MemSaver: Enabling an All-in-memory Switch Experience for Many Apps in a Smartphone

Prajwal Challa
vxc5208@mavs.uta.edu
University of Texas at Arlington
Arlington, Texas, USA

Baohua Song
21cnbao@gmail.com
Coredump Limited
Auckland, New Zealand

Song Jiang
song.jiang@uta.edu
University of Texas, Arlington
Arlington, Texas, USA

ABSTRACT

The availability of diverse applications (apps) and the need to use many apps simultaneously have propelled users to constantly switch between apps in smartphones. For an instantaneous switch, these apps are often expected to stay in the memory. However, when a user opens more apps and memory pressure increases, Android kills background apps to relieve the memory pressure. When the user switches a killed app back to the foreground, the user experiences a laggy response that compromises his experience. To delay this killing under memory pressure for a smoother user experience, we propose *MemSaver*, a low-cost approach for preemptively swapping selected pages of the background apps out of memory to avoid or postpone the killing of apps while ensuring their near-ideal switch time. MemSaver uses pages accessed during events similar to the switch and about the same app context for predicting the pages to be accessed in the next switch. Evaluations on OnePlus 9 Pro using representative apps show that up to 60% of app's memory (RSS) can be saved while maintaining the switch time within the acceptable range.

CCS CONCEPTS

• **Software and its engineering → Memory management**; • **Human-centered computing → Ubiquitous and mobile computing systems and tools**.

KEYWORDS

Android, Memory reclamation, App hot launch

ACM Reference Format:
Prajwal Challa, Baohua Song, and Song Jiang. 2024. MemSaver: Enabling an All-in-memory Switch Experience for Many Apps in a Smartphone. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3629526.3645050

1 INTRODUCTION

The explosive advancement of mobile technology has carved itself into the daily lives of people. Combining ever-increasing computational power along with a wide variety of feature-rich mobile applications (apps), use of smartphones have become an integral part in one's daily life. To accommodate their diverse needs, studies have shown that smartphone users usually run ten or more apps daily [7], often requiring a large memory capacity for a smooth user experience. However, smartphone manufacturers often have to limit the amount of DRAM due to a trade-off with affordability and battery capacity. This leads to high memory pressure when the user wants to keep more apps alive in memory.

Mobile OSes, like Android OS, follow the foot steps of traditional desktop/server OSes and support virtual memory, which is an approach that allows multiple processes to run concurrently even with limited physical memory via page swapping. The swapping strategy searches the space occupied by any of the in-memory processes for pages that are less likely to be accessed and swap them out to the secondary storage to make more free memory available. However, this widely-accepted practice of memory usage control becomes highly undesirable in the smartphone environment. In a smartphone, there is only one foreground app at a time that is actively interacting with the user and the remaining apps run in the background. Performance of the foreground app is of the highest priority as it directly determines user experience. In mobile devices, the swapping strategy, despite its benefits, may potentially bring unacceptable performance issues for the foreground app. As some of its pages may be selected for swapping due to lack of recent accesses, a shift of working set to access these pages again will result in a surge of page faults and turbulent user experience. Even worse, using the swapping approach for tackling high memory pressure could lead to memory page thrashing and render the mobile device inoperative. Disrupting use experience of the app the user is actively interacting with should be avoided at any cost and be used only as a last line of defence under extremely high memory shortage.

To this end, Android chooses to relieve high memory pressure by first killing background apps using its Low Memory Killer Daemon (*lkmd*) [12]. To protect the foreground app from being subject to swapping, lkmd kills the least essential app(s) to free memory as a response to high memory pressure. Killing apps is carried out in a selected manner where apps are assigned different priorities based on their execution state. For example, an app that contains Android Activities that are still visible to the user or runs background services is of higher priority than apps that are not visible to the user (background state). It is noted that the killing usually does not affect the app's functionalities. When an Android app correctly implements its Activity component for saving its current state, the state will be saved before its killing. When the user switches back to the app, its Activity will restore all its visible state as if it remained in the background.

ICPE '24, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645050

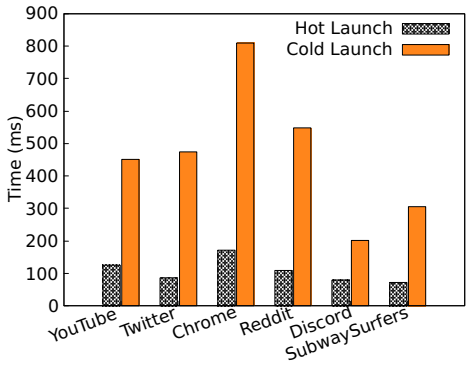


Figure 1: Hot and cold launch latencies for various apps in OnePlus 9Pro.

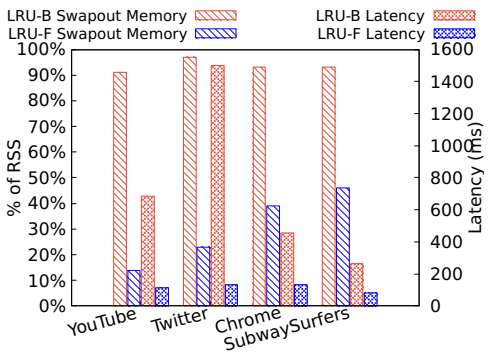


Figure 2: Amount of memory swap-out and hot launch latency with different LRU strategies for various apps, including LRU-background (denoted "LRU-B") and LRU-foreground (denoted "LRU-F").

As users interact only with the app displayed on the screen and usually keep many apps in the background, frequent switches between the apps are expected. Studies have found that users switch between apps over 100 times a day [2]. With such a high switch frequency, switch time is highly impactful on users' perceived smartphone service quality. An app may be switched from its background state to the foreground state when it still runs in the background. This switch is named *hot launch*. In contrast, a switch of an app to the foreground after it has been killed is named *cold launch*. To understand the impact of lkmd on the switch time (which turns a hot launch into a cold launch), we select some popular apps and experimentally compare their hot and cold launch latencies in an Android smartphone (see Section 4 for details). These apps include entertainment app (YouTube), social media apps (Twitter, Reddit, Discord), utility app (Chrome), and gaming apps (Subway Surfers). The results are shown in Figure 1. The hot launch latency of each of the apps is much lower than its cold launch latency. As shown, the hot launch latency is around 100 ms. Studies have suggested that when an event's response time is less than 100 ms people feel that the event is instantaneous [1]. And a response time less than 150ms does not compromise user satisfaction [14]. A time that is significantly higher than the 150 ms latency indicates a laggy response. For example, cold launch latencies of most of the apps are over 400 ms. Repeated cold-launch experience due to the aggressive lkmd's action to reclaim memory is annoying, though the effort is currently deemed necessary and often unavoidable.

While it is known that system-wide page swapping takes the risk of compromising foreground app's user experience, selected killing of background apps leads to laggy smartphone use experience. In this paper, we propose a solution, named *MemSaver*, that reclaims memory pages to ease memory pressure without killing apps or compromising the foreground-app's user experience. It carries out page swapping for background apps with minimal impacts on their hot launch latency. To this end, there are some significant challenges to address, including how to predict pages currently in the memory space occupied by a background app, or its RSS (Resident Set Size), that will be (or equivalently, will not be) accessed in its next hot launch, how to collect history access information for the prediction without disruption of foreground app's execution, how to store the information in a light-weight way, and how to strike a trade-off between saving memory and keeping launch latency low.

To this end, we make a number of contributions in the paper to address the challenges.

- We show that the commonly-used LRU-like history access information becomes much less relevant for prediction of pages to be used in the next hot launch.
- For the first time we found that pages accessed in the history hot launches of a context are highly predictive of those to be accessed in the upcoming hot launch of the same context.
- We developed a series of low-cost approaches to collect and record the relevant history accesses.
- We implemented Memsaver into Android and extensively evaluate its performance on six commonly-used apps. The results show that application's RSS can be reduced by up to 60% while keeping the hot0launch latency in an acceptable range.

2 RELEVANT HISTORY FOR PREDICTION

In the swapping of a background app, we need to predict its pages that are likely to be accessed in its next hot launch and only keep them in the memory. Obviously, we have to make the prediction based on access history. A common wisdom is to look into the recent accesses and identify pages that have been recently and frequently accessed - the LRU strategy used in Linux (the Android's kernel).

2.1 Recency-based History

The LRU policy is based on recent access history, or recency-based history. To understand the impact of using recent access in a page swapping strategy on an Android app, we design two controlled experiments. When an app is in the background, a time is chosen as its *swap moment* when its selected pages are swapped out of the memory to the flash. In this study, the intended swap moment is usually well before the available memory is to be exhausted to keep enough idle memory in the system always available. This is important for smooth user experience because the foreground app

or newly started apps may need substantial amount of free memory at any time to quickly expand its memory allocation.

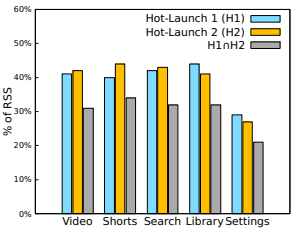
The two experiments differ at how the "recently" in the LRU policy is defined. In the first experiment named *LRU-background*, only the time period from when the app has completed its transition to the background and before the swap moment is considered as "recent". In the second experiment named *LRU-foreground*, the time period starts when the app is still in its previous foreground (10 seconds before a switch to the background). That is, it covers recent accesses in the last foreground execution. In each of the experiments, only pages that are accessed during its defined recent time period are kept in the memory in the swapping. This is a simulation of LRU's behavior.

We are interested in knowing (1) how many pages can be swapped and (2) how longer the next hot launch latency will become after a swapping. The desired result is that significant memory can be saved without substantial increase of the latency. Figure 2 shows percentage of the app's RSS that can be reclaimed and the corresponding hot launch latency for various apps. We observe that in *LRU-background* very few pages are accessed and most of the RSS pages (over 90%) can be swapped. However, this large memory saving comes with an unacceptably high hot launch latency. For example, for YouTube the latency is increased by over 6X over its 120ms ideal latency (with no swapping). In contrast, *LRU-foreground* has little increase over the latency. However, it saves much less memory (*LRU-background* can swap 2.1X-6.5X as many pages as *LRU-foreground* from the memory). However, the experiments suggest that making the choice by adjusting the recency does not lead to a solution with both goals (large memory swapout and low hot launch latency) well achieved. The key to a success relies on the accurate prediction of pages to be accessed in the upcoming hot launch.

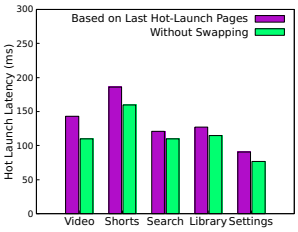
2.2 Event-based History

While the recency-based history is not well indicative of pages required in the next hot launch (aka hot-launch pages), we need to turn to more relevant history. Similar to process scheduling, an app's hot launch involves a fixed set of operations and data accesses to re-establish its previous execution state. Accordingly, the set of pages in different hot launches are likely to bear some similarity, and provide a clue on which pages should be retained in the memory. As a hot launch is a user-triggered event, we are defining and exploiting a locality based on the same type of events in the history.

To observe whether such similarity exists across the hot launches in an app's execution, we examine and compare the pages accessed in consecutive hot launches of YouTube. To assess the potentially maximal similarity, we consider an app's activity context in the investigation. An app usually has a number of predefined activity contexts, such as video, shorts, search, and library in YouTube. A hot launch resumes its foreground execution in the same context as the one it stayed in immediately before its switch to the background. Resuming the execution in different contexts may require different context-specific pages. Therefore, in this experiment we only conduct the comparison between hot launches of the same context.



(a) Consecutive hot launch page's overlap



(b) Hot launch latency with and without swapping

Figure 3: Overlap of hot launch pages between two consecutive hot launches (H1 and H2) with the same context as well as impact of using the last hot-launch pages for swapping on the hot launch latency.

Figure 3a shows the size of hot-launch page set as a percentage of its RSS for each of the two consecutive hot launches of a specific context. The RSS represents the amount of memory held by the app in the background. The figure also presents size of the intersection of the two sets. As shown, the two sets contain about 80% of common pages, suggesting that accesses in a recent hot-launch event are predictive of that in the next hot launch. The hot launch pages are almost all in the memory before a swapping. This is the reason why a background app has a satisfactory hot launch latency if no swapping was conducted. If we only keep the last hot launch pages in the memory (about 30-45% of the RSS), about 55-70% of the memory pages can be swapped. The potential memory saving is significant. Meanwhile, Figure 3b shows next hot launch latency with or without swapping based on the event-based history. The time increase is about 10-30 ms, which is moderate and stays in the acceptable range.

2.3 Challenges of Exploiting History

While we reveal the more relevant locality in the event-based history, there are a number of challenges on translating this finding into an online system design. First, it can be too expensive to online detect pages accessed during a hot launch. A straightforward approach for the detection requires two scans of the app's entire page table: in the initial scan reset reference bits in the PTEs (Page Table Entry) and then check them in the following scan when the launch is completed. Each of the full-table scans can be very expensive. The cost is especially problematic for the first one, as it takes place when the user initiates the app's hot launch. It would essentially block user's interaction during the scan.

The second challenge is on the cost of recording the detected pages. Following the conventional wisdom, it is tempting to consider introducing a new data structure to accurately record the pages. However, this approach would carry significant time and space overheads. In particular, the data structure for tracking all accessed pages would increase the memory usage, which is in conflict with the goal of this work for reducing memory footprint. In addition, it may be necessary to record multiple sets of pages, each about a different history event/context, which further inflates the costs.

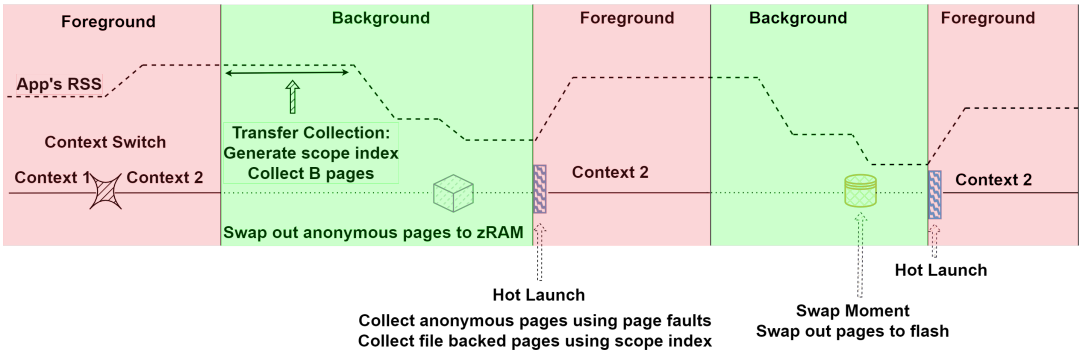


Figure 4: Representative events in an app's life time.

Third, assuming that a light-weight approach for detecting and storing hot-launch pages is available, we may have multiple hot-launch page sets recorded during an app's execution, each for one recent history hot launch. These hot launch events are of different recency, and may be associated with different contexts. It is unknown which of the page sets should be used in the decision of page swapping for more memory saving and lower launch latency. For example, in a situation where a hot launch of the same context hasn't been recorded, or the most recent hot launch is about a different context, it is not straightforward to understand the impact of using the sub-ideal history data and to make the best page swapping decision. The design of MemSaver addresses the challenges.

3 THE MEMSAVER DESIGN

An ideal design should have a light-weight approach to accurately detect history pages, along with an efficient way to store the pages. Such approach should also ensure that history information is utilized in a manner such that app's RSS is effectively reduced and does not lead to undesirable latency in the next hot launch. In this section we present MemSaver, an efficient design for identifying and recording pages that are likely to be accessed in an app's upcoming hot launch and selectively retaining them in memory and swapping the remaining pages to the flash. We dive into various design choices and take a close look at their corresponding performance implications. As MemSaver is deployed in Android, the design is tailored to the smartphone's hardware and software designs.

3.1 Phases in an Android App's Execution

It is necessary to know the phases and relevant events experienced during an app's execution that are relevant to the MemSaver's design. As illustrated in Figure 4, the app constantly alternates between the foreground state (the red zone in the figure) and background state (the green zone). In the red zone, any substantial overhead added by the external facility is likely to be felt by users and compromises their use experience. In contrast, a green zone allows such overhead without users' notice. Therefore, all MemSaver's operations are carefully carried out in the green zones.

As the swapping takes place at a moment during the background execution, we need to understand the memory usage during the time period to select the moment. In Android, after an app is sent to the background, within the first few seconds (about 5 seconds)

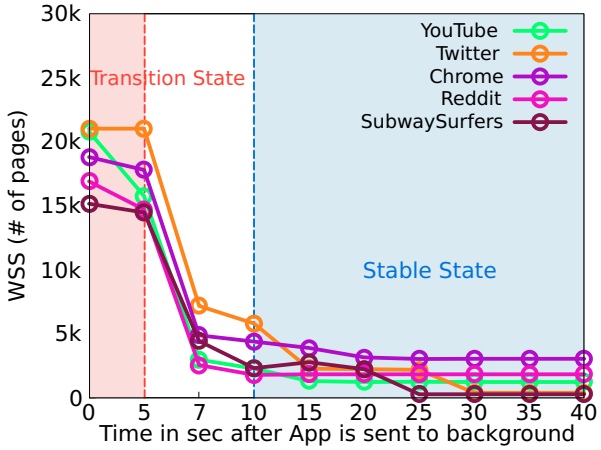


Figure 5: Background App's WSS reduction

we observe that the app's resident memory space (RSS) remains about the same as that of the app during its foreground execution, as shown in Figure 4. After this it reduces to a smaller size by releasing its allocated space and then stays at the memory footprint. We name these two time periods in the background as the transition and stable states, respectively. In the two states the size of actively accessed memory space (aka working set size or WSS) is very different. Figure 5 shows the change in the WSS of various apps over time after the app is sent to the background. During the transition state, the Android app is in the *onPause()* life-cycle activity state, a state where Android OS expects the app likely to go back to *onResume()* activity state (re-launch back to the foreground) soon. Consequently, an app's RSS is observed to remain the same as that of in the foreground to continue its normal execution in the background for a short time period in anticipation of a possible quick switch back. After this period WSS is dramatically reduced and stabilized as the app continues its execution into the stable state. This unique WSS/RSS behaviour of Android apps is one of the necessary elements the design of MemSaver will take into account. Integrating Android specific functionality along with careful design enables MemSaver to efficiently detect, collect, and use hot launch pages.

3.2 Detection and Storage of Hot-Launch Pages

To allow access prediction based on prior hot-launch events, MemSaver needs to collect pages accessed during a hot launch (hot launch pages). As a hot launch needs to be associated with an activity context, MemSaver integrates itself with Android OS to identify the current context when user interacts with the app in the foreground. As only the last context before a switch to the background will resume its execution in the next hot launch, MemSaver appends the last context's name before the app goes into the background to the next hot launch. That is, the context of a set of history hot launch pages is known when they are used for deciding swappable pages.

When MemSaver is informed by Android of an app's hot launch, MemSaver starts to collect accessed pages during the launch. As we have mentioned, the two full-scan approach requires a traversal of an app's entire address space, whose cost is unacceptable. As an example, for YouTube it represents a coverage of a 17GB address space, much larger than its RSS at the time of the hot launch (about 400MB). Accordingly, the time spent on one scan is about 100ms, which is too expensive.

In an app's address space, there are two types of pages: anonymous and file-backed pages. Since they have different properties, we design different light-weight approaches for access detection. As anonymous pages are more scattered in a larger space, it's not effective to identify a much reduced scope for efficient scanning. Instead, MemSaver artificially generates minor page faults to reveal what pages are accessed in a hot launch. Specifically, we leverage a Linux facility – zRAM – that is enabled in Android by default. zRAM is an in-memory compression-based swap space. When the app is in the background, MemSaver swaps out all its anonymous pages to the zRAM using *madvise()* syscall with the *MADV_PAGEOUT* flag. At this time the hot launch has not yet started. The swapout cost during the background execution is not felt by the user

During the hot launch, any accessed page will trigger a page fault for loading it from the zRAM. Consequently, MemSaver can identify the hot-launch pages by intercepting page faults during their handling in Linux. As page faults in zRAM do not involve any flash I/O operations, and only decompression of anonymous pages instead, the cost is moderate. Compared to that of true page faults from the flash, this cost is close to that of minor page faults.

The approach of using zRAM for generating page faults cannot be applied to the file backed pages, as these pages would only be swapped to the file system on the flash. To address the issue, we choose to narrow the scan scope. This is made possible by the observation that file-backed pages (1) are often clustered and (2) are mostly also accessed in the foreground phase. The idea is to collect the VMAs (the data structure the Linux kernel uses to manage contiguous virtual memory areas) covering file backed pages that have been accessed in the foreground. This set of VMAs will be the focused scope where MemSaver will look for accessed file-backed pages in the next hot launch.

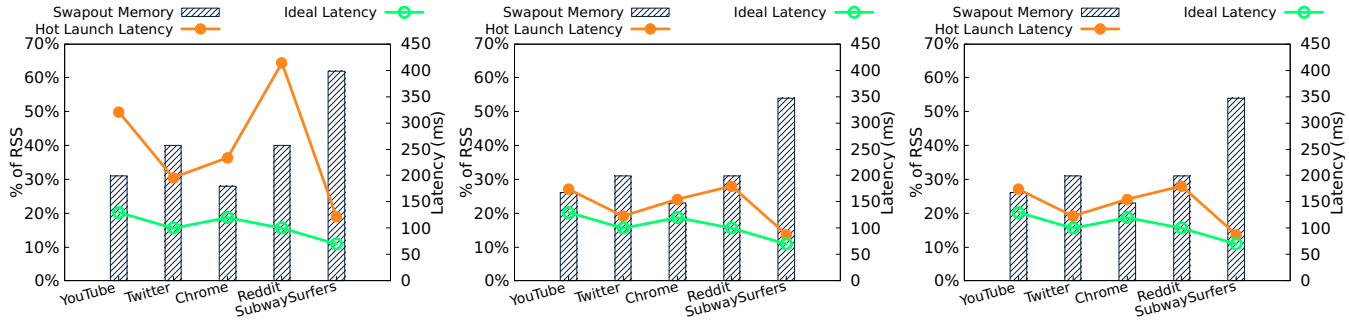
However, detection of the VMAs for accessed file-backed pages during the foreground execution is intrusive to the foreground app's execution. To address this issue, we leverage the unique WSS behaviour of Android apps, which is the existence of the transition state. The brief pseudo background period is in the green zone but retains foreground's access behavior. Therefore, MemSaver moves its detection of the focused scope from the foreground to this special background period. As this operation, named *transfer collection*, is in the green zone, it is affordable to use two scans to identify the VMAs that have been accessed. The VMAs will then be the scope for the two-scan operation in the next hot launch for file-backed pages. To further reduce the scope we use the 2MB aligned memory region in a VMA as the unit (representing 512 4KB-pages) for tracking file-back accesses. All detected VMA regions are recorded in an index (the scope index) for quick access. Within this focused page-scan scope, the two scans in the reduced scope takes only about 1 ms. This time overhead becomes acceptable even in the red zone.

Instead of using a data structure to store the detected hot launch pages, MemSaver records them in the unused PTE bits (bits 60-63) in the page table. At the time when an accessed page is detected, its PTE is likely to be the CPU cache. The cost for updating the PTEs is negligible. To determine pages for swapping, MemSaver does need to scan the page table to know the recorded hot launch pages. However, this will take place only when the app is in the background. Furthermore, using the idle PTE bits avoids inflating memory usage. The availability of four bits for recording accesses in the history presents MemSaver with choices of tracking multiple selected events. As a page swapping policy, MemSaver needs to know the specific events whose page accesses should be collected and which of the history events should be involved in a swapping decision making. To this end, we adopt a heuristic approach whose design is driven by targeted experiments (to be discussed).

3.3 Incorporating Working Sets in the Recent History

We have suggested retaining history hot launch pages in memory during the swapping. In the meantime, there are two sets of actively accessed pages (working sets) that may need to be kept in memory. One of them is that right before the app switches to the background. A hot launch is essentially a resumption of the last foreground execution. Pages in the working set at the end of the foreground phase is likely to be accessed at the beginning of the next foreground phase (i.e., the hot launch time period). The other working set is the one during the stable state in the background. Its pages need to be kept in memory even after the swapping to support its background execution. One challenge is that the detection of pages in the foreground working set would take place in the red zone in Figure 4. Its cost is simply not acceptable during the foreground execution. Fortunately, the existence of the transition state right after the switch to the background makes a non-intrusive detection of the pages possible. As it is in the background, we can use the two-scan approach to get its working set. Because the app maintains its foreground activities in the transition state, the working set right after the switch can be an approximation of the one before the switch. We denote this background working set B_1. In contrast, the working set in the stable state is denoted B_2.

MemSaver initiates its swapping during the background execution after the transition state. To understand the impact of additionally keeping B_1 and/or B_2 on the swapping efficacy in terms of swap-out memory amount and hot launch latency, we experiment with keeping different combinations of the working sets along with



(a) Keep hot launch pages H_1 and Working set B_2 in memory

(b) Keep hot launch pages H_1 and Working sets B_1 and B_2 in memory

(c) Keep hot launch pages H_1 and Working set B_1 in memory

Figure 6: Impact of Background Active pages on RSS reduction and Hot launch latency

the set of pages accessed in the last hot launch in memory (denoted as H_1). Note that we assume the app stays in the same context across the events. Apparently, it is necessary to keep B_2 in memory, as pages in B_2 continue to be accessed after the swapping (during the background execution period). The question is whether B_1 also needs to be considered. In the first experiment, we leave out B_1. Figure 6a shows the amount of swap-out memory (as a percentage of its RSS at the time of the swapping) and the corresponding hot launch latency after swapping any pages in the RSS that are not in either H_1 or B_2 for various apps. As seen, though an app's RSS can be effectively reduced (up to 60%), the hot launch latency is much higher than the ideal hot launch latency (the one without swapping). As it is necessary to maintain an acceptable hot launch latency, it is not sufficient to only consider B_2 in the swapping decision.

In an attempt to reduce the hot launch latency, we add the B_1 to the set of pages that are kept in memory (i.e., B_1, B_2, and H_1) in the swapping decision. Figure 6b shows its swapout memory amount and the corresponding launch latency. As expected, the amount of swapout memory is reduced. However, the reductions represent small percentages of respective RSSes. In the meantime, the hot launch latencies become close to their respective ideal ones. To know if it is still necessary to consider B_2 after B_1 is included, we then remove B_2 (i.e., only B_1 and H_1 pages are not swapped out). Figure 6c shows that both the memory swap-out amount and the hot launch latency do not have any substantial changes. Examining background active pages, we observe that some of the B_1 pages continue to be active over the execution of the app in the background and consequently cover about 95% of B_2 pages. Figure 7 shows the overlap of B_1 and B_2 for various apps. For this reason, regarding B_1 and B_2, B_1 MemSaver only collects B_1 pages of an app and retains these pages in memory (if they have been in memory before the swapping).

3.4 Incorporating Context-aware Hot-launch Pages

As discussed, the set of pages that have been accessed during the prior hot launch is a potentially strong indicator of pages to be used in the upcoming hot launch. Multiple such prior page sets may have been recorded, and they may be associated with different

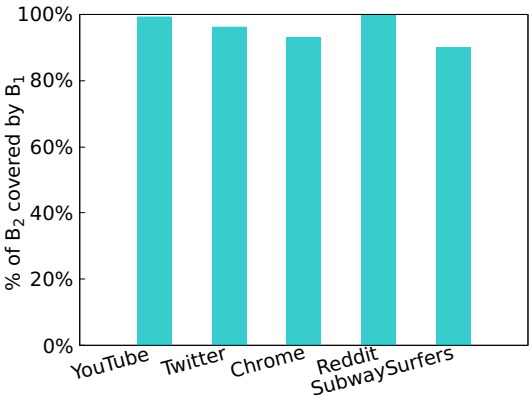


Figure 7: Overlap between B_1 and B_2

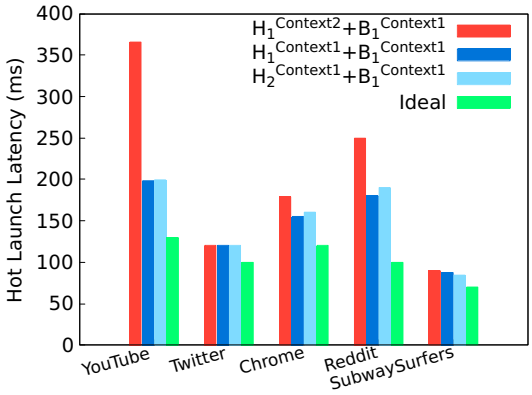


Figure 8: Hot launch latency when different history hot launch page set is used for the swapping before the hot launch.

app contexts. Intuitively, the most relevant one among all possibly recorded page sets is the one about the most recent hot launch and of the same context as the one for the upcoming hot launch. To

Table 1: Apps and Contexts

App	Context1	Context2
YouTube	Video	Shorts
Twitter	Search	Tweet
Chrome	Web page	Tabs
Reddit	Message	Search
Subway Surfers	Game Pause	Shop

have a sensible design, we need to experimentally confirm or disconfirm this conjecture and the impact of different sets of history hot launches. In the interest of clarity, the set of pages accessed in a history hot launch (H) is denoted H_k^{cxt}, where cxt is the context when the hot launch take places, k indicates how recently the launch is. For $k = 1$, it is the most recent one, or the last one, and for $k = 2$, it is the second to the last one, and so on. Figure 8 shows the hot launch latency when different history access information was used for a swapping in the background before the hot launch for various apps. The latency is compared with the ideal latency, the one when swapping is not conducted. In the experiment, two contexts were selected for each of the apps (Youtube, Twitter, Chrome, Reddit, and SubwaySurfers), namely *context1* and *context2*. Table 1 depicts the contexts for each of the apps. In the experiments, *context1* is designated as the current background context, which is also the one for the upcoming hot launch. *Context2* is a context appearing in the past. In a swapping, we keep pages in the most recent background working set B_1 (with *context1*) and pages in an H set. As shown in the figure, if the history H is $H_1^{context1}$ (the assumed most relevant H), or pages in the $H_1^{context1} UB_1^{context1}$ are retained in memory, the next hot launch latencies stay in the acceptable range (about less than 200ms). Admittedly, some of them are substantially higher than the ideal one (e.g., Reddit). In practice, smartphone use experience is more impacted by the unexpectedly long latency than by latency variations within an acceptable range.

While an H is associated with a context, we need to understand how strongly it is correlated with its context. This is important because sometimes a history H with *context1* (the current context) is not available. In the situation, MemSaver may have to choose an H whose context is different from *context1* for its swapping decision. Figure 8 also shows the hot launch latency after retaining pages in $H_1^{context2} UB_1^{context1}$ in memory during the swapping. Using an H with an unmatched context (*context2*), the hot launch latency can increase to an unacceptable level (370ms for YouTube). The latency for Reddit also increases to over 250ms. These drastic increases do not take place for all the apps (e.g., the increases for Twitter and SubwaySurf are moderate.). To be a reliable design, memSaver chooses to be conservative by not carrying out swapping if an H with a matched context has not yet been recorded.

A more common situation is that a history H with a matched context does exist but it is not the most recent one. To understand the impact of the recency on the swapping effectiveness, we experiment with the case where pages in $H_2^{context1} UB_1^{context1}$ are retained in memory. Figure 8 also shows its hot launch latency. As shown, using $H_2^{context1}$ produces the hot launch latency almost the same as that using $H_1^{context1}$. This observation suggests that

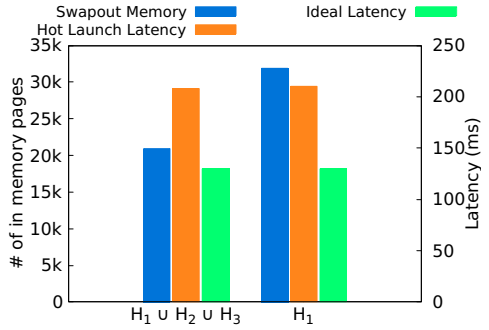


Figure 9: Comparison of swapout memory and hot launch latency for YouTube between using its three recent Hs and using only one recent H. All Hs are associated with the same context ('video').

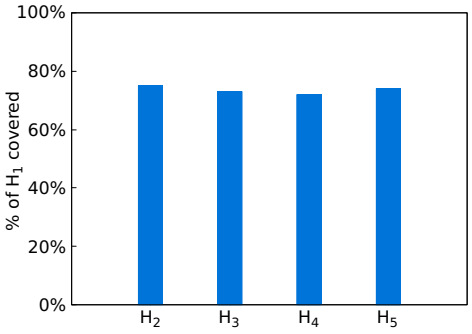
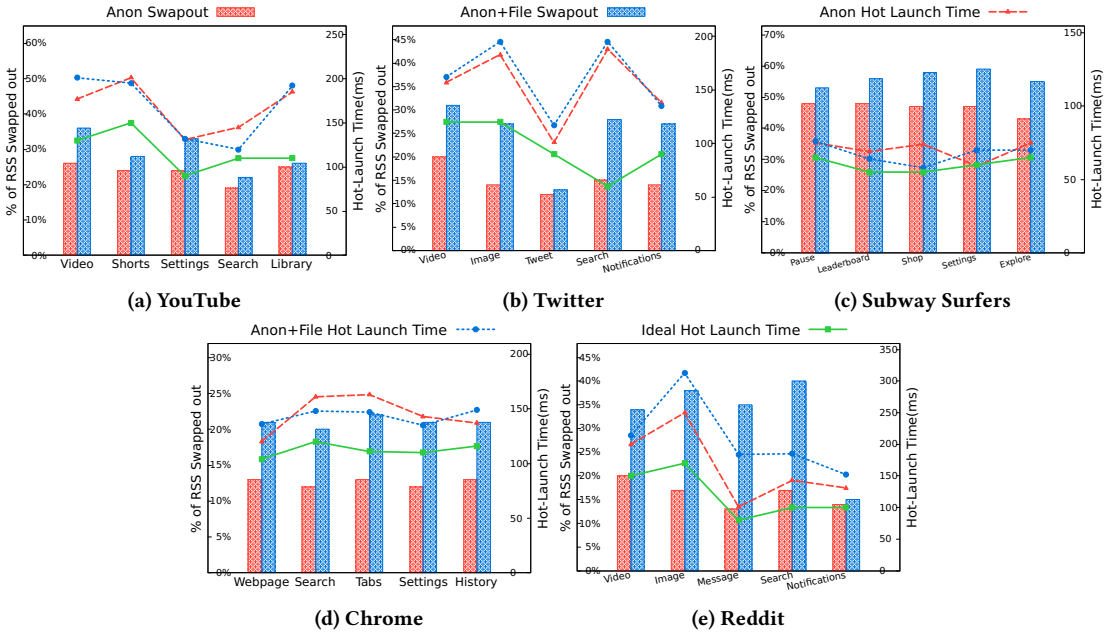


Figure 10: Overlap of earlier H with the most recent H (H_1 for YouTube. All Hs are associated with the same context ('video').

recency, or time-defined locality, is less relevant for identifying useful access history. This is consistent with the rationale of adopting event-based history in this study. Therefore, MemSaver chooses to use the latest H with the matched context (if available), whether it is for the latest hot launch or not, in its recorded history for swapping out the app's memory.

The last design question is on the choice of H when there are multiple Hs with the matched context in the recorded history. One might expect that using all the 'qualified' Hs would help to keep more pages to be used in the next hot launch in memory and thus further reduce the hot launch latency. To understand its impact, we carried out swapping for YouTube with 'video' as *context1* (the current context) and using either one or three history Hs of the same *context1*. Figure 9 shows their respective hot launch latency and number of swapout pages. Using additional Hs doesn't help to further reduce the hot launch time. It is not necessary to consider more than one (earlier) Hs. Furthermore, using multiple Hs has a side effort. As shown, using three Hs reduces the number of swapout pages by about 32%. Figure 10 shows overlap of each of the four earlier Hs with the H_1 (all Hs are associated with the same context). The overlap is consistently about 75%. This suggests that this 75% subset represents an invariant that persists over the different hot launches. It is this invariant that contributes to



(a) YouTube **(b) Twitter** **(c) Subway Surfers**

(d) Chrome **(e) Reddit**

Figure 11: RSS reductions and hot launch times with different swapping methods for apps at different contexts.

MemSaver's effective prediction. The remaining 25% subsets in the respective Hs are not essential for the prediction. This corroborates the finding shown in Figure 8 about using a non-most-recent H. As long as it is an H of the same context, the H can effectively serve the purpose. For this reason, as an optimization MemSaver doesn't have to collect and record H for every hot launch.

3.5 A Summary of MemSaver Policy

MemSaver consists of two operations: access collection and page swapping. There are four bits for each page in an app's page table entry for recording history, each for one history event. Because of limited number of bits and cost of access collection, MemSaver follows this collection policy.

(1) One bit is reserved for the B_1 pages. B_1 is collected with each switch to the background (during the transition phase);
(2) The remaining three bits are used for up to three Hs. They are logically organized as an LRU stack. The most recently recorded H is at the stack top. The one at the bottom is to be replaced by recording of a new H;
(3) For a new hot launch with *context1*:
  (a) If there hasn't been an H of *context1* recorded in the stack, collect its accessed pages and place the corresponding H at the stack top (by replacing the one at the bottom);
  (b) Otherwise, an H of *context1* exits in the stack. In the case, if it's in the stack bottom, collect a new H to replace it. Otherwise, skip the collection for this hot launch. This is for three reasons. (1) A history H of the same context is still usable; (2) Making fewer collections helps improve efficiency. And (3) an H earlier than two other Hs of different contexts is updated for freshness.

In the meantime, MemSaver follows this swapping policy.

(1) If there are not any H of the matching context, do not swap.
(2) Otherwise, use the H of the matching context and B_1 to decide the pages that will be retained in the memory and swap out other in-memory pages.

4 ADDITIONAL EVALUATION RESULTS

To understand the efficacy of MemSaver, we implemented and evaluated it in Android using various representative apps (YouTube, Twitter, Subway Surfer, Chrome, and Reddit). We compared its hot launch latency after an app's RSS has been reduced against the hot launch latency of all-in-memory apps (the ideal case). The evaluations were performed on a OnePlus 9 Pro phone with Qualcomm® Snapdragon™ 888 processor containing 12GB RAM and running Android version 11.

As mentioned, MemSaver has two objectives for a background app, which are to reduce its memory footprint and keep its hot launch latency within an acceptable range. While MemSaver uses different methods for detecting hot-launch pages (page faults from zRAM for anonymous pages vs. page table scans in a reduced scope for file-backed pages), we break down their impacts in the experiments. For each app, we assume availability of $H_1^{context1}$ with various current contexts (*context1*). Figure 11 shows the RSS reduction in percentage and the corresponding hot launch latency when swapping is applied only on anonymous pages or on both anonymous and file-backed pages. It also shows the ideal hot launch latency.

There are some interesting observations. First, RSS can be substantially reduced. With only anonymous pages are considered, up to 48% of the memory can be saved with an average of 23%. If file-based pages are also considered, up to 60% of memory can be saved with an average of 34%. Second, the increase of hot launch latency is mostly in the range of 10-50 ms, leaving the latency usually under 150 ms and thus making users mostly feel it as an instantaneous

launch. This is an impressive result. As an example, with an average of 33% memory saving, a smartphone that currently can hold 10 background apps in the memory without being killed will be able to keep 15 apps alive in the background with MemSaver. For a user whose number of actively used apps is moderately over existing limit, with MemSaver his bumpy app switch experience is removed. Third, additionally considering file-backed pages for swapping either doesn't increase the launch time substantially or sometime even reduce the time. This is because file-backed pages are usually sequentially accessed and prefetching could be activated. Four, we do observe that for some apps (e.g., Figure 11b and Figure 11e) at some contexts (e.g, search and image) MemSaver may occasionally produce a hot launch latency as high as 200-300 ms due to excessive number of page faults. In summary, in general MemSaver can reduce apps' RSS while maintaining near-ideal hot launch latency. In a few cases the latency can be high when compared to the ideal time. However, if compared to the often much higher cold launch times (see Figure 1), the time still represents an improvement.

5 RELATED WORKS

User experience is often dictated by the availability of memory in smartphones. Prior works have focused on improving user experience on many fronts including improving app's memory management and launch performance. To improve app switch time, ASAP [13] uses multiple threads to prefetch pages that are likely to be accessed during a switch. IORap [4] in Android 11 predicts data required by an app ahead of time by profiling its I/O in several cold runs. FALCON [15] uses information including user location and temporal access patterns to predict an app's launch time and preload its data.

Under high memory pressure Android triggers *lkmd* [9] to kill background apps to ease memory pressure. To avoid the killing, Marvin [6] swaps out memory that is less likely to be used in the object granularity with ahead-of-time swap, which requires Android Run Time (ART) modifications. In contrast, MemSaver swaps out pages unlikely to be used in an app's upcoming hot launch in an attempt to postpone the inevitable killing. It doesn't require any modification of apps themselves. SmartSwap [16] predicts least likely used app using information like location and usage history to swap out pages of those apps. Rather than predicting which app will be relaunched, MemSaver only considers app's event-based access history to selectively swap pages out of memory. To avoid disruption of foreground app, Acclaim [8] frees pages from background apps and provide them to foreground apps. A2S [5] integrates process-level kill approach and page-level swap approach by using a threshold to decide processes for killing and by estimating page lifetime to decide pages for swapping out. Instead of predicting which pages to swap out or prefectch or predicting apps as a whole [10, 11], MemSaver uses more intricate information of app-specific context to reduce app's RSS while maintaining near-ideal hot launch time. A more extensive coverage and analysis of related efforts on the improvement of app launch time can be found in the survey paper [3]. It helps to further understand that MemSaver's unique approach and techniques represent a step forward in the improvement of smartphones' use experience.

6 CONCLUSIONS

In this paper we present MemSaver, a low-cost approach for preemptively swapping selected pages of the background apps out of memory while ensuring their near-ideal hot-launch time. Different from conventional LRU-like strategies for selecting memory pages for swapping, MemSaver uniquely resorts to event-specific history to accurately determine the pages for swapping. Evaluation of an implementation of MemSaver in Android shows that up to 60% of application's RSS can be reduced while ensuring that its hot launch time remains in the range friendly to users for real apps of representative types.

REFERENCES

[1] Stuart K Card, George G Robertson, and Jock D Mackinlay. 1991. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*. 181–186.

[2] Tao Deng, Shaheen Kanthawala, Jingbo Meng, Wei Peng, Anastasia Kononova, Qi Hao, Qinhao Zhang, and Prabu David. 2019. Measuring smartphone usage and task switching with log tracking and self-reports. *Mobile Media & Communication* 7, 1 (2019), 3–23.

[3] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. 2022. A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2879–2904. https://doi.org/10.1109/TSE.2021.3071193

[4] IORAP. 2023. IORAP. https://medium.com/androiddevelopers/improving-app-startup-with-i-o-prefetching-62fbdb9c9020

[5] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. 2017. Application-aware swapping for mobile systems. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–19.

[6] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. 2020. End the senseless killing: Improving memory management for mobile operating systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 873–887.

[7] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association. https://www.usenix.org/conference/atc20/presentation/liang-yu

[8] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 897–910.

[9] LKMD. 2023. Low Memory Killer Daemon. https://source.android.com/docs/core/perf/lmkd

[10] Nagarajan Natarajan, Donghyuk Shin, and Inderjit S. Dhillon. 2013. Which App Will You Use next? Collaborative Filtering with Interactional Context. In *Proceedings of the 7th ACM Conference on Recommender Systems* (Hong Kong, China) *(RecSys '13)*. Association for Computing Machinery, New York, NY, USA, 201–208. https://doi.org/10.1145/2507157.2507186

[11] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. 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* (Zurich, Switzerland) *(UbiComp '13)*. Association for Computing Machinery, New York, NY, USA, 275–284. https://doi.org/10.1145/2493432.2493490

[12] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W. Lee, and Hongil Yoon. 2021. ASAP: Fast Mobile Application Switch via Adaptive Prepaging. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 365–380. https://www.usenix.org/conference/atc21/presentation/son

[13] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W Lee, and Hongil Yoon. 2021. ASAP: Fast Mobile Application Switch via Adaptive Prepaging.. In *USENIX Annual Technical Conference*. 365–380.

[14] N. Tolia, D.G. Andersen, and M. Satyanarayanan. 2006. Quantifying interactive user experience on thin clients. *Computer* 39, 3 (2006), 46–52.

[15] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (Low Wood Bay, Lake District, UK) *(MobiSys '12)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2307636.2307648

[16] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. 2017. SmartSwap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.