Fast Application Launch on Personal Computing/ Communication Devices

Junhee Ryu, *SK hynix;* Dongeun Lee, *Texas A&M University - Commerce;*
Kang G. Shin, *University of Michigan;* Kyungtae Kang, *Hanyang University*

https://www.usenix.org/conference/fast23/presentation/ryu

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

Fast Application Launch on Personal Computing/Communication Devices

Junhee Ryu[1], Dongeun Lee[2], Kang G. Shin[3], and Kyungtae Kang[4]
[1]*SK hynix,* [2]*Texas A&M University - Commerce,* [3]*University of Michigan,* [4]*Hanyang University*

Abstract

We present Paralfetch, a novel prefetcher to speed up app launches on personal computing/communication devices by: 1) accurate collection of launch-related disk read requests, 2) pre-scheduling of these requests to improve I/O throughput during prefetching, and 3) overlapping app execution with disk prefetching for hiding disk access time from the app execution. We have implemented Paralfetch under Linux kernels on a desktop/laptop PC, a Raspberry Pi 3 board, and an Android smartphone. Tests with popular apps show that Paralfetch significantly reduces app launch times on flash-based drives, and outperforms *GSoC Prefetch* and FAST, which are representative app prefetchers available for Linux-based systems.

1 Introduction

Quick app launches are of great importance to user experience on personal computing/communication devices such as laptop/tablet PCs, single-board computers, and smartphones [17,18,22,24,26,34]. The latency of launching an app mainly depends on the performance of the underlying CPU and flash-based disks. Despite continuing improvements in the performance of these components, the launch latencies, especially of large apps and games, still remain an important problem for three reasons.

First, the performance of flash storage does not always meet users' expectations/desire. For example, it has been predicted [53] that in 2025 around 50% of the data on flash will be stored in QLC (quad-level cell) flash, which has 2.1× slower read and 5.7× slower write times than TLC (triple-level cell) flash [4]. The use of affordable QLC SSDs was found to extend the launch latency of the popular Blade and Soul game from 91s to 114s [46], and that of Horizon Zero Dawn from 15.7s to 21.4s [47], compared to high-end SSDs. Many Windows apps take a similar amount of time [48] to launch from the Samsung QLC SSD as they do from the Intel X25-M G2 SSD, which was released in 2009. Furthermore, recent entry-class SSDs widely adopt DRAM-less architecture [35], which leads to additional flash accesses for translating logical-to-physical addresses. A Raspberry Pi is also widely used to run desktop applications [57], but it only supports the sluggish MicroSD.

Second, the complexity of apps is continuously growing due to the addition of new features and functionality to software [50]. Unfortunately, complex software also requires higher-level programming languages and libraries, generating slower code, thus extending their launch latencies [54].

Third, although parallelism is effectively utilized in modern multicore CPUs and solid-state disks [8], app launches can seldom exploit existing sources of parallelism. It has also been shown [25] that CPUs and disks are seldom utilized simultaneously during a launch because synchronous disk reads are dominant. Making better use of parallelism is, therefore, a major consideration in the design of app prefetchers [24].

Launch latencies depend on the previous state of the system, especially the disk cache. A *cold start* occurs when the disk cache does not hold any data required by the app, either because it is the first time the app has been launched, or because all of the app's data has been evicted since its last run. A *system cold start* is a special case of cold start, which occurs when no user-launched app is already running. A *warm start* occurs when the app being launched has been running recently, so the disk cache still holds all, or most, of the data that it needs. A warm start is much faster than a cold start, because no, or very little, data has to be fetched from the disk. This avoids the concomitant file system operations, thus saving CPU time as well as disk time.

An app prefetcher [6, 7, 9, 11, 28, 36, 40] can reduce the time needed for a cold start: during *learning phase*, which corresponds to the first launch of an app, the prefetcher collects launch-related blocks and/or their access sequences (the term *launch sequence* is used interchangeably). This is usually achieved by monitoring disk reads and/or page faults. A *prefetching phase* occurs during subsequent launches of the app, in which case this launch sequence is used for disk prefetching to accelerate loading.

Different prefetching strategies are required for the different seek characteristics of mechanical and flash disks. These storage devices have different performance bottlenecks which have been addressed in well-known ways. *Threaded prefetching* is designed for SSDs. A dedicated thread is used to prefetch blocks in the order of their collection during monitoring. The prefetching thread runs concurrently with the app, reducing the launch time. On the other hand, *Sorted prefetching* is designed for HDDs. Data is read from the disk in logical block address (LBA) order to reduce seek times [5, 19, 20], which account for most of the launch time. Sorted prefetching is not done concurrently with the app because the app's disk I/O would disrupt prefetching in the LBA sequence.

In this paper, we define three fundamental challenges in reaping the potential speed-up with an app prefetcher, and then explain how `Paralfetch` addresses these issues that previous approaches fail to achieve. Overall, this paper makes the following main contributions:

- **Accurate tracking of launch-related blocks (§3.1):** Most monitoring methods fail to locate a significant number of blocks during the learning phase [23]. In threaded prefetching on SSDs, an access tracer should collect not only accessed blocks but their access order. To do this, a viable solution is to monitor at the disk I/O level after performing the invalidation of unused entries in the disk cache. Unfortunately, metadata and data blocks would not be detected by imperfect OS-level disk cache invalidation. To address this problem, `Paralfetch` introduces a file-system-level block dependency check and low-overhead page-fault monitoring.
- **Pre-scheduling of these blocks to increase prefetch throughput (§3.2):** Although the I/O involved in prefetching frequently becomes a bottleneck in threaded prefetching on commodity SSDs, prior work does not address this issue. We observe I/O dependencies between prefetch blocks to significantly hinder the asynchrony of I/O requests, reducing prefetch throughput. We address this problem with a new I/O reordering method called *metadata shift* that places more I/O requests between dependent I/O requests, issuing more I/O requests asynchronously. A *range merge* is also introduced to combine nearby I/O requests into one large request, improving I/O throughput.
- **Tailored overlapping of application execution with prefetching (§3.3)**: We find that aggressive prefetching with excessive pre-scheduling can actually increase launch latencies because of I/O contention between the app and prefetching threads. Modern SSDs' reordering of outstanding I/O operations can aggravate this contention [41]. We vary the amount of I/O optimization in response to a prefetching bottleneck. This avoids the I/O contention caused by an excessive optimization, and thus helps `Paralfetch` find a better optimization level.
- **Implementation (§4) and evaluation (§5) of `Paralfetch`:** We evaluate `Paralfetch` in the launch of common apps on a laptop PC, a Raspberry Pi 3, and an Android smartphone. With the aforementioned features, `Paralfetch` achieves launch performance close to the warm start: On a PC, `Paralfetch` reduced the average system cold start time (favoring competitors) of 16 benchmark apps by 48.0%, this number corresponds to 11% and 22% further reductions from `FAST` and GSoC Prefetch, respectively. `Paralfetch` also reduced the average app launch time on a Raspberry Pi 3 by 31%, and on an Android phone by 11%. `Paralfetch` is publicly available[1]

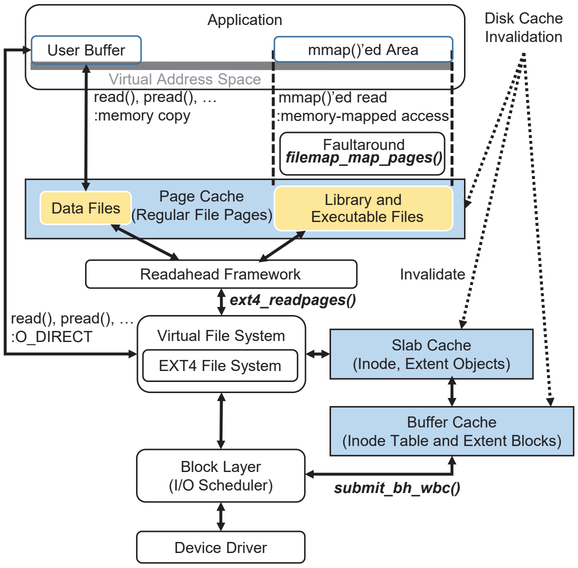

Application
User Buffer
mmap()'ed Area
Virtual Address Space
Disk Cache Invalidation
read(), pread(), … :memory copy
mmap()'ed read :memory-mapped access
Faultaround
filemap_map_pages()
Data Files
Page Cache (Regular File Pages)
Library and Executable Files
Readahead Framework
Invalidate
ext4_readpages()
read(), pread(), … :O_DIRECT
Virtual File System
EXT4 File System
Slab Cache (Inode, Extent Objects)
Buffer Cache (Inode Table and Extent Blocks)
Block Layer (I/O Scheduler)
submit_bh_wbc()
Device Driver

Figure 1: I/O Stack in Linux. Linux includes three disk caches: page cache for regular files, slab (or slub) cache for metadata objects, and buffer cache for metadata blocks. The slab is used as an object-granular metadata cache for buffer cache. `read` system call explicitly fills page cache based on its arguments, while page cache for `mmap`ed files is populated through page fault mechanism. Readahead framework is responsible for filling the contents of page cache, and it determines how many blocks to be prefetched based on the access sequentiality. Note that metadata blocks can be prefetched by EXT4 file system.

2 Background and Motivation

2.1 Targets of `Paralfetch`

Linux-based systems using EXT4 file system. We implemented and tested a `Paralfetch` prototype on EXT4 file system on a laptop with SSD, a Raspberry Pi 3 with microSD card, and a Pixel smartphone with universal flash storage (UFS).

Large apps with highly deterministic I/O. Other applications do not benefit much from `Paralfetch`: I/O requests from text-based apps such as `cp`, `gcc` and `find` largely depend on input parameters that can change with every launch; and apps such as `pwd` and `ssh` are too small to amortize prefetch overhead, and are usually warm started.

2.2 Disk Caching in Linux

Figure 1 provides a summary of the Linux I/O stack from disk caching perspectives.

Page cache and buffer cache. The Linux kernel provides two cache mechanisms for disk blocks in terms of API and unit size [15]: The *page cache* holds file pages, whereas the *buffer cache* contains data blocks corresponding to block devices. The contents and lookup spaces of these caches are managed using a radix tree for each regular file or block device file.

[1] https://github.com/optios/paralfetch

In EXT4 file system, blocks of data from regular files are cached in the page cache, while the buffer cache is used for caching metadata blocks (*e.g.*, inode table blocks, directory blocks, and extent blocks). The contents of regular files can be prefetched using a combination of *device number*, *inode number*, *offset*, and *size*. On the other hand, metadata blocks can be prefetched using a combination of *device number* and *block number*. It should be noted that there are no prefetching-level dependencies among buffer-cached (metadata) blocks, whereas I/O requests for page-cached (data) blocks are delayed until relevant metadata blocks are cached.

Slab for caching file system metadata at object granularity. Metadata objects in EXT4 file system, namely, the inode, directory entry, and extent, are smaller than a file system block but must nevertheless be managed individually so that important objects are kept in memory, even when the memory is under pressure. Therefore, the Linux slab object allocator caches these objects without reference to the contexts of the buffer cache. Thus an `inode` can be simultaneously stored in both the slab and buffer caches.

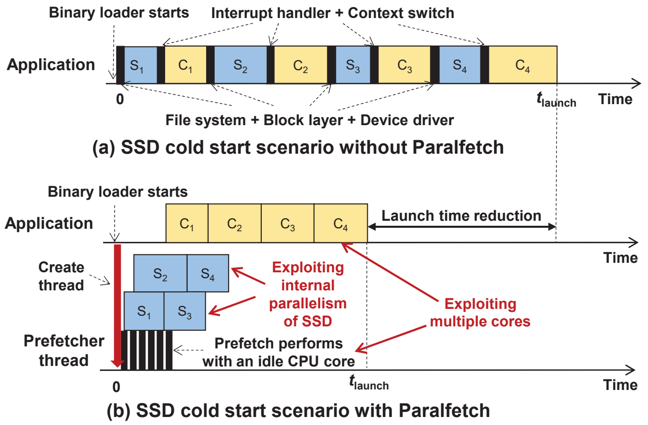
Page cache accessing methods. A process can copy the contents of the page cache into a user buffer using a `read` or a file-related syscall. Alternatively, a process can map the extent of a file to its virtual address space using the `mmap` syscall. In the latter case, attempting to access an unmapped address in the page table causes a page fault. To reduce the number of page faults, Linux employs an interesting feature, called *faultaround* [49], which pre-faults a 64KB-aligned chunk of the address space around the fault address.

Disk cache invalidation. The Linux kernel provides functions to invalidate disk caches. A user or process with root permission can invalidate these caches by writing a predefined value ("1" for the page and buffer caches, "2" for the slab cache, and "3" for all these) into the `/proc/sys/vm/drop_caches` proc file. This method can only invalidate unused entries with zero reference counts.

2.3 Representative App Prefetchers

Windows prefetcher [37]**.** Since XP, Windows has included a prefetcher for launch and system boot. The *Windows prefetcher* is customized for HDDs, but it can also be used with SSDs, although user configuration is required to make best use of more capable SSDs. In its learning phase, the copies of file-backed memory pages which are required by an application are identified by the Windows working-set manager. The generated information, which is file-level data, determines the disk blocks to be prefetched during subsequent application launches. By defragmenting these blocks to make their file-level prefetch blocks correspond to their LBA order, the Windows prefetcher optimizes the disk head movements of HDD. This time-consuming process is scheduled to happen every three days.

GSoC Prefetch [29], which was selected for the Google Summer of Code 2007, is a Linux-based prefetcher for HDDs. It obtains launch-related block information in its learning phase by first clearing the bit in every OS-managed page descriptor (not page table) which indicates that the page has been referenced. After a predefined monitoring time (10 seconds by default), GSoC Prefetch traces those referenced pages with 'referenced' bits on. It then extracts a file identifier (device number, inode number, and offset) from each of the traced pages. Next, GSoC Prefetch sorts the pages based on these identifiers and stores the sorted pages in a file. On subsequent launches, launch-related blocks are prefetched in the order recorded in that file. This reduces both seek and rotational latencies in HDDs. GSoC Prefetch has a defragmentation tool similar to that in the Windows prefetcher.



(a) SSD cold start scenario without Paralfetch

(b) SSD cold start scenario with Paralfetch

Figure 2: SSD cold start scenarios with and without Paralfetch. S_i is the i^{th} block requested from the SSD, and C_i is the corresponding CPU computation. Paralfetch expedites an application launch by exploiting parallelism of each resource (i.e., multicore activation and internal parallelism on SSDs) and utilizing these resources concurrently.

FAST [24] is a recent Linux-based prefetcher for SSDs. It starts by clearing the slab, buffer, and page caches. Then, `FAST` begins its learning phase, during which it creates a prefetch program by monitoring the LBAs of blocks using the `blktrace` tool and converting them to prefetchable system calls with arguments. On subsequent launches, `FAST` executes this prefetch program at the same time as the application. Disk blocks are prefetched in order without any I/O optimization.

2.4 Cold Start with Paralfetch

Figure 2a shows a cold start scenario without `Paralfetch`, and Figure 2b shows the same scenario in which `Paralfetch` runs the application concurrently with a prefetch thread. The computations run on multiple CPU cores, in parallel with the SSD accesses, which are issued in a way that exploits the internal parallelism of the SSD. This is effected by issuing concurrent asynchronous I/O requests using the command queuing (CQ) feature. If an SSD does not support CQ, `Paralfetch` merges I/O requests, which have consecutive LBAs and are close in the block access sequence, so as to promote internal parallelism.

Table 1: Metadata and data block requests required to launch applications with missing metadata blocks. Note that ‘regular’ files include `mmap`ed files, and that files `mmap`ed by running applications are not subject to disk cache invalidation. The last column shows the number of I/O operations that were not captured by Paralfetch, which varies from run to run.

		Read requests traced by `Paralfetch`		Number of missing	Number of accessed files		
	Application	Metadata accesses	File data accesses	metadata blocks	regular	`mmap`ed	Number of
		(total size in KB)	(total size in KB)	detected	files	files	missing I/Os
Ubuntu Linux (Laptop PC)	Android Studio	1,330 (6,844)	3,845 (197,932)	58	954	10	38
	Chromium Browser	612 (3,048)	1,135 (130,728)	37	629	108	34
	Eclipse	565 (3,348)	1,669 (67,256)	28	744	328	49
	GIMP	489 (2,620)	1,026 (38,512)	20	975	474	28
	LibreOffice Impress	590 (2,900)	706 (83,004)	37	438	232	32
	LibreOffice Writer	552 (2,800)	729 (83,824)	25	476	227	33
	Okular	1,093 (5,720)	426 (23,640)	41	349	238	36
	Scribus	840 (5,984)	1,560 (141,056)	35	1,230	682	21
	VLC Player	682 (5,420)	444 (20,192)	41	375	104	32
	Xilinx ISE	573 (3,024)	1,028 (176,504)	42	657	273	33
Raspbian OS (Raspberry Pi 3)	Chromium Browser	496 (1,984)	2,017 (138,600)	40	473	68	41
	Frozen Bubble	605 (2,420)	3,769 (32,992)	25	3,425	26	12
	GIMP	618 (2,472)	1,863 (46,664)	38	991	296	47
	LibreOffice Writer	596 (2,384)	911 (35,164)	33	395	154	36
	Scratch 2	332 (1,328)	839 (48,580)	40	294	73	19
	Xpdf	127 (508)	169 (7,236)	15	75	21	11
	0 A.D.	206 (509)	669 (86,272)	19	162	139	21
Android 8.0 (Google Pixel XL)	Asphalt 8	131 (988)	838 (217,240)	49	179	N/A	11
	Dragon Quest 8	95 (852)	4,339 (333,812)	46	335	N/A	12
	FIFA 16 UT	76 (772)	805 (166,120)	39	265	N/A	47
	GTA SA	104 (560)	377 (82,928)	41	95	N/A	36
	Truck Pro	96 (792)	1,792 (115,732)	41	175	N/A	19
	Devil May Cry	237 (1,728)	1,904 (316,004)	45	407	N/A	19
	The War of Mine	127 (696)	517 (128,300)	43	101	N/A	11

3 Paralfetch Design and Preliminary Results

3.1 Accurate Tracing

The benefit from an application prefetching is limited by the tracing accuracy with which launch-related blocks are traced. In particular, accurate tracing is essential to prevent a launching application’s wait for missing blocks from disk when several concurrent threads are causing lots of I/O contention. Note that the threaded prefetching can marginally benefit from Windows prefetcher and GSoC Prefetch which cannot trace the block access sequence because they rely on a snapshot of the working set or of the referenced pages after a launch.

There are also issues with the tracing method used by GSoC Prefetch: it only traces pages for regular files, and missing metadata limits the benefit of prefetching; a significant number of pages are also accessed more than once during a launch. This latter issue is particularly problematic because, when a page with the ‘referenced’ bit set on is accessed for the second time, Linux OS turns off the ‘referenced’ bit and promotes the page from the inactive list to the active list. As a result, some pages are never traced. In the case of Eclipse, we found 2,782 file-backed pages not traced.

Potentially, the highest accuracy would be achieved by monitoring page faults and data accesses at all disk caching layers (*e.g.*, slab, buffer, and page caches). But such exhaustive tracing would produce significantly more data than I/O-level monitoring (37× during an Eclipse launch), incurring unacceptable memory and computation overheads. Furthermore, a log of I/O operations obtained by monitoring disk cache accesses is likely to include many useless cached entries created by I/O operations of background tasks.

This issue is successfully mitigated by monitoring I/O requests: In the learning phase, `Paralfetch` invalidates unused entries in the disk cache, so that `Paralfetch` collects a proper set of blocks for subsequent launches of the application. It then records I/O requests for blocks not found in these caches by instrumenting file system functions with I/O logging codes, and these requests are used to prefetch those additional blocks during launches. In this paper, we use the term *log entry* to refer to a log of I/O request collected during a launch, while the term *prefetch entry* refers to an entry used for prefetching disk blocks. The latter includes arguments for prefetching function calls.

Unfortunately, as mentioned earlier, the invalidation of disk caches (slab, buffer, and page caches) is not perfect because only unused entries can be invalidated; a working set of blocks for running applications is always retained. This issue has been overlooked in previous schemes (including `FAST`), i.e., their evaluation was restricted to system cold start scenarios. Table 1 classifies traced blocks with `Paralfetch`. Note that metadata blocks and `mmap`ed file blocks are potential missing blocks when using `FAST`. Since usually many user and system processes run in the background, this issue can significantly degrade tracing accuracy. For example, 225 files of this kind were accessed by both LibreOffice Impress and LibreOffice Writer (on a laptop) during a launch of either. Thus, an attempt to trace launch blocks for LibreOffice Writer just after LibreOffice Impress launched (and started running in the background) returns only 700 log entries (27,688 KB) compared to 1,281 log entries (83,824 KB) during a system cold start. We conducted further experiments by substituting Android Studio, Chromium Browser, Eclipse, and GIMP for LibreOffice Impress. Surprisingly, imperfect cache invalidation still resulted in many missing data and associated metadata blocks: 5.0%, 12.0%, 14.4%, and 6.6% of the total in each case. The launch time impact of missing blocks is significant as shown in §5.2.

We have therefore developed two methods to detect missing metadata and data blocks.

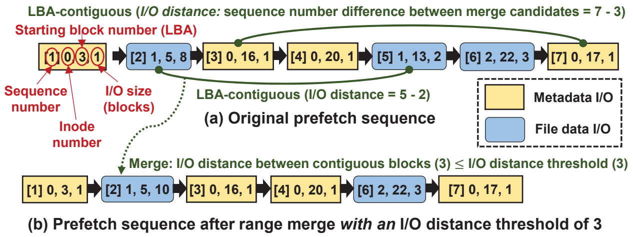
1) Finding missing metadata blocks. We first introduce a file system-level dependency check, called *missing metadata block detection*, which identifies launch-related metadata blocks (*i.e.*, inode and extent blocks) that have not been traced due to the imperfect invalidation of the slab and buffer cache, but nevertheless share a dependency with traced data blocks. To address this issue, `Paralfetch` implements a function (§4.2) that tracks associated metadata blocks for each log entry for a regular file. Table 1 shows that 15 – 58 missing metadata blocks were found during launches, and these numbers vary with the number of irreclaimable entries in the disk caches under use by running applications. When these missing blocks are found, `Paralfetch` inserts new log entries for them just before other log entries of associated data blocks.

2) Page fault monitoring. Page cache invalidation is also imperfect because file-backed pages which are dirty, under writeback, or accessed through `mmap`, are not invalidated. To trace pages which are dirty or under writeback, `Paralfetch` flushes them out via a `sync` operation before the disk cache is cleared. However, pages accessed through `mmap`, such as shared library files, are more challenging. When these are shared with running applications, tracing accuracy is compromised. To address this issue, we arranged for `Paralfetch` to trace previously untraced blocks accessed through `mmap` calls by instrumenting the faultaround [49] handler with page fault tracing code. The handler proactively maps 16 boundary-aligned (page-cached) pages around the page-faulted address.

3.2 Prefetch Scheduling

Upon completion of collection of disk I/O requests during an application launch, `Paralfetch` pre-schedules these requests to speed up the prefetching phase, *merging* and *reordering* requests so as to exploit the internal parallelism of an SSD.

Range merging. Merging small I/O requests into a single large request enhances the throughput of an SSD [12, 27, 32, 43]. Figure 3b shows a range merge in which two requests for blocks with consecutive LBAs that are within a predefined *I/O distance* threshold are combined where the I/O distance is defined as difference in the locations of blocks in the launch sequence. This threshold prevents the merging of far-apart log entries in the launch sequence, as they can hinder timely prefetching of subsequent blocks. Overly-aggressive merging can be bad especially for applications with CPU-bound launches, in which I/O optimization is less influential in meeting prefetching deadlines. Figure 4 shows plots of prefetch time against the I/O distance threshold on SSD, UFS flash, and MicroSD. The performance gain from range merging tails off as the threshold increases mainly because EXT4 tries to locate metadata and data blocks for related files close together in terms of LBA.


LBA-contiguous (I/O distance: sequence number difference between merge candidates = 7 - 3)
Starting block number (LBA)
[1] 0, 3, 1
[2] 1, 5, 8
[3] 0, 16, 1
[4] 0, 20, 1
[5] 1, 13, 2
[6] 2, 22, 3
[7] 0, 17, 1
Sequence number
Inode number
I/O size (blocks)
LBA-contiguous (I/O distance = 5 - 2)
Metadata I/O
File data I/O
Merge: I/O distance between contiguous blocks (3) ≤ I/O distance threshold (3)
[1] 0, 3, 1
[2] 1, 5, 10
[3] 0, 16, 1
[4] 0, 20, 1
[6] 2, 22, 3
[7] 0, 17, 1

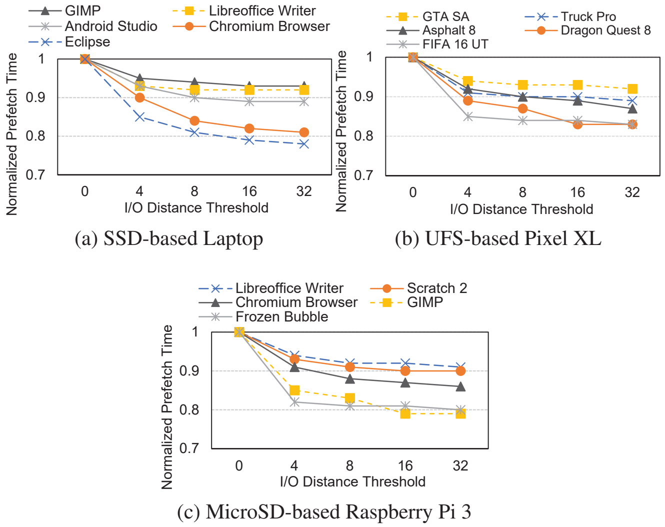
(a) Original prefetch sequence

(b) Prefetch sequence after range merge *with an* I/O distance threshold of 3

Figure 3: Range merge. Merging nearby I/O operations into a single large operation improves throughput while keeping changes to the I/O order within a predefined limit so that the target application and prefetch thread can run concurrently. Range merge combines LBA-contiguous I/O requests of the same type (*e.g.*, metadata or data block) into the preceding one.

Metadata shifting. Every file system has its own particular I/O dependencies for prefetching between metadata and data blocks (and between metadata blocks). In EXT4, a request for a data block can only be issued after the associated metadata block, which contains the LBA of that data block, has been read. The metadata for a data block is often requested just before the corresponding data block.

Thus this dependency tends to limit the number of commands that can be queued, and this in turn limits the effectiveness of command queuing, which yields maximum benefit when there are many commands in the queue which can potentially be executed in parallel [39].



(a) SSD-based Laptop

(b) UFS-based Pixel XL

(c) MicroSD-based Raspberry Pi 3

Figure 4: Normalized prefetching times with varying I/O distance thresholds.

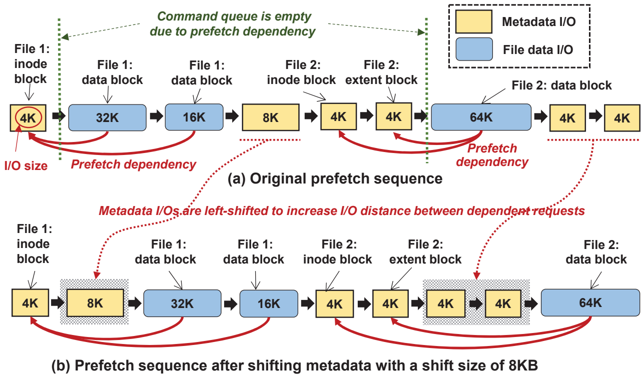


Figure 5: Metadata shifting to boost the outstanding I/O size in the command queue of an SSD controller. An I/O request for data blocks should wait for the associated metadata blocks to be read. By left-shifting I/O requests for metadata, more I/O requests can be issued asynchronously. The shift size controls the extent to which metadata blocks can be left-shifted.

This issue can be addressed by bringing forward requests for metadata blocks. This is facilitated in EXT4, where there are no read dependencies among buffer-cached (metadata) blocks, while I/O requests for page-cached data blocks can only be issued after associated metadata blocks are buffer-cached. Figure 5a shows the processing of an example prefetch thread, in which dependencies on metadata blocks cause the command queue to become empty on two occasions. Figure 5b shows how Paralfetch brings forward metadata block requests in the prefetch thread to increase the interval between requests for dependent blocks. Figure 6a shows that the average prefetching time on a CQ-enabled SSD was reduced by 21.6% through shifting metadata requests forward by 128 KB, when combined with the tracing of missing metadata blocks.

An SSD without CQ support can also benefit from shifted metadata (Figure 6c): requests to the I/O scheduler can be issued in advance, so that the storage driver receives a request earlier from the I/O scheduler queue, rather than later by the application; and an MMC/SD driver (for eMMC flash and SD cards) overlaps flash access for the current I/O request with DMA preparation for the next I/O request. A metadata shift of 4 KB reduced prefetch times by 19.3% on the Raspberry Pi 3 using a MicroSD.

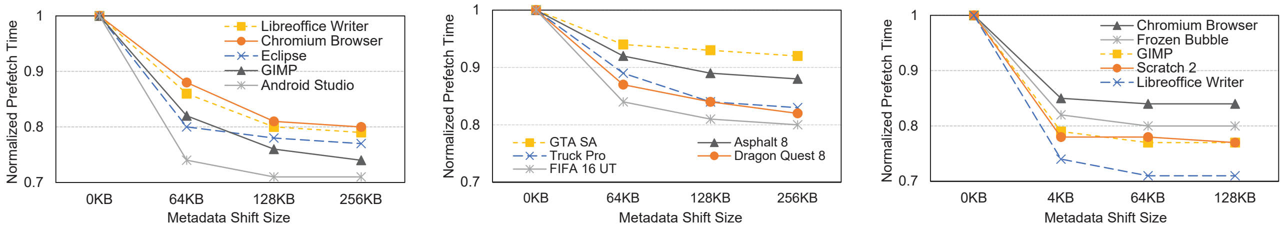
Correctness. The read requests from the prefetch thread go through disk caches, and hence reordering and merging of a launch sequence have no implications on correctness. Even if a prefetch entry is outdated, it only affects the launch performance.

3.3 Parallelized Execution: Overlapping Application Execution with Disk Prefetching

Timely prefetching can better overlap application execution with prefetching. Reordering or merging blocks far apart could improve prefetch throughtput but could also hinder timely prefetching. Experimental results in Figures 7 and 8 substantiate the claim by showing prefetching throughput does not always correspond to launch performance. Paralfetch avoids this pitfall by tailoring metadata shift and range merge dynamically. A challenge is how to find near-optimal threshold values in an automatic manner. To address this, Paralfetch employs *dynamic scheduling* which reschedules prefetch entries with an increased I/O distance threshold and/or metadata shift size when a prefetching bottleneck is detected.

The ability of shifting metadata and merging nearby requests to reduce prefetching time on SSD-based systems is limited by contentions between I/O requests from the prefetch thread and I/O requests which must be issued by the application because they were omitted from the prefetch thread. As shown in Table 1, we found that an average of 2.8% of requested blocks were not traced despite the improved tracing features of Paralfetch. These missing blocks are inevitably requested by the application, which waits until the blocks are loaded from the disk. Contention between the application and the prefetch thread becomes critical when there are too many I/O requests in the I/O scheduler or command queue [13] in an SSD. This can occur when metadata blocks are shifted too far, or when an oversize I/O request is created by range merging with a large threshold. From an experiment with Eclipse, we found that the effect of missing blocks on latency was increased by 3.2× and 8.7× when the largest allowable shifts were 128KB and 256KB, respectively.

To avoid the need to optimize the thresholds for metadata shifting and range merge over a number of trial runs, Paralfetch gradually increases the threshold if prefetching

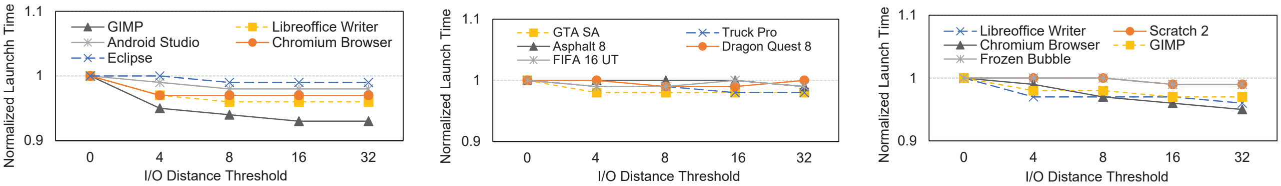


(a) SSD-based Laptop

(b) UFS-based Pixel XL

(c) MicroSD-based Raspberry Pi 3

Figure 6: Normalized prefetching times for different metadata shift sizes.

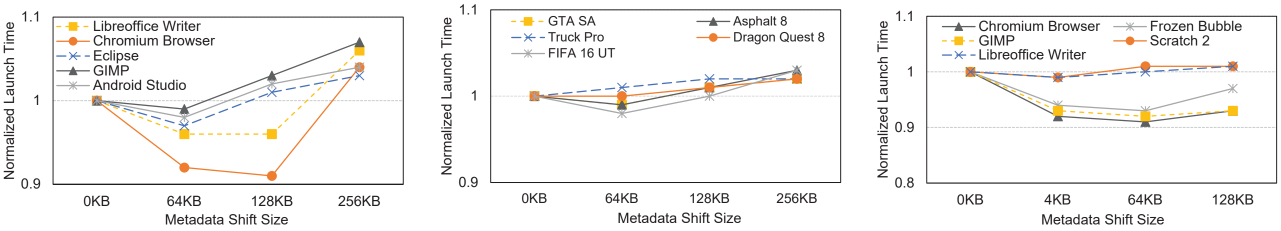


(a) SSD-based Laptop

(b) UFS-based Pixel XL

(c) MicroSD-based Raspberry Pi 3

Figure 7: Normalized launch times with varying I/O distance thresholds.



(a) SSD-based Laptop

(b) UFS-based Pixel XL

(c) MicroSD-based Raspberry Pi 3

Figure 8: Normalized launch times for different metadata shift sizes.

Table 2: Default configuration for prefetch optimization.

	SSD without CQ feature	SSD with CQ feature
I/O distance threshold for range merging	Starts at 8 and can be increased	8
Metadata shift size (KB) for metadta shifting	4	Starts at 64 and can be increased

is not effective. Next, we describe how to control the extent of dynamic scheduling and how to measure the effectiveness of prefetching.

Optimizing prefetch entries with dynamic scheduling. Initially, Paralfetch uses default thresholds for metadata shifting and range merge shown in Table 2. It subsequently increases the threshold for only one of these methods, depending on the availability of CQ support. The metadata shifting threshold is increased in increments of 16KB and the I/O distance threshold in increments of 4.

The best combination of scheduling methods depends on the type of disk. For example, on a CQ-supported SSD, range merge gains little beyond a threshold of 8, which can, therefore, be used as a default during the learning phase. Similarly, metadata shifting yields little benefit on MicroSD-based devices without CQ support beyond a threshold of 4KB.

Detecting prefetch bottleneck. An application experiences more context switches when it has to wait for the blocks requested by the prefetch thread, implying that the prefetch thread is not prefetching in time. Specifically, the prefetch thread collects the number of context switches made by the launching application during the prefetching period. Paralfetch ends dynamic scheduling if the quantity of context switches is below a user-defined threshold (by default, 5% of the number of prefetch entries). The overall disk read size is checked by Paralfetch in order to remove the results from the warm cache.

4 Implementation of Paralfetch

This section details the workflow of Paralfetch and the interaction among its main components described in Figure 9.

4.1 Launch Phase Management

Native Linux: The next launch type for each application is determined by reading the 9-th byte of the header of its executable and linkable format (ELF) binary file. This byte (referred to as the *phase byte*) is normally used for memory alignment (padding), and has a default value of 0. It is set to PHASE_LEARNING (3) for a learning phase and PHASE_THREADED_PREFETCHING (1) for a prefetching phase. A user can also set this value to PHASE_DISABLE (9) to disable prefetching altogether, for small applications or utilities that frequently experience warm starts. The phase byte is passed to the ELF binary loader (load_elf_binary).

Paralfetch supports two modes for launch phase management. In manual mode, a user explicitly selects applications that will use Paralfetch, by calling pfsetmode, which takes a value for the phase byte and an ELF binary path as arguments. pfsetmode can be also invoked from a desktop icon (*i.e.*, mouse right-click menu). In contrast, Paralfetch is applied to all installed applications in automatic mode, which is similar to the management method used in FAST.

Android: zygote is a process that creates a native Android application in Java by forking and loading the main class of a program [30]. zygote invokes the handleChildProc method to create and run a new Android application. To re-

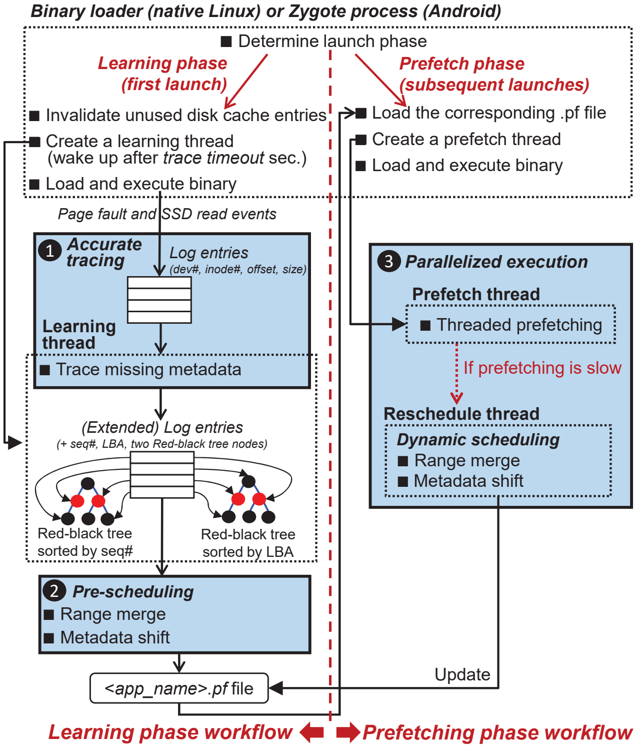

Binary loader (native Linux) or Zygote process (Android)
Determine launch phase
Learning phase (first launch)
Prefetch phase (subsequent launches)
Invalidate unused disk cache entries
Create a learning thread (wake up after trace timeout sec.)
Load and execute binary
Load the corresponding .pf file
Create a prefetch thread
Load and execute binary
Page fault and SSD read events
1 Accurate tracing
Log entries (dev#, inode#, offset, size)
Learning thread
Trace missing metadata
(Extended) Log entries (+ seq#, LBA, two Red-black tree nodes)
Red-black tree sorted by seq#
Red-black tree sorted by LBA
2 Pre-scheduling
Range merge
Metadata shift
<app_name>.pf file
3 Parallelized execution
Prefetch thread
Threaded prefetching
If prefetching is slow
Reschedule thread
Dynamic scheduling
Range merge
Metadata shift
Update
Learning phase workflow
Prefetching phase workflow

Figure 9: Paralfetch workflow. Boxes with dotted edges denote threads, and boxes with solid edges identify the three major components of Paralfetch. During a learning phase, `Paralfetch` records an I/O log as a form of log entry. Upon the completion of the launch, collected log entries are passed to missing metadata detector, generating additional log entries for missing metadata. Then, the log entries are passed to pre-scheduling functions as a form of red-black tree. The details of pre-scheduling are described in Algorithm 1 and 2.

duce launch times, `zygote` preloads classes and resource files used by many applications, quickly creating a process which shares these preloaded classes. Unlike native Linux processes, a native Android process remains in the background even after a user quits the application, and can be resumed by moving the process to the foreground (the resuming procedure). However, when free memory is in short supply, Android wakes up the low memory killer (LMK) to reclaim memory space by removing less important processes completely.

To interface `Paralfetch` with the Android platform, we created a file named `fetch_app` using `sysfs`, which provides a communication interface between the Linux kernel and a user process. On Android, `Paralfetch` uses automatic launch management mode, in which `Paralfetch` tailors each launch to the type of application. When the main class name of an application is written to the `fetch_app` file, `Paralfetch` determines how to perform the launch phase based on the following rules: if there is no corresponding `<class_name>.pf` file[2] in the `/persist/paralfetch` directory, then `Paralfetch` starts a learning phase for that application; but if the file exists, then `Paralfetch` performs prefetching. To implement this, we augmented the `handleChildProc` method to write the main class name of the application being launched to the `fetch_app` file. `Paralfetch` does not begin a prefetching for the resuming procedure that does not invoke `handleChildProc`.

4.2 Learning Phase

I/O logging. To collect blocks required for a launch, `Paralfetch` first invalidates unused entries in the slab (for file system objects), buffer and page caches, and temporarily disables the inode read-ahead functionality of EXT4 so as to prevent I/O contention resulting from unnecessary inode blocks being read during the prefetching phase. Next, `Paralfetch` sets a trace timeout, with the default value of 30 seconds, and also sets the `trace_flag` to true to activate logging. Then, `Paralfetch` resumes loading and execution of the application. During the execution, the I/O requests for buffer-cached blocks caused by disk cache misses are logged by code introduced into the metadata access function (`submit_bh_wbc`). Similarly, code introduced into the functions `ext4_readpage`, `ext4_readpages`, and `filemap_map_pages` logs read requests associated with page-cached blocks.

Page fault monitoring. The `filemap_map_pages` function is called by the OS when a page fault occurs. It pre-faults the 16 boundary-aligned pages which contain the faulting page, provided that these pages are in the page cache [49]. Performing this reduces the overhead of tracing page faults.

Tracing missing metadata blocks. Block tracing ends when the trace times out, and the launch is deemed to be complete when fewer than 10 block read requests occur in a second [25]. We refer to the corresponding block of an application as the *completion block*. To detect missing metadata blocks, we implemented the `ext4_fiedep` function, a variant of the `ext4_fiemap` function that must in any case access the metadata blocks associated with file blocks during the mapping of logical-to-physical extents. Unlike the original version that returns file extents for arguments (*i.e.*, a file and query range of the file), the `ext4_fiedep` function returns a list of associated metadata blocks along with file extents.

As shown in Figure 9, `Paralfetch` builds two red-black binary search trees for log entries that are used for prefetch scheduling: `Paralfetch` reads log entries in their access order and inserts each of them to the trees. It invokes the `ext4_fiedep` function for each log entry for a regular file. If the corresponding metadata blocks are missing from the tree, `Paralfetch` allocates and inserts new log entries for them right before the entry for the corresponding data blocks.

This operation consumes little CPU time (17 ms for Android Studio) and incurs no disk I/Os because the procedure runs in the warm cache condition (*i.e.*, after the completion of a launch process).

[2] `<class_name>.pf` file is equivalent to `<app_name>.pf` in native Linux.

Algorithm 1: Metadata Shift Procedure

```
Input: Log entries sorted by their access order ($rbtree\_seq$),
       Metadata shift size ($ms\_size$)
Result: Metadata-shifted log entries (accessed via $rbtree\_seq$)
1  $log \leftarrow$ first_log_entry($rbtree\_seq$)
2  $out\_meta\_size \leftarrow 0$
3  while $log \neq NULL$ do
4      if $is\_metadata\_log\_entry(log)$ then
5          move_to_MS_queue($log$)
6          $out\_meta\_size \leftarrow out\_meta\_size + log.size$
           /* expired entries ($log.expire$ <= $out\_meta\_size$)
              in wait queue are moved to MS queue      */
7          move_expired_wait_queue_entries_to_MS_queue()
8      else
9          $log.expire = out\_meta\_size + ms\_size$
10         move_to_wait_queue($log$)
11     $log \leftarrow$ next_log_entry_seq($log$)
12 drain_wait_queue_entries_to_ms_queue()
13 rebuild_rbtree_seq_to_correspond_to_ms_queue_order()
```

Algorithm 2: Range Merge Procedure

```
Input: Log entries sorted by their LBA ($rbtree\_lba$) and access
       order ($rbtree\_seq$), IO distance threshold ($dist\_thr$)
Result: Range-merged log entries (accessed via $rbtree\_seq$)
1  $curr \leftarrow$ first_log_entry($rbtree\_lba$)
2  $next \leftarrow$ next_log_entry_lba($curr$)
3  while $next \neq NULL$ do
4      if $curr.inode\_num = next.inode\_num$ &
5      $curr.start\_lba + curr.size = next.start\_lba$ &
6      $next.seq\_num - curr.seq\_num <= dist\_thr$ then
7          $curr.size \leftarrow curr.size + next.size$
8          unlink_log_entry_from_rbtrees_lba_and_seq($next$)
9          remove_log_entry($next$)
10         $next \leftarrow$ next_log_entry_lba($curr$)
11         continue
12     $curr \leftarrow next$
13     $next \leftarrow$ next_log_entry_lba($curr$)
```

Pre-scheduling. Paralfetch schedules the collected log entries. Algorithm 1 describes the procedure of metadata shift: Paralfetch accesses log entries in their access order (lines 1, 3, 11). A log entry for metadata blocks moves right away to the *MS* queue[3] (lines 4–5), while a log entry for data blocks remains in the wait queue until enough subsequent metadata blocks (at least the metadata shift size) are moved to the MS queue (lines 9–10) in order to left-shift metadata I/O requests When enough metadata blocks are left-shifted, the accompanying wait queue log entries are transferred to the MS queue (line 7). Finally, the red-black tree *rbtree_seq* is rebuilt with the metadata-shifted order (line 13) once the remaining log items in the wait queue are transferred to the MS queue (line 12).

To perform range merge (as described in Algorithm 2), Paralfetch accesses log entries in their LBA-sorted order. This makes it easy to detect log entries that have consecutive LBAs (line 5) of the same inode (line 4). Range merge then combines consecutive I/O operations (lines 7–9) that are within a predefined threshold for I/O distance in the launch sequence (line 6).

Different thresholds of metadata shift and range merge are used for SSDs with and without command queuing (CQ). To discover whether an SSD supports CQ, the Paralfetch initialization process, executed by the systemd daemon or a startup script (*e.g.*, rc.local), examines sysfs files. For example, the CQ support for an SATA SSD is determined by the value of /sys/block/*<root device>*/device/queue_depth.

Storing scheduled log entries. Scheduled log entries (*i.e.*, prefetch entries) are stored in the file <app_name>.pf (*e.g.*, eclipse.pf for Eclipse). This file consists of a 24-byte Paralfetch header, followed by prefetch entries. The header contains the version number, the inode number of the executable file, the metadata for dynamic scheduling, the number of obsolete entries, and the number of prefetch entries. Each prefetch entry contains the device number, the inode number, its offset and size. The inode number for a metadata block is set to 0. The size of each prefetch entry is 20(24) bytes on a 32(64)-bit system.

4.3 Prefetching Phase

During the prefetching phase, Paralfetch creates the prefetch thread, following the sequence stored in the <app_name>.pf file.

For EXT4 file system, Paralfetch uses the __breadahead function to prefetch metadata blocks, and the force_page_cache_readahead function to prefetch data blocks for regular files. While these functions try to perform block caching asynchronously (or in a non-blocking manner), data blocks can be prefetched asynchronously only when the associated metadata blocks are ready. Paralfetch uses explicit I/O plugging [3] to merge contiguous metadata (bio) requests into a single request, which is then delivered to the dispatch queue of device drivers. This reduces the amount of computation required for dispatching and completing I/O requests.

Changing from prefetching back to the learning phase. The set of blocks required for the first launch of some applications is significantly different from that required for subsequent launches. For example, Eclipse and GIMP only configure their environments on their first launch: Paralfetch detects this behavior by counting I/O requests issued by an application during its launch, which is easily done by counting synchronous readahead requests [38] in the Linux readahead framework [33]. If the count is greater than 10% of the total number of prefetch entries, Paralfetch returns to the learning phase.

5 Evaluation

5.1 Methodology

Launch time measurement. Like [24], we measure the launch time of an application between two events: in the

[3] The MS queue stores the metadata-shifted order of log entries.

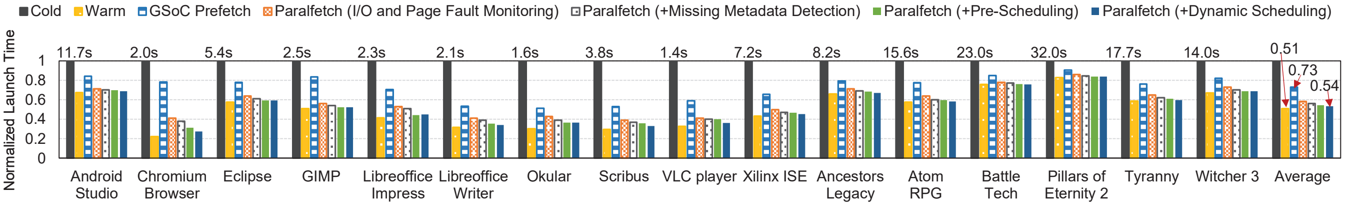


Figure 10: Launch times on a laptop equipped with a QLC SSD, normalized to cold start times. Optimizations for Paralfetch are incrementally applied.

case of Linux, the launch is deemed to start when the load_elf_binary function is called, and to finish when the completion block request has itself completed. To identify the latter event, we remove the completion block request from the prefetch file, allowing it to be issued by the application. After a warm start, we call posix_fadvise with the argument POSIX_FADV_DONTNEED to evict the completion block request from the page cache.

Comparisons with other prefetchers. We ported the GSoC Prefetcher to the Linux kernel 5.4.51 and set its trace timeout to the value used by Paralfetch. We temporarily modified Paralfetch to bring its operation in line with three key features of the GSoC Prefetcher: 1) the way in which it traces referenced file pages during an application launch, 2) its method of pre-scheduling disk I/O using inode numbers and in-file offsets as sort key, and 3) the way in which it holds an application until prefetching is completed, rather than allowing the application and the perfetcher thread to compete.

FAST only supports EXT3 file system, so we temporarily modified Paralfetch's function for detecting missing metadata to support EXT3. We could only compare FAST with Paralfetch on a PC because the Android and Raspbian OS do not support EXT3 file system.

5.2 On a PC

We conducted experiments on a laptop PC equipped with an Intel Core i5-8265 CPU and 16 GB of RAM, running Linux kernel 5.4.51. This PC has a 1 TB Samsung 860 QVO QLC SSD, which uses native command queuing. We tested Paralfetch, GSoC Prefetch and FAST on 16 applications, 6 of which were games. The 10 non-game applications were Android Studio, Chromium Browser, Eclipse, GIMP, LibreOffice Impress, LibreOffice Writer, Okular, Scribus, VLC player, and Xilinx ISE; and the 6 games were Ancestors Legacy, Atom RPG, Battle Tech, Pillars of Eternity 2, Tyranny, Witcher 3.

QLC SSDs typically employ a small pseudo-SLC (single-level cell) cache. To reduce the effects of this cache, we conducted evaluation after installing all benchmark apps.

Comparison with the GSoC prefetcher. Figure 10 shows Paralfetch to reduce the average launch time of these 16 applications by 44.2% with pre-scheduling alone. After four launches of each application, a 1.8% more reduction was achieved on average by using dynamic scheduling to increase prefetch throughput.

It should be noted that the naïve use of excessive metadata shift (of 256KB) led to a 3.8% increase in average launch time: as previously shown in Table 1, Paralfetch fails to trace a few launch blocks. A launching application should wait for these missing blocks to be read while a large number of outstanding I/O requests due to excessive metadata shift increase the waiting time.

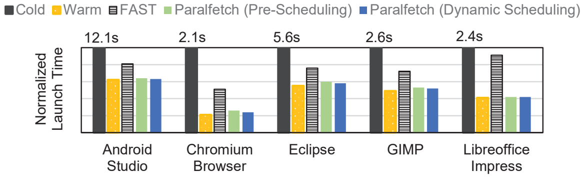


Figure 11: Comparison of Paralfetch and FAST launch times on a laptop PC, normalized to cold start times. Tracing of each application is performed when LibreOffice Writer is running in the background. The results show that running applications can significantly degrade tracing accuracy of FAST and its performance benefit.

Comparison with FAST. FAST is the closest to ours in that its target media is SSDs. In §3.1 we described how disk cache clearing affects tracing accuracy. The most serious drawback of FAST seems to be that the accuracy of its tracing depends greatly on the other applications that are running, because files accessed by these applications through mmap are not traced. Also, metadata used by the applications are not traced. We believe that this issue is frequently occurred in common scenarios. Figure 11 shows the significance of this issue. Conversely, the page fault monitoring and detecting missing metadata used by Paralfetch leads to launch times similar to that of a warm start.

Although tracing under a system-cold state favors FAST, the launch times averaged across all 16 applications were 11% less with Paralfetch than with FAST as shown in Figure 12. The relatively poor performance of FAST can be attributed to its reliance on system calls, which limits both the accuracy of tracing and its scheduling options, in particular its use

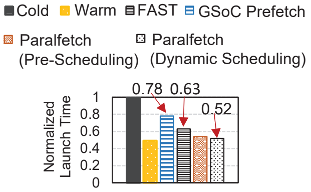


Figure 12: Average launch time for 16 apps on a laptop equipped with a QLC SSD, normalized to cold start times.

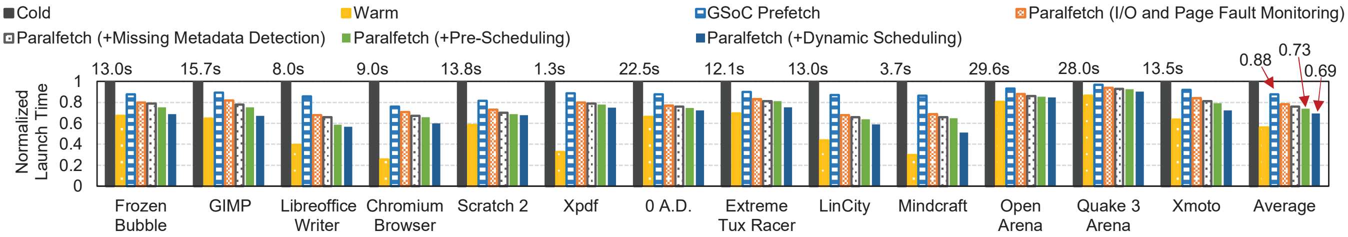


Figure 13: Launch times on a Raspberry Pi 3, normalized to cold start times. Optimizations for Paralfetch are incrementally applied.

of synchronous I/O for prefetching metadata blocks makes it difficult to exploit parallelism.

5.3 Raspberry Pi 3

Our second evaluation of Paralfetch was conducted on a Raspberry Pi 3 running the Raspbian OS (Linux kernel 4.9.56) with a Samsung 16 GB MicroSD (class 10). This flash storage does not support CQ (although more recent A2-class MicroSD has both CQ and an SLC cache).

We used 13 applications, 8 of which were games: Frozen Bubble, GIMP, LibreOffice Writer, Chromium browser, Scratch 2, Xpdf, 0 A.D., Extreme Tux Racer, LinCity, Mindcraft, Open Arena, Quake 3 Arena, and Xmoto. The launch times in Figure 13 show that frequent flash accesses contribute about 45% of the delay in application launches. This provides a considerable opportunity for I/O scheduling. After four launches with dynamic scheduling, launch times are further reduced by an average of 4.8% compared to Paralfetch with pre-scheduling only. We attribute this reduction to: 1) an application launch on a Raspberry Pi 3 board is a disk-bound process, and 2) the throughput of a MicroSD is usually improved by merging I/O operations: for example, the bandwidth of random reads of 128KB on the MicroSD we used is 28.6 MB/sec, which is 6.7× higher than that of 4KB (only 4.3 MB/sec). Chromium Browser and Xpdf application launch times are more heavily influenced by disk performance than by CPU performance. Due to the significant limitations of timely prefetching with prefetch scheduling, it is difficult to achieve warm start launch performance, especially for SSDs without command queuing.

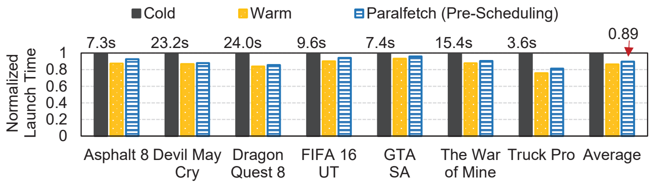


Figure 14: Launch times on an Android smartphone (Google Pixel XL), normalized to cold start times.

5.4 Google Pixel (Android)

Paralfetch can be easily ported to Linux variants, such as Android. Android has its own launch mechanism, and hence we needed to modify 180 lines of the Android source code to accommodate Paralfetch.

To test Paralfetch on Android, we used a new set of seven games: Asphalt 8, Devil May Cry, Dragon Quest 8, FIFA 16 UT, GTA SA, The War of Mine, and Truck Pro. We measured the launch times for these games on a Google Pixel XL smartphone with UFS flash (which supports CQ) running Android 8.0 (Oreo) with the Linux kernel 3.18.52. As shown in Figure 14, the pre-scheduling performed by Paralfetch reduced launch times by 11% on average, which equates to as much as 3.5 seconds for Dragon Quest 8. However, dynamic scheduling offers little benefit because 1) application launches are CPU-bound (86% on average in our benchmarks) rather than disk-bound, and 2) launches encounter little dependencies between metadata and data blocks. Another distinct characteristic of an Android app launch is that a number of write and fdatasync syscalls are issued by SQLite during the launch, making a gap between the times for a warm start and a cold start with Paralfetch.

5.5 Overhead

We measure Paralfetch's overheads on a laptop PC from 4 aspects: tracing, pre-scheduling, prefetching and storage.

Tracing overhead. The I/O-based tracing used by Paralfetch has a low instrumentation overhead, and in most cases log entries are relatively short (*e.g.*, less than 3000 entries). Android Studio is an exception, as it creates lots of log entries. Nevertheless, the difference in cold start launch time with and without Paralfetch was only 136ms. Disk cache invalidation can produce some latency, but this does not affect the working set of pages. Thus, it should not affect the users. In any case, the cache is only invalidated during the learning phase.

Pre-scheduling overhead. In our experiments, the time required by the background jobs which perform pre-scheduling, including missing metadata detection, metadata shift, and range merge, varied between 42ms for VLC Player and 153ms for Android Studio, whereas FAST took 21 seconds to generate the prefetch program for Android Studio. When there is an idle CPU core, pre-scheduling delays can be hidden from users because Paralfetch creates a dedicated thread for that.

Prefetching overhead. Paralfetch employs threaded prefetching, imposing extra overhead from management perspective. However, we observed that threaded prefetching can reduce CPU usage for an application launch in the cold start. As shown in Figure 2, a synchronous I/O incurs two context switches. On the other hand, the asynchronous I/O requests issued by the prefetch thread significantly reduce the overall number of context switches. In our sampling-based CPU utilization measurement [22], we found that the number of context switches during a launch of Android Studio with Paralfetch was reduced from 9,902 to 1,035, resulting in a 3.2% reduction in CPU usage.

In the warm start where prefetching is unnecessary, Paralfetch still runs the prefetch thread, but this only incurs a delay of hundreds of microseconds if an available CPU core exists. Even if there was no available CPU core, where prefetching overhead could not be hidden, Paralfetch extended Android Studio launch by only 2.8ms for (Eclipse by 3.1ms, which was the worst case).

Storage overhead. Paralfetch used 672 KB of SSD to store the <app_name>.pf files for the 16 applications, whereas FAST required 8.2 MB.

6 Future research direction

Non-intrusive tracing. Paralfetch instruments some kernel functions to trace disk accesses. The (low) instrumentation overhead can be effectively removed by employing dynamic instrumentation tools, such as SystemTap [55] and eBPF [56].

Sophisticated prefetch scheduling. Paralfetch applies metadata shifting and range merging to the entire launch sequence, leaving room for further improvement: by applying prefetch scheduling only to prefetch-bottlenecked regions of the launch sequence, Paralfetch can avoid unnecessary I/O contention between the prefetch thread and the launching application, achieving a better launch performance.

Prefetch scheduling considering internal behaviors of disks. If Paralfetch schedules prefetch entries considering internal behaviors and performance of storage devices, it can schedule them better at the pre-scheduling stage, thus reducing the need for rescheduling with dynamic scheduling.

7 Additional Related Work

Previous application prefetchers are discussed in §2. We now summarize various other approaches to reducing application launch times, which are orthogonal or complementary to Paralfetch.

Predictive disk prefetchers, such as Preload [14] and Windows Superfetch [37], analyze the pattern and frequency of application usage, predict the applications that are likely to be loaded soon, and then preload them. Falcon [42] is a predictive prefetcher that considers mobile context such as location and battery state. Falcon launches an application in advance rather than merely prefetching launch-related blocks. Obviously, the merit of this strategy depends heavily on the accuracy of the prefetcher's predictions [34].

General-purpose disk prefetcher. It has been demonstrated that general-purpose prefetching [11, 28] can also be beneficial in reducing application launch times. However, it can limit the accuracy of tracing launch-related blocks because block-level I/O patterns depend greatly on the contents of disk caches.

A block I/O cache provides another way of reducing latency. Intel Turbo Memory [31], Intel Smart Response Technology [51], and AMD StoreMI [52] store delay-sensitive data in a relatively fast SSD and other data in a larger region of slower storage. A similar behavior is provided by software caching methods, which operate in the device mapping layer [1] and the block layer [2].

I/O scheduling can reduce I/O contention between a launch process and background processes. Several schemes have been proposed: FastTrack [16] prioritizes I/O requests generated by the foreground application, and the BFQ I/O scheduler [10] gives new processes extra I/O bandwidth. Boosting the priority of an I/O request, which is issued asynchronously but results in blocking the issuing process, can also expedite a launch [21].

Memory management can also reduce latency. Re-assigning pages from background apps to foreground apps can improve user experience of mobile operating systems [44]. Similarly, pre-swapping of unused memory can reduce delays by avoiding page reclamation latencies [45]. These schemes can reduce app launch times by timely provision of memory when it is under pressure.

8 Conclusion

We have presented Paralfetch, which achieves launch performance close to the warm start through more accurate tracing, pre-scheduling for fast I/O reads, and prefetch thread overlapping. Paralfetch incurs negligible overhead in terms of CPU, memory, and storage. We have also shown Paralfetch to significantly outperform existing prefetchers on various personal computing/communication devices running Linux.

Acknowledgments

We would like to thank our shepherd Nitin Agrawal and the anonymous reviewers for their valuable feedback and suggestions. This work was partly supported by the IITP grant (No. RS-2022-00155885, Artificial Intelligence Convergence Innovation Human Resources Development (Hanyang University ERICA)) and the National Research Foundation (NRF) of Korea grant funded by the Korean government (MSIT) (No. NRF-2022R1F1A1074505). K. Kang and D. Lee are the corresponding authors.

References

[1] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 355–369, 2016.

[2] L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving virtualized storage performance with Sky. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 112–128, 2017.

[3] J. Axboe. Explicit block device plugging. https://lwn.net/Articles/438256/, 2011.

[4] Y. Takai, M. Fukuchi, R. Kinoshita, C. Matsui, and K. Takeuchi. Analysis on heterogeneous SSD configuration with quadruple-level cell (QLC) NAND flash memory. In *Proceedings of the 11th IEEE International Memory Workshop (IMW)*, pages 169–172, 2019.

[5] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Lip-Tak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for self-optimizing storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 183–196, 2009.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 188–197, 1995.

[7] F. Chang, and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 1999.

[8] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–277, 2011.

[9] L. Colitti. Analyzing and improving GNOME startup time. In *Proceedings of the 5th System Administration and Network Engineering Conference (SANE)*, pages 1–11, 2006.

[10] J. Corbet. The BFQ I/O scheduler. https://lwn.net/Articles/601799/, 2014.

[11] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC)*, pages 261–274, 2007.

[12] C. Dirik, and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 279–289, 2009.

[13] A. Eisenman, I. Abdelrahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*,pages 42:1–13, 2018.

[14] B. Esfahbod. Preload–An adaptive prefetching daemon. Master's thesis, Graduate Department of Computer Science, University of Toronto, Canada, 2006.

[15] W. Fengguang, X. Hongsheng, and X. Chenfeng. On the design of a new Linux readahead framework. *ACM SIGOPS Operating Systems Review*, 42(5):75–84, 2008.

[16] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim. FastTrack: Foreground app-aware I/O management for improving user experience of Android smartphones. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 15–28, 2018.

[17] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–83, 2011.

[18] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, pages 155–168, 2012.

[19] W. W. Hsu, A. J. Smith, and H. C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems*, 23(4):424–473, 2005.

[20] B. Hubert. On faster application startup times: Cache stuffing, seek profiling, adaptive preloading. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 245–248, 2005.

[21] D. Jeong, Y. Lee, and J.-S. Kim. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of The 13th USENIX Conference on File and Storage Technologies (FAST)* pages 191–202, 2015.

[22] Y. Joo, Y. Cho, K. Lee, and N. Chang. Improving application launch times with hybrid disks. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* pages 373–382, 2009.

[23] Y. Joo, J. Ryu, S. Park, H. Shin, and K. G. Shin. Rapid prototyping and evaluation of intelligence functions of active storage devices. *IEEE Transactions on Computers*, 63(9):2356–2368, 2014.

[24] Y. Joo, J. Ryu, S. Park, and K. G. Shin. FAST: Quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 259–272, 2011.

[25] Y. Joo, J. Ryu, S. Park, and K. G. Shin. Improving application launch performance on SSDs. *Journal of Computer Science and Technology*, 27(4):727–743, 2012.

[26] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 209–222, 2012.

[27] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *Proceedings of the 9th ACM/IEEE International Conference on Embedded software (EMSOFT)*, pages 295–304, 2009.

[28] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)* pages 173–186, 2004.

[29] K. Lichota. Prefetch: Linux solution for prefetching necessary data during application and system startup. http://code.google.com/p/prefetch/, 2007.

[30] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 383–400, 2016.

[31] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel®Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Transactions on Storage*, 4(2):1–24, 2008.

[32] D. T. Nguyen, Improving smartphone responsiveness through I/O optimizations. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication (UBICOMP Adjunct)*, pages 337–342, 2014.

[33] R. Pai, B. Pulavarty, and M. Cao, Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 105–116, 2004.

[34] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UBICOMP)*, pages 275–284, 2013.

[35] K. Kim, E. Lee, and T. Kim, HMB-SSD: Framework for efficient exploiting of the host memory buffer in the NVMe SSD. *IEEE Access*, vol. 7, pp. 150403-150411, 2019.

[36] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 79–95, 1995.

[37] M. Russinovich, D. Solomon, and A. Lonescu. Windows Internals, Part 2, 6th ed. Microsoft Press, pages 324-350, 2012.

[38] W. Mauerer. Professional Linux Kernel Architecture. Wrox Press, pages 970–974, 2008.

[39] J. Ryu, Y. Joo, S. Park, H. Shin, and K. G. Shin. Exploiting SSD parallelism to accelerate application launch on SSDs. *IET Electronics Letters*, 47(5):313–315, 2011.

[40] S. Vandebogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *Proceedings of the 2009 USENIX Annual Technical Conference (ATC)*, pages 299–312, 2009.

[41] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 211–226, 2018.

[42] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fastapp launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, pages 113–126, 2012.

[43] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 759–771, 2017.

[44] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *Proceedings of*

the 2020 USENIX Annual Technical Conference (ATC), pages 873–887, 2020.

[45] Y. Liang, J. Li, R. Ausavarungnirun, R. Pan, L. Shi, T.-W. Kuo, and C. J. Xue. Acclaim: Adaptive memory reclaim to improve user experience in Android systems. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 897–910, 2020.

[46] BodNara. ADATA Ultimate SU630 960GB. https://www.bodnara.co.kr/bbs/article.html?num=154114, 2019.

[47] T. Schiesser. Storage Game Loading Test: PCIe 4.0 SSD vs. PCIe 3.0 vs. SATA vs. HDD. https://www.techspot.com/review/2116-storage-speed-game-loading, 2019.

[48] T. Thomas. Samsung's 860 QVO 1-TB SSD reviewed. https://techreport.com/review/34281/samsungs-860-qvo-1-tb-ssd-reviewed, 2018.

[49] K. A. Shutemov. mm: map few pages around fault address if they are in page cache. https://lwn.net/Articles/588802, 2014.

[50] R. Nelson. The size of Iphone's top apps has increased by 1,000% in four years. https://sensortower.com/blog/ios-app-size-growth, 2017.

[51] Intel® Smart Response Technology: Technology Brief. https://www.intel.com/content/www/us/en/architecture-and-technology/smart-response-technology-brief.html, 2014.

[52] AMD StoreMI Technology. https://www.amd.com/en/technologies/store-mi, 2021.

[53] S. Sivaram, and C. Bergey, Zoned storage for the zettabyte age, https://www.flashmemorysummit.com/Proceedings2019/08-06-Tuesday/20190806_Keynote2_WesternDigital_Sivaram_Bergey.pdf, 2019.

[54] James Larus. Spending Moore's dividend. *Communications of the ACM*, 2009.

[55] SystemTap. https://sourceware.org/systemtap, 2022.

[56] eBPF. https://ebpf.io, 2022.

[57] Install Ubuntu on a Raspberry Pi. https://ubuntu.com/download/raspberry-pi, 2022.