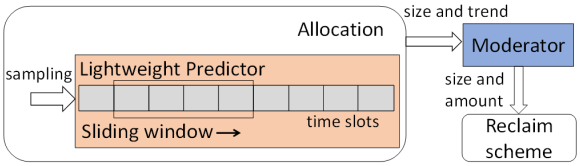


Figure 11: The framework of LWP reclaim scheme.

5.2.2 Lightweight Predictors (LWP)

Using recent information processed by a sliding window to predict the future workloads is commonly applied for prediction [24] [8] [25]. The challenge is how to implement a

lightweight predictor. Recent information-based prediction should make sure the correctness of stored information. Page allocation procedure supports concurrency, thus the sliding window needs to guarantee correctness by using locks which will greatly degrade the performance. To predict allocation workloads including size and frequency, the system needs to store the size and time of historical allocation requests. However, storing all historical information precisely will occupy significant space and CPU time and thus degrade the overall performance. To reduce the overhead, the proposed predictor is designed as a lock-free sliding window, and it only stores limited historical information.

Limited historical information. To reduce the stored information, the sliding window is carefully designed in two aspects. First, the sliding window is designed based on time slots to avoid storing the time information. Second, the sliding window stores sampled historical information to reduce the amount of stored information. Each element of the sliding window is the allocation size of a sampling allocation request. For example, when the sampling period equals 10 *ms*, the predictor will pick one of allocation requests that happened in this 10 *ms* and add its allocation size to the sliding window.

Lock-free sliding window. We first analyze the impact of being lock-free on the sliding window. When the window is lock-free, the following three things could happen. (1) Data disorder and data missing could occur; (2) When the predictor samples the historical information, the penalty of being lock-free reduces as there are fewer access to locks; (3) Being lock-free does not affect the system consistency as we only use it to store the memory historical information. We further evaluate the accuracy of the lock-free and sampling predictor. Let sampling = 10 *ms*, sliding = 10 *ms*, and window = 1000 *ms*, to get the sum of the reclaim sizes in a window. To compare three cases, the log of sum value is shown in Figure 12 as the sum in the sampling and lock-free cases are much smaller than that in the lock-free only case and original cases. Three usage behaviors, launching Chrome, launching YouTube, and launching and using five applications, are evaluated. The x-axis is the index of the sliding window. The y-axis is the log of the sum of the reclaim sizes in a window. We use vertical red lines to show the trend changes in the first figure. The results show that the lock-free and sampling case captures the same trend as the original case. (For using the ACFYT case, too much data makes the trend of the sampling case not obvious. It's trend also same as the original case.) Thus, the lightweight predictor can predict the trend of the amount of allocation requests in the next window correctly. In this way, prediction can be achieved with a low overhead in both storage and latency.

5.2.3 LWP-Based Moderator

The moderator is used to tune the reclaim size and amount of the background reclaim according to the predicted results.

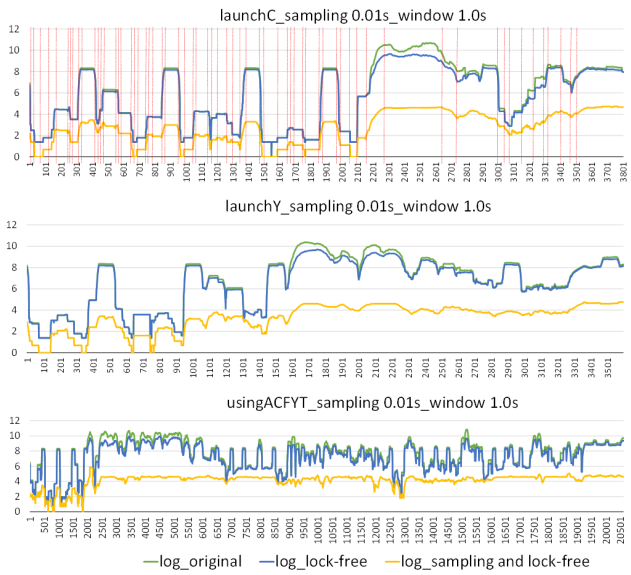


Figure 12: Predicted sum trend of reclaim sizes in the lock-free and sampling sliding window.

The reclaim amount of the background reclaim could be tuned by modifying its stop setting ($watermark_{high}$). The original background reclaim will stop when the number of free pages is above the $watermark_{high}$, which is a fixed value (a proportion of the total number of pages). The original reclaim size of each background reclaim is the maximum number of requested pages until the time of the reclaim. To exploit the trade-off between reclaim size and performance, the LWP base moderator tries to make the background reclaim reclaims just enough free pages just in time. The main idea is to tune the reclaim size of each background reclaim according to the predicted allocation size and tune the $watermark_{high}$ according to the predicted sum trend. "sum trend" is defined as sum/lastsum, where "sum" is the sum of the reclaim sizes in the current window and "lastsum" is the sum of the reclaim sizes in the last window. For the reclaim size, if the trend is larger than a threshold T_1, that means the amount of allocation in the future will be much increased. Thus, the reclaim size should be increased.

- Reclaim size = $P_1 * predicted\ size$ when $trend \geq T_1$
- Reclaim size = $P_2 * predicted\ size$ in other cases, $(P_2 < P_1)$

For the reclaim amount, the moderator tunes the amount of reclaim based on the default value that is set by the mobile device manufacture.

- Reclaim amount = $min(watermark_{high} * (1 + trend), watermark_{high} + T_2)$ when $trend > 1$
- Reclaim amount = $max(watermark_{high} * (1 - trend), watermark_{low})$ in other cases

Reclaim amount is limited. If it is smaller than the default $watermark_{low}$, the performance will be degraded because direct reclaim will be triggered often.

Table 4: Summary of parameters used in LWP.

Symbols	Semantics	Default values
P_1	Amplification factor of reclaim size when I/O is intensive.	4
P_2	Amplification factor of reclaim size when I/O is sparse.	2
T_1	Defines "sudden change" and thus decides the reclaim size.	2
T_2	A threshold to stop background reclaim.	$watermark_{high} - watermark_{low}$

6 Evaluation

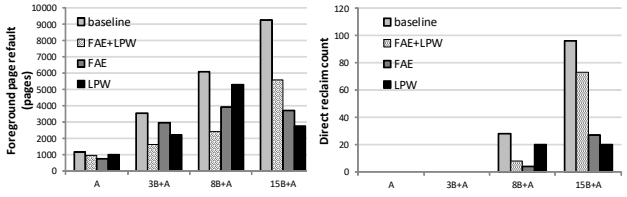
To evaluate Acclaim, we set sampling duration to 10 *ms*, window duration to 100 *ms*, sliding duration to 100 *ms* for sliding window of LWP. Both memory overhead and the predict accuracy of LWP are sensitive to these three parameters.

Moreover, we set the parameters for LWP-based moderator in Table 4. Reclaim size should be larger than the predicted allocation size to avoid memory once heavy workloads are arriving. However, according to the analysis in Section 4, reclaim scheme should not be over aggressive on mobile devices. Thus, we choose small values (4 and 2) as amplification factors (P_1 and P_2) under different workloads.

Furthermore, T_1 determines the sensitivity to the increment in workloads. We configure Acclaim to be sensitive to changes in workloads and responds in time, thus we choose a relatively small value (2). T_2 is the threshold that ensures that the reclaim amount is not too large. Like the default $watermark_{high}$, it is an empirical value. We evaluate Acclaim on three aspects: impact on foreground applications, impact on background applications, and overhead.

6.1 Impact on Foreground Applications

Reduction in page re-fault for foreground applications and direct reclaim of OS. Page re-fault and direct reclaim are closely related to user behaviors. To compare the solution and baseline, we need to choose an application with little change in user behaviors. Thus, a single game, AngryBird, is used as a foreground application for five minutes in this evaluation. The page re-fault and direct reclaim results under the kernel with the original reclaim scheme and the kernel with our solution are shown in Figure 13.



(a) Page re-fault. (b) Direct reclaim.

Figure 13: The page re-faults in a foreground application (AngryBird) and direct reclaims in the whole OS, showing the benefit of each solution is different in various scenarios.

The results demonstrate the efficacy of Acclaim. For the benchmark (AngryBird), Acclaim reduces page re-faults by 16.3% – 60.2%; it reduces direct reclaims of the whole OS by from 23.9% – 70%. In the experiments, the proposed two techniques play vital, complementary roles. FAE shows higher benefits as the number of background applications increases, as it seizes free pages from background applications to relieve memory pressure. LPW's benefit depends on the accuracy of its prediction, based on which it dynamically tunes the background reclaims. Notably, in case of sudden changes in application workloads, LPW may suffer from accuracy loss and thus underperforms.

Benefit to read/write performance. Reduction in page re-faults and direct reclaims could improve read and write performance. To quantify the impact on read and write operations, we show the read and write performance by using read and write micro benchmarks, [4] and the results are shown in Figure 14. Since most page allocation request sizes on mobile devices are in the size of 4KB [10], we write or read 512MB or 1GB of data in size of 4KB.

This may not be a typical write access pattern, it could happen in some cases. For example, when installing games and applications, more than 1GB of an apk file could be downloaded and written back to flash storage. Moreover, this is a stress test to show Acclaim's benefit under intensive I/O requests. Thus, these evaluation results show the performance impact under intensive I/O requests.

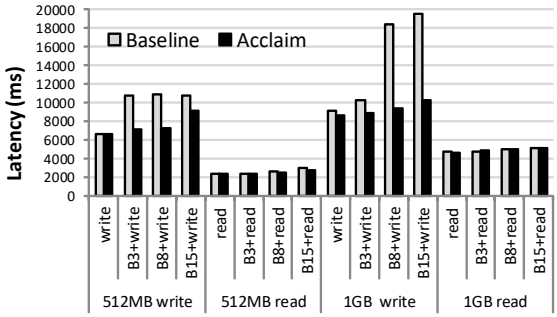


Figure 14: Read and write performance.

The results show that Acclaim improves write performance by up to 49.3% (when writing 1GB of data). This is because page allocation, which Acclaim optimizes, constitutes a significant portion of the delay in writes because of the write-back operations of dirty pages. The read performance is only improved slightly as the latency of page allocation is only a small part of read latency in this set of test cases as no dirty pages are generated and written back during reads.

[4] https://github.com/MIoTLab/Acclaim

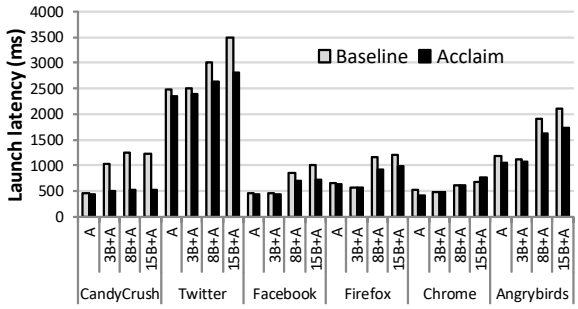


Figure 15: The launch latency of foreground applications.

Benefit to user experience in application launch. To quantify the impact on user experience, the launch time of various foreground applications are evaluated. The evaluation results are shown in Figure 15. The results show that the launch latency improvement varies for different applications. The benefit of Acclaim is more pronounced when an application is launched with multiple memory-hungry applications in background. For example, the launch latencies of CandyCrush and Facebook are reduced by up to 58.8% and 28.8%, respectively. Acclaim can have negative impact when foreground and background applications share common files. For example, the launch latency of Chrome could be prolonged up to 12.3% by Acclaim when there are many background applications. This may because Acclaim evicts common files between background applications and Chrome [3]. However, this penalty can be eliminated by combining with mlocking common files [3]. In summary, Acclaim outperforms the baseline in most test cases. Of all the 24 test cases, it reduces latencies in most of them (20, with median reduction of 19.1% and max reduction of 58.8%) while incurring additional latencies in 4 (with median increase of 3.1% and max increase of 12.3%).

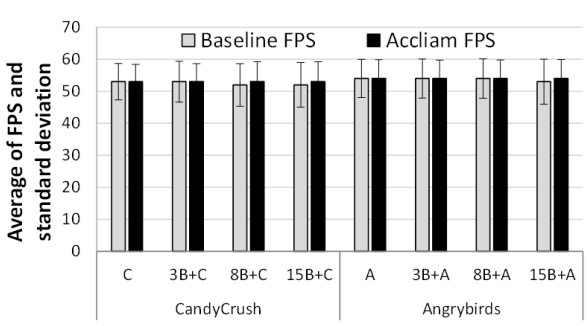


Figure 16: Average FPS and the standard deviation of FPS of foreground applications.

Impact on FPS during active user interactions. Acclaim reduces the launch latency by employing policies on how application uses the memory to cache files. This policy might impact user experience negatively after application launch, as launched applications will have a different amount of data in page cache. Thus, we measure the possible loss in user experience as FPS in KFMARK, a popular gaming benchmark. [12]. The average FPS and the standard deviation of FPS of foreground applications are shown in Figure 16. For the average FPS, the larger the value, the better. While the smaller the value, the better for the standard deviation of FPS. The results in Figure 16 suggest no noticeable impact: the difference between the mean values of the baselines and Acclaim are smaller than their standard deviations.

6.2 Impact on Background Applications

Because Acclaim evicts more pages from background applications, it can negative these applications' re-launch time. We evaluate the re-launch time for the first-launched background application to show the upper bound of the penalty. In this evaluation, we use Facebook as the first launched background application is evaluated when it is launched for one or ten minutes, and the results are shown in Figure 17.

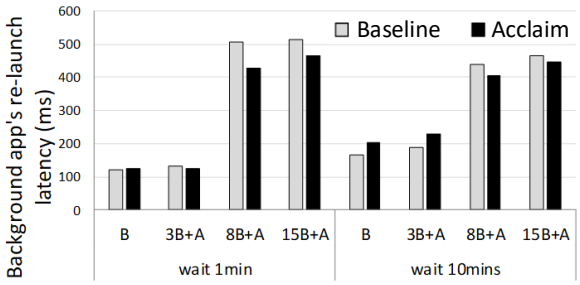


Figure 17: Re-launch time of the first-launched background application (Facebook).

Based on Figure 17, we observe that re-launch penalty only occurs when there are a few background applications after ten minutes. In most cases, the benefit is larger than the penalty. When there are many background applications, Acclaim effectively seizes free pages from background applications, which can be used to aid background applications' re-launch process. Moreover, Acclaim also reduces the number of direct reclaims, benefiting all applications. When the time during which the application is used after it is launched is too short (one minute), Acclaim may not have enough time to move out of the evaluated background application's pages. After system starts, many applications are partially run in the background even if the user does not use them. To this end, their pages will be firstly evicted from LRU lists by Acclaim.

Additionally, notice that the penalty of Acclaim can be eliminated by using it in combination with application prediction [8, 27, 31]. If a background application is predicted as the next used application, Acclaim removes it from the background application list in FAE, and thus the priority of its pages will not be degraded in LRU lists when it is in background. Thus, the penalty will be eliminated.

6.3 Overhead Analysis

Additional memory overhead. FAE adds a uid to PTE, incurring a space overhead of an integer space (4 Bytes). For a device with 3GB of memory, the maximum memory overhead is 3MB. Moreover, FAE needs to store the uids of applications in the application list. If a user installs 100 applications, the total memory cost is only 0.4 KB.

LWP needs to store the sampling historical allocation size (4 Bytes per entry). When the sampling duration is 10 *ms* and the window is 100 *ms*, only 10 values need to be stored. Even if there are 100 values in the window, the LWP only takes up 400B storage overhead. In summary, the total memory overhead is about 3MB (0.1% of memory capacity).

Performance overhead. FAE's performance overhead can be broken down into three parts. First, after system starts, it needs to check the configure file to get the UIDs of the applications. It only happens when the system starts. Second, when the user switches applications, the new foreground UID needs to be delivered to FAE from the framework layer. Third, FAE needs to check if the UID equals to one of the background applications during each page eviction. Only a few comparisons are conducted, thus the performance overhead is negligible.

The performance overhead of LWP includes two parts. First, prediction has a small cost because of the lock-free sliding window. Second, reducing the reclaim size could prolong the wake up time of the background reclaim . However, LWP dynamically tunes the amount of background reclaim according to the allocation workloads to reduce the CPU time consumption. In summary, the performance overhead of Acclaim is trivial.

7 Related Work

Application launch. Existing studies on context-awareness led to the development of application pre-loading algorithms [8, 27, 31]. These algorithms greatly reduce the application launch latency by preparing required resources before they are requested.

Application foreground/background behaviors. Many mobile applications are designed to run in background to enable a model of always-on connectivity and to provide fast response time. This means that once installed and initiated by the user, applications can register themselves with the services provided by the OS framework for background activities, regardless of the user's actual usage of the app. This is true of both iOS and Android OS [4, 7, 26].

Memory management. Many previous works were focusing on the design of the buddy system for managing memory. Burton [6] proposed a generalized buddy system. By using the Fibonacci numbers as block size, Knuth [17] proposed the Fibonacci buddy system. Moreover, this idea was complemented by Hirschberg [15], and was optimized by Hinds [14], Cranston and Thomas [11] to locate buddies in time similar to the binary buddy system. Shen and Peterson [36] proposed the weighted buddy system. Page and Hagins [30] proposed the dual buddy system, an improvement to the weighted buddy system, to reduce the amount of fragmentation to that of the binary buddy system. A buddy system designed for disk-file layout with high storage utilization was proposed by Koch [18]. Brodal et al. [5] improved the memory management for accelerating allocation and deallocation. Marotta et al. [22] proposed a non-blocking buddy system for scalable memory allocation on multi-core machines. Yu et al. [21] show that the existing reclaim scheme is not working well for Android mobile devices. Consequently, this paper proposes a new smart reclaim scheme for Android mobile devices.

Mobile device-specific memory management. Due to mobile OSes have poor insight into application memory usage, the memory allocation may take a long latency, especially under memory pressure. Marvin [28] implements most memory managements in the language runtime, which has more insight into an app's memory usage. They target the same problem at a different layer. By predicting allocation workloads and with foreground and background information, Acclaim improves memory management efficiency at the system level.

8 Conclusion

Existing Linux memory reclaim scheme is designed for servers and PCs. Android inherits Linux kernel and thus the memory reclaim scheme is transplanted to mobile devices. The experimental results show that these algorithms become less effective for the characteristics of applications running on Android mobile devices due to two main reasons. First, background applications has less impact on the user experience than foreground applications. However, they continually consume free pages that increase the frequency of page re-fault and direct reclaim. Second, the large-size reclaim aggravates this problem on mobile devices which involve with almost exclusively small allocation requests. In this work, we propose Acclaim. Acclaim consists of the Foreground aware eviction (FAE), which is designed to relocate free pages from background applications for foreground applications, and the lightweight prediction-based reclaim scheme (LWP), which is used to dynamically tune the size and amount of the background reclaim according to the predicted allocation workloads. Evaluation results show that Acclaim improves the performance in general with a trivial overhead.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Prof. Felix Xiaozhu Lin for their feedbacks and guidance. This paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (11204718) and National Natural Science Foundation of China under Grant No. 61772092.

References

[1] Android open source project. low memory killer. https://source.android.com/devices/tech/perf/lmkd, 2017.

[2] Linux kernel code. lru scheme in the kernel. https://www.kernel.org/, 2019.

[3] Android open source project. mlock commonly-used files. https://source.android.com/devices/tech/debug/jank_jitter, 2020.

[4] AMALFITANO, D., AMATUCCI, N., TRAMONTANA, P., FASOLINO, A., AND MEMON, A. A general framework for comparing automatic testing techniques of android mobile apps. *Journal of Systems and Software 125* (12 2016).

[5] BRODAL, G. S., DEMAINE, E. D., AND MUNRO, J. I. Fast allocation and deallocation with an improved buddy system. *Acta Informatica 41*, 4 (Mar 2005), 273–291.

[6] BURTON, W. A buddy system variation for disk storage allocation. *Commun. ACM 19*, 7 (July 1976), 416–417.

[7] CHEN, X., JINDAL, A., DING, N., HU, Y. C., GUPTA, M., AND VANNITHAMBY, R. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *MobiCom '15* (2015).

[8] CHU, D., KANSAL, A., AND LIU, J. Fast app launching for mobile devices using predictive user context. In *ACM MobiSys* (June 2012), ACM.

[9] CORBET, J. Proactively reclaiming idle memory. https://lwn.net/Articles/787611/, 2019.

[10] COURVILLE, J., AND CHEN, F. Understanding storage i/o behaviors of mobile applications. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)* (May 2016), pp. 1–11.

[11] CRANSTON, B., AND THOMAS, R. A simplified recombination scheme for the fibonacci buddy system. *Commun. ACM 18*, 6 (June 1975), 331–332.

[12] FVIEW. Fps test tool kfmark. https://kfmark.com/, 2017.

[13] GAO, C., SHI, L., XUE, C. J., JI, C., YANG, J., AND ZHANG, Y. Parallel all the time: Plane level parallelism exploration for high performance ssds. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2019), pp. 172–184.

[14] HINDS, J. A. An algorithm for locating adjacent storage blocks in the buddy system. *Commun. ACM 18*, 4 (Apr. 1975), 221–222.

[15] HIRSCHBERG, D. S. A class of dynamic memory allocation algorithms. *Commun. ACM 16*, 10 (Oct. 1973), 615–618.

[16] JANSEN, M. Common google Pixel 3 problems, and how to fix them. https://www.digitaltrends.com/mobile/common-google-pixel-3-xl-problems-and-how-to-fix-them/, 2019.

[17] KNUTH, D. Dynamic storage allocation. *In: The art of computer programming 1*, 435–455.

[18] KOCH, P. D. L. Disk file allocation based on the buddy system. *ACM Trans. Comput. Syst. 5*, 4 (Oct. 1987), 352–370.

[19] LEE, C., SIM, D., HWANG, J. Y., AND CHO, S. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[20] LEE, K., AND WON, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT)* (2012), ACM, pp. 23–32.

[21] LIANG, Y., LI, Q., AND XUE, C. J. Mismatched memory management of android smartphones. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)* (2019), USENIX Association.

[22] MAROTTA, R., IANNI, M., SCARSELLI, A., PELLEGRINI, A., AND QUAGLIA, F. A non-blocking buddy system for scalable memory allocation on multi-core machines. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)* (2018), pp. 164–165.

[23] MATHUR, A., CAO, M., BHATTACHARYA, S., AND DILGER, A. The new ext4 filesystem : current status and future plans. In *In Proceedings of Linux Symposium* (2007), pp. 21–33.

[24] MEI, L., HU, R., CAO, H., LIU, Y., HAN, Z., LI, F., AND LI, J. Realtime mobile bandwidth prediction using lstm neural network. In *Passive and Active Measurement* (Cham, 2019), D. Choffnes and M. Barcellos, Eds., Springer International Publishing, pp. 34–47.

[25] MITTAL, G., YAGNIK, K. B., GARG, M., AND KRISHNAN, N. C. Spotgarbage: smartphone app to detect garbage using deep learning. *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (2016).

[26] MUCCINI, H., FRANCESCO, A., AND ESPOSITO, P. Software testing of mobile applications: Challenges and future research directions. *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings* (06 2012).

[27] NATARAJAN, N., SHIN, D., AND S. DHILLON, I. Which app will you use next? collaborative filtering with interactional context. pp. 201–208.

[28] NIEL LEBECK, ARVIND KRISHNAMURTHY, H. M. L., AND ZHANG, I. End the senseless killing: Improving memory management for mobile operating systems. In *USENIX Annual Technical Conference (USENIX ATC '20)* (2020), USENIX Association.

[29] OH, G., KIM, S., LEE, S.-W., AND MOON, B. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1454–1465.

[30] PAGE, AND HAGINS. Improving the performance of buddy systems. *IEEE Transactions on Computers C-35*, 5 (May 1986), 441–447.

[31] PARATE, A., BÖHMER, M., CHU, D., GANESAN, D., AND MARLIN, B. M. 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* (2013), UbiComp '13, ACM, pp. 275–284.

[32] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (2006), CASES '06, ACM, pp. 234–241.

[33] PELEGRIN, W. Google Pixel 3 is unable to shuffle between a few apps at a time. https://www.androidauthority.com/google-pixel-3-memory-issues-917255/, 2018.

[34] PYROPUS TECHNOLOGY. Memory test tool memtester. http://pyropus.ca/software/memtester/, 2017.

[35] SCHOON, B. Google Pixel 3 kills background apps. https://9to5google.com/2018/10/22/pixel-3-memory-management-issue-background-apps/, 2018.

[36] SHEN, K. K., AND PETERSON, J. L. A weighted buddy method for dynamic storage allocation. *Commun. ACM 17*, 10 (Oct. 1974), 558–562.

[37] SHIMP208. Android debug bridge (adb) tool. https://androidmtk.com/download-minimal-adb-and-fastboot-tool, 2019.

[38] SIMS, G. How much ram does your phone really need in 2019? https://www.androidauthority.com/how-much-ram-do-you-need-in-smartphone-2019-944920/, 2019.

[39] SISOFTWARE. Memory perfromance. https://www.sisoftware.co.uk/author/cas-admin/page/5/, 2017.

[40] YOO, Y.-S., LEE, H., RYU, Y., AND BAHN, H. Page replacement algorithms for nand flash memory storages. In *Proceedings of the 2007 International Conference on Computational Science and Its Applications - Volume Part I* (Berlin, Heidelberg, 2007), ICCSA'07, Springer-Verlag, pp. 201–212.