ASAP: Fast Mobile Application Switch via Adaptive Prepaging

Sam Son¹  Seung Yul Lee¹  Yunho Jin¹  Jonghyun Bae¹  Jinkyu Jeong²
Tae Jun Ham¹  Jae W. Lee¹  Hongil Yoon³

¹ Seoul National University  ² Sungkyunkwan University  ³ Google
Memory Pressure in Today’s Smartphone Usage

- Memory capacity is becoming a scarce resource on mobile devices
  - The application size and memory footprint have been growing
  - Users run more than 5 applications concurrently\(^1\)
- However, the cost/power/area budget often limits its size

\[1\] Yu Liang et al., “Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems” in ATC’20
Memory Pressure Degrades UX

- Causes latency when users switch applications

- Maintaining low latency is crucial
  - Users switch applications more than 100 times a day\(^2\)

\(^2\) Tao Deng et al., “Measuring smartphone usage and taskswitching with log tracking and self-reports” in Mobile Media & Communications 2018
Android Memory Management
**Application Launch** creates an application process from scratch → takes long time
Android Memory Management

Launch other apps

Memory Pressure ↑

Launching more apps uses up all the memory
Android Memory Management

Launch other apps

(1) Page Eviction

To secure free memory, OS **compresses** anonymous pages (compression-based swap)
Launch other apps

(1) Page Eviction
To secure free memory, OS **discards** file-backed pages
Switching to Calendar is delayed due to on-demand page fetching
Android Memory Management

Switch to Calendar

Switching to Calendar is delayed due to decompressing anonymous pages
Switching to Calendar is delayed due to reading file-backed pages from disk.
(2) Low Memory Killer (LMK)
Killing background application frees up pages
This time, switching to Calendar causes slow re-launching of Calendar.
Application Switching Latency under Memory Pressure

Observation 1: Launch time is longer than switch time even when most pages not in memory
Implication: It is better to avoid relaunching by disabling LMK
Observation 2: Switch time can increase by 4x on average under memory pressure.

Implication: Retrieving relevant pages on-demand increases switch time a lot.
Limitation of Demand-Paging

- Both CPU and disk BW are under-utilized during switch time
  - Page decompression is delayed until anonymous page fault occurs → low CPU utilization
  - Disk I/O is delayed until file-backed page fault occurs → low disk BW utilization
- On average, only **34%** of CPU and **15%** of disk BW are utilized during the switch time
Opportunity of Prepaging

- Switch time can be improved by leveraging prepaging at the beginning of switch
- By doing so, available system resources (i.e., CPU cycles and disk bandwidth) can be fully utilized
Opportunity of Prepaging

- Switch time can be improved by leveraging prepaging at the beginning of switch
- By doing so, available system resources (i.e., CPU cycles and disk bandwidth) can be fully exploited

Our Goal

Reducing switching latency by leveraging prepaging
Challenges of Prepaging

What to Prepage?
- Applications’ contexts keep changing
- Achieving both high coverage and low misprediction ratio

How to Prepage?
- Maximizing the efficiency by achieving high system resource utilization
- Minimizing contention with application threads
Application Switch via Adaptive Prepaging (ASAP)

- ASAP maintains low switching latency without LMK
- ASAP is application-agnostic, and requires no changes to applications codes

What to Prepage?

- Logging both page faults and I/O syscalls → High coverage
- Adaptively update based on feedback → Low misprediction

How to Prepage?

- Multiple prepaging threads → High utilization
- Opportunistically prepaging to minimize contention → Low contention
ASAP: Design Overview

What to prepage?

How to prepage?
Switch Footprint Estimator (SFE)

What to prepage?

- Logging page access information
- Maintaining Candidate Table
- Maintaining Prepaging Target Table
Switch Footprint Estimator: Mechanism

Switching to App X

(inode, index)

(32, 2)
(32, 3)
(32, 5)

Logging
inode(/A.vdex) = 32

Fault Buffer

(inode, index)

(32, 2)
(32, 3)
(32, 5)

Candidate Table for X

<table>
<thead>
<tr>
<th>inode</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>(1, 2, 3, 4)</td>
</tr>
</tbody>
</table>

Prepaging Target Table for X

<table>
<thead>
<tr>
<th>inode</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>(2, 3)</td>
</tr>
</tbody>
</table>

Eviction

Promotion
Optimized SFE for Each Type of Pages

- Anonymous pages and file-backed pages have different access patterns.
- About 75% of all accessed file-backed pages are invariant across switches, while only 44% of anonymous pages are invariant.

Access Frequency of **File-backed Pages** (over 10 switches)

Access Frequency of **Anonymous Pages** (over 10 switches)
Optimized SFE for Each Type of Pages

[Diagram of the system architecture showing page fault handler, Linux kernel, Android apps, Android framework, Swap cache, Page cache, In-memory comp. swap, Flash storage, Data load operation, I/O operation, Read/write system call, SFE for file-backed pages, SFE for anonymous pages, Static Candidate Table, Dynamic Candidate Table.]

- Android apps
- Linux kernel
- Page fault handler
- Swap cache
- Page cache
- In-memory comp. swap
- Flash storage
- Data load operation
- I/O operation
- Read/write system call
- SFE for file-backed pages
- SFE for anonymous pages
- Static Candidate Table
- Dynamic Candidate Table

App switching start / end

ASAP

Prepaging Manager

Static Candidate Table

Dynamic Candidate Table
Prepaging Manager

Prepaging Manager

Android apps

Linux kernel

Page fault

Read/write system call

Swap cache

Prepaging Manager

Android framework

App switching start/end

Prepaging target

File-backed pages

Anonymous pages

Linux kernel

Data load operation

I/O operation

In-memory comp. swap

Flash storage

Fetching prepaging target

Optimizing prepaging threads

How to prepage?
Optimizing Prepaging Threads

- Batch processing minimizes lock contention between prepaging threads
  - 16 pages for anonymous pages
  - All target pages of one file for file-backed pages

- Giving low schedule priority to avoid CPU contention with app threads
  - SCHED_IDLE(lowest) for prepaging threads
  - Opportunistically prepaging
Evaluation Methodology

- Integrated ASAP into Android OS
- Evaluated ASAP on high-end and mid-end devices (Google Pixel 4 and Pixel 3a)
- 8 popular mobile applications with diverse automated usage patterns

<table>
<thead>
<tr>
<th>Device</th>
<th>Google Pixel 4</th>
<th>Google Pixel 3a</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Snapdragon 855</td>
<td>Snapdragon 670</td>
</tr>
<tr>
<td>DRAM</td>
<td>6GB (effective 4GB)</td>
<td>4GB</td>
</tr>
<tr>
<td>Storage</td>
<td>UFS 2.1</td>
<td>eMMC 5.1</td>
</tr>
<tr>
<td>OS</td>
<td>Android 10.0.0(r41) with Linux kernel 4.14</td>
<td>Android 10.0.0(r41) with Linux kernel 4.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Usage Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angry Bird (AB)</td>
<td>Play a stage</td>
</tr>
<tr>
<td>Candy Crush (CC)</td>
<td>Play a stage</td>
</tr>
<tr>
<td>New York Times (NY)</td>
<td>Browse and read articles</td>
</tr>
<tr>
<td>Youtube (YT)</td>
<td>Watch Videos</td>
</tr>
<tr>
<td>Facebook (FB)</td>
<td>Browse and read posts</td>
</tr>
<tr>
<td>Twitter (TW)</td>
<td>Browse and read posts</td>
</tr>
<tr>
<td>Chrome (CH)</td>
<td>Browse keywords</td>
</tr>
<tr>
<td>Quora (QR)</td>
<td>Browse questions and answers</td>
</tr>
</tbody>
</table>

Device Specification

Benchmark Applications and Usage Pattern
Switching Latency Reduction

- Baseline: switching latency when 8 applications run concurrently (high memory pressure)
- Up to 33.3% (22.2% on average) latency reduction on Google Pixel 4
- Up to 35.7% (28.3% on average) latency reduction on Google Pixel 3a
Improved CPU Utilization

- Noticeable increase in the CPU cycles at the early phase of switching
- Higher CPU utilization (Up to 35%, average 18%)
Improved Disk Bandwidth Utilization

- Noticeable increase in the I/O bandwidth at the early phase of switching
- Higher disk BW utilization (Up to 35%, average 25%)
Switch Footprint Estimator Efficiency

- Higher Precision $\rightarrow$ Lower misprediction
- Higher Recall $\rightarrow$ Higher coverage
- SFE for file-backed pages shows **better precision** due to static access pattern
- SFE for anonymous pages shows **better recall** due to dynamic candidate table
Conclusion

**ASAP** provides better UX to mobile users by reducing latency of application switch

Contributions:
- Identified performance bottlenecks of application switching time
- Identified the root cause of low resource utilization during application switch
- Designed an application-agnostic prepaging technique
- Achieved up to 35.7% latency reduction on Google Pixel devices
Thank You!

ASAP’s Android kernel code is available at https://github.com/SNU-ARC/atc21-asap-kernel

Sam Son, sosson97@snu.ac.kr