Paralfetch: Fast Application Launch on Personal Computing/Communication Devices

Junhee Ryu, Dongeun Lee, Kang G. Shin, *Life Fellow, IEEE*, and Kyungtae Kang, *Member, IEEE*

***Abstract*—Paralfetch speeds up application launches on personal computing/communication devices, by means of: 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 application execution with disk prefetching for hiding disk access time from the execution of the application. We implemented Paralfetch under Linux kernels on a desktop/laptop PC, a Raspberry Pi 3 board, and an Android smartphone. Tests with popular applications show that Paralfetch significantly reduces application launch times on flash-based drives and hard disk drives, and it outperforms *GSoC Prefetch* Lichota et al. 2007 and FAST Joo et al. 2011, which are representative application prefetchers available for Linux-based systems.**

***Index Terms*—Application launch, application prefetch, disk prefetch.**

I. 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 [14], [15], [19], [21], [23], [30], [43], [44]. The latency incurred by the launch of an application mainly depends on the performance of the underlying CPU and disks. Despite continuing improvements in the performance of these components, launch times, especially of large applications and games, remain an important problem, for three reasons.

First, the performance of flash storage does not always meet expectations. 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 [3]. The use of affordable QLC SSDs was found to extend the launch time of Blade and Soul, a popular game, from 91s to 114s [45], and that of Horizon Zero Dawn from 15.7s to 21.4s [46], compared to high-end SSDs. Many Windows applications launched from Samsung QLC SSDs show launch times similar to those launched from Intel X25-M G2 SSD, which was released back in 2009 [47]. Furthermore, recent entry-class SSDs widely adopt DRAM-less architecture [5], which leads to additional flash accesses in translating logical addresses into physical addresses. A Raspberry Pi is also widely used to run desktop applications [56], [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 [49]. Unfortunately, complex software also requires higher-level programming languages and libraries, generating slower code, thus extending their launch latencies [54].

Third, although parallelism (i.e., multicore platforms, internal parallelism on SSDs [6], and command queuing features) is effectively utilized in modern multicore CPUs and disk drives, app launches can seldom exploit existing sources of parallelism. Moreover, it has been shown [22] 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 [21].

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 an app, either because it is the first time the app is launched, or because all of the app's data has been evicted since its last run (e.g., after a large file copy, video play, or heavy game play). A *system cold start* is a special case of cold start, which occurs when no user-launched app is running. On the contrary, a *warm start* occurs when an app being launched ran recently, so the disk cache still holds all, or most, of the data that it needs.

An app prefetcher [4], [8], [26], [31], [34], [38], [39] 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 optionally 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 the learned launch sequence is used for disk prefetching to accelerate loading. A well-designed prefetcher can reduce the time needed for a cold start, so that it approaches the time for a warm start.

Received 13 October 2023; revised 16 October 2024; accepted 25 December 2024. Date of publication 2 January 2025; date of current version 3 March 2025. This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP), funded in part by the Korean government (Ministry of Science and ICT) under Grant RS-2022-00155885, Grant RS-2024-00431388, and Grant IITP-2024-RS-2024-00423071 for the Artificial Intelligence Convergence Innovation Human Resources Development Program at Hanyang University ERICA, in part by the Global Research Support Program in the Digital Field, and in part by the Convergence Security Core Talent Training Program, respectively. Recommended for acceptance by D. Feng. *(Corresponding authors: Kyungtae Kang; Dongeun Lee.)*

Junhee Ryu is with the SK hynix, Seongnam 13558, South Korea (e-mail: jhryu79@gmail.com).

Dongeun Lee is with the Department of Computer Science, East Texas A&M University, Commerce, TX 75428 USA (e-mail: dongeun.lee@tamuc.edu).

Kang G. Shin is with the Department of Electrical and Engineering, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: kgshin@umich.edu).

Kyungtae Kang is with the Department of Artificial Intelligence, Hanyang University, Ansan 15588, South Korea (e-mail: ktkang@hanyang.ac.kr).

Digital Object Identifier 10.1109/TPDS.2024.3525337

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 the logical block address (LBA) order to reduce seek times [40], [41], [42], 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 order to avoid long disk seek operations caused by I/O contention between them, sorted prefetching is performed within the context of launching app.

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

- *Accurate tracking of launch-related blocks:* Most monitoring methods fail to locate a significant number of blocks during the learning phase [20]. 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 disk requests 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 traced blocks to increase prefetch throughput:* 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 that significantly hinder the asynchrony of I/O requests and reduce 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, enabling more asynchronous I/O requests. A *range merge* is also introduced to combine nearby I/O requests into one large request, improving I/O throughput. For HDDs, `Paralfetch` employs a new LBA sorting method that streamlines disk head movements, even on aged file systems.
- *Tailored overlapping of application execution with prefetching:* 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 [35]. 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.

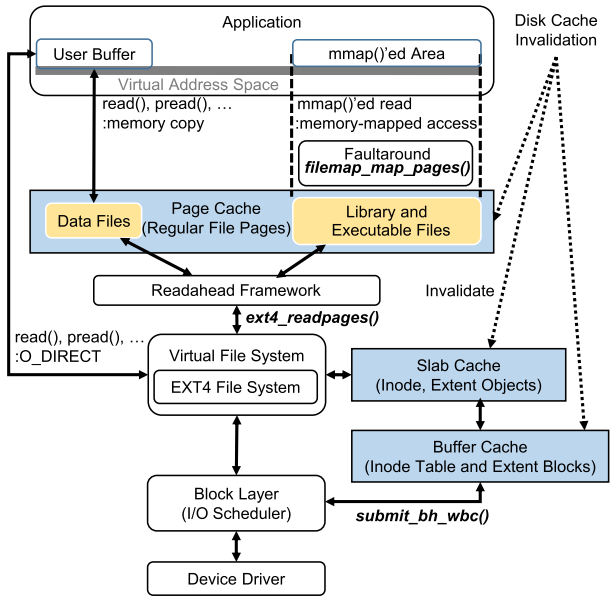


Fig. 1. I/O Stack in Linux. Linux includes three disk caches: page cache for regular files, slab (or slub) cache for metadata objects, and buffer cache for metadata blocks. The slab is used as an object-granular metadata cache for buffer cache. There are different paths to populate page cache in accordance with an access method: `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 the number of blocks to be prefetched based on the access sequentiality.

- *Implementation and evaluation 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 of 16 benchmark apps by 48.0% , this number corresponds to 11% and 22% further reductions from the results of `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% .

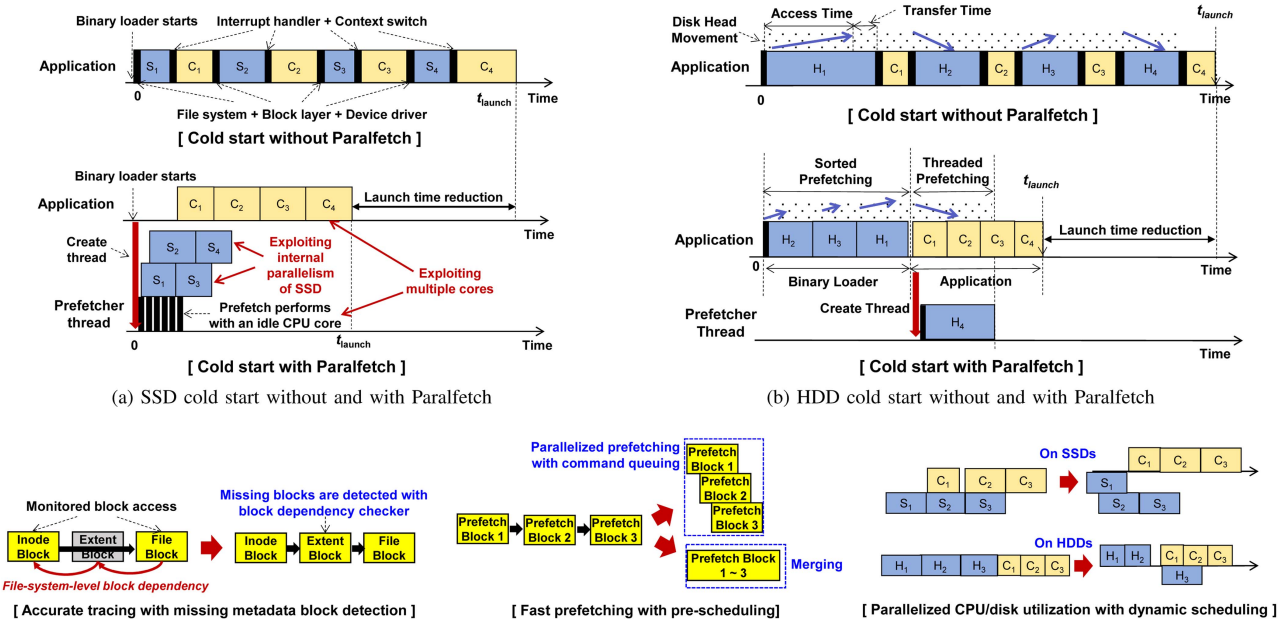
The remainder of this paper is organized as follows: In the next section, we briefly present the background and motivation for this work. In Section III we introduce `Paralfetch`, and describe its three main components. In Section IV we describe the implementation of those components on a Linux platform using EXT4 file system. In Section V we evaluate `Paralfetch` on a laptop PC, a Raspberry Pi 3, and an Android smartphone. In Section VI we discuss the deployment and limitations of `Paralfetch`. In Section VII, we present prior work related to `Paralfetch`, and finally, we conclude the paper in Section VIII.

II. Background and Motivation

A. Disk Caching in Linux

Fig. 1 provides a brief 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 [12]: The *page cache* holds file pages, whereas the *buffer*



(a) SSD cold start without and with Paralfetch

(b) HDD cold start without and with Paralfetch

(c) Key optimization strategies

Fig. 2. Cold start scenarios with and without `Paralfetch` on SSDs and HDDs. 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. H_i is the i-th block requested from the HDD. On systems based on HDDs, the benefit from threaded prefetching is limited because an application launch on HDDs is mostly disk bound and reducing disk time mainly comes from generating a sorted I/O prefetching pattern.

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.

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*.

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 context of the buffer cache. As a result, 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 system call. Alternatively, a process can map the extent of a file to its virtual address space using the `mmap` system call. 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* [33], 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) to the `/proc/sys/vm/drop_caches` proc file. This method can only invalidate unused entries with zero reference counts. Because of this, the invalidation of these caches is not perfect.

Command queuing at disk level: The majority of today's commodity disks supports command queuing (CQ). For example, native command queuing (NCQ) in the AHCI interface for SATA drives supports up to 32 outstanding commands in only one queue; the NVMe interface theoretically allows up to 64 K commands in each of 64 K queues. Even a USB-connected storage device can exploit CQ if the storage controller in the enclosure supports USB attached SCSI protocol (UASP). Outstanding commands in the command queue can be processed out of order, optimizing disk head movements on HDDs. On SSDs, CQ enables SSD controllers to accept multiple commands and leverage their parallelization [6]. For mobile storage, UFS, eMMC 5.1, and Application-class MicroSD cards also support CQ.

B. Representative Application Prefetchers

We briefly introduce alternative schemes, focusing on the reduction of launch time. Moreover, we discuss related studies leveraging different approaches in Section VII.

Windows prefetcher [17]: Since XP, Windows has included a prefetcher for launch and system boot. The *Windows prefetcher* is customized for HDDs, but can also be used with SSDs, although user configuration is required to make the 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. The resulting data is file-level information, which determines the disk blocks that will be prefetched in subsequent application launches. Unfortunately, the defragmentation procedure is time-consuming and scheduled to happen every three days.

GSoC Prefetch [18], 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 (i.e., `struct page`) which indicates that the page has been referenced. After a predefined monitoring period (10 seconds by default), GSoC Prefetch traces referenced pages with the 'referenced' bits on. It then extracts a file identifier (i.e., 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 the file. This reduces both seek and rotational latencies in HDDs. GSoC Prefetch also has a defragmentation tool similar to that in the Windows prefetcher.

FAST [21] is a recent Linux-based prefetcher for SSDs. In the learning phase, `FAST` invalidates the slub, buffer, and page caches. Then, it executes the target app and 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 launch. `FAST` prefetches disk blocks in the order of their accesses without any I/O optimization.

C. Cold Start With Paralfetch

On SSDs: Fig. 2(a) shows a cold start scenario without and with `Paralfetch`. As shown in the figure, the scenario with `Paralfetch` runs the application concurrently with a prefetch thread. The computations run on multiple CPU cores, in parallel with SSD access, 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 CQ. If an SSD does not support CQ, `Paralfetch` merges I/O requests that have consecutive LBAs and are close in the block access sequence, so as to promote internal parallelism.

On HDDs: Fig. 2(b) shows a launch scenario on an HDD without and with `Paralfetch`. The app launch on an HDD is usually I/O bound because of heavy disk head movements. To reduce the mechanical access time, 'Binary Loader' first performs sorted prefetching without creating a prefetch thread, which also stops loading the binary of the target application until sorted prefetching is completed. Next, `Paralfetch` performs threaded prefetching for loading a small portion of blocks that are accessed late (we refer to these blocks as *late-deadline blocks*) by the target application to overlap the computation with HDD access. On the other hand, early-deadline blocks should be prefetched during the sorted prefetching and within the context of the launching process; otherwise, these blocks could delay the target application.

III. Paralfetch Design and Preliminary Results

In this section, we describe the design and preliminary analysis of the three major components of `Paralfetch`.

A. Accurate Tracing

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

There are also issues with the tracing method used by GSoC Prefetch. Since it only traces pages for regular files, missing metadata limits the benefit of prefetching. Furthermore, a significant number of pages are accessed more than once during a launch. This is problematic because the Linux memory manager tracks the importance of a page based on both activeness and reference bits (stored in page descriptors). When a page with its reference bit set on is accessed for the second time, the Linux kernel turns off the 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 (i.e., slab, page, and buffer 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 logs are used to prefetch corresponding blocks during subsequent 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.

As mentioned earlier, the invalidation of disk caches (slab, page, and buffer caches) is not perfect because only unused entries can be invalidated; a working set of 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 I classifies traced blocks with `Paralfetch`. Note that metadata blocks and `mmap`ed file blocks are potential missing blocks when using `FAST`.

Since many user and system processes can run in the background, it can significantly degrade tracing accuracy. For example, 225 files of this kind were accessed by both LibreOffice

TABLE I
METADATA AND DATA BLOCK REQUESTS REQUIRED TO LAUNCH APPLICATIONS

	Application	Read requests traced by Paralfetch: Metadata accesses (total size in KB)	Read requests traced by Paralfetch: File data accesses (total size in KB)	Number of missing metadata blocks detected	Number of accessed files: regular files	Number of accessed files: mmaped files	Number of missing I/Os
Ubuntu Linux (SSD-based laptop)	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

Note that 'regular' files include mmaped files, and that files mmaped by running applications are not subject to disk cache invalidation. Therefore, mmaped files are potential contributors to missing i/os for regular files. The last column shows the number of I/O operations that were not captured by paralfetch, which varies from run to run.

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 with 1,281 log entries (86,624 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, respectively.

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

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 (Section IV-B) that tracks associated metadata blocks for each log entry for a regular file. Table I 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 other running applications. When these missing blocks are found, `Paralfetch` inserts new prefetch entries for them just before other log entries of associated data blocks.

A log entry for data blocks is contiguous in terms of file-level block numbers but may consist of non-contiguous blocks in LBA. During the process of detecting missing metadata blocks, `Paralfetch` identifies this type of log entry and splits it into multiple LBA-contiguous entries (or extents). Otherwise, intermixed chunks can interfere with LBA sorting.

Page fault monitoring: Page cache invalidation is also imperfect because file-backed pages that 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 to monitor. 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 [33] handler with page fault tracing code. By default, the faultaround handler proactively maps 16 boundary-aligned page-cached pages around the page-faulted address.

Missing blocks: Despite efforts to collect an accurate set of launch-related blocks, `Paralfetch` still misses a few, as indicated by the number of missing I/Os in Table I: First, an application may perform metadata-only access when reading: 1) directory blocks to search for a file, 2) an inode block to access file statistics, or 3) extent blocks to retrieve the logical-to-physical mapping of a file (e.g., using `fiemap`). If the missing metadata blocks are accessed solely for metadata purposes, `Paralfetch`'s detection feature cannot identify them. Second, files in a user's home directory frequently change, modifying both associated metadata and data blocks. Applications like

Android Studio, LibreOffice, GIMP, and Xilinx ISE read the home directory during the final stage of launch to display files or classify application-relevant files stored in the directory. `Paralfetch` is inadequate at tracking these dynamic changes in real time after generating the prefetch file.

Incorrect prefetch entry removal: `Paralfetch` has a special provision for finding incorrect information in prefetch entries, such as entries generated by irrelevant applications running in the background, or by changes to files used by the application being launched. At predefined intervals (10 days by default), a daemon process of `Paralfetch` checks the prefetch entries in all the <app_name>.pf files to see whether each entry is obsolete: the daemon compares the time of the latest access to <app_name>.pf with the corresponding access time for each regular file that is referred to in the <app_name>.pf file. If the regular file corresponding to prefetch entry has not been accessed more recently than the <app_name>.pf file itself, `Paralfetch` removes the corresponding prefetch entry from the <app_name>.pf file.

This technique can be readily applied when the mount option of the file system is configured as *atime*, causing the access time to be updated each time a file is read through system calls. Thus we know that a file has been accessed by an application if the access time in its file metadata has been updated. This is effective because the access time of a file is not updated via prefetching functions used by `Paralfetch`.

However, if the mount option is *relatime*, the access time of a file is updated during a file read only if it precedes modified time, or the access time was updated a long time ago (one day by default). To avoid being misled by this feature of the Linux kernel, the daemon checks whether the file to be updated was last accessed over 24 hours ago.

B. 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. In doing so, `Paralfetch` *merges* and *reorders* those requests so as to exploit the internal parallelism of an SSD. In the case of an HDD, `Paralfetch` sorts I/O requests by their logical block addresses to reduce disk head movement, which is further optimized by hardware-level reordering.

Range merging: Merging small I/O requests into a single large request enhances the throughput of an SSD [9], [25], [29], [37]. Fig. 3(b) 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 the difference in the locations of blocks in the launch sequence. This threshold prevents merging of far-apart log entries in the launch sequence, as they can hinder timely prefetching of subsequent blocks. Fig. 4 shows plots of prefetch time against the I/O distance threshold on SSD, UFS, 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. This nature of block allocation policy results in range-merging candidates adjacent to each other. In addition, overly aggressive merging

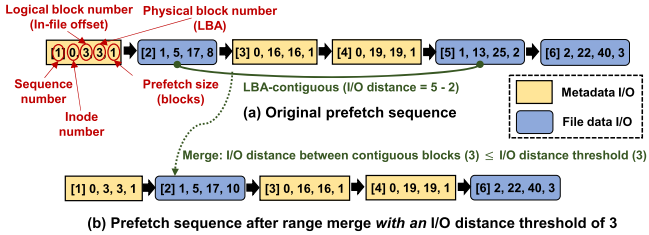


Fig. 3. Range merging. Merging nearby I/O operations into a single large operation improves throughput while keeping changes to the I/O order within a predefined limit so that the target application and prefetch thread can run concurrently. Range merge combines LBA-contiguous I/O requests of the same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

can be bad especially for applications with CPU-bound launches, in which I/O optimization has less impact on timely prefetching.

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 that can potentially be executed in parallel [24].

This issue can be addressed by bringing forward requests for metadata blocks. This can be facilitated in EXT4, as 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. Fig. 5(a) shows the processing of an example prefetch thread, in which dependencies cause the command queue to become empty on two occasions. Fig. 5(b) shows how `Paralfetch` brings forward metadata block requests in the prefetch thread to increase the interval between requests for dependent blocks. Fig. 6(a) shows that the prefetching time on a CQ-enabled SSD was reduced by 21.6% on average through shifting metadata block requests by 128 KB, when combined with the tracing of missing metadata blocks. Similarly, the prefetching time on a UFS storage was incrementally reduced as the shift size increases, which is shown in Fig. 6(b).

An SSD without CQ support can also benefit from shifted metadata (Fig. 6(c)). For example, requests to the I/O scheduler can be issued in advance, so that the storage driver receives requests earlier from the I/O scheduler queue, than later from 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. In Fig. 6(c), a metadata shift of 4 KB reduced prefetch times by 19.3% on average on the Raspberry Pi 3 using a MicroSD.

LBA sorting: Random I/O requests during an app launch result in heavy disk head movements of an HDD [28], [40]. A well-known way to optimize disk head movements is to reorder I/Os by their LBAs [8] and reorganize disk blocks to be close together in order to reduce seek distance [40], [41]. Previous prefetch schemes for HDDs, including GSoC prefetch and Windows prefetcher, employ file-level I/O sorting that uses

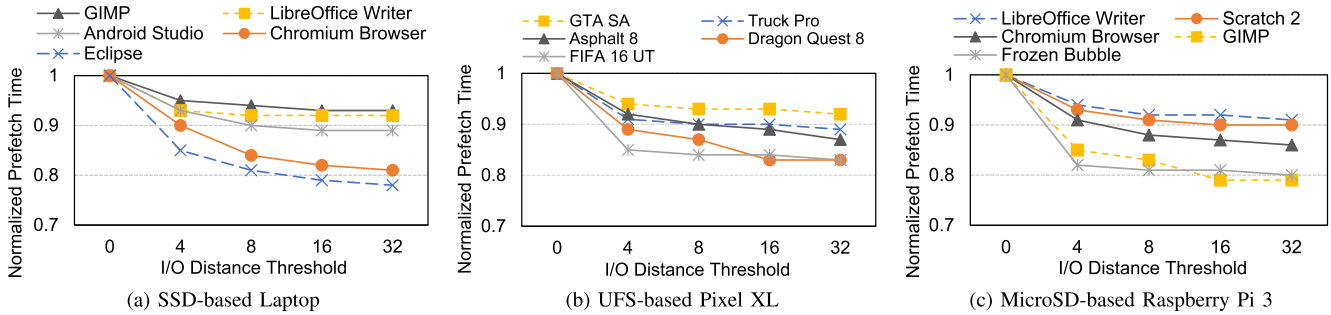


Fig. 4. Normalized prefetching times with varying range merge thresholds.

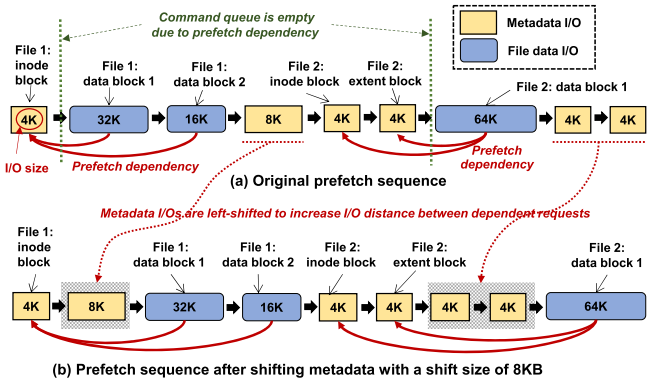


Fig. 5. Metadata shifting to boost the outstanding I/O size in the command queue of an SSD controller. An I/O request for data blocks should wait for the associated metadata blocks to be read. By left-shifting I/O requests for metadata, more I/O requests can be issued asynchronously. The shift size controls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

device number, inode number, and offset as sorting keys, but this still incurs unnecessary disk head movements during prefetching since this ordering is not the same as the LBA order [20]. To address this problem, `Paralfetch` explicitly monitors and sorts I/O requests by their LBAs. Unlike Windows prefetcher and GSoC prefetch, `Paralfetch` performs LBA-sorted prefetching without disk defragmentation. As shown in Fig. 7, a large proportion of disk prefetching time (75.2% on average) can be reduced by minimizing disk head movements.

Infill option: Infill merge [34] was initially invented to avoid unnecessary disk rotations on HDDs by reading extra blocks. And we have found that the throughput of an SSD can also be improved by merging I/O operations with infilling: for example, the bandwidth of random reads of 128 KB on the MicroSD we used is 6.7× higher than that of 4KB. Therefore, there is significant room for reducing the prefetch time via merging I/O requests. Experimental results in Figs. 8 and 9 show the effectiveness of infill merge on flash and rotational disks, respectively. A user can adjust the allowable infill size in the unit of file system block.

The infill option consumes more memory for reading extra disk blocks. To avoid this, `Paralfetch` provides an option that these extra blocks are freed by the prefetcher right after completing the prefetch (at the cost of adding a log entry to describe infill range). However, `Paralfetch` disables this option by default because clean page cache pages are reclaimed preferentially by the Linux kernel with little cost.

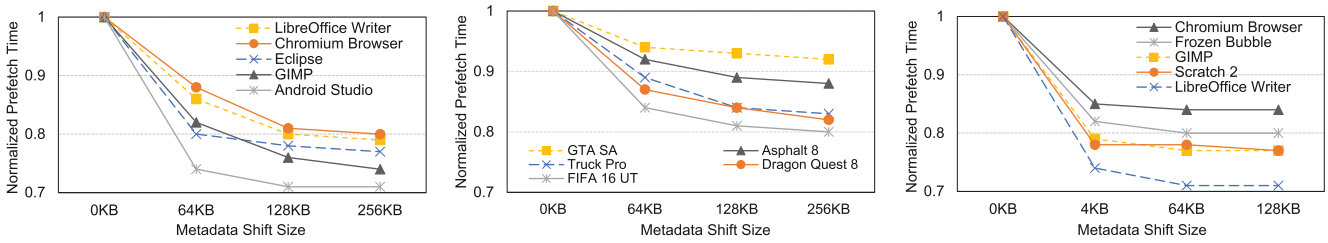
In our experiments, there was no noticeable improvement in prefetch throughput when we applied an infill option simultaneously with metadata shifting on an SSD with CQ support. Therefore, `Paralfetch` applies the infill option only on the SSD without CQ support and the HDD. It is noted that we used empirical values for infill sizes: 32 KB for SSDs and 64 KB for HDDs. As shown in Figs. 8 and 9, with these infill sizes, the average disk prefetching times on the SSD and HDD were reduced by 4.8% and 5.3% , respectively. The figures also show that the benefit of infill merging varies depending on the app, as the impact is largely influenced by the on-disk layout of each app's launch sequence. For instance, in the case of Scratch 2 (Fig. 8), infill merging with a 32 KB threshold merges only 1.2% of the prefetch entries from the original sequence. In contrast, the same threshold merges 8.5% of the prefetch entries for Chromium Browser.

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.

C. 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 Figs. 10 and 11 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. Here 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 a metadata shift size when a prefetching bottleneck is detected.

On SSDs: 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 I, we found that an average of 2.8% of requested blocks were not traced despite the improved tracing capability of `Paralfetch`. These missing blocks are inevitably to be



(a) SSD-based Laptop (b) UFS-based Pixel XL (c) MicroSD-based Raspberry Pi 3

Fig. 6. Normalized prefetching times for different metadata shift sizes.

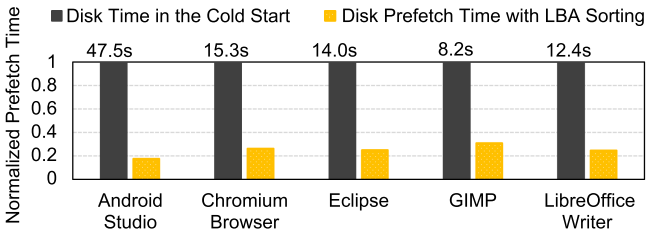


Fig. 7. Disk times in the cold start and prefetching times with LBA sorting on an HDD-based Laptop, normalized to the disk time in the cold start.

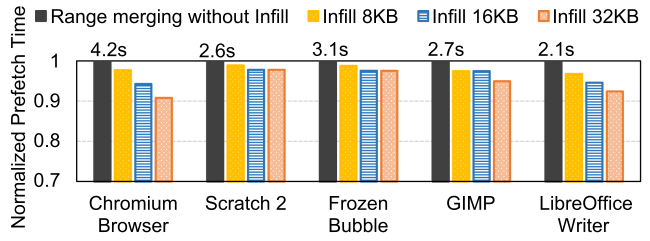


Fig. 8. Prefetching times for different allowable infill sizes on a MicroSD-based Raspberry Pi 3 board, normalized to the prefetch time with range merging. All test cases set a range merging with an I/O distance threshold of 16.

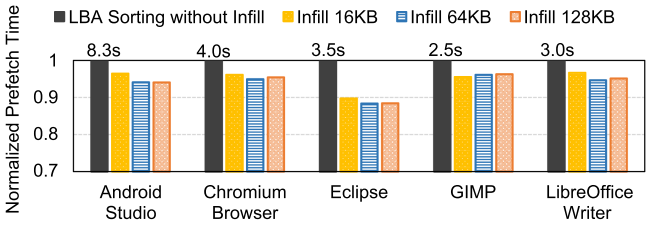


Fig. 9. Prefetching times for different allowable infill sizes on an HDD-based laptop, normalized to the prefetch time with LBA sorting.

requested by the application, which has to wait 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 the command queue [10] in an SSD. This can occur when metadata blocks are shifted too far, or when an oversized 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 merging over a number of trial runs, `Paralfetch` gradually increases the thresholds if prefetching

TABLE II
DEFAULT CONFIGURATION FOR PREFETCH OPTIMIZATION

	SSD without CQ feature	SSD with CQ feature
Range merge (I/O distance)	Starts at 8 and can be increased	8
Metadata shift (KB)	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.

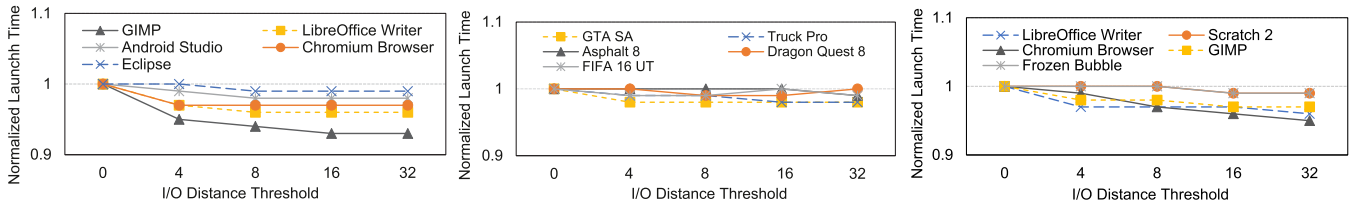
Optimizing prefetch entries with dynamic scheduling. Initially, `Paralfetch` uses default thresholds for metadata shift and range merge shown in Table II. It subsequently increases the threshold for only one of these methods, depending on the availability of CQ support. The metadata shifting threshold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

The best combination of scheduling methods depends on the type of disk. For example, on a CQ-supported SSD, range merging gains little beyond the 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 the threshold of 4 KB.

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. As such, 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). To exclude warm cache results, `Paralfetch` only admits the case where the overall disk read size is larger than a predefined threshold, 90% of disk reads in the cold start in our setting.

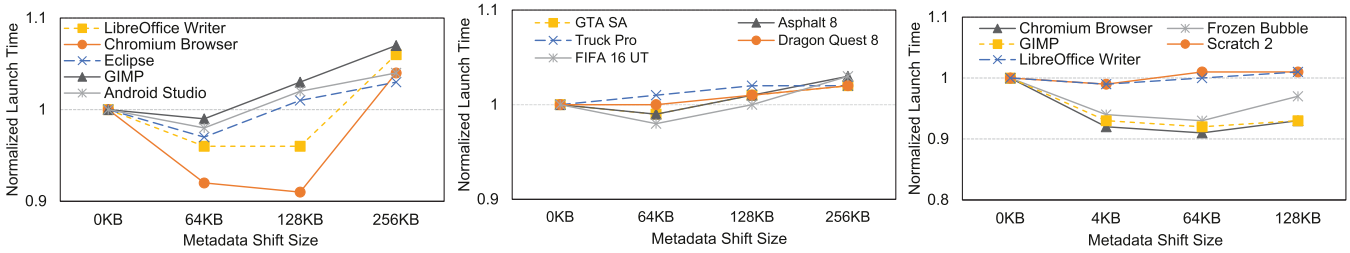
On HDDs: LBA sorting is a main contributor to reduce an app launch time on HDD-based systems. We also found that the launch times can be further reduced by prefetching late-deadline blocks with the prefetch thread.

A challenge is to find the largest quantity of prefetch entries that does not incur prefetch bottleneck with threaded prefetching. To address this, dynamic scheduling is applied with two empirical rates. `Paralfetch` allots 12% (by default) of late-deadline blocks to the threaded prefetching phase at the pre-scheduing stage. If the prefetch bottleneck is detected in the



(a) SSD-based Laptop (b) UFS-based Pixel XL (c) MicroSD-based Raspberry Pi 3

Fig. 10. Normalized launch times with varying I/O distance thresholds.



(a) SSD-based Laptop (b) UFS-based Pixel XL (c) MicroSD-based Raspberry Pi 3

Fig. 11. Normalized launch times for different metadata shift sizes.

threaded prefetching phase, Paralfetch incrementally moves 3% (by default) of prefetch entries from the threaded prefetching phase to the sorted prefetching phase. When a user does not set to use dynamic scheduling, whole log entries are sorted at the pre-scheduling stage.

D. Summary of the Paralfetch Design

- Paralfetch traces the launch sequence at the disk I/O level, imposing minimal overhead. However, this method is affected by imperfect disk cache invalidation. To mitigate this issue, Paralfetch incorporates missing metadata block detection and page fault monitoring. These features enable Paralfetch to trace I/Os and their sequence more accurately. Additionally, Paralfetch removes outdated prefetch entries that are fetched but not accessed by the app.
- Slow prefetch throughput often limits the benefits of app prefetching. Paralfetch improves prefetch throughput by using pre-scheduling techniques, such as range merging and metadata shifting, that exploit the internal parallelism of SSDs. In contrast, FAST does not reorder or merge prefetch sequences, even though these techniques have been shown to be crucial for SSD performance. For rotating disks, Paralfetch performs LBA sorting without requiring disk defragmentation. On the other hand, Windows Prefetcher and GSoC Prefetch rely on file-level block sorting combined with periodic block defragmentation to ensure that the file-level order matches the LBA order
- The primary benefit of app prefetching on SSDs comes from the parallel utilization of the CPU and disk. However, excessive pre-scheduling can hinder timely prefetching by causing reordering of prefetch I/Os. To find a near-optimal pre-scheduling strength, Paralfetch dynamically adjusts the pre-scheduling with enhanced optimization levels, retrying as long as slow prefetching delays the launch procedure.

IV. Implementation of Paralfetch

This section details the workflow of Paralfetch and the interaction among its main components described in Fig. 12.

A. Launch Phase Management

Native Linux: The next launch type for each application is determined by reading the ninth 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 the learning phase, and PHASE_THREADED_PREFETCHING (1) and PHASE_SORTED_PREFETCHING (2) for the 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 or whose I/O patterns highly depend on input arguments. 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.

Android: zygote is a process that creates a native Android application in Java by forking and loading the main class of a program [27]. zygote invokes the handleChildProc method to create and run a new Android application. To reduce launch times, zygote preloads classes and resource files used by many applications, quickly creating a process which shares these preloaded classes. However, this scheme cannot cover all class and resource files, thus leaving significant room for further launch time reduction with Paralfetch.

Unlike native Linux processes, a native Android process remains in the background even after a user quits the application,

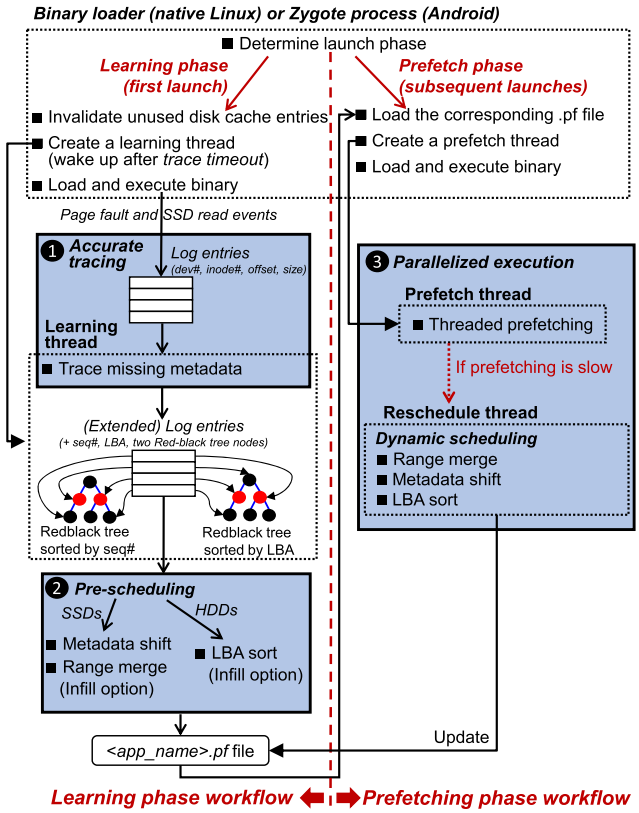


Fig. 12. Paralfetch workflow. Boxes with dotted edges denote threads, and boxes with solid edges identify the three major components of Paralfetch. During a learning phase, `Paralfetch` records an I/O as a form of log entry, which contains device number, inode number, offset (start_blk), and size. Upon the completion of the launch, collected log entries are passed to missing metadata detector, generating additional log entries for missing metadata. The output is a list of (extended) log entries, each of which has additional four fields: sequence number, LBA (start_lba), and two tree nodes for two redblack trees sorted by sequence number and LBA, respectively. Then, redblack trees are passed to pre-scheduling functions, the details of which are described in Algorithms 1, 2, and 3.

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. `Paralfetch` does not begin prefetching for a resuming procedure that does not invoke `handleChildProc`.

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` automatically 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[1] 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.

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

B. Learning Phase

I/O logging: To collect blocks required for a launch, `Paralfetch` first invalidates unused entries in the slab (for file system objects), the buffer cache, and the page cache. Then `Paralfetch` 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 (30 seconds for SSDs and 60 seconds for HDDs), and also sets `trace_flag` to true to activate logging. `Paralfetch` then 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 16 boundary-aligned pages which contain the faulting page, provided that these pages are in the page cache [33]. This feature reduces the overhead of tracing page faults.

Tracing missing metadata blocks: Block tracing ends when it is timed out, and the launch is deemed to be complete when fewer than 10 block read requests occur in one second [22]. We refer to the corresponding last 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 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 the file extents.

As shown in Fig. 12, `Paralfetch` builds two redblack 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 data block entry.

In the pre-scheduling phase, `Paralfetch` uses the LBAs of data blocks obtained using the `ext4_fiedep` function. It is noted that the LBAs of metadata blocks are easily calculated by adding the block number of the log entry to starting block number of the partition.

Missing metadata detection using the `ext4_fiedep` function 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).

Pre-scheduling: `Paralfetch` schedules the collected log entries. To do this, as discussed earlier, `Paralfetch` builds two redblack binary search trees for log entries, each of which uses the starting LBA and the access order of each log entry as the sort key, respectively. `Paralfetch` uses the LBAs of data blocks

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 ← first_log_entry(rbtree_seq)
2  out_meta_size ← 0
3  while log ≠ NULL do
4      if is_metadata_log_entry(log) then
5          move_to_MS_queue(log)
6          out_meta_size ← 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 ← 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 ← first_log_entry(rbtree_lba)
2  next ← next_log_entry_lba(curr)
3  while next ≠ NULL do
4      if curr.inode_num = next.inode_num &
5      curr.start_lba + curr.size = next.start_lba &
6      curr.start_blk + curr.size = next.start_blk &
7      abs(next.seq_num − curr.seq_num) <= dist_thr then
8          if curr.seq_num < next.seq_num then
9              curr.size ← curr.size + next.size
10             unlink_log_entry_from_rbtrees_lba_and_seq(next)
11             remove_log_entry(next)
12             next ← next_log_entry_lba(curr)
13         else
14             next.start_lba ← curr.start_lba
15             next.start_blk ← curr.start_blk
16             next.size ← next.size + curr.size
17             unlink_log_entry_from_rbtrees_lba_and_seq(curr)
18             remove_log_entry(curr)
19             curr ← next
20             next ← next_log_entry_lba(next)
21         continue
22     curr ← next
23     next ← next_log_entry_lba(curr)
```

Algorithm 3: LBA Sort Procedure.

Input: Log entries sorted by their LBA ($rbtree\_lba$) and access order ($rbtree\_seq$)
Result: LBA-sorted log entries (accessed via $rbtree\_lba$)

```
1  curr ← first_log_entry(rbtree_lba)
2  next ← next_log_entry_lba(curr)
3  while next ≠ NULL do
4      if curr.inode_num = next.inode_num &
5      curr.start_lba + curr.size = next.start_lba &
6      curr.start_blk + curr.size = next.start_blk then
7          if curr.seq_num < next.seq_num then
8              curr.size ← curr.size + next.size
9              unlink_log_entry_from_rbtrees_lba_and_seq(next)
10             remove_log_entry(next)
11             next ← next_log_entry_lba(curr)
12         else
13             next.start_lba ← curr.start_lba
14             next.start_blk ← curr.start_blk
15             next.size ← next.size + curr.size
16             unlink_log_entry_from_rbtrees_lba_and_seq(curr)
17             remove_log_entry(curr)
18             curr ← next
19             next ← next_log_entry_lba(next)
20         continue
21     curr ← next
22     next ← next_log_entry_lba(curr)
```

obtained during the detection of missing metadata. `Paralfetch` classifies disks into three types: CQ-enabled SSD, SSD without CQ support, and HDD. This classification is performed automatically through `sysfs`-managed files exposed by mounted devices.[2] To optimize prefetching time, `Paralfetch` performs I/O scheduling based on the type of the disk.

Algorithm 1 describes the procedure of metadata shift: `Paralfetch` accesses log entries in their access order (lines 1, 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 redblack 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 (lines 5–6) of the same inode (line 4). Range merge then combines consecutive I/O operations (lines 9–12 for back merge or lines 14–20 for front merge) that are within a predefined threshold for I/O distance in the launch sequence (line 7).

Different pre-scheduling methods are applied in accordance with the device type ((i.e., rotational media or not), and different thresholds of metadata shift and range merge are used for SSDs with and without CQ. To discover device type and command queuing support (in case of SSDs), 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`.

Algorithm 3 describes the procedure of lba sorting: `Paralfetch` accesses log entries in their LBA-sorted order (lines 1–3, 21 and 22), and it merges log entries (lines 8–11 or lines 13–19) that have consecutive LBAs (lines 4–7, 12) of the same inode (line 4).

The implementation of infill option is rather simple; `Paralfetch` checks that two consecutive log entries in the LBA-sorted red-black tree have consecutive LBAs with an infill of an allowable size. To do this, lines 5–6 in Algorithm 2 and lines 5–6 in Algorithm 3 are modified correspondingly.

Storing scheduled log entries: Scheduled log entries (i.e., prefetch entries) are stored in the file `<app_name>.pf` (e.g.,

[2]The value of `/sys/block/<device>/queue/rotational` is "0" in the case of an SSD and "1" in the case of an HDD. The CQ support is determined by the value of `/sys/block/<device>/device/queue_depth`.

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

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. Finally, Paralfetch changes the launch phase of the target application from the learning phase to the prefetching phase by updating the value of the phase byte of the application.

C. 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 [48] 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 [32] in the Linux readahead framework [12]. If the count is greater than 10% of the total number of prefetch entries, Paralfetch returns to the learning phase. If this situation occurs repeatedly, Paralfetch sets the phase value to PHASE_DISABLE (9) to disable prefetching for applications.

V. Evaluation

We first describe how we measure the launch time of an application and how we compare Paralfetch with other prefetchers. We then present the results of performance evaluations on a PC, a Raspberry Pi 3 board, and a Google Pixel smartphone.

A. Methodology

Launch time measurement: Like [21], we measure the launch time of an application between two events: in the 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.

B. 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: Fig. 13 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 256 KB) led to a 3.8% increase in average launch time: as previously shown in Table I, 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.

Fig. 14 shows experimental results with WD Black 2.5" 7200 RPM 1 TB HDD (WD10JPLX). Because Paralfetch performs LBA sorting and accurate tracing of launch-related blocks, it outperforms GSoC prefetch, which implements file-level I/O sorting without prefetching metadata blocks explicitly. Paralfetch shows an additional 24.9% reduction in launch time. LBA sorting of Paralfetch reduces 73.1% of visible disk time in the cold start, and infill merge reduces the launch time further up to 3.6% in Eclipse with a threshold value of 128 KB.

Comparison with FAST: FAST is the closest to ours in that its target media is SSDs. In Section III-A we described how disk

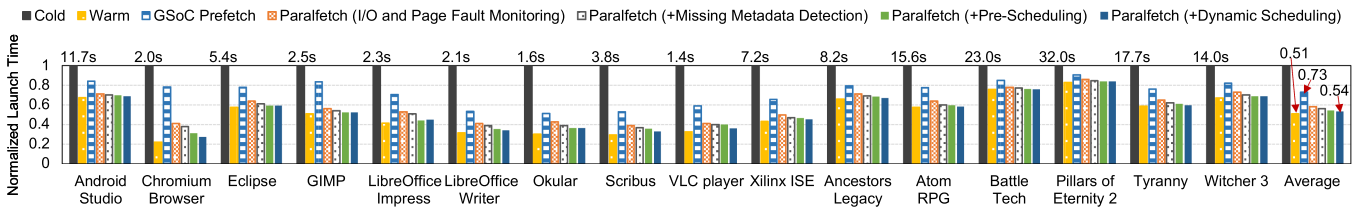


Fig. 13. Launch times on a laptop equipped with a QLC SSD, normalized to cold start times. Optimizations for `Paralfetch` are incrementally applied.

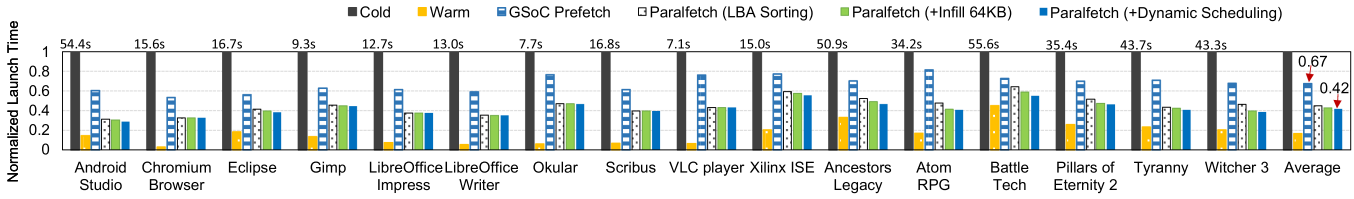


Fig. 14. Launch times on a laptop equipped with an HDD, normalized to cold start times. Optimizations for `Paralfetch` are incrementally applied.

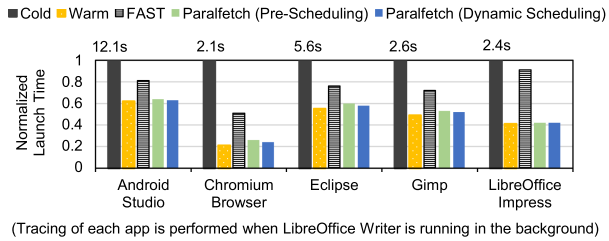


Fig. 15. Comparison of `Paralfetch` and `FAST` launch times on a laptop PC, normalized to cold start times. Tracing of each application is performed when LibreOffice Writer is running in the background. The results show that running applications can significantly degrade tracing accuracy of FAST and its performance benefit.

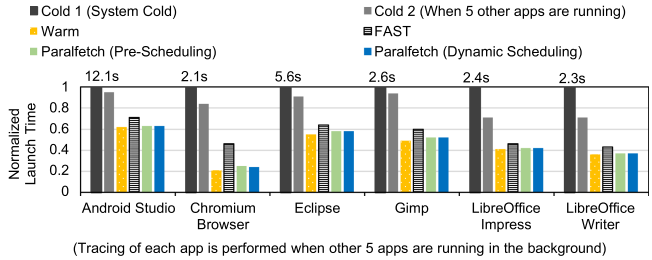


Fig. 16. Comparison of launch times with `Paralfetch` and `FAST` on a laptop PC, normalized to system cold start times (cold 1). Aside from the cold 1 scenario, each application's launch was traced and evaluated while five other applications ran in the background.

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. Fig. 15 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 the primary goal of application prefetchers is to reduce launch times during system cold starts, many applications typically run in the background. Fig. 16 illustrates the launch times of each test application when five other apps are running in the background. In this situation, many I/O requests are served by disk caches, limiting the improvements provided by application prefetchers. As mentioned earlier, LibreOffice Impress and LibreOffice Writer share 68% of commonly accessed blocks, significantly reducing the launch time of one when the other is already running. For disk-intensive applications like the Chromium Browser, `FAST` performs noticeably slower than `Paralfetch` due to its slower prefetch throughput. `FAST` prefetches metadata blocks one at a time, using synchronous system calls. Even worse, it responds to directory data requests by prefetching every block of the directories the target application accesses, further slowing down the process.

Although launch sequence 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 Fig. 17. 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 of synchronous I/O for prefetching metadata blocks makes it difficult to exploit parallelism.

Limitations: For some applications, it is challenging to achieve a launch time close to warm start time even with aggressive dynamic scheduling. Chromium Browser in Fig. 13 is an application where most of launch times are spent on I/O instead of CPU. It is challenging to make their launch times close to a warm start because even for SSD with command queueing, too many I/O commands will trigger its internal reordering, which will, in turn, destroy timely prefetching (simultaneous usage of CPU and SSD).

C. 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

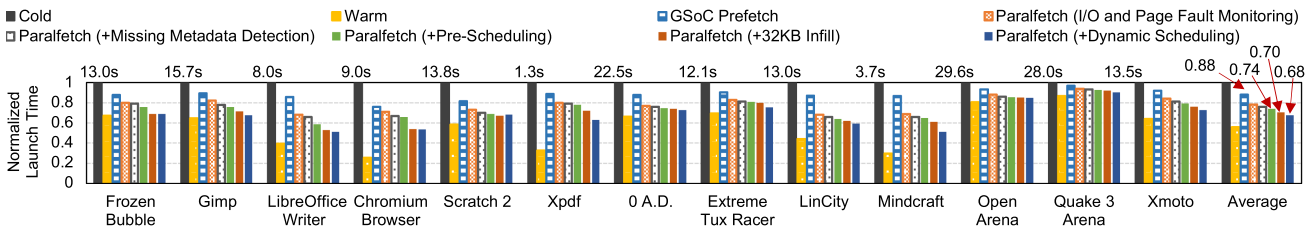


Fig. 17. Launch times on a Raspberry Pi 3, normalized to cold start times. Optimizations for `Paralfetch` are incrementally applied.

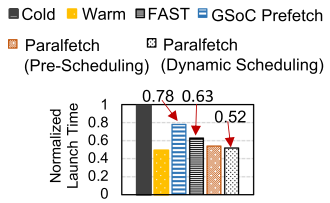


Fig. 18. Average launch time for 16 apps on a laptop equipped with a QLC SSD, normalized to cold start times.

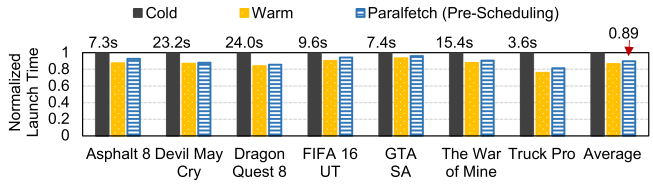


Fig. 19. Launch times on an Android smartphone (Google Pixel XL), normalized to cold start times.

2, Xpdf, 0 A.D., Extreme Tux Racer, LinCity, Mindcraft, Open Arena, Quake 3 Arena, and Xmoto. The launch times in Fig. 18 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 6.5% and 2.3% compared to `Paralfetch` with pre-scheduling and pre-scheduling with a 64 KB infill option, respectively. 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 128 KB on the MicroSD we used is 28.6 MB/sec, which is 6.7× higher than that of 4KB (only 4.3 MB/sec).

Limitations: Chromium Browser and Xpdf application launch times are more heavily influenced by disk performance than by CPU performance. Especially for SSDs without command queuing, prefetch time reduction with pre-scheduling is significantly limited. Due to the limitations of timely prefetching, it is difficult to achieve warm start launch performance.

D. 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 (which supports CQ) running Android 8.0 (Oreo) with the Linux kernel 3.18.52. As shown in Fig. 19, 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.

Limitations: Although the launch procedures of most Android apps are highly CPU-intensive, their launch times with `Paralfetch` are still not comparable to warm start scenarios. This is primarily due to a unique characteristic of Android app launches: multiple `write` and `fdatasync` system calls are issued by SQLite during the process [58]. These synchronous writes compete with disk prefetch requests, causing delays. In contrast, no such synchronous `write` or `fdatasync` system call is issued during the launch of the PC and Raspberry Pi apps listed in Table I.

E. 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 136 ms. 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 42 ms for VLC Player and 153 ms 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: On SSDs, `Paralfetch` employs threaded prefetching, imposing extra overhead from the management perspective. However, we observed that threaded prefetching can reduce CPU usage for an application launch in the cold start. As shown in Fig. 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 [19], 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. On HDDs, `Paralfetch` gets a similar benefit via I/O merging (after LBA sorting) and asynchronous I/O operations.

In the warm start where prefetching is unnecessary, `Paralfetch` on SSDs 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 2.8 ms for (Eclipse by 3.1 ms, which was the worst case). On HDDs, `Paralfetch` performs sorted prefetching which delays the launch procedure. In our experiments, sorted prefetching in the warm start required only 4 ms for Eclipse, 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. On HDDs, `Paralfetch` used only 326 KB of disk space for the same applications. This is because prefetch entries are easily merged after LBA sorting.

VI. Discussion

Applicability to other file systems: Key features of `Paralfetch`, such as finding missing metadata blocks and metadata shifting, rely on file system-level block dependencies. Since most file systems used by major operating systems have their own block dependencies, the core concepts of `Paralfetch` can be extended to other file systems, even though it is built on EXT4. For example, F2FS [50], a popular file system for Android platforms, has its own block dependencies that can be leveraged to detect missing metadata and perform metadata shifting. In F2FS, the node allocation table (NAT) contains the addresses of all node blocks, meaning a node block can only be prefetched after the corresponding NAT entry is loaded. Similarly, a data block depends on the node block that references it. Notably, `Paralfetch` explicitly invokes the readahead functions of the mounted file system, meaning it operates independently of the readahead policy implemented by that file system. Even if the on-disk layout of prefetch entries is altered or split by file system-level garbage collection, the changes can easily be reflected in the `<app_name>.pf` file using the `f2fs_fiemap` or `f2fs_fiedep` function.

Effectiveness of `Paralfetch` *on a newer Linux kernel:* Slow application launches, even on newer Android smartphones or Raspberry Pi boards, remain problematic due to the ever increasing complexity of software and the minimal improvements in flash access latency for secondary storage, particularly with small synchronous random I/O requests. In contrast, SSD throughput has significantly improved with newer interfaces, such as the latest PCIe generation. `Paralfetch` can take advantage of this increased throughput through its pre-scheduling feature. Additionally, recent Linux kernels have shifted the I/O scheduling burden from software to hardware [59], opening up further optimization opportunities for `Paralfetch`'s pre-scheduling capabilities. Despite the presence of an I/O scheduler, I/O merging rarely occurs at the scheduler level unless explicit I/O plugging [48] is used. When an SSD has an available slot in the command queue, the scheduler immediately inserts the request into the hardware queue without delay. For SSDs without a command queue, dependencies between mergeable I/O requests can prevent their merging. This makes pre-scheduling essential for improving prefetching throughput.

VII. Additional Related Work

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

Predictive disk prefetchers, such as Preload [11] and Windows Superfetch [17], analyze the pattern and frequency of application usage, predict the applications that are likely to be loaded soon, and then preload them. Falcon [36] 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 [30].

General-purpose disk prefetcher: It has been demonstrated that general-purpose prefetching [8], [26] 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 [28], Intel Smart Response Technology [51], AMD StoreMI [52], and Solidigm Fast Lane [55] store delay-sensitive data in a relatively fast SSD (or pseudo-SLC space) 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 [13] prioritizes I/O requests generated by the foreground application, and the BFQ I/O scheduler [7] 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 [16].

Memory management can also reduce latency. Re-assigning pages from background apps to foreground apps can improve user experience of mobile operating systems [43]. Similarly, pre-swapping of unused memory can reduce delays by avoiding page reclamation latencies [44]. These schemes can reduce app launch times by timely provision of memory when it is under pressure.

VIII. 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.

REFERENCES

[1] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "Cloudcache: On-demand flash cache management for cloud computing," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 355–369.

[2] L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Improving virtualized storage performance with sky," in *Proc. 13th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2017, pp. 112–128.

[3] 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 *Proc. 11th IEEE Int. Memory Workshop*, 2019, pp. 1–4.

[4] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proc. 3rd Symp. Operating Syst. Des. Implementation*, 1999, pp. 1–4.

[5] 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.

[6] 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 *Proc. 17th IEEE Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 266–277.

[7] J. Corbet, "The BFQ I/O scheduler," 2014. [Online]. Available: https://lwn.net/Articles/601799/

[8] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: Exploiting disk layout and access history to enhance I/O prefetch," in *Proc. USENIX Annu. Tech. Conf.*, 2007, pp. 261–274.

[9] 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 *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 279–289.

[10] A. Eisenman et al., "Reducing DRAM footprint with NVM in facebook," in *Proc. 13th Eur. Conf. Comput. Syst.*, 2018, pp. 42:1–42:13.

[11] B. Esfahbod, "Preload–An adaptive prefetching daemon," Master's thesis, Graduate Dept. Comput. Sci., Univ. Toronto, 2006.

[12] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, pp. 75–84, 2008.

[13] 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 *Proc. 2018 USENIX Annu. Tech. Conf.*, 2018, pp. 15–28.

[14] 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 *Proc. 23 rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 71–83.

[15] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in *Proc. 10th Int. Conf. Mobile Syst. Appl. Serv.*, 2012, pp. 155–168.

[16] D. Jeong, Y. Lee, and J. Kim, "Boosting quasi-asynchronous I/O for better responsiveness in mobile devices," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 191–202.

[17] M. Russinovich, D. Solomon, and A. Lonescu, *Windows Internals, Part 2*, 6th Ed. Microsoft Press, 2012, pp. 324–350.

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

[19] Y. Joo, Y. Cho, K. Lee, and N. Chang, "Improving application launch times with hybrid disks," in *Proc. 7th IEEE/ACM Int. Conf. Hardware/Softw. Codesign System Synth.*, 2009, pp. 373–382.

[20] Y. Joo, J. Ryu, S. Park, H. Shin, and K. G. Shin, "Rapid prototyping and evaluation of intelligence functions of active storage devices," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2356–2368, Sep. 2014.

[21] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: Quick application launch on solid-state drives," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 259–272.

[22] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "Improving application launch performance on SSDs," *J. Comput. Sci. Technol.*, vol. 27, pp. 727–743, 2012.

[23] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 17.

[24] J. Ryu, Y. Joo, S. Park, H. Shin, and K. G. Shin, "Exploiting SSD parallelism to accelerate application launch on SSDs," *IET Electron. Lett.*, vol. 47, pp. 313–315, 2011.

[25] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proc. 9th ACM/IEEE Int. Conf. Embedded Softw.*, 2009, pp. 295–304.

[26] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou, "C-Miner: Mining block correlations in storage systems," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 173–186.

[27] 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 *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 383–400.

[28] 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 Trans. Storage*, vol. 4, 2008, Art. no. 4.

[29] D. T. Nguyen, "Improving smartphone responsiveness through I/O optimizations," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.: Adjunct Publication*, 2014, pp. 337–342.

[30] 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 *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2013, pp. 275–284.

[31] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. 15th ACM Symp. Operating Syst. Princ.*, 1995, pp. 79–95.

[32] W. Mauerer, *Professional Linux Kernel Architecture*. Hoboken, NJ, USA: Wrox Press, 2008.

[33] A. KirillShutemov, "mm: Map few pages around fault address if they are in page cache," 2014. [Online]. Available: https://lwn.net/Articles/588802/

[34] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing seek overhead with application-directed prefetching," in *Proc. 2009 USENIX Annu. Tech. Conf.*, 2009, pp. 299–312.

[35] Y. Won et al., "Barrier-enabled IO stack for flash storage," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 211–226.

[36] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proc. 10th Int. Conf. Mobile Syst. Appl. Serv.*, 2012, pp. 113–126.

[37] S. S. Hahn et al., "Improving file system performance of mobile storage systems using a decoupled defragmenter," in *Proc. 2017 USENIX Annu. Tech. Conf.*, 2017, pp. 759–771.

[38] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 1995, pp. 188–197.

[39] L. Colitti, "Analyzing and improving GNOME startup time," in *Proc. 5th Syst. Admin. Netw. Eng. Conf.*, 2006, pp. 1–11.

[40] M. Bhadkamkar et al., "BORG: Block-reORGanization for self-optimizing storage systems," in *Proc. 7th USENIX Conf. File Storage Technol.*, 2009, pp. 183–196.

[41] W. W. Hsu, A. J. Smith, and H. C. Young, "The automatic improvement of locality in storage systems," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 424–473, 2005.

[42] B. Hubert, "On faster application startup times: Cache stuffing, seek profiling, adaptive preloading," in *Proc. Ottawa Linux Symp.*, 2005, pp. 245–248.

[43] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the senseless killing: Improving memory management for mobile operating systems," in *Proc. 2020 USENIX Annu. Tech. Conf.*, 2020, pp. 873–887.

[44] Y. Liang et al., "Acclaim: Adaptive memory reclaim to improve user experience in android systems," in *Proc. 2020 USENIX Annu. Tech. Conf.*, 2020, pp. 897–910.

[45] BodNara, "ADATA ultimate SU630 960 GB," 2019. [Online]. Available: https://www.bodnara.co.kr/bbs/article.html?num=154114

[46] T. Schiesser, "Storage game loading test: PCIe 4.0 SSD vs. PCIe 3.0 vs. SATA vs. HDD," 2019. [Online]. Available: https://www.techspot.com/review/2116-storage-speed-game-loading

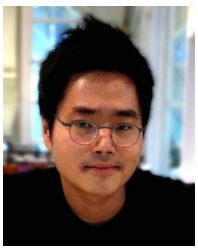
[47] T. Thomas, "Samsung's 860 QVO 1-TB SSD reviewed," 2018. [Online]. Available: https://techreport.com/review/34281/samsungs-860-qvo-1-tb-ssd-reviewed

[48] J. Axboe, "Explicit block device plugging," 2011. [Online]. Available: https://lwn.net/Articles/438256/

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

[50] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.
[51] , "Intel smart response technology," 2014. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/smart-response-technology.html
[52] "AMD StoreMI technology," 2021. [Online]. Available: https://www.amd.com/en/technologies/store-mi
[53] S. Sivaram and C. Bergey, "Zoned storage for the zettabyte age," 2019. [Online]. Available: https://www.flashmemorysummit.com/Proceedings2019/08-06-Tuesday/20190806_Keynote2_Western_Digital_Sivaram_Bergey.pdf
[54] J. Larus, "Spending moore's dividend," *Commun. ACM*, vol. 52, pp. 62–69, 2009.
[55] Boost Your, "PC's performance with solidigm synergy™ software," 2024. [Online]. Available: https://www.solidigm.com/products/client/synergy.html
[56] "Install ubuntu on a raspberry PI," 2022. [Online]. Available: https://ubuntu.com/download/raspberry-pi
[57] A. Nuñez-Unda, A. Vera, L. Haz, V. Pinos, R. Zurita, and S. Medina, "The raspberry pi as a computer substitute at elementary schools in developing countries: A pilot experiment in Ecuador," in *Proc. 22nd Int. Conf. Circuits, Systems, Commun. Comput.*, 2018, Art. no. 04023.
[58] S. Jeong, K. Lee, S. Lee, and S. Son, "Won I/O stack optimization for smartphones," in *Proc. 2013 USENIX Annu. Tech. Conf.*, 2013, pp. 309–320.
[59] Z. Ren, K. Doekemeijer, N. Tehrany, and A. Trivedi, "BFQ, Multiqueue-Deadline, or Kyber? Performance characterization of linux storage schedulers in the NVMe Era," in *Proc. 15th ACM/SPEC Int. Conf. Perform. Eng.*, 2024, pp. 154–165.

Junhee Ryu received the BS degree in computer engineering from Korea Aerospace University, Korea, in 2003, and the MS and PhD degrees in electrical engineering and computer science from Seoul National University, Korea, in 2005, and 2013, respectively. He is currently working with SK hynix as a principal engineer. His current research interests include operating systems, mobile systems, file systems, and real-time embedded systems.



Dongeun Lee received the BS degree in computer science and engineering and the PhD degree in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 2006 and 2014, respectively. He was a computer systems engineer with Lawrence Berkeley National Laboratory and also jointly affiliated as a postdoctoral research associate with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, Korea. He joined the Department of Computer Science, East Texas A& M University, in 2016, where he is currently an associate professor. His current research interest includes Big Data analytics for streaming data and scientific machine learning.

Kang G. Shin (Life Fellow, IEEE) received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is the Kevin and Nancy O'Connor professor of computer science and founding director of the Real-Time Computing Laboratory with the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan. At Michigan, he has supervised the completion of 93 PhDs and also chaired the Computer Science and Engineering Division with Michigan for three years starting 1991. From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. His current research focuses on QoS-sensitive computing and networks as well as on embedded real-time and cyber-physical systems. He has authored/coauthored more than 1000 technical articles (more than 350 of which are published in archival journals) and more than 60 patents or invention disclosures. He has co-authored (with C. M. Krishna) a textbook "Real-Time Systems," McGraw Hill, 1997. He has received numerous awards, including 2023 IEEE TCCPS Technical Achievement Award, 2023 SIGMOBILE Test-of-Time Award, 2019 Caspar Bowden Award for Outstanding Research in Privacy Enhancing Technologies, and best paper awards from 2023 VehicleSec, 2011 ACM International Conference on Mobile Computing and Networking (MobiCom'2011), the 2011 IEEE International Conference on Autonomic Computing, the 2010 & 2000 USENIX Annual Technical Conference, the 2003 IEEE IWQoS, and 1996 IEEE Real-Time Technology and Application Symposium. He also won the 2003 IEEE Communications Society William R. Bennett Prize Paper Award and the 1987 Outstanding IEEE Transactions on Automatic Control Paper Award. He has also received several institutional awards, including the Research Excellence Award, in 1989, Outstanding Achievement Award, in 1999, Service Excellence Award, in 2000, Distinguished Faculty Achievement Award, in 2001, and Stephen Attwood Award, in 2004 from The University of Michigan (the highest honor bestowed to Michigan Engineering faculty); a Distinguished Alumni Award of the College of Engineering, Seoul National University in 2002; 2003 IEEE RTC Technical Achievement Award; and 2006 Ho-Am Prize in Engineering. He is fellow of ACM, and overseas member of the Korean Academy of Engineering, served as the general co-chair for 2009 ACM Annual International Conference on Mobile Computing and Networking (MobiCom'09), was the general chair for 2008 IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'08), the 3rd ACM/USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys'05) and 2000 IEEE Real-Time Technology and Applications Symposium (RTAS'00), the program chair of the 1986 IEEE Real-Time Systems Symposium (RTSS), the general chair of the 1987 RTSS, a program co-chair for the 1992 International Conference on Parallel Processing, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991–1993, an editor of *IEEE Trans. on Parallel and Distributed Computing*, and an area editor of *International Journal of Time-Critical Computing Systems, Computer Networks*, and *ACM Transactions on Embedded Systems*. He has also served or is serving on numerous government committees, such as the US NSF Cyber-Physical Systems Executive Committee and the Korean Government R& D Strategy Advisory Committee. He was a co-founder of two startups and is serving as an executive advisor for Samsung Research.

Kyungtae Kang (Member, IEEE) received the BS degree in computer science and engineering, and the MS and PhD degrees in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 1999, 2001, and 2007, respectively. From 2008 to 2010, he was a postdoctoral research associate with the University of Illinois at Urbana-Champaign. In 2011, he joined the Department of Computer Science and Engineering, Hanyang University, Korea, where he is currently a tenured professor with the Department of Artificial Intelligence. His research interests lie primarily in systems, including operating systems, mobile systems, distributed systems, and real-time embedded systems. His recent research interest is in the interdisciplinary area of cyber-physical systems.