2023 Design, Automation & Test in Europe Conference (DATE 2023)

# TPP: Accelerate Application Launch via Two-Phase Prefetching on Smartphone

Ying Yuan[†], Zhipeng Tan[†],Shitong Wei[†], Lihua Yang[†], Wenjie Qi[†], Xuanzhi Wang[†], Cong Liu[†]
[†]WNLO, Huazhong University of Science and Technology
Corresponding Email: tanzhipeng@hust.edu.cn*

*Abstract*—**The fast app launch is crucial to users' experience and it is one of the eternal pursuits of manufacturers. Page fault is a critical factor leading to long app launch latency. Prefetching is the current method of reducing page faults during app launch. Before app launch, prefetching all demanded pages of the target app can speed up the app launch effectively, but it always uses the memory of several hundred MB, leading to low memory and slowing other apps' launch. Prefetching during application launch uses memory effectively, however, current methods are not aware of the order of pages accessed, causing noticeable accessing-prefetching order inversions, which results in limited acceleration of app launch.**

**In order to accelerate the application launch effectively with little memory usage, we propose a Two-Phase Prefetching schema (TPP), which performs prefetching via two phases: 1) Before the app launch, to increase the efficiency of memory usage in prefetching, TPP prefetches few critical pages with app prediction, which is based on Long Short-Term Memory (LSTM) with high accuracy. 2) During app launch, TPP prefetches the rest of the critical pages via an order-aware sliding window method, resolving the accessing-prefetching order inversions and significantly reducing the app launch latency. We evaluate TPP on Google Pixel 3, compared to the state-of-the-art method, TPP reduces the application launch time by up to 52.5%, and 37% on average, and the data prefetched before the target application started is only 1.31 MB on average.**

## I. Introduction

Providing a huge number of useful functions, smartphones have gone deep into people's daily life, work, and study. According to the most recent study from IDC, worldwide smartphone shipments are 2.86 billion in the second quarter of 2022 [**1**], and 85% of them use Android system [**2**]. Android users have always suffered the problem of system lag, especially during the app launch. The interaction between users and smartphones begins with an app launch, and the fast app launch is a precondition a the good user experience. In addition, users are used to launching applications more than 100 times a day [**3**], resulting in frequent experiences with system lag.

During the app launch, the app accesses a large amount of pages with small I/O. The speed of reading data from memory is 1000 times greater than that of flash storage [**4**], which will be longer when the read I/Os wait for direct reclaims to free memory or wait to get scheduled in the block layer. Research shows that the read I/O latency contributes to more than 75% of the time spent during lagging rendering processes [**5**].

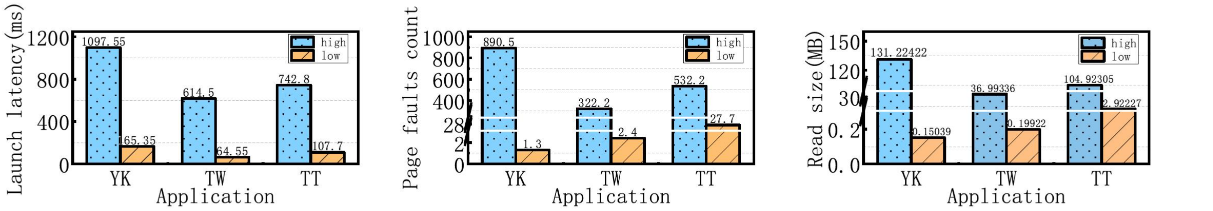
Most of the data accessed during app launch is invariant [**6**, **7**], and prefetching the invariant data is effective in reducing app launch latency. One current prefetching method is to perform **P**refetching **B**efore the **A**pp **L**aunch (PBAL), which preloads the context that the launch process needs [**8**, **9**], reduces all of the latency caused by reading data from flash storage, and accelerate app launch effectively. This method must be aware of which app to be used next by app prediction. However, the prediction is not accurate, and the accuracy of the advanced studies is less than 70% [**10**, **11**, **12**]. The memory overhead of preloading is up to hundreds of megabytes. If the prediction is wrong, the prefetched data will not be accessed and the memory is used ineffectively, leading to extra latency for other apps' launch. Another current prefetching method is to perform **P**refetching **D**uring the **A**pp **L**aunch (PDAL). This method only prefetches the data of the app which is launching, most of the prefetched data will be accessed by the app launch process, and the memory is used effectively[**6**, **7**]. However, it isn't aware of the order of read I/Os, the page accessed first may be prefetched later. That is, accessing-prefetching order inversions occur. In this case, when the pages are accessed, they have not been loaded to the memory, leading to long read I/O latency and ineffective accelerating app launch.

There is a contradiction between the effect of acceleration and memory usage. We propose a **T**wo-**P**hase **P**refetching schema (TPP) to effectively accelerate the app launch with minimal memory consumption. TPP is based on the following two observations.

**Observation1: few key launch pages are accessed in a burst at the beginning of app launch**. We trace the read I/Os during app launch with popular apps on Google Pixel3. At the beginning of the app launch, the apps demand several megabytes of pages in a short time. If these pages are prefetched during the app launch, it will be too late to load these pages into memory before they are accessed, slowing down the process of application launch. These pages are few but critical to accelerate app launch, and we call them key launch pages. To effectively accelerate app launch, these pages should be prefetched before app launch instead of during app launch. Based on this observation, we perform predictive prefetching, prefetching the few key launch pages before the target application launch based on app prediction, effectively decreasing launch latency with little memory consumption.

**Observation2: except at the beginning of app launch, the read traffic is gentle and sequential**. We record the read I/Os during app launch according to their arrival time. In spite of the beginning, the read traffic is gentle mostly, which is 1/70 of key launch pages' traffic on average. The observation gives us a chance to prefetch the pages (except the key launch pages) by sliding window. The prefetching process has enough time to load the pages to memory before they are demanded for the gentle traffic, and we call these pages common launch pages.

978-3-9819263-7-8/DATE23/© 2023 EDAA



(a) Applicaiton launch latency (b) Page faults count (c) Size of data read from storag

Fig. 1: Application launch latency and page faults count/size during app launch with popular apps, YK: Youku, TW: Twitter, TT: TikTok.

What's more, the pages in the same time window are demanded during the same time range. The sliding window prefetching loads pages by the order they are accessed, resolving accessing-prefetching order inversions during app launch and effectively accelerating app launch.

We implemented TPP on the Android kernel, and TPP includes three parts: 1) **A**pp **P**rediction (AP), predicting which application to be used next based on LSTM; 2) **L**aunch **P**ages **M**anagement (LPM), collecting which pages are demanded during application and the order they are accessed; 3) **P**refetching **M**anagement (PM), performing two-phase prefetching: predictive prefetching reads key launch pages before app launch to decrease the memory overhead, sliding window prefetching reads common launch pages to effectively accelerate app launch.

This paper is organized as follows. Section II introduces the related work for accelerating application launch and discusses their effects. Section III describes the motivation of our work. Section IV details the design of TPP. Section V evaluates performance of TPP with public application usage dataset and we conclude this paper with section VI.

## II. Related Work

**Accelerate app launch via PBAL**. POA [**8**] identifies app usage patterns by analyzing the app usage log, and prelaunches the app which will be used next. FALCON [**9**] predicts the next app based on the location and application usage sequences and preloads apps from storage. If the target app is not used, PBAL will consume the memory of several hundred megabytes. App usage prediction has been researched by several works [**18**, **19**, **20**], and the studies show that app usage is sensitive to context-info including time of the day [**18**] and app usage sequences [**19**, **20**]. However, the accuracy of the advanced studies is less than 70% [**10**, **11**, **12**], and the prelaunching and preloading based on prediction consume a large amount of memory when the prediction is wrong.

**Accelerate app launch via PDAL**. FAST [**6**] collects all of the block requests during application launch and prefetches these requests. ASAP [**7**] tracks the switch footprint for file page faults and anonymous page faults and reduces the switch time via adaptively prepaging. However, PDAL is not aware of the order of data access, which leads to a large amount of accessing-prefetching order inversions, and it has a limited effect on improving application launch latency.

**Accelerate app launch via other methods**. Many Studies improve app launch speed via memory management, reducing the direct reclaim count and refault count [**15**], avoiding making swap during the application launch [**13**, **14**]. Some studies focus on I/O management [**16**, **17**], accelerating app launch via giving higher priority to foreground app I/Os. Both of memory management and I/O management are orthogonal to PBAL and PDAL.

## III. Motivations

### A. Key Factors that delay app launch

**Page faults prolong app launch latency**. We record the page faults during app launch with three popular apps (YK: Youku, TW: Twitter, TT: TikTok). We evaluate app launch with a different count of page faults. We evaluate high count of page faults with start app from scratch, and the pages accessed are all in the flash storage, which presents the highest count of page faults. Besides, we evaluate low count with start the app twice in a raw, and most of the pages accessed during the second launch are in memory, which presents the lowest count of page faults. Fig. 1 (a) shows that the latency of the high count is much longer than which of low count for all applications. Fig. 1 (b) and Fig. 1 (c) show that the page faults count and size of the high count are larger than those of low count, which indicates that page fault is the crucial factor in delaying app launch. When the pages accessed by the app launch task are not in memory, page faults occur, and the CPU proceeding with the app launch task waits until the pages are read from storage, which increases the app launch latency.

### B. Observations on the read I/Os during app launch

**Few key launch pages give an opportunity to minimal memory overhead**. We trace the read I/Os during the app launch with three popular apps (YK: Youku, TW: Twitter, TT: TikTok). As shown in the orange circles of Fig. 2, for all apps, read I/Os arrives in a burst at the beginning of the app launch, which leads to page faults occurring in a burst, delaying app launch, and we call these pages key launch pages. The read size is little (1-2M), but the read I/Os are busy, they arrive during the first 5ms of the app launch process. If the prefetching task fetches them during the app launch, the prefetching task doesn't have enough time to fetch the key launch pages before they are demanded, which fails to accelerate the app launch. If the key launch pages are prefetched before the app launch, it will be effective in accelerating the app launch, which gives us an opportunity to minimize memory overhead on accelerating the app launch.

**Common launch pages give an opportunity to effectively accelerate app launch**. As shown in the dot-lines of Fig. 2, the read I/Os are not busy for the rest of the app launch, and

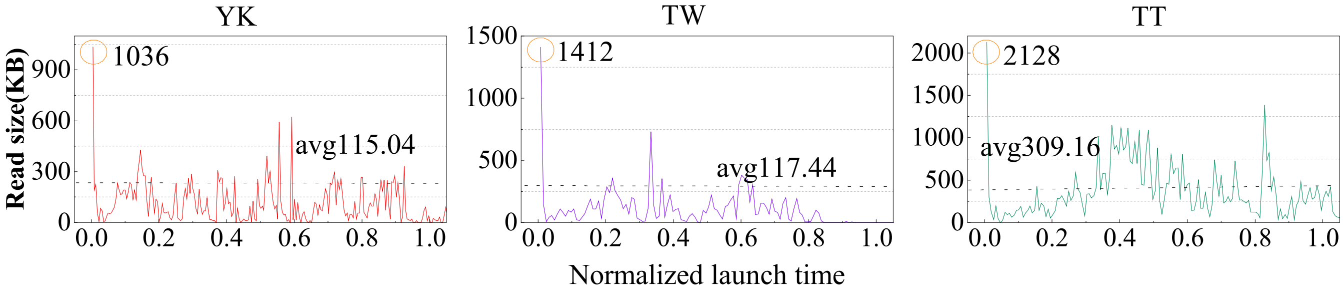


Fig. 2: Read size during cold app launch.

the average read size in a time window(5ms) is 100-300KB, which is about 1/70 of key launch pages, and we call these pages common launch pages. That gives us an opportunity to prefetch the pages via sliding window prefetching, the pages in the same time window can be prefetched before they are demanded. Most important of all, the pages in the same time window are demanded during the same interval, and the sliding window prefetching fetches the pages according to the order of access, reducing page faults caused by accessing-prefetching order inversions. In the ideal condition, the effect of sliding window prefetching is the same as low page faults in Fig. 1.

## IV. Design and Implementation of TPP

Based on the observation above, we propose a **T**wo-**P**hase **P**refetching schema (TPP), in order to effectively accelerate app launch with minimal memory overhead. TPP consists of three components: 1) **A**pp **P**rediction (AP), accurately predicting app. 2) **L**aunch **P**ages **M**anagement (LPM), tracking exactly which pages are accessed and the order of access. 3) **P**refetching **M**anagement (PM), prefetching few but critical launch pages before app launch via predictive prefetching to minimize the memory overhead, and prefetching common launch pages during app launch via sliding window prefetching to effectively accelerate app launch. Fig. 3 shows the overall design of the TPP, where the core components of the TPP are shown in orange. AP is implemented in user space and consists of a log collector and app usage predictor. The log collector records app usage information and the app usage predictor trains the app prediction model and accurately predicts the app to be used next based on historical app usage sequences, and sends the prediction results to PM for prefetching.

LPM and PM are implemented in kernel space. LPM tracks the key and common launch pages during app launch and records the order of access. Based on app prediction results, PM prefetches few but critical pages before app launch. During app launch, the PM prefetches the common pages into memory in the order of they are accessed.

### A. App Prediction based on Word-LSTM

Users have a specific pattern of using the app. App usage patterns are closely related to historical app usage sequences [12]. Example 1: Amy uses PayPal to pay for her order after placing it with Amazon Shopping. Example 2: when Amy is browsing videos using TikTok, she receives a Facebook message. Amy switches to Facebook for viewing the message and re-launches TikTok to continue browsing videos. For user Amy, after using Amazon Shopping, she is probably using PayPal, and after using TikTok, she may frequently switch back to TikTok. From the two examples, the target app is determined by the historical app usage sequences.

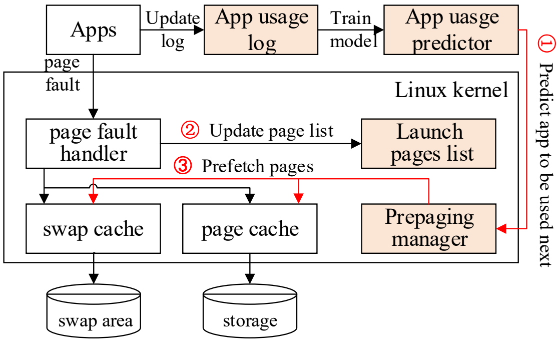


Fig. 3: TPP desgin overview.

**Problem Formulation**. App usage prediction based on historical app usage sequences can be described as (1), where $A_m$ is the sequence $(a_0, a_1, \ldots, a_{m-1})$ of the most recent historical usage with m apps, $a_i$ means the i-th app used, $a_i \in A$, and $A$ is the given set of apps. $P$ denotes the probability that each app is to be used. The app with the highest probability is the one most likely to be used.

$$P = h(A_m) \tag{1}$$

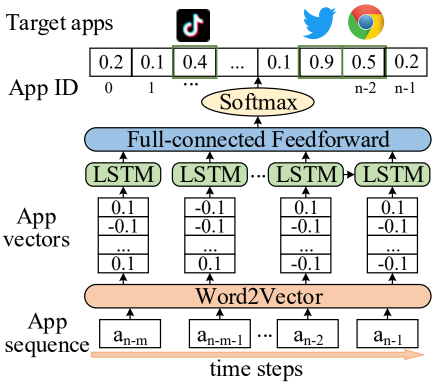


Fig. 4: Illustration of Word-LSTM Model.

**Word-LSTM prediction model**. As shown in Fig. 4, the App prediction model is mainly composed of two components, including the embedding layer and the prediction component. The embedding layer maps the app to a d-dimensional vector $v_a$. Long Short-Term Memory (LSTM) is adopted as the prediction component. The results of the LSTM modeling app sequence $(v_0, v_1, \ldots, v_{m-1})$ are input into the full-connected layer with the activation function $tanh$. Since our prediction objective is to obtain the probability that each app to be used, $softmax$ is used as the final activation function. The k apps with the highest probability are selected as the target apps. In Fig. 4, the target apps are TikTok, Twitter, and Chrome.

**Embedding layer**. The task of the embedding layer is to map the app to the d-dimensional vector space. There are millions of apps in the app market, and a fast and efficient solution is needed to map the apps. The traditional one-hot coding method is simple and fast, but it does not identify the relationship between apps. For example, Facebook and WeChat are similar, but one-hot can only simply map these apps into different vectors. In natural language processing, Word2vec is able to perform training quickly on millions of dictionaries and data, and it can capture the correlation between words properly to obtain the word vector. Inspired by this, each app can be viewed as a word in a document, and we use Word2vec for app embedding to make full use of the context in the app sequence and preserve the rich semantic information of the app.

**Prediction component**. Currently there are many app prediction studies using Markov [**20**] or Bayesian [**13**, **19**]models. Markov model is capable to identify the correlation of adjacent apps, but it assumes that the target app is only correlated with the previous one, ignoring the impact of historical app sequences on the target app, and it is not able to identify the usage pattern of Example 2. The Bayesian model can identify contextual information, but it assumes that each app usage record is isolated and fails to identify the relationship between app usage records [**12**]. Long short-term memory neural network (LSTM) is able to capture time dependence and process time series data efficiently. We predict reserved files based on historical information via Markov, Bayesian and LSTM, the performance of different methods is shown in Table II. We use LSTM for app usage prediction based on historical app usage sequences for the high accuracy of LSTM.

### *B. Launch Page Management (LPM)*

The object of LPM is to accurately capture pages that are repeatedly accessed during app startup in different turns, including file pages and anonymous pages. The access pattern is related to the type of files. Most file pages are invariant during app startup [**7**], such as executable files with extension of .bak, .vdex, .odex, .so, etc. The cache files and database files of the app are gradually changed as the users use the apps. Besides, the anonymous page is closely related to the running state of the app and the user's operation on the app, and the anonymous page is variant over multiple launches. In order to record the repeated accessed pages accurately, LPM generates a candidate table (CT) and a target table (TT) for every app.

**Data structures**. The data structure of TT is the same as CT. For file pages, every entry of CT is a file I/O list, and each item of the list is a pair (index, len), which represents the first accessed page's index in the file and the count of pages to be read. For anonymous pages, every entry of CT is the swap-in pages list, and each item of the list is a pair (pid, vaddr), which presents the process the page belongs to and the virtual address of the anonymous page. The CT is divided into $n$ time windows (TW) with the same $timezone$. The pages accessed in the same interval are recorded in the same TW. The management of CT and TT is the same for file pages and anonymous pages.

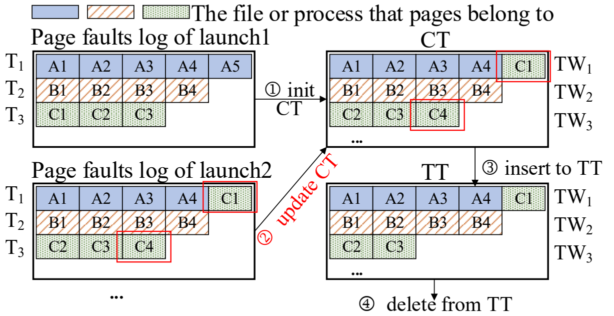


Fig. 5: Profiling pages for prefetching.

**Trace read I/Os and swap-in pages during app launch via logger**. We build logger in kernel to record the read I/Os and swap-in pages. For file pages, we insert routines to the Virtual File System (VFS) to trace read I/Os during app launch. During App startup, LPM traces mmap read I/Os via the filemap_fault() function and traces non-mmap read I/Os via do_generic_file_read() function. For anonymous pages, we modified the do page_fault() function to trace the anonymous pages to be accessed during startup. The logger records the pages' metadata: (index, len, interval) for file pages, (pid, vaddr, interval) for anonymous pages, where the interval presents the duration from the beginning of the app launch to the pages accessed.

**Record all demanded pages and the order of access**. CT records all the demanded pages as the candidate for pretching. For file pages, when the app is installed or updated, LPM deletes the old CT/TT and builds a new one for the app. Because the files app accessed are changed and the pages recorded in old CT/TT are invalid. Similarly, for anonymous pages, when the app starts from scratch, LPM deletes the old CT/TT and builds a new one. Because the anonymous pages are bound to the process, new processes are created when the app starts from scratch. When app is launched, CT is inited with the data in the logger. Record the pages in different TW according to the interval of pages to identify the order of access, and the TW number of pages is (2). As shown in Fig. 5 ① , during the first app launch, the pages A1-A5 are accessed at T1, the TW number is 1 according to (2), and the CT records A1-A5 in TW1. Besides, if a page is accessed for times during the same launch, CT only records it once with the smallest interval. That is, record when the page is first accessed during one launch.

$$TWnumber = interval/timezone \qquad (2)$$

**Update variant pages and evict inactive pages**. In order to increase the hit rate and accuracy of prefetching, fresh active pages should be identified timely, and the recorded inactive pages should be evicted on time. If the logged pages are recorded in CT, these pages are accessed before, which means that these pages are fresh and active, and they are most likely to be accessed again during subsequent app launches. In order to increase the hit rate of prefetching, these active pages' metadata should be inserted into TT timely. If the pages traced are not recorded in CT, these pages are not repeatedly accessed before and should be inserted into CT as candidates for prefetching pages, as shown in Fig. 5 ② ③ .If the page recorded in TT is not accessed during two consecutive launches, the page becomes an inactive page, which indicates that the page is most likely not accessed during future app launches. After the app is launched, these inactive pages' metadata is removed from TT, as shown in Fig. 5 ④ , to increase the accuracy of pre-reading.

### *C. Prefetching management*

PM is the critical component for TPP to achieve effective acceleration with little memory overhead.

**Prefetching of key launch pages**. After AP predicts the target app to be used next, PM reads the TT of the target app to get the key launch pages' metadata. Based on observation 1, PM only prefetches key launch pages (pages within the first k time windows) before the target app launch to prevent page faults burst at the beginning of the app launch, as shown in Fig. 6. Compared to traditional PBAL, the memory waste of TPP is negligible.

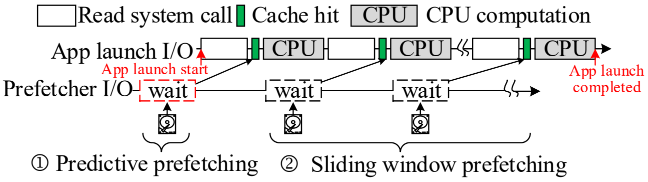


Fig. 6: Prefetching diagram of TPP.

**Prefetching of common launch pages**. When the target app starts, the prefetching of common launch pages (pages after k-th time windows) is triggered immediately. Based on observation 2, prefetch common pages in the order they were accessed, i.e., time window order. Prefetching is performed concurrently with multiple threads, where one thread is assigned one time window at a time to speed up the prefetching task. Compared with traditional PDAL, TPP is aware of the order of pages accessed, and the page accessed first is prefetched first, which solves the problem of access-pre-reading order reversal, effectively reducing page faults during app launch.

## V. EVALUATION

### *A. Device*

We implement TPP in Android for Google Pixel 3 (see Table I for the spec and configurations). In this section, we evaluate modules of TPP separately, which includes (1) analyzing app usage prediction accuracy, (2) evaluating the performance of application launch acceleration, and (3) measuring TPP overhead, the metadata size in LPM, and the memory overhead in PM.

TABLE I: Device Specifications.

| Device | Google Pixel3 |
|---|---|
| CPU | Snapdragon 845 |
| DRAM | 4GB LPDDR4X |
| Storage | 64GB eMMC 5.1 |
| OS | Android9.0(r21) Linux 4.9.124 |
| ZRAM | 2GB |

### *B. Prediction*

To evaluate the app usage prediction, we use traces of LSApp[**21**]. We merge the logs of the same app in LSApp to extract the app history usage records. After pre-processing, there are 61564 app usage records from 292 users, and the training set and test set are divided by 7:3. The prediction takes $m$ historical app usage records as input sequence, and Word2Vec is used to map each app to a d-dimensional vector, and in the experiment, $d = 10$.

We only concern whether the app that the user actually uses is in the list of predicted results, so the precision of the algorithm is evaluated using recall. To determine the number $m$ of apps in the app history sequence, we test the prediction accuracy with different $m$. When $m < 6$, the prediction accuracy increases rapidly with the increase of $m$, and is almost unchanged when $m > 6$. Therefore, in the experiment $m = 6$. App prediction results are shown in Table II, Recall@3 of Word-LSTM is 82.99%, which is much higher than other methods.

TABLE II: Performance Comparison with Baselines.

| Method | Recall@1 | Recall@2 | Recall@3 |
|---|---|---|---|
| Bayesian | 0.191196 | 0.329098 | 0.384015 |
| Markov | 0.350325 | 0.515617 | 0.60127 |
| Word-LSTM | **0.645011*** | **0.77721*** | **0.829869*** |

### *C. Performance*

The workloads are shown in Table III. We separately evaluate the impact of the predictive prefetching size and the time window size on the app launch time by Twitter. To evaluate the effect of predictive prefetching size, we set the time window size to 1ms. We prefetch pages before the app launch in the first $n$ time windows and use the number of time windows $n$ to represent the pages' size. The experimental results show that when $n < 10$, the launch delay decreases significantly as n increases, and when $n > 10$, the launch delay changes little. It is because when the predictive prefetching size is small, most of the key launch pages are prefetched during the app launch, and as the predictive prefetching size increases, more key launch pages are prefetched before the app launch, effectively reducing the app launch latency. When $n = 10$, all of the key launch pages are prefetched before the app launch. So, we take 10ms as the predictive prefetching size in the following experiments. Similarly, the experimental results show when the time window size is 10ms, the app launch speed is the fastest, and we set the time window size as 10ms.

TABLE III: Workloads.

| Application | Action |
|---|---|
| Twitter (TW) | Browse and read posts |
| TikTok (TT) | Watch videos |
| Candy Crush (CC) | Play a stage |
| Amazon Shopping(AS) | Browse products |
| Firefox (FF) | Search via keyword and browse |
| Earth (ET) | Search via keyword and browse |

Based on the prediction results, select three apps with the highest probability as target apps. Before the target app launch, prefetch the key launch pages accessed during the first 10ms by the launch process. During app launch, the time window size is set as 10ms, and prefetch the common launch pages via 8 threads in the order of time windows.

App launch latency.As shown in Fig. 7, compared to the Android 9.0 origin prefetching, TPP reduces app launch latency by up to 67.8% and 41% on average. Compared to the most advanced accelerate app launch method ASAP, TPP reduces app launch latency by up to 52.5% and 37% on average.

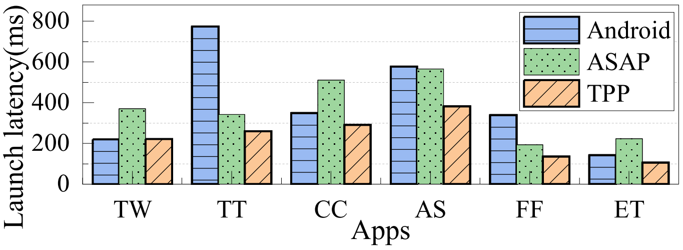


Fig. 7: App launch latency on popular Apps.

To verify the efficiency of TPP in reducing page faults, we chose three apps (TT, AS, and ET) to measure the count and size of page faults during app launch with different prefetching methods. As shown in Fig. 8 (a), TPP reduces the page faults count by 57.3% - 97.3% compared to the Android 9.0 native prefetching and 54.5% - 87.1% compared to ASAP. As shown in Fig. 8 (b), TPP reduces the page faults size by 91.5% - 99.5% compared to the Android 9.0 native prefetching and 72.3% - 96.7% compared to ASAP.

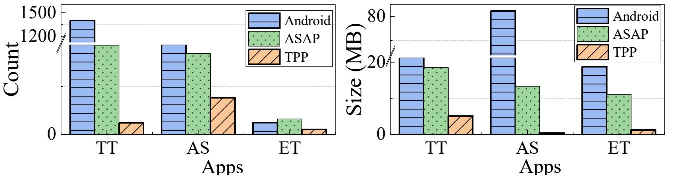


Fig. 8: Page faults count and size during app launch for different prefetching schema.

### D. Overhead

Memory is a scarce resource on mobile devices. Besides, as an interactive device, the mobile is latency sensitive. Therefore, we evaluate the memory usage and latency of each module in TPP. App usage prediction. We train the prediction model 100 times, the prediction model average size is 109KB, and the model offline training latency is 100.798s on average. We predict ten thousand sequences in batch 100 times, the model online prediction latency is 30.04µs on average, which is negligible for users. Launch Page Management. The prefetching metadata of per app is 338.45KB on average, which can be resident in memory.

Prefetching Management. As shown in Fig. 9, the predictive prefetching method preloads data with an average size of 1.31M, which is much lower than the traditional PBAL. For time-sharing prefetching, the average number of pages recorded in a time window is 45.1, i.e. 180.4KB, and the average delay of reading the pages in a time window is 2.12ms.

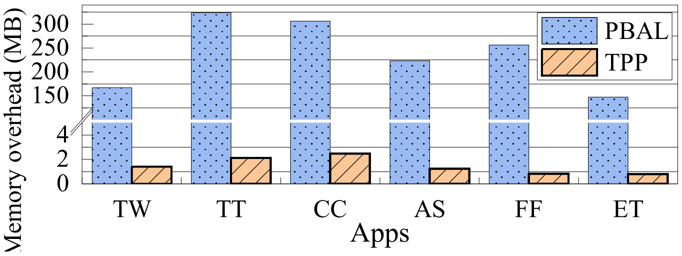


Fig. 9: Memory Overhaed.

## VI. Conclusion

In this paper, we explored the temporal characteristics of the read I/Os during app launch, and first propose the predictive prefetching for the little but crucial pages, avoiding preloading all of the launch pages. Besides we design a novel time-sharing prefetching method, to resolve the problem of access-prefetching order reversal, effectively reducing the number of page faults that occur during app startup. The experimental results show that TPP effectively accelerates app launch with minimal resources.

## Acknowledgment

This work was supported by Key Laboratory of Information Storage System and Engineering Research Center for Data Storage Systems and Technology, Ministry of Education, China.

## References

[1] “Smartphone Market Share,” https://www.idc.com/promo/smartphone-market-share. 2022.

[2] Umar Farooq and Zhijia Zhao, “Runtimedroid: Restarting-free runtime change handling for android apps,” in MobiSys, 2018, pp. 110–122.

[3] Deng et al, “Measuring smartphone usage and task switching with log tracking and self-reports,” in Mobile Media & Communication, vol. 7, no. 1, 2019, pp. 3–23.

[4] Al Maruf, Hasan, and Mosharaf Chowdhury, “Effectively prefetching remote memory with leap,” in ATC, 2020, pp. 843–857.

[5] Li, Changlong, et al, “SEAL: User experience-aware two-level swap for mobile devices,” in TCAD, vol. 39, no. 11, 2020, pp. 4102–4114.

[6] Joo, Yongsoo, et al, “FAST: Quick Application Launch on Solid-State Drives,” in FAST, 2011, pp. 259–272.

[7] Son, Sam, et al, “ASAP: Fast Mobile Application Switch via Adaptive Prepaging,” in ATC, 2021, pp. 365–380.

[8] Song, Wook, et al, “Personalized optimization for android smartphones,” in TECS, vol. 13, no. 2, 2014, pp. 1–25.

[9] Yan, Tingxin, et al, “Fast app launching for mobile devices using predictive user context,” in MobiSys, 2012, pp. 113–126.

[10] Shen, Zhihao, et al, “Deepapp: a deep reinforcement learning framework for mobile application usage prediction,” in SenSys, 2019, pp. 153–165.

[11] Tian, Yuan, et al, “What and how long: Prediction of mobile app engagement,”, in TOIS. vol. 40, no. 1, 2021, pp. 1–38.

[12] Zhao, Sha, et al, “Appusage2vec: Modeling smartphone app usage for prediction,” in ICDE, 2019, pp. 1322–1333.

[13] Zhu, Xiao, et al, “SmartSwap: High-performance and user experience friendly swapping in mobile systems,” in DAC, 2017, pp. 1–6.

[14] Lebeck, Niel, et al, “End the senseless killing: Improving memory management for mobile operating systems,” in ATC, 2020, pp. 873–887.

[15] Liang, Yu, et al, “Acclaim: Adaptive memory reclaim to improve user experience in android systems,” in ATC, 2020, pp. 897–910.

[16] Hahn, Sangwook Shane, et al, “FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones,” in ATC, 2018, pp. 15–28.

[17] Kim, Sangwook, et al, “Enlightening the I/O Path: A Holistic Approach for Application Performance,” in FAST, 2017, pp. 345–358.

[18] Liao, Zhung-Xun et al, “On mining mobile apps usage behavior for predicting apps usage in smartphones,” in CIKM, 2013, pp. 609–618.

[19] Zou, Xun, et al, “Prophet: What app you wish to use next,” in UbiComp, 2013, pp. 167–170.

[20] Parate, Abhinav, et al, “Practical prediction and prefetch for faster access to applications on mobile phones,” in UbiComp, 2013, pp. 275–284.

[21] Aliannejadi, Mohammad, et al, “Context-aware target apps selection and recommendation for enhancing personal mobile assistants,” in TOIS, vol. 39, no. 3, 2021, pp. 1–30.