Integrated Host-SSD Mapping Table Management for Improving User Experience of Smartphones

Yoona Kim and Inhyuk Choi, *Seoul National University;* Juhyung Park, Jaeheon Lee, and Sungjin Lee, *DGIST;* Jihong Kim, *Seoul National University*

https://www.usenix.org/conference/fast23/presentation/kim-yoona

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

NetApp®

Integrated Host-SSD Mapping Table Management for Improving User Experience of Smartphones

Yoona Kim
Seoul National University

Inhyuk Choi
Seoul National University

Juhyung Park
DGIST

Jaeheon Lee
DGIST

Sungjin Lee
DGIST

Jihong Kim
Seoul National University

Abstract

Host Performance Booster (HPB) was proposed to improve the performance of high-capacity mobile flash storage systems by utilizing unused host DRAM memory. In this paper, we investigate how HPB should be managed so that the user experience of smartphones can be enhanced from HPB-enabled high-performance mobile storage systems. From our empirical study on Android environments, we identified two requirements for an efficient HPB management scheme in smartphones. First, HPB should be managed in a foreground app-centric manner so that the user-perceived latency can be greatly reduced. Second, the capacity of the HPB memory should be dynamically adjusted so as not to degrade the user experience of the foreground app. As an efficient HPB management solution that meets the identified requirements, we propose an integrated host-SSD mapping table management scheme, HPBvalve, for smartphones. HPBvalve prioritizes the foreground app in managing mapping table entries in the HPB memory. HPBvalve dynamically resizes the overall capacity of the HPB memory depending on the memory pressure status of the smartphone. Our experimental results using the prototype implementation demonstrate that HPBvalve improves UX-critical app launching time by up to 43% (250 ms) over the existing HPB management scheme, without negatively affecting memory pressure. Meanwhile, the L2P mapping misses are alleviated by up to 78%.

1 Introduction

User experience (UX) design is one of the topmost tasks in designing modern smartphones. In order to create a high-quality UX from a smartphone, it is essential for the smartphone to react promptly to user inputs without a noticeable delay. For example, when an application (app) is launched, if there exists a considerable user-perceived delay, the quality of UX would be significantly degraded. Since user-perceived delays play a key role in realizing high-quality UX, many researchers have investigated various system resource management schemes so that user-perceived delays can be minimized for the user-facing foreground (FG) apps [1–4].

Although existing techniques have explored the most plausible sources that influence user-perceived delays, a storage system has not been actively investigated from the perspective of user-perceived delays. As the capacity of a mobile storage system quickly increases (*e.g.,* a 1-TB Universal Flash Storage (UFS) device [5]), the read latency of the mobile storage system is emerging as a key factor that can negatively affect user-perceived delays [1, 4, 6, 7]. Since the overall quality of UX is determined by how promptly a smartphone responds to a user's input, storage responsiveness has a significant correlation with improved user responsiveness. There are two main reasons why the read latency of the mobile storage system has a high impact on the UX quality. First, the read latency of a mobile storage system accounts for the largest portion of the total latency of a host request in modern smartphones [1]. For example, when an app is launched on an Android smartphone, approximately more than half of the total app launching time is taken by the storage read time [1, 6].

Second, the read latency of a mobile storage system varies significantly because of the limited SRAM capacity in the mobile storage system. Since SRAM in the mobile storage system is used for implementing a logical-to-physical (L2P) mapping table, which is an essential component of a flash-based storage system, the performance of the mobile storage system is highly dependent on the capacity of SRAM. Unfortunately, the capacity of SRAM is quite limited for large-capacity mobile storage systems. Under this design constraint, an L2P mapping table is commonly managed by an on-demand scheme (*e.g.,* DFTL [8]) that only loads a small portion of the entire L2P mapping entries in (fast) SRAM while the complete L2P mapping table is stored in (slow) flash memory. When most host read requests cannot find their L2P mapping entries from SRAM, their read latency can be significantly longer, thus causing a large increase in user-perceived delays. For example, in our exploratory evaluation, we observed that the app launching time increases by up to 50% when the SRAM only contains a portion of the L2P mapping entries as opposed to when it contains all entries.

To overcome the performance problem from the limited

SRAM capacity within a mobile storage system, Host Performance Booster (HPB) [9, 10] was introduced to store L2P mapping entries in the host memory. It was first shipped in production by Google's Pixel 3 in 2018 with Linux kernel v4.9.96 [11] shortly after its introduction. By exploiting the host memory as a (fast) L2P mapping cache in addition to the SRAM of a mobile storage system, HPB can improve I/O performance by reducing costly SRAM L2P cache misses that require slow flash read accesses. Although several researchers have successfully shown that exploiting the host memory is an effective approach for improving I/O performance [9, 12–14], few work has treated the problem of utilizing the host memory for high-performance I/O *from the UX perspective* in a holistic fashion. The main goal of our work is to comprehensively investigate how HPB should be managed so that the UX of smartphones can be enhanced from HPB-enabled high-performance mobile storage systems.

In order to understand how HPB should be managed in a UX-aware fashion, we evaluate how various UX-related performance metrics are affected by different HPB settings on Android environments. To this end, we measure performance metrics that are relevant to UX quality, such as the app launching, switching, and loading times. We measure this systematically using repeatable and reproducible benchmarks to enable accurate and reliable UX-quality evaluation, eliminating the possibility of human errors. From our empirical study, we identify two key requirements for an efficient HPB management scheme in smartphones. First, the existing HPB management scheme [15] is FG app-oblivious in that the app status is not actively considered in managing the HPB memory. For example, HPB only focuses on caching L2P entries with high reference counts without considering its impact on UX. In order to improve UX from a smartphone, HPB memory should be managed in an FG app-centric manner so that the user-perceived delay of a user-facing FG app can be effectively minimized. Second, the capacity of the HPB memory reserved from the host memory should be dynamically adjusted during run time so that no apps suffer extra memory pressure from the HPB-allocated DRAM. For example, when a large amount of memory is statically reserved for HPB, the low memory killer daemon (LMKD) [16] is triggered more frequently to relieve increased memory pressure, significantly degrading the UX. Allocating small-size memory to avoid such cases is not ideal either as it negates the potential performance improvement from deploying HPB.

As an efficient HPB management solution that meets two key requirements, we propose HPBvalve (Hvalve in short), an integrated host-SSD mapping management scheme for HPB-enabled smartphones. Unlike the existing HPB management scheme [15], Hvalve prioritizes FG app in caching entries to HPB by integrating app status (FG or BG) for every submitted I/O. For further UX improvement, Hvalve detects every app launch event which is one of the most important activities of smartphones that highly impacts UX. Then, Hvalve utilizes the profiled launch-time-referenced L2P list ahead of time to reduce user-perceived delays of an app launch. Additionally, Hvalve adjusts the maximum capacity of the reserved HPB memory according to the current memory pressure status of a smartphone. When memory pressure is monitored, Hvalve selectively returns HPB memory to apps. Through dynamic HPB memory size adjustment, Hvalve can utilize the unused host memory efficiently while preventing inadvertent UX regression from using HPB.

In order to validate the effectiveness of the proposed Hvalve, we develop a prototype Hvalve that supports the internal operational logic of Hvalve on a hardware development kit (HDK) based on the Snapdragon 888 SoC [17] (see Section 6.1 for details). Our experimental results show that Hvalve can effectively manage the HPB memory, reducing the user-perceived delays of an FG app by up to 43% over the existing scheme without increasing the overall memory pressure.

The remainder of this paper is organized as follows. We first review how HPB-enabled smartphones work in Section 2 and review related work in Section 3. In Section 4, we present the key design requirements of a UX-aware HPB management scheme based on our empirical observations. In Section 5, we describe the design and implementation of Hvalve. The experimental results are reported in Section 6. Finally, we conclude with a summary in Section 7.

2 Background

In this section, we briefly explain the basics of L2P mapping structures and policies of the conventional and the HPB-enabled storage systems.

2.1 Controller-side L2P Mapping Structure

The latency of I/O requests submitted to a UFS device varies greatly depending on how the underlying L2P mapping scheme works. As the capacity of UFS devices increases, its L2P mapping table size also increases accordingly. Keeping such a large mapping table in a small SRAM inside the UFS device is technically impossible. Therefore, the UFS device employs an on-demand cache scheme that stores the entire L2P mapping table in flash, caching popular mapping entries in SRAM. On a cache hit, the UFS device provides excellent performance. On a cache miss, however, it suffers from long I/O latency because the missing L2P entry must be fetched from the flash first before serving an I/O request.

Fig. 1 illustrates how the UFS deals with I/O requests from the host in detail. When a read request is received (❶), the flash translation layer (FTL), which is responsible for translating a logical page address (LPA - file-system managed address) to a physical page address (PPA - storage device managed address), looks up cached L2P entries in SRAM (❷). If the desired L2P entry is found in cache, the FTL reads the requested data from the flash by consulting the translation information and returns it to the host. To keep track of hot entries, the FTL internally maintains a pseudo-LRU list for

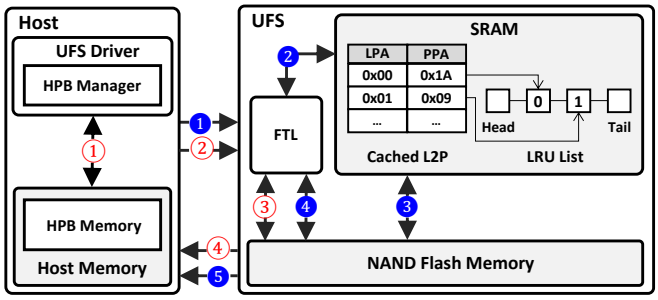


Figure 1: A read operation path.

L2P entries. The hit L2P entry moves to the head of the list. If the FTL fails to find the matched entry, it has to choose and evict a victim entry to make room in the SRAM. The entry at the tail of the list is evicted and the FTL reads in the wanted L2P entry from the flash to the SRAM (❸). Finally, the FTL reads the data from the flash (❹), and delivers the read data to the host (❺).

The on-demand cache scheme performs well when the size of the SRAM cache is large enough to accommodate most of the hot L2P entries. However, the capacity of a UFS device scales much faster than that of SRAM, which makes it difficult to cache sufficient hot entries in the SRAM. For example, the latest UFS device offers up to 1 TB [5] capacity, but its internal SRAM capacity is known to be only several hundred kilobytes [18]. Considering that the mapping table size is estimated as 0.1% of the UFS capacity, only the top 0.0005% of the table entries can be cached in the SRAM when its size is 512 KB, which is too small to keep hot entries.

2.2 Host-side L2P Mapping Structure

The constrained capacity of SRAM results in inconsistent I/O latency, which degrades UX. To overcome this problem, Jeong *et al.* [9] have introduced a Host Performance Booster (HPB) which extends a storage mapping space by exploiting the host memory. The HPB borrows a specific portion of the host memory and then keeps popular L2P entries to improve a mapping cache hit ratio.

Fig. 1 illustrates how the UFS device handles a read request when the HPB is enabled. The HPB manager is implemented in the UFS device driver of the Android kernel and manages the host memory space dedicated to caching L2P entries. Before sending a read request to the UFS, the HPB manager first searches for its L2P mapping entry in the host memory using the logical block address (LBA) of the request. If the desired L2P entry is found, the corresponding PPA is piggybacked on the read request (①), which is then submitted to the UFS (②). Upon the receipt of the request, the FTL in the UFS first verifies the integrity of the given PPA [9] and then directly issues a page read request to fetch the data of the designated PPA (③). It is unnecessary to look up the device-side mapping table. Finally, the FTL delivers the data to the host (④).

The HPB manager is responsible for selecting which L2P entries to fetch from UFS and keep in the HPB-designated host memory, based on its predefined conditions. A single HPB entry is 4 KB in size and stores 512 L2P entries. The HPB manager retrieves 512 L2P entries from UFS through one fetch command. The fetch command involves a normal 4-KB block read request to UFS, so the latency of a single fetch command is comparable to regular read latency. Whenever PPAs of L2P entries are changed due to internal operations such as a garbage collection on the UFS device side, the HPB manager is informed of the invalidated PPAs.

Using HPB, the overall I/O performance can be greatly improved by minimizing L2P misses. However, this benefit comes at the cost of reduced working memory space for apps. When integrating the HPB to the system, the following two technical issues should be carefully considered. The first is to properly decide the size of the HPB-designated memory. If the HPB size is too small, I/O performance gains by the HPB would be marginal. Conversely, if it is overly provisioned, the performance of apps would drop significantly as the HPB steals too much system memory which was to be used for apps. The second is to appropriately choose L2P entries to cache within the limited HPB memory, in a UX-centric manner. While HPB parameters are set vendor-specifically [10], to the best of our knowledge, there are currently no HPB systems in production that actively consider the state of apps [11, 19–28]. The current upstream HPB device driver (included in the Android Common Kernel since v5.10 [15]) employs the *counter-based caching policy* and the *timer-based eviction policy* for efficient HPB memory management. However, we argue that both of these policies fail to improve user-perceived delays which we discuss in Section 4.

3 Related Work

Classifications of FG and BG apps are pivotal in maintaining good UX on both mobile [29–31] and desktop environments [32]. Academia also follows this trend and makes use of FG/BG separation to further improve UX. Marvin [33] and Acclaim [34] modify the memory management subsystem and improve the FG app's performance by de-prioritizing BG apps' memory pages. ASAP [7] categorizes memory pages and prefetches FG app-related pages to improve app switching time. FastTrack [1] accelerates FG I/O requests by resolving I/O priority inversion caused by BG apps.

Despite the great impact of storage performance on UX, little attention has been paid to optimizing L2P caching under mobile device environments. To the best of our knowledge, FOAM [6] is the only work that sophisticates an L2P cache to enhance user-perceived performance. FOAM assigns different priorities to L2P entries, depending on the type of apps (FG or BG) and the type of I/O requests (read or write) that access them. They argue that the FG apps and the read requests precede other counterparts in terms of user-perceived performance. As such, FOAM divides the L2P cache into four partitions, FG-read (FR), FG-write (FW), BG-read (BR), and

BG-write (BW), and it accordingly moves the L2P entries across partitions whenever they are referenced. When choosing a victim, FOAM evicts the partitions from the lowest to the highest priority (*i.e.,* BR, BW, FR, and FW).

While FOAM enhances UX by prioritizing the eviction of L2P entries associated with BG apps, it has two limitations. First, FOAM only assumes an in-device cache, having no consideration of an HPB-enabled system. Thus, its effectiveness is limited to the latest mobile environments where HPB is used due to the increased mapping table size. Second, FOAM does not perform well when the FG and BG apps are switched quickly. This situation happens when the user runs multiple apps simultaneously. In this case, because the effective distinction between the FG and BG apps is not clear, the eviction policy of FOAM might lead to an unintended result.

4 Empirical Study of HPB on Smartphones

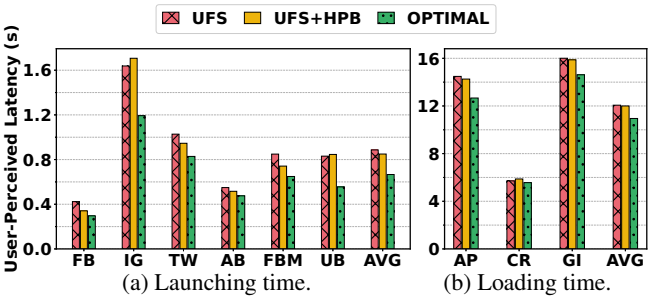
In this section, we empirically investigate how much the performance of FG apps is affected by the storage L2P cache. We first examine how the storage mapping cache affects the quality of UX on smartphones. Then, we assess the effectiveness of the existing HPB cache management policy and how it should be managed to boost UX.

4.1 Evaluation Study Setup

We conduct a set of experiments with a mobile hardware development kit based on the Snapdragon 888 SoC [17]. As for the benchmarks, we use nine popular smartphone apps [1] that are categorized into three types: games, social media, and utilities. We run the nine apps according to a predefined scenario that mimics real-world app-usage patterns of smartphones. In evaluating the UX, there exist various metrics such as app launching [35–41], app switching [7] and app loading [1]. These metrics are directly affected by the I/O performance as numerous libraries and files have to be loaded. In this section, we target app launching and loading times for the key metrics to assess the impact of L2P cache misses on UX, as they are the biggest contributors to the user-perceived latency.

We modify HPB in the Android kernel to implement various HPB cache management policies and to collect various performance-related statistics (*e.g.,* user-perceived latency and mapping cache hit ratios). Unfortunately, it is impossible to modify the firmware of UFS products. As an alternative, we develop a custom-emulated UFS device that mimics the behavior of production UFS devices using an Ultra-Low Latency SSD (ULL-SSD) [42]. The ULL-SSD has very low I/O latency (<20 μs) with extremely low variations, which makes it the perfect vehicle to emulate a slower UFS device.

We attach a 1-TB ULL-SSD as the main storage device for our custom-emulated UFS device. In between the HPB and the ULL-SSD, we run a UFS layer that implements UFS firmware algorithms, including L2P address translation, mapping cache management policies, and garbage collection. To emulate the I/O latency of UFS devices over the ULL-SSD, we also include a UFS I/O latency model [2] on the UFS layer. The UFS layer borrows a part of the host memory space and uses it as an L2P cache. For the UFS layer, we assign 512 KB of memory as an L2P cache space [18]. The HPB-allocated host memory size is set to 256 MB out of the 12 GB of host DRAM. Note that 1 GB of memory is required to cache the entire L2P mapping table. Since we use the same system and benchmark setups used in Section 6, more details of the experimental settings are explained in Section 6.1.



(a) Launching time. (b) Loading time.

Figure 2: Impact of L2P cache misses on user-perceived latency.

4.2 Impact of L2P Cache Misses on UX

In order to understand how much L2P cache misses affect the quality of UX, we quantitatively measure the user-perceived latency when an app is being launched and loaded. We measure the app launching time of apps from social media and utilities, and the app loading time of games while executing the app-usage scenario as described in Section 6.1.

Fig. 2 shows our experimental results. We compare the app launching time of three system setups: `UFS`, `UFS+HPB`, and `OPTIMAL`. `UFS` only uses a small cache (*i.e.,* 512 KB) to keep L2P entries. In addition to the UFS-level cache, `UFS+HPB` expands the capacity of the L2P cache by borrowing the host memory, 256 MB in our setup. `OPTIMAL` represents the optimal case that assumes the underlying UFS has sufficient memory space to keep the entire L2P entries. The `OPTIMAL` setup neither suffers from extra I/Os caused by L2P cache misses nor needs to steal host memory to expand its L2P cache size. Note that the difference in the latency between apps is due to different amounts of data needed for the execution of each app. By monitoring the memory consumption of each app, we observe that the maximum memory consumption gap is approximately 1 GB (between *FBM* and *GI*).

As expected, `OPTIMAL` exhibits the best performance across all apps, outperforming `UFS` and `UFS+HPB` by up to 50% for *UB* and 43% for *IG*, respectively. These results confirm that

[1] Asphalt9 (*AP*), Clash Royale (*CR*), Genshin Impact (*GI*), Facebook (*FB*), Instagram (*IG*), Twitter (*TW*), Airbnb (*AB*), Facebook Messenger (*FBM*), and Uber (*UB*) (see Section 6.1 for app usage scenarios).

[2] We acquired the numbers for the latency model through a discussion with a storage vendor since the official datasheet is not publicly available.

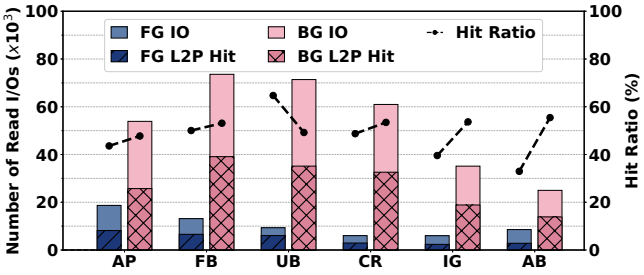


Figure 3: The number of read I/Os and the HPB hit ratios of FG and BG apps.

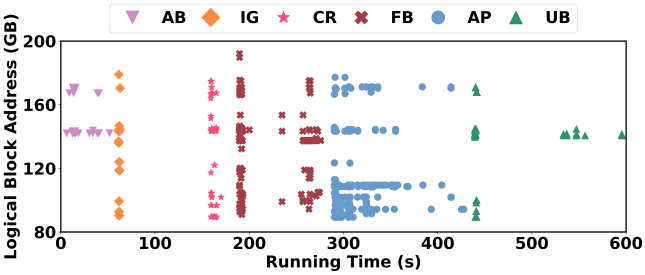


Figure 4: Read I/O access patterns of FG apps.

L2P cache misses greatly impact user-perceived delays. Even worse, absolute launch-time gaps are much wider than our expectations: 220 ms between `OPTIMAL` and `UFS`; 183 ms between `OPTIMAL` and `UFS+HPB`, on average. In order to deliver high-quality UX, reducing every millisecond matters [43, 44]. This is further emphasized by the recent mobile hardware trend of shipping displays with higher refresh rates [45–48]. For example, just 3.5 ms of delay can result in a noticeable stutter with a 144-Hz display [49]. It is important to optimize the user-perceived latency since it is well-known in the industry that a delay of just 100 ms can have significant consequences in online marketplaces [43].

We make two prominent observations from the above results. First, even though `UFS+HPB` borrows relatively a large amount of memory – 256 MB that can cache 25% of the entire L2P entries in its cache – from the host, it shows a marginal improvement in the app launching time. According to our observations (see Section 4.3), the L2P cache management policy fails to cache useful L2P entries that have a high impact on user-perceived latency. Instead, it often caches less important entries associated with BG apps, wasting valuable memory. Second, `UFS+HPB` shows worse performance than `UFS` for some apps – *CR*, *IG*, and *UB*. Our analysis reveals that stealing too much memory from the host incurs severe memory pressure. This leads to the frequent killing of apps, which results in many additional I/Os when the killed apps are launched again (see Section 4.4).

4.3 Impact of HPB Management Policy on UX

To figure out the root causes of why HPB performs poorly with a large mapping cache memory, we compare the hit ratios of FG and BG apps. We observe that FG apps suffer from higher miss ratios than BG apps, regardless of the cache size. Fig. 3 counts the number of I/Os issued by FG and BG apps and also displays how many of them are hit by the HPB cache. Except for *UB*, FG apps experience more L2P cache misses than BG apps.

We analyze detailed behaviors of state-of-the-art HPB management techniques. We find that the low hit ratios of FG apps are mainly due to wrong decisions made by a reference count-based L2P fetch policy and a timer-based eviction policy employed by the HPB manager in the Android kernel [15]. The HPB manager measures reference counts of LBAs and fetches L2P entries from the storage that have a large number of reads. However, as shown in Fig. 3, the number of read I/Os issued by BG apps is relatively larger than those by FG apps. L2P entries associated with BG apps are likely to have larger reference counts than those of FG apps. This results in unintended consequences that the HPB fetches L2P entries for BG apps. Simply fetching LBAs with large reference counts cannot guarantee improved UX.

The timer-based eviction policy is another root cause that makes the HPB inefficient. Even when the HPB cache space is not full, HPB evicts a cached L2P entry that is not referenced for a predefined time (*e.g.,* 100 seconds in the Android Common Kernel v5.10 [15]). This timer-based eviction policy also does not consider the app usage patterns of the user, and thus often evicts L2P entries associated with FG apps. In general, after using an FG app for a while, a user moves to another app and then returns to the former FG app again. If the former FG app has not been used for a relatively long time, the timer-based policy would have evicted its L2P entries. When the user re-launches the former FG app, its L2P entries will no longer exist in the HPB memory, which results in mapping misses and may increase user-perceived delays.

Random I/O patterns that typically occur when an app is launched make it challenging for the HPB to provide high L2P hit ratios. Fig. 4 illustrates partial LBA access patterns of FG apps when they are launched and run for a while. As shown in Fig. 4, we observe that many small random reads, which span a wide range of LBAs, are heavily issued at the beginning of app launches. This randomness results in high L2P cache misses. Fig. 5 illustrates trends of the number of L2P cache misses over time for some selected apps in

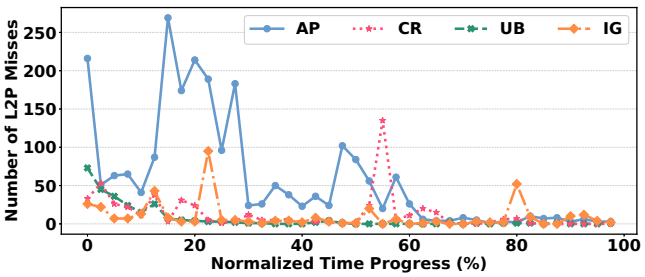


Figure 5: Distributions of the total number of L2P cache misses of FG apps over execution time.

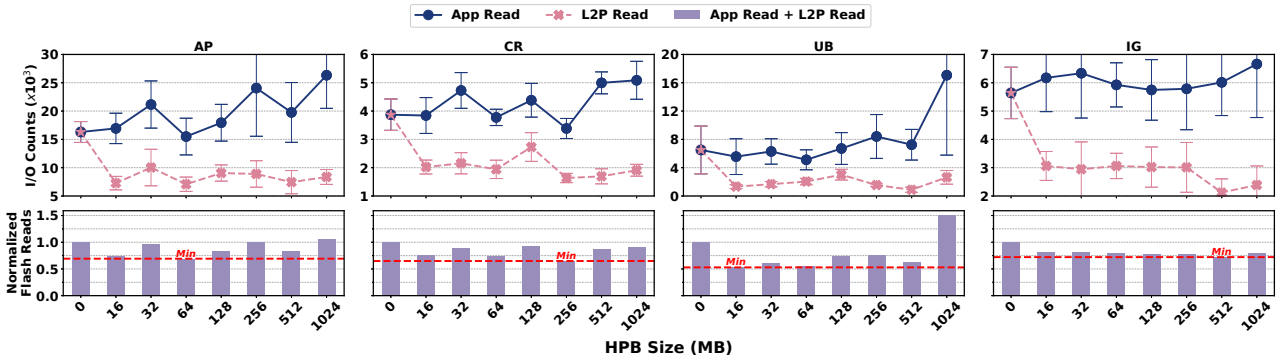


Figure 6: Number of read I/Os issued by FG apps and the corresponding HPB L2P misses with different HPB sizes.

Fig. 4. As expected from I/O patterns in Fig. 4, many L2P cache misses are concentrated in the early stages of an app's execution. For example, *UB* experiences 74% and 90%, and *AP* experiences 20% and 51% of its cache miss in the first tenth and the first fifth of its total execution time respectively. It is worth noting that even if a smartphone user does not manually close the used apps, it is practically infeasible to keep all apps open in BG due to memory constraints [16, 33, 50], even on devices with large amounts of DRAM [51]. Thus, random I/Os are inevitable in the mobile environment, and improving them is a key factor in providing a better UX.

4.4 Impact of HPB Size on UX

HPB shares the same host memory with the Android platform. To provide optimal performance to users, the size of HPB memory should be carefully tuned. Allocating large amounts of host memory to HPB is beneficial in improving L2P hit ratios. On the other hand, as mentioned in Section 4.2, assigning too much memory to HPB might result in UX degradation due to an increase in memory pressure. To prevent HPB from over-consuming memory, the HPB manager employs a timer-based eviction policy. However, as shown in Section 4.3, its FG app-oblivious decisions often cause side effects resulting in evictions of the FG app's cached L2P entries.

To understand how much the HPB memory size affects the user-perceived latency, we observe how the number of read I/Os changes while varying the HPB size usage from 0 to 1 GB. In our evaluation setup, 1 GB of memory is large enough to keep all of the L2P entries in HPB. With recent mobile devices with more and more DRAM (*e.g.,* 18 GB) [52], this amount may sound trivial. However, memory pressure is still often observed in Android systems [51, 53]. Contrary to server or desktop systems, Android tries to maximize memory utilization to maximize its caching capabilities by default [54]. Also, due to the general trend of apps using more resources [55], Android is often susceptible to high memory pressure even with a large capacity of memory. Consequently, relieving memory pressure on the Android system depends on low memory killer by terminating the least important apps. Thus, statically reserving a large amount of memory for improving storage performance is a short-sighted decision with no consideration of its impact on the overall UX.

From the experimental results shown in the Fig. 6, we make two key observations on the impact of the different HPB sizes. First, the optimal HPB size, which results in the minimal number of flash reads (*i.e.,* app reads + L2P reads), is different for each individual app. For example, *IG* shows the minimum number of flash reads with 512 MB whereas *UB* only needs 16 MB. Second, the number of FG apps issued I/Os (app reads) gradually increases as more and more memory is allocated to the HPB. Fig. 6 counts the number of I/Os issued from FG apps and the HPB. As the HPB size increases, thanks to the improved L2P hit ratios, L2P reads from the HPB tend to decrease. While at the same time, since HPB increases the memory pressure of the system, FG apps tend to issue an increased number of read I/Os (*e.g., UB* issues 142% more read I/Os with 1-GB HPB memory when compared to none of the host memory is allocated to the HPB).

Under memory pressure, Android starts killing apps to relieve memory pressure. LMKD uses pressure stall information (PSI) [50] provided by the Linux kernel to detect memory pressure situations, and decides when and how to kill apps. Using PSI, LMKD monitors memory pressure levels and kills the least important app repeatedly until the memory pressure is relieved. If the system consumes more memory, it naturally leads to LMKD killing more apps. As shown in Table 1, killed apps (cold state) not only take much longer to launch, up to 6.2×, but it also incurs much more I/Os, up to 12×, further degrading the UX [7, 34]. Hence, the HPB size is a trade-off regarding the overall UX which should be carefully tuned.

To understand how HPB affects the behavior of LMKD (*e.g.,* how often it kills and how important the victim app is), we analyze the LMKD kill counts for each priority cate-

	Warm state		Cold state	
	Launching time (ms)	I/O counts	Launching time (ms)	I/O counts
AP	352.6	250	2188.6	2164
CR	239	217	668.4	2879
UB	366.4	28	563	60
IG	482.3	349	1245.1	4224

Table 1: App launching time and the corresponding I/O counts of two different launching states.

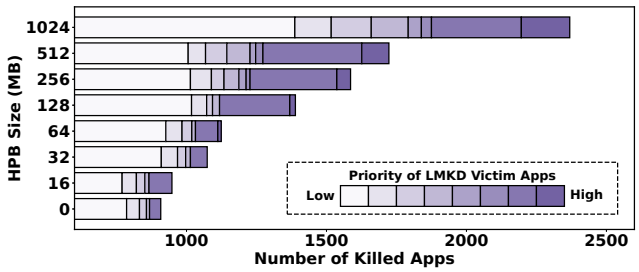


Figure 7: Number of LMKD killed apps and the proportion of the killed apps' priority with different HPB sizes.

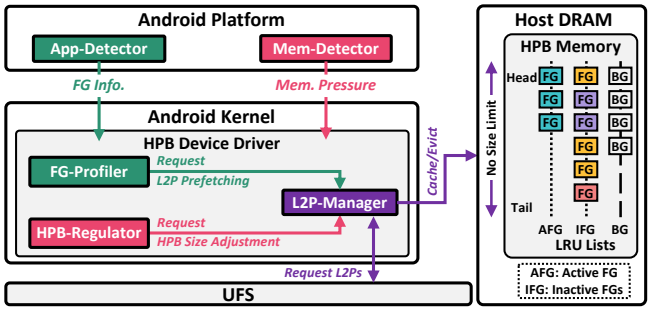


Figure 8: An overall architecture of HPBvalve.

gory. Fig. 7 shows the histogram of LMKD kill counts with different HPB sizes. The higher the priority is, the more user-perceptible the app is (*e.g.,* the second-highest apps are the user recently used ones but in the BG) [56]. As expected, an increase in the HPB size results in a greater number of apps and a higher proportion of high-priority apps being killed by LMKD. Even though the existing HPB scheme might provide better L2P hit ratios, UX degradation is inevitable.

Based on our observations, we conclude that the state-of-the-art timer-based HPB size adjustment policy is suboptimal in two aspects. First, while the timer-based HPB size adjustment could lower the HPB's memory usage, it cannot dynamically relieve memory pressure as HPB is unaware of the current memory pressure status. Second, when users run multiple apps simultaneously the system will suffer from severe memory pressure due to the increased memory utilization by both user apps and the HPB. In such a case, the timer-based eviction policy is unable to proactively and selectively evict cached entries as most HPB cached entries are recently referenced. Managing HPB memory with unawareness of the memory pressure status poses a significant risk of degrading the UX. To achieve the best HPB performance, the size of HPB memory should be dynamically adjusted by considering the memory pressure status while not sacrificing the L2P cache performance.

5 Design and Implementation of HPBvalve

Our empirical study presented in Section 4 reveals that the naïve integration of HPB to Android does not guarantee improved UX. Moreover, the existing techniques neither efficiently cache or evict L2P entries in the HPB memory, nor decide a proper size of the HPB memory from the perspective of maximizing the user-perceived performance.

To improve the user-perceived latency of smartphones, we should minimize the L2P cache misses of I/O requests from FG apps. If UX-sensitive I/O requests are always hit by the HPB memory, smartphone users experience the equivalent performance as if the entire L2P entries are cached. At the same time, to prevent user-noticeable and important apps from being killed by LMKD, we should wisely adjust the HPB size according to the status of the system memory pressure.

To accomplish the above goals, the existing HPB layer needs to be improved in two aspects. First, HPB should identify which I/Os are user-latency sensitive or not. Once identified, HPB should appropriately manage associated L2P entries in the L2P cache, particularly in an FG app-centric manner. Second, HPB should be aware of the memory pressure status of the Android system. Then, it should decide whether to increase or decrease the HPB memory size for higher L2P cache hit ratios or for relieving memory pressure.

Keeping the above observations in mind, we propose an integrated host-SSD mapping management scheme, called HPBvalve (Hvalve in short), which addresses the limitations of existing techniques. We aim to design Hvalve to be simple yet effective for its wide adoption in real-world devices. To this end, Hvalve leverages information that is already collected by other existing modules in the Android platform, which enables Hvalve to exploit a variety of information in a vertically-integrated manner at a low cost.

5.1 Overall Architecture of HPBvalve

Fig. 8 illustrates an overall architecture of Hvalve that is composed of five key modules – App-Detector, Mem-Detector, FG-Profiler, L2P-Manager, and HPB-Regulator. Hvalve has a cross-layered design that spans across a wide range of system layers from the Android platform to the kernel. Two modules, App-Detector and Mem-Detector, implemented in the Android platform monitor the system status and collect a set of information, including (*i*) the type of apps (*i.e.,* FG or BG) that issue I/Os, (*ii*) app state changes, and (*iii*) the memory pressure status. This information is then delivered to the HPB device driver in the Android kernel. Based on the delivered information, three modules implemented in the kernel, FG-Profiler, L2P-Manager, and HPB-Regulator, manage HPB in a UX-centric manner by (*i*) separately managing and profiling L2P entries of FG apps, (*ii*) prefetching the profiled L2P entries on every app launch, and (*iii*) adjusting the HPB size dynamically depending on the memory pressure status.

5.2 FG App-centric HPB Management

In this section, we explain how Hvalve manages L2P entries using the *FG app-centric caching policy*.

FG/BG classification: In order to identify user-latency-sensitive I/O requests, every submitted I/O has to be distinguished whether it is submitted by an FG or a BG app. To this

end, we extend the kernel and Android framework so that the kernel I/O stack becomes aware of the app-level information. With our extension, every I/O request holds its caller UID (a unique number that the Android system assigns to every app). When a regular I/O system call is invoked, a new `struct bio` is allocated under the same process context. Since the same process context is maintained, the caller's process control block (`struct task_struct`) is accessible from the `bio` allocation step. We add a new member field in the `bio` to copy the caller's UID from the PCB. The new UID field can be used in deciding whether a `bio` belongs to an FG or BG app.

In order to distinguish whether the submitted I/O is from an FG app, the UID embedded in the request header has to be compared to the UID of a current FG app. **App-Detector** is designed to detect an FG app in the system. The App-Detector keeps track of every app state change (*e.g.,* a new FG launch-start and launch-end) by referring to Android's activity task manager [57]. Upon every app state change detected, App-Detector delivers a state change message to Hvalve. For example, when a new FG launch is detected, App-Detector delivers a launch-start signal (*e.g.,* a new FG app is launched) to the HPB in the kernel along with its UID. Hvalve makes use of the delivered information for distinguishing every FG app-submitted I/O. If an I/O request passed to the block layer has a different UID from the App-Detector-passed current FG app's UID, it is considered as a BG I/O. This makes it possible for Hvalve to manage HPB memory to assign higher priority to L2P entries of an FG app (*i.e.,* UX-sensitive L2P entries).

L2P management: In order to prioritize L2P entries of FG apps, the **L2P-Manager** manages cached L2P entries in three separate LRU lists depending on their importance –*AFG* (an active FG app), *IFG* (inactive FG apps), and *BG* (BG apps) lists, as illustrated in the Fig. 8. The reason for Hvalve managing cached entries with three different LRU lists is to differentiate the priority upon eviction. With these three separate LRU lists, Hvalve is able to give different priorities to the cached entries that are referenced by a user currently interacting FG app, previously user-interacted FG apps, and BG apps. Whenever an L2P of the current FG app gets cached to the HPB memory, it is first inserted into the AFG list. If a user launches a new FG app, the L2P entries in the AFG list get demoted to the IFG list as they now belong to the previous FG app. On the other hand, if an L2P entry from a BG app gets cached to the HPB memory, it directly goes into the BG list. Moving between lists, inserting or removing entries from each list is trivial as all lists are implemented with hash lists. With these three different LRU lists, Hvalve is able to manage the HPB memory in an FG app-centric manner.

Caching policy of Hvalve: Hvalve respects the HPB's counter-based caching policy, thus it also caches frequently referenced L2P entries to the HPB memory. In addition to this, Hvalve cache L2P entries that satisfy the following conditions. First, for every HPB cache miss that originated from the current FG app, the L2P-Manager immediately caches the corresponding L2P entry to the HPB memory to prevent further cache misses of the said L2P. Second, a set of L2P entries of the launched FG app are directly cached to the HPB memory with the help of the **FG-Profiler** upon every app launch signal. FG-Profiler is designed to collect information on the current FG app to help HPB prioritize FG L2P entries to improve the launching time as well as the total user-perceived latency.

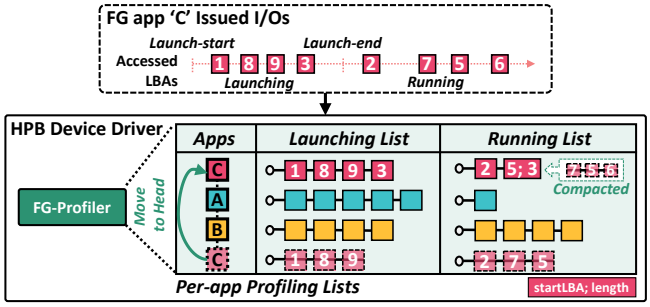


Figure 9: An example of per-app profiling lists being updated.

FG app profiling and prefetching: As discussed in Section 4.3, every app has its unique I/O patterns. To better manage the L2P cache based on apps' unique I/O characteristics, FG-Profiler maintains an LRU list of recently used FG apps, each containing two separate L2P profiling lists: *app launch list* and *app running list*. The app launch list contains a list of LBAs that are referenced during an app launch (*i.e.,* from a launch-start signal to a launch-end signal, delivered by App-Detector). The app running list holds a list of LBAs that are accessed during the execution of the app (*i.e.,* after a launch-end signal).

Fig. 9 illustrates an example of how the FG-Profiler maintains the L2P profiling lists of FG apps. Once it receives a launch-start message, it adds or moves the new FG app to the head of the app-LRU list. Until a launch-end signal arrives, it profiles every L2P entry needed by the launched app.

Fig. 10 shows an overview of how the L2P-Manager prefetches L2P entries for every app launch. When the App-Detector is notified of a launch-start of a new FG app it sends a launch-start signal to HPB (❶). Upon every launch-start signal, the Hvalve tracked UID of the current FG app gets updated to the UID of the new FG app. Then, the FG-Profiler searches for the previously created profiling list of the new FG app (❷). If the profiled list is found, the FG-Profiler requests the L2P-Manager, (❸), to prefetch the L2P entries from the list. This hides mapping miss penalties of the new FG app during an app launching, as well as running.

Since every L2P prefetch request results in a new flash page read operation, the FG-Profiler first prioritizes prefetching L2P entries from the launch list. Only after prefetching the L2P entries profiled in the launch list, the entries in the running list are requested to be prefetched. The L2P-Manager preferentially fetches the FG-Profiler-requested L2P entries from the UFS (❹). Those prefetched FG L2P entries are then inserted to the tail of the FG LRU list, AFG, rather than the head of the

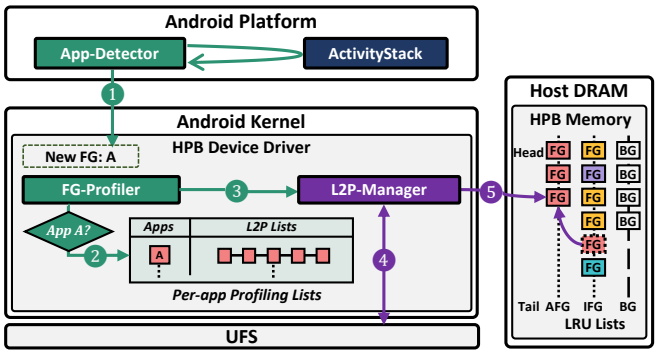


Figure 10: L2P prefetching mechanism of a new FG app.

list (❺). This is to differentiate priority upon victim selection between the actually referenced entries by the FG app and the Hvalve-prefetched entries. Even if the prefetched entries are inserted into the tail of the AFG list, Hvalve does not attempt to evict them until they are demoted to the IFG or BG lists.

Managing L2P lists per app is not only efficient for better L2P hit ratios of FG apps, but also space-efficient in utilizing the host memory. The profiling list is maintained as a pair of the start chunk of the accessed LBA and the length of the neighboring referenced chunks. Each of the profiling lists only requires a few tens of kilobytes for most apps, 7 KB on average. Moreover, to prevent the per-app L2P profiling lists from excessively consuming the host memory, we statically limit the maximum size of the total profiling lists to 1 MB. Hvalve does not provide special handling for deduplicating entries between app lists. Since the size of an entry of the profiled per-app list is only 4 bytes, it is unlikely that significant memory gain can be achieved by removing duplicate entries between lists. If the number of profiled apps exceeds the predefined limit, the FG-Profiler frees the least recently used app from the list to allow the newly launched FG app to be profiled. Our current implementation sets this number to 20, which corresponds to the average smartphone usage [34]. This keeps the entire per-app L2P lists in less than 1 MB of memory – a negligible space overhead. It can also be easily extended to dynamically adjust if needed (see Section 6.3 for more details).

5.3 Dynamic HPB Size Adjustment

To the best of our knowledge, there exist no techniques that optimally decide the HPB size to provide the best performance by considering its impact on the overall UX quality. Hvalve neither insists on statically allocating small HPB size to minimize its impact on memory pressure nor large HPB size to boost L2P hit ratios. Instead, Hvalve proposes a *dynamic HPB size adjustment scheme* that adjusts the HPB size based on the monitored memory pressure status. Hvalve adaptively controls the HPB memory size for higher I/O performance while no user-interacting apps are mistakenly killed by LMKD. Hvalve is unique in that I/O performance improvements are achieved without negatively affecting UX-critical factors.

With Hvalve, if a non-memory-intensive app runs and the system has enough free memory, the HPB size can increase to cover the entire L2P mapping table. This enables us to maximally exploit the full benefits of HPB. However, whenever the system starts to experience memory pressure, Hvalve immediately adjusts the HPB size accordingly, returning memory for apps to use. As a result, the degradation of UX by excessively assigning host memory to HPB does not occur.

Fig. 11 illustrates how Hvalve dynamically adjusts the allocated HPB memory. As discussed in Section 4.4, Android employs LMKD which selectively kills running apps to relieve memory pressure. Once LMKD decides on a victim app to kill, the **Mem-Detector** notifies the **HPB-Regulator** with the target's UID before LMKD starts killing the victim app (①). The HPB-Regulator decides how important the victim app is by checking the per-app L2P profiling list (②). If the victim app is not found in the per-app L2P profiling list (*i.e.,* not a user recently used app), the signal is ignored and leaves LMKD to continue killing the victim app. On the other hand, if the victim app exists in the per-app L2P profiling list, it is treated as an important app (*i.e.,* a user recently interacted app). Then to prevent the important app from being killed by LMKD, Hvalve preferentially reclaims the HPB memory by aggressively evicting low-priority cached L2P entries.

To decide how much memory to free from HPB, the properties of the LMKD victim app are taken into consideration. As LMKD calculates the expected amount of memory to be freed upon its victim selection, Hvalve attempts to free as much as the LMKD desired amount. When the HPB-Regulator tells L2P-Manager how much HPB memory to free (③), it delivers how many cached entries in the HPB memory should be evicted to prevent the LMKD killing its victim app.

The L2P-Manager considers the priority of cached L2P entries when choosing which entries to evict. Hvalve marks every cached L2P entry with its last-referenced UID, and stores to an LRU list (*e.g.,* AFG, IFG or BG) depending on the type of the app it belongs to. With the three separate LRU lists, Hvalve can make a fine-grained decision on which entry to evict first as Hvalve is aware of which entries belong to the

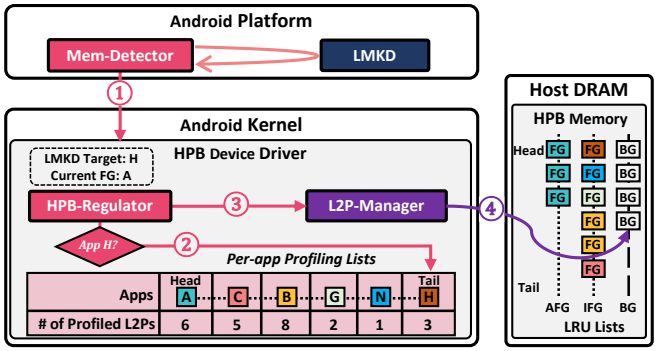


Figure 11: HPB size adjustment mechanism of Hvalve.

BG apps (*BG list*), a user recently used FG apps (*inactive FG list*), and a currently user-facing FG app (*active FG list*). The L2P-Manager starts to evict entries from the tail of the *BG list*, (④). If there are no more entries from the *BG list* to evict, the ones from the *inactive FG list* are tried next. L2P-Manager never evicts entries from the *active FG list* to avoid UX degradation. The entries in the *inactive FG list* will naturally get evicted by the L2P-Manager if it is left unused for a long time. On the other hand, the entries in the *BG list* can be promoted to the *FG list* if an FG app references entries in the *BG list*.

The proposed dynamic HPB size adjustment policy of Hvalve may result in an increase in the execution time of the LMKD's app-killing process. This occurs when there is an insufficient number of HPB cached entries that can be evicted to alleviate the memory pressure. In such a case, LMKD has to resume the paused app-killing process to reserve free memory space. The increased amount of execution time, however, is marginal compared to the relatively longer procedure of LMKD. In addition, from a long-term perspective, the dynamic HPB size adjustment is much more beneficial to the overall UX quality as it prevents user apps from being killed. A more detailed analysis is described in Section 6.3.

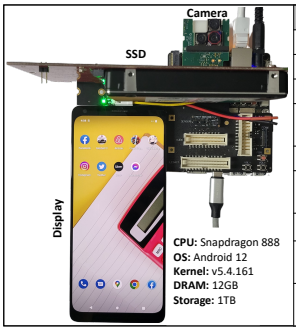
6 Experimental Results

In this section, we evaluate the overall quality of UX when Hvalve is applied to an HPB-enabled system.

6.1 Experimental Setup

To evaluate the effectiveness of the proposed techniques, we implement the App-Detector, Mem-Detector, FG-Profiler, L2P-Manager, and HPB-Regulator of HPBvalve on a Snapdragon 888 Mobile HDK [17], which is illustrated in Fig. 12. Our evaluation platform uses the same board support package that is used on other production smartphones using the same SoC. This HDK has 12 GB of DRAM (effectively 8 GB) and PCIe 3.0 x2 connectivity, which we use to connect PCIe peripherals. We use Android 12 and Linux kernel v5.4.161 to implement Hvalve. As it is practically infeasible to modify the UFS firmware, we use an ultra-low latency NVMe SSD [42] described in Section 4.1 and implement a lightweight FTL in the kernel to mimic UFS storage, which consumes approximately 4 GB of memory to run our test scenarios. We have written about 1,000 LOC to implement Hvalve, which we open-sourced on GitHub[3], including the changes made to the Android platform and the kernel.

We use `am start` command [35] to measure app launching and switching times. For apps with multiple loading stages, the `am start` command is unable to measure the total time taken until the device is ready to take user inputs. For example, *GI* goes through three separate loading stages until the gameplay button appears while the `am start` command only measures the time taken until the first stage. To precisely measure the app loading time, we employ an external high-speed camera, which captures 120 frames per second that match the refresh rate of our evaluation platform's display.



Apps	App Usage Scenario
AP	Play a stage.
CR	Play a stage.
GI	Explore the map.
FB	Browse Feed, Watch, Stories, Notifications, and other pages.
IG	Browse Feed, Reels and write comments.
TW	Browse Timeline, Trending, Notifications , and reply to a tweet.
AB	Browse the globe map and opens multiple listings.
FBM	Browse Chats, Calls, People, Stories, and send messages.
UB	Browse Activity, Account settings, and search the map.

Figure 12: Prototype HPBvalve setup and app usage scenarios.

As for the benchmarks, we evaluate nine popular mobile apps listed in Section 4.1. To automatically run multiple mobile apps under realistic app usage scenarios, we use the Android debug bridge (*adb*) [58]. The predefined app usage scenarios are described in Fig. 12. To avoid cherry-picking sequences that would favor Hvalve, the sequence of the nine apps is randomized and run multiple times to reduce variables. The chosen random sequence is executed for each technique for a fair comparison.

Even though the same randomized sequence was run for each technique, there still exists run-to-run variations due to noises such as network conditions, random advertisement occurrences, and others. In order to minimize run-to-run variations, we fully automated the evaluation process to repeat the same scenario twenty times for each case. We also disabled the checkpoint on the underlying file system, f2fs [59], so that the entire *userdata* partition could be rolled back to the previous state to further minimize variances. After running twenty times, we averaged the results of fifteen runs, excluding the five outliers. We compared Hvalve with the typical HPB system that employs FOAM [6] as its L2P eviction policy, `UFS+HPB`, and an ideal system where all L2P entries are cached in memory, `OPTIMAL`, and a conventional UFS system without HPB, `UFS`. We also compared Hvalve-Only with `HPB-Only` where the underlying UFS SRAM is not considered to evaluate the effectiveness of caching policy of Hvalve.

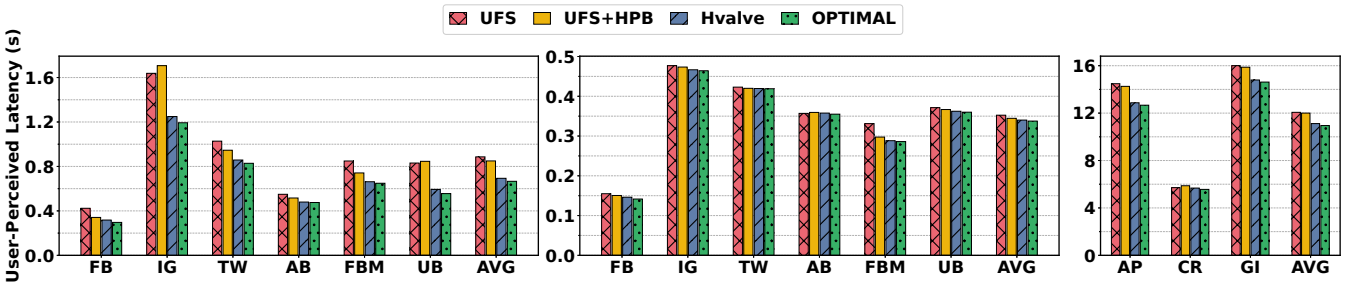
6.2 Performance Evaluation

In order to validate the effectiveness of Hvalve, we assess the impact of FG app-centric HPB management and dynamic HPB size adjustment techniques on the overall UX quality compared against `UFS+HPB`.

6.2.1 FG app-centric HPB management

User-perceived latency: As for the most important performance evaluation metrics in deciding the quality of UX, we assess the app launching, switching, and loading times of various HPB configurations.

[3]The source code is available at https://github.com/cares-davinci/Hvalve.



(a) App launching time of social media and utilities. (b) App switching time of social media and utilities. (c) App loading time of games.

Figure 13: Evaluation results of the user-perceived latency.

Fig. 13(a) shows the experimental results of the app launching time. The average app launching time of Hvalve is improved by 28% and 23% when compared to UFS and UFS+HPB, respectively. The maximum and minimum app launching time improvements between Hvalve and UFS+HPB are 43% for *UB* and 7% for *FB* where the corresponding absolute app launching time improvements are 237 ms and 23 ms, respectively. We also compare the app launching time of Hvalve to OPTIMAL. The minimum and the maximum absolute launch-time differences between OPTIMAL and Hvalve are only by 4 ms (+0.85%) for *AB* and 55 ms (+4.7%) for *IG*. On the other hand, UFS+HPB takes 40 ms (+8.5%) for *AB* and 514 ms (+43.1%) for *IG* more when compared to OPTIMAL.

We also measure the app switching time – the latency of when a user switches back to an app that is recently launched. As shown in Fig. 13(b), Hvalve outperforms UFS+HPB for all cases. The maximum app switching time difference between Hvalve and OPTIMAL is only 4.6 ms (+3.15%) with *FB* while it is 13.4 ms (+8.65%) on UFS+HPB in the same scenario. The minimum increase in app switching time of Hvalve compared to OPTIMAL is 0.5 ms (+0.12%) with *TW*, while the gap between UFS+HPB and OPTIMAL of the same case is 4 ms (+0.95%). Since a comparatively small number of I/Os are issued while an app is being switched compared to the app launch process, as described in Table 1, the performance increase with Hvalve for the app switching time is quite marginal compared to the other two metrics.

We also evaluate the app loading time of games by using an external high-speed camera to measure the time from a user launch of an app until the device is ready to take user inputs. As shown in Fig. 13(c), the app loading time of all three games is improved which provides almost the same user-perceived latency as OPTIMAL. The minimum increase in app loading time of Hvalve compared to OPTIMAL is 111.4 ms (+2%) with *CR* while the gap between UFS+HPB and OPTIMAL of the same case is 308.45 ms (+5.5%). The maximum app loading time difference between Hvalve and OPTIMAL is only 205 ms (+1.6%) with *AP* while it is 1589 ms (+12.5%) on UFS+HPB with the same scenario. When comparing Hvalve to OPTIMAL, the increase in user-perceived latency, mainly caused by mapping misses, is very marginal.

As discussed in Section 4.2, every millisecond of responsiveness greatly impacts the UX. According to our evaluation results, the storage mapping miss penalties, resulting in user-perceived delays, are significantly alleviated with Hvalve. These benefits come from the FG app-centric HPB management policy employed in Hvalve, which gives higher priority to L2P entries of FG apps to be managed in the HPB memory. To summarize, Hvalve alleviates the storage mapping miss penalties of the baseline UFS+HPB by 80% for app launching time and by 86% for app loading time on average.

L2P miss patterns: To analyze the impact of our proposed FG app-centric HPB management of Hvalve on FG apps, we first observe how much L2P miss distributions differ from UFS+HPB over an app execution. Due to the page limit, we include four representative apps, *AP*, *CR*, *UB*, and *IG*, two from games and one each from social media and utilities.

As examined in Section 4.2, most of the L2P cache misses occur during the early stages of the total app execution time. Fig. 14 shows the distributions of L2P misses over the execution time of each FG app. Such mapping misses that occur in the early stage are one of the root causes that increase user-perceivable delays. Hvalve significantly reduces the peak number of L2P misses during the early stage as well as the total number of the L2P cache misses compared to UFS+HPB. The maximum L2P miss reduction in the first tenth of the total normalized execution time is 88% with *CR* over UFS+HPB, while the minimum is 45% with *AP*. During the first fifth of the app execution time, 71% and 75% of the mapping misses are alleviated for *UB* and *IG*, respectively.

The peak L2P misses are greatly reduced not only in the early stages, but also throughout the entire execution time. This advantage comes from both the caching and the eviction policy of Hvalve. Hvalve prepares L2P lists by prefetching when an app is being launched, and those of the prefetched or cached entries of the current FG L2P entries never get evicted from HPB memory as long as it remains as an FG. The mapping miss penalties included in the user-perceived latency, which could severely degrade the overall quality of UX, are greatly reduced.

Hit ratios: In addition to the reduced user-perceived latency and L2P misses, we also quantitatively evaluate the

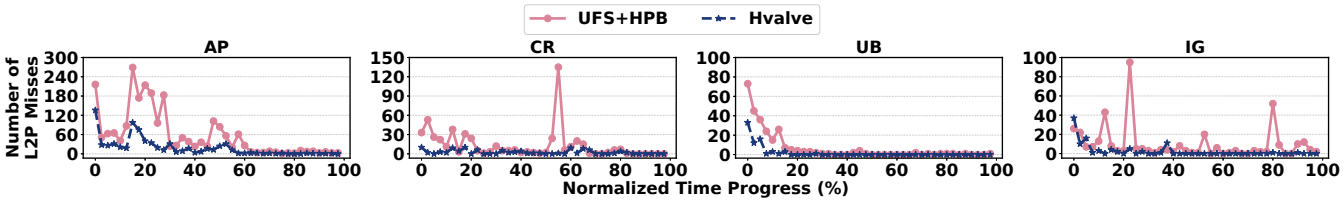


Figure 14: Distributions of L2P misses of FG apps over execution time.

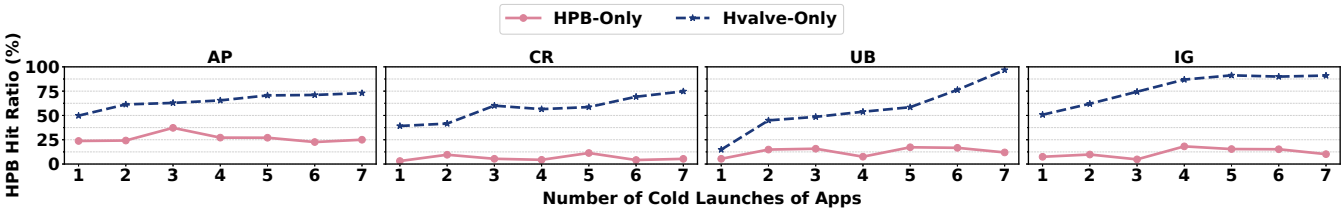


Figure 15: HPB hit ratios of each launch when consecutively launched for seven times.

effectiveness of the FG app-centric L2P management policy of Hvalve, by comparing `HPB-Only` to Hvalve-Only where the underlying UFS SRAM is not considered. Fig. 15 illustrates the HPB hit ratios of each FG app. The hit ratios of Hvalve-Only outperform `HPB-Only` for all cases. In order to evaluate the effectiveness of the L2P prefetching scheme of Hvalve, we observe how the hit ratios of each FG app change over a few numbers of consecutive app launches.

As the number of app launch counts increases, the hit ratios of Hvalve keep increasing while the hit ratios of `HPB-Only` remain consistently low. The largest hit ratio difference between `HPB-Only` and Hvalve-Only is 84.79% with *UB*. This result proves that the L2P prefetching mechanism successfully improves the performance of FG apps by hiding miss penalties. The above results also confirm that Hvalve is effective in providing a better quality of UX as it actively reflects the app usage patterns of individual smartphone users in managing the HPB memory.

6.2.2 Dynamic HPB size adjustment

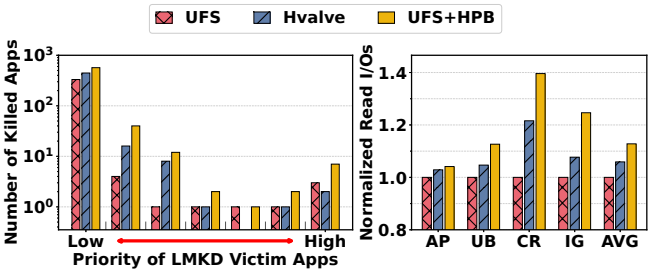
We compare Hvalve with `UFS+HPB` to evaluate the impact of our proposed dynamic HPB size adjustment scheme on UX. While Hvalve dynamically adjusts the HPB size depending on the monitored memory pressure status, `UFS+HPB` statically allocates the HPB size and adjusts it with a simple timer-based eviction policy. In this evaluation, we set `UFS` as a baseline, which does not require extra host memory to load storage mapping entries (*i.e.,* no impact on the memory pressure).

Fig. 16(a) shows a log-scaled histogram of the number of LMKD killed apps with seven different priority categories. As no extra host memory is used for allocating HPB memory (*i.e.,* no extra memory pressure), `UFS` results in the lowest number of kills on every priority category. On the other hand, the result of `UFS+HPB` shows much more apps were killed on every priority category when compared to `UFS`. The number of killed apps in the top three priority categories was increased by 5× with `UFS+HPB` when compared to the `UFS`. While providing much higher L2P cache hit ratios, Hvalve reduces the number of apps killed by LMKD in the top three priority categories by 70% compared to `UFS+HPB`. Therefore, we again prove that simply integrating HPB into the system inevitably increases the memory pressure resulting in high-priority user apps being killed.

To further investigate the consequences of high-priority apps being killed by LMKD, Fig. 16(b) shows the change in the number of read I/Os issued by FG apps, normalized to the number of read I/Os of `UFS`. The number of read I/Os with `UFS+HPB` is increased by 13% on average when compared to `UFS`, while Hvalve is only increased by 5% as it reduces the impact on the FG apps as well.

To demonstrate how each `UFS+HPB` and Hvalve adjust the allocated HPB memory, Fig. 17 illustrates changes in HPB memory size along with the reported memory pressure signals. Unsurprisingly, `UFS+HPB` allocates and frees HPB memory in a non-harmonized manner with the overall memory pressure status. Even when the memory pressure is present, `UFS+HPB` still allocates host memory to HPB (*i.e.,* adding more memory pressure to the system) which increases the possibility of important apps getting killed by LMKD.

On the other hand, Hvalve allocates host memory to the HPB memory and also effectively returns the HPB memory to



(a) Histogram of LMKD killed apps.

(b) Number of app-issued read I/Os.

Figure 16: Impact of HPB on FG apps.

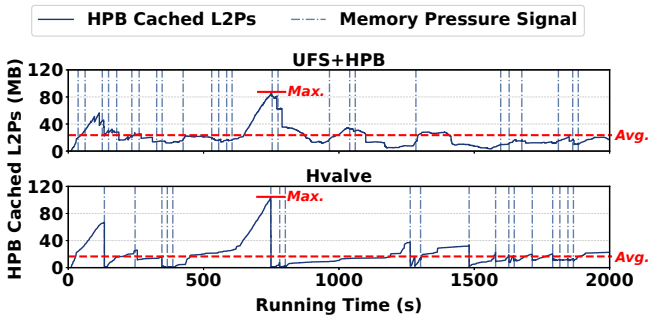


Figure 17: Changes in HPB memory size with the memory pressure signals.

user apps in a harmonized manner with the memory pressure signals. As a consequence, shown in Fig. 16(a), the number of important apps killed by LMKD with Hvalve is much lower than that of `UFS+HPB`. This is because Hvalve proactively prevents LMKD from killing high-priority apps by promptly returning the HPB-allocated memory to the user apps under the monitored memory pressure signals.

6.3 Overhead Analysis

Space overhead: Unlike the existing HPB technique, Hvalve consumes extra memory space for maintaining the per-app profiling lists to prefetch L2P entries upon every FG app launch. To avoid excessive memory consumption in managing the per-app profiling lists, Hvalve profiles only a moderate number (*e.g.,* 20) of recently used apps (which has a high impact on UX). The total profiling app list size is also regulated to 1 MB. The number of profiled L2P entries of each app differs based on the app access patterns. The total memory consumption for per-app profiling lists of the nine apps we use throughout the evaluations only consumes 99 KB of memory. It can be further minimized by merging neighboring groups as each node can be transformed into a compact profiling list that contains `<Start LBA, Length>`.

Performance overhead: While Hvalve manages the cached L2Ps with three separate LRU lists, AFG, IFG, and BG, moving cached entries between the LRU lists does not necessitate exhaustive search overhead since all lists are managed with hash lists. Retrieval and relocating a specific cached L2P entry from one list to another list can be done in $O(1)$ time complexity. The process of promoting or demoting an entry from one list to another is also simple as it requires updating only a few associated pointers.

The overhead of dynamic HPB size adjustment is also negligible as the process of returning HPB memory can be executed comparatively faster than that of the LMKD's app-killing process. Hvalve takes about 1.8 ms on average to free HPB memory whereas the app-killing process of LMKD typically takes hundreds of milliseconds. Although unlikely, in the worst case when Hvalve cannot evict sufficient cached entries to free the requested amount of memory, an extra time overhead (a few ms) can incur as LMKD must resume the suspended app-killing procedure. Despite the potential of Hvalve increasing the LMKD's execution time, preferentially freeing the HPB memory under memory pressure is more advantageous to the overall system performance. For example, if an app is killed by LMKD and re-launched after a while, the evicted app-related data has to be reloaded. On the other hand, re-fetching the HPB entries only requires a much smaller number of I/Os.

Energy Consumption: Employing Hvalve does not require extra energy consumption, since the two proposed HPB management schemes of Hvalve do not introduce severe search or space overheads to the system. As the total execution time of apps is reduced by integrating Hvalve, the total energy consumption with Hvalve is even decreased by 3.51% compared to `UFS+HPB`. When comparing to `OPTIMAL`, `UFS+HPB` increases the total energy consumption by 4.02% while Hvalve only increases by 0.56%. Hvalve successfully mitigates the negative impacts of `UFS+HPB` on resource consumption and the overall quality of UX.

7 Conclusion

In this paper, we present a novel FG app-centric L2P mapping cache management scheme, HPBvalve, for the HPB-enabled system. Hvalve is motivated by the fact that the existing HPB management scheme fails to improve the UX of smartphones due to two main reasons revealed through our empirical investigations. First, the priority of app status (FG or BG) is not considered while managing cached L2P entries in the HPB memory. Second, the memory pressure of the system could get critically high as host memory is allocated to the HPB without considering the memory pressure status. To improve the overall quality of UX upon these shortcomings of the existing HPB, Hvalve prioritizes the FG app in managing the cached L2P entries in HPB memory and dynamically adjusts the size of the HPB-designated host memory by monitoring the current memory pressure status. This allows Hvalve to reduce app kills in the top three priority categories by 70% while achieving significantly higher L2P hit ratios for FG apps. Our experimental results show that Hvalve successfully improves the overall UX quality of smartphones and provides almost equivalent performance as if most entries are cached in the HPB memory.

Acknowledgments

We would like to thank Ali R. Butt, our shepherd, and anonymous reviewers for their valuable suggestions. This work was supported by Samsung Electronics Co., Ltd (IO201207-07809-01) and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2022R1A2C2010529). The ICT at Seoul National University provided research facilities for this study. Sungjin Lee was supported by the National Research Foundation of Korea (NRF-2018R1A5A1060031). (Co-corresponding Authors: Sungjin Lee and Jihong Kim)

References

[1] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones. In *2018 USENIX Annual Technical Conference (ATC)*, pages 15–28. USENIX Association, 2018.

[2] Google - Android Open Source Project. Performance Management. https://source.android.com/devices/tech/power/performance.

[3] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *15th USENIX Conference on File and Storage Technologies (FAST)*, pages 345–358. USENIX Association, 2017.

[4] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices. In *13th USENIX Conference on File and Storage Technologies (FAST)*, pages 191–202. USENIX Association, 2015.

[5] Samsung. UFS. https://semiconductor.samsung.com/estorage/ufs.

[6] Chao Wu, Qiao Li, Cheng Ji, Tei-Wei Kuo, and Chun Jason Xue. Boosting User Experience via Foreground-Aware Cache Management in UFS Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11):3263–3275, 2020.

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

[8] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. *SIGARCH Comput. Archit. News*, 37(1):229–240, 2009.

[9] Wookhan Jeong, Hyunsoo Cho, Yongmyung Lee, Jaegyu Lee, Songho Yoon, Jooyoung Hwang, and Donggi Lee. Improving Flash Storage Performance by Caching Address Mapping Table in Host Memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, page 19. USENIX Association, 2017.

[10] JEDEC. Universal Flash Storage (UFS) host performance booster (HPB) extension, version 2.0. https://www.jedec.org/standards-documents/docs/JESD220-3A.pdf, 2020.

[11] Google - Android Open Source Project. Google Pixel 3 kernel source - drivers/scsi/ufs/ufshpb.c. https://android.googlesource.com/kernel/msm/+/23d68f4b84c3c6a309512f9fef6d80072fb8364a, 2018.

[12] Masafumi Takahashi. UFS Unified Memory Extension. JEDEC Mobile Forum, 2014.

[13] Konosuke Watanabe, Kenichiro Yoshii, Nobuhiro Kondo, Kenichi Maeda, Toshio Fujisawa, Junji Wadatsumi, Daisuke Miyashita, Shouhei Kousai, Yasuo Unekawa, Shinsuke Fujii, Takuma Aoyama, Takayuki Tamura, Atsushi Kunimatsu, and Yukihito Oowaki. 19.3 66.3KIOPS-random-read 690MB/s-sequential-read universal Flash storage device controller with unified memory extension. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 330–331, 2014.

[14] NVM Express. NVMe specifications 1.2. http://www.nvmexpress.org/specifications, 2014.

[15] Google - Android Open Source Project. Android Common Kernel's HPB. https://android.googlesource.com/kernel/common/+/refs/heads/android12-5.10/drivers/scsi/ufs/ufshpb.c.

[16] Google - Android Open Source Project. Low Memory Killer Daemon. https://source.android.com/docs/core/perf/lmkd.

[17] Qualcomm. Snapdragon 888 HDK. https://developer.qualcomm.com/hardware/snapdragon-888-hdk, 2021.

[18] Jung-Hoon Kim, Sang-Hoon Kim, and Jin-Soo Kim. Utilizing Subpage Programming to Prolong the Lifetime of Embedded NAND Flash-Based Storage. *IEEE Transactions on Consumer Electronics (TCE)*, 64(1):101–109, 2018.

[19] OnePlus - GitHub. OnePlus 9 kernel source - drivers/scsi/ufs/ufshpb.c. https://github.com/OnePlusOSS/android_kernel_oneplus_sm8350/blob/1c052de944b391dd50957b03e1fc92f93f35e125/drivers/scsi/ufs/ufshpb.c, 2022.

[20] OnePlus - GitHub. OnePlus 9 kernel source - drivers/scsi/ufs/ufshpb_skh.c. https://github.com/OnePlusOSS/android_kernel_oneplus_sm8350/blob/1c052de944b391dd50957b03e1fc92f93f35e125/drivers/scsi/ufs/ufshpb_skh.c, 2022.

[21] OnePlus - GitHub. OnePlus 9 kernel source - fs/hpb_supp.c. https://github.com/OnePlusOSS/android_kernel_oneplus_sm8350/blob/1c052de944b391dd50957b03e1fc92f93f35e125/fs/hpb_supp.c, 2022.

[22] Xiaomi - GitHub. Redmi 10X, Redmi 10X Pro, Redmi K30 Ultra kernel source - drivers/scsi/ufs/ufshpb.c. https://github.com/MiCode/Xiaomi_Kernel_OpenSource/blob/9be022c0db171ac622e528752410d613c0e4e64d/drivers/scsi/ufs/ufshpb.c, 2021.

[23] Xiaomi - GitHub. Redmi 10X, Redmi 10X Pro, Redmi K30 Ultra kernel source - drivers/scsi/ufs/ufshpb_skh.c. https://github.com/MiCode/Xiaomi_Kernel_OpenSource/blob/9be022c0db171ac622e528752410d613c0e4e64d/drivers/scsi/ufs/ufshpb_skh.c, 2021.

[24] Samsung - Samsung Open Source. Galaxy S20 kernel source - drivers/scsi/ufs/ufshpb.c, fs/hpb_supp.c. https://opensource.samsung.com/uploadSearch?searchValue=G981NKSU1FUL9, 2021.

[25] Motorola - GitHub. Motorola Edge 30 kernel source - Samsung HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/a6cc1ed04b2a76fd325929a7925c01a99a310e0d, 2022.

[26] Motorola - GitHub. Motorola Edge 30 kernel source - SK Hynix HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/e297cf514d64bf85fdc2a1ee8722da8baf0afd4c, 2022.

[27] Motorola - GitHub. Motorola Edge 30 kernel source - Micron HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/26150eb5bb48d37b60f279b9766761926ba17de3, 2022.

[28] Motorola - GitHub. Motorola Edge 30 kernel source - Kioxia HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/f4da8e2db63f5d5818abc37ba1677b79e604bacd, 2022.

[29] Google - Android Open Source Project. Identifying Capacity-Related Jank. https://source.android.com/docs/core/debug/jank_capacity.

[30] Google - Android Open Source Project. Cgroup Abstraction Layer. https://source.android.com/docs/core/perf/cgroups.

[31] Google - Android Open Source Project. Cached Apps Freezer. https://source.android.com/docs/core/perf/cached-apps-freezer.

[32] Howard Oakley. How M1 Macs feel faster than Intel models: it's about QoS. https://eclecticlight.co/2021/05/17/how-m1-macs-feel-faster-than-intel-models-its-about-qos, 2021.

[33] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the Senseless Killing: Improving Memory Management for Mobile Operating Systems. In *2020 USENIX Annual Technical Conference (ATC)*, pages 873–887. USENIX Association, 2020.

[34] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems. In *2020 USENIX Annual Technical Conference (ATC)*, pages 897–910. USENIX Association, 2020.

[35] Google - Android Developers. App startup time. https://developer.android.com/topic/performance/vitals/launch-time.

[36] Google - Android Developers. Josh sees increased customer retention by improving app startup time by 30%. https://developer.android.com/stories/apps/josh.

[37] Google - Android Developers Blog. Improving App Performance with Baseline Profiles. https://android-developers.googleblog.com/2022/01/improving-app-performance-with-baseline.html, 2022.

[38] Google - Android Developers Blog. Improving app performance with ART optimizing profiles in the cloud. https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art.html, 2019.

[39] Google - Android Developers (Medium). Testing App Startup Performance. https://medium.com/androiddevelopers/testing-app-startup-performance-36169c27ee55, 2020.

[40] Google - Android Developers. Increasing app speed by 30%: a key ingredient in Zomato's growth recipe. https://developer.android.com/stories/apps/zomato.

[41] Google - Android Developers (Medium). Improving app startup with I/O prefetching. https://medium.com/androiddevelopers/improving-app-startup-with-i-o-prefetching-62fbdb9c9020, 2020.

[42] Samsung. Z-SSD. https://semiconductor.samsung.com/ssd/z-ssd.

[43] Forbes. Why Brands Are Fighting Over Milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds, 2016.

[44] NVIDIA. Analysing Stutter – Mining More from Percentiles. https://developer.nvidia.com/content/analysing-stutter-%E2%80%93-mining-more-percentiles-0, 2014.

[45] Engadget. Razer Phone hands-on. https://www.engadget.com/2017-11-01-razer-phone-hands-on.html, 2017.

[46] Samsung. Galaxy S20 Display Developers on What Makes the 120Hz Display Special. https://news.samsung.com/global/interview-galaxy-s20-display-developers-on-what-makes-the-120hz-display-special, 2020.

[47] Apple. Apple unveils iPhone 13 Pro and iPhone 13 Pro Max — more pro than ever before. https://www.apple.com/newsroom/2021/09/apple-unveils-iphone-13-pro-and-iphone-13-pro-max-more-pro-than-ever-before, 2021.

[48] ASUS. The ROG Phone 3 turns mobile gaming up to 144Hz. https://rog.asus.com/articles/product-news/the-rog-phone-3-turns-mobile-gaming-up-to-144hz, 2020.

[49] Google - Android Open Source Project. Identifying Jitter-Related Jank. https://source.android.com/docs/core/debug/jank_jitter.

[50] Johannes Weiner. PSI - Pressure Stall Information. https://docs.kernel.org/accounting/psi.html, 2018.

[51] Google - Android Open Source Project. Google Pixel 7 Pro device tree - device-panther.mk. https://android.googlesource.com/device/google/pantah/+/c9139250db92931907cd2bba1b5253846c389711, 2022.

[52] ASUS. ROG Phone 6 Pro - Tech Specs. https://rog.asus.com/phones/rog-phone-6-pro-model/spec, 2022.

[53] Yu Zhao. Multigenerational LRU Framework. https://lore.kernel.org/linux-mm/20220208081902.3550911-1-yuzhao@google.com, 2022.

[54] Google - Android Developers. Memory allocation among processes. https://developer.android.com/topic/performance/memory-management.

[55] Google - Google Play Apps & Games (Medium). Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2, 2017.

[56] Google - Android Open Source Project. ProcessList. https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-12.1.0_r27/services/core/java/com/android/server/am/ProcessList.java#188, 2022.

[57] Google - Android Developers. ActivityManager.AppTask. https://developer.android.com/reference/android/app/ActivityManager.AppTask.

[58] Google - Android Developers. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb.

[59] Daniel Rosenberg. f2fs: checkpoint disabling. https://lore.kernel.org/lkml/20180807234843.129387-1-drosen@google.com, 2018.