ClusterFetch: A Lightweight Prefetcher for Intensive Disk Reads

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

Abstract—By overlapping disk accesses with computation-intensive operations, prefetching can reduce delays in launching an application and in loading significant amounts of data while the application is running. The key to effective prefetching is making the tradeoff between the mining accuracy of selecting relevant blocks, and the time to decide those blocks. To address this problem, we propose a new prefetcher called *ClusterFetch*. In its learning mode, ClusterFetch detects periods of intensive disk accesses by monitoring the speed at which read requests are queued; it re-organizes these reads and locates the file opened by the application just before each such period. During subsequent runs of the same application, ClusterFetch prefetches the data associated with the opening of a "trigger" file. Our experimental results show that ClusterFetch implemented in Linux can reduce the application launch time by up to 41.3 percent and the loading time by up to 38.2 percent, while taking up less than 200 KB of main memory.

Index Terms—Disk prefetching, disk read bursts detection, quick application loading, user-perceived latency improvement

1 Introduction

Application launch times, and the delays when a running application has to load additional data from disk, which we call 'loading times', strongly affect the satisfaction of PC users [1], [2], [3], [4], [5], [6], [7]. Since the secondary storage devices in a PC, hard disk drives (HDDs) or solid-state disks (SSDs), are much slower than the main memory and CPU, they become bottlenecks. In particular, long delays are inevitable during a cold start when an application runs for the first time after the OS has been rebooted, where all blocks must be fetched from disk. A cold start also occurs if all the blocks needed by an application have been evicted from the disk cache. In the Linux kernel, this cache is in main memory and eviction is performed by the memory management subsystem [8].

Prefetching disk blocks into the disk cache can mitigate disk access latency so that applications launch more quickly [1], [4]. Ideally, a prefetcher should exactly predict the disk blocks about to be required, and transfer these blocks from secondary storage to the disk cache while the CPU is processing data from previously fetched blocks. This ideal cannot be realized; nevertheless, efficient prefetching can arrange for most blocks to be ready for the CPU in the disk cache, thus mitigating the delay associated with accessing a disk.

The accuracy of a prefetching scheme comes at an expense in processing time and memory. For instance, C-Miner [9] analyzes frequently occurring block access sequences by employing a data mining technique based on CloSpan [10]; this technique estimates disk block correlations conservatively, so that only blocks that are very likely to be accessed are prefetched. This restricts the extent to which C-Miner can improve performance, despite its low processing and memory overheads. For example, blocks of library data are often required by many applications, and the need for these blocks may well be correlated with different block access sequences in each application. Such blocks are not identified by C-Miner.

DiskSeen [11] is a prefetching scheme that analyzes the temporal relationship between accesses to pairs of blocks with adjacent logical block numbers (LBNs). This is effective, but it requires table of block correlations that occupies 4 GB of main memory for a 1TB HDD (assuming each disk block can have up to four correlations), and this requirement rises linearly with disk size.

There are also very different schemes, such as Prefetch [12], which was selected for the Google Summer of Code (GSoC) 2007, and Preload [13]. Both of these schemes are specifically designed to reduce application launch times, and require a clear signal that an application is being launched (e.g., a function call for a binary loader or the creation of an entry in the `proc` file system for a new application). These schemes have a low overhead, but they cannot expedite the loading of blocks after an application has launched. This is particularly significant for games in which the user moves through an environment that must be frequently updated. A comparison of ClusterFetch with existing prefetchers is summarized in Table 1.

ClusterFetch is a prefetcher which expedites both launching and subsequent loading without the need for a large table of correlations: this makes it suitable for applications like games which run on a PC but which also need to load significant amounts of data after the launch. The design of ClusterFetch is based on the observation that data is usually loaded in bursts, and each burst is associated with the opening of many different files. Therefore ClusterFetch is designed to recognize bursts of I/O and not to search for individual block correlations. It detects these bursts by counting the frequency of read operations by logging them in a circular queue. This approach has a low CPU and memory overhead. And we identify each burst of input by linking it to a trigger file, which is opened just before that burst occurs. ClusterFetch then prefetches the same blocks when the same file is opened in a subsequent run. An example sequence of bursts and the trigger files associated with them for the Savage 2 game can be seen in Fig. 1.

ClusterFetch tries to identify the file opening event that is best correlated with a particular burst of input, and no others. We therefore determine what types of files are commonly opened during many I/O bursts, and prevent them from becoming trigger files. The remaining types of files are eligible for trigger files, and the four eligible files which are opened most immediately preceding a burst of input can be considered for the *trigger files* associated with that burst. If a file is associated with another burst, then it is discarded and the next candidate is considered.

ClusterFetch runs within the Linux kernel and requires only 200 KB of memory to store correlations between disk blocks. It reduces application launch times by up to 41.3 percent and application loading times by up to 38.2 percent.

The rest of this paper is organized as follows. In Section 2 we discuss trigger file selection and the recognition of bursts of I/O. Section 3 describes the overall framework of ClusterFetch and its major components: the I/O miner, the prefetcher, and the I/O prioritizer. In Section 4 we evaluate the performance of ClusterFetch with 15 launch and loading scenarios on a Linux desktop equipped with HDDs or SSDs, and we conclude the paper in Section 5.

2 Block Mining

2.1 Trigger File Candidates

The ClusterFetch algorithm relies on the selection of suitable trigger files. Unsuitable files include those regularly opened by the Linux daemon, even when the system is idle, and those used by several different programs. We empirically examined file opening

- *J. Ryu and K. Kang are with the Department of Computer Science & Engineering, Hanyang University, Ansan 15588, Korea. E-mail: {junhee, ktkang}@hanyang.ac.kr.*
- *D. Lee is with the Department of Computer Science, Texas A&M University – Commerce, Commerce, TX 75428. E-mail: dongeun.lee@tamuc.edu.*
- *K. G. Shin is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: kgshin@umich.edu.*

Manuscript received 22 Feb. 2017; revised 30 July 2017; accepted 28 Aug. 2017. Date of publication 3 Sept. 2017; date of current version 22 Jan. 2018.
(Corresponding authors: Kyungtae Kang and Dongeun Lee.)
Recommended for acceptance by P. Gratz.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2748939

TABLE 1
Comparison of Alternative Prefetching Schemes

Scheme	Scenario	Mining accuracy	I/O optimization	Memory overhead
C-Miner [9]	All cases	Low	X	Low
DiskSeen [11]	All cases	Adjustable	LBN sorting	High
GSoC Prefetch [12]	Launch	Mid	File-level sorting	Low
Preload [13]	Launch	Mid	X	Low
Application-directed prefetching [14]	All cases	Mid	LBN sorting	Low
ClusterFetch	I/O bursts (launch and loading)	High	LBN sorting & I/O plugging	Low

TABLE 2
Analysis of Files Opened During Launch and Loading of Sample Applications

Applications	Unique files	Special files	Files opened by windows manager	Shared library files	Trigger file candidates
Eclipse (launch)	980	33	179	140	628
Firefox (launch)	1,065	401	160	162	342
OOWriter (launch)	1,303	52	399	255	597
OOImpress (launch)	645	25	150	247	223
FlightGear (launch)	1,418	93	20	112	1,193
Savage 2 (launch)	210	41	11	84	74
Savage 2 (loading 1)	138	40	5	6	87
Savage 2 (loading 2)	159	39	8	9	103
0 A.D. (launch)	684	68	160	126	330
0 A.D. (loading)	78	12	6	8	52
BFW (launch)	719	32	126	105	456
BFW (loading 1)	427	27	28	3	369
BFW (loading 2)	354	19	28	3	304
TORCS (launch)	220	44	9	95	72
TORCS (loading)	279	26	5	9	239

patterns for 15 scenarios involving nine applications, with the results summarized in Table 2. Here, the column "unique files" represents the number of files uniquely opened in each scenario.

We can group files that are unlikely to be of use as trigger files into three categories as follows. *Special files* (e.g., device files and pseudo files) are unlikely to be related to a specific application. These files tend to be opened by the Linux kernel or daemon. For instance, `/proc/interrupts` and `/proc/irq/<CPU #>/smp_affinity` files are opened periodically by the interrupt load balancing daemon. We found that these files were opened by all the applications in our test (several times in some scenarios) and might also be accessed when the system is idle. ClusterFetch therefore ignores file paths with `/proc/` or `/sys/` prefixes.

Files opened by the windows manager occur when an application runs on X-Windows. Typical examples include icon and font files: ClusterFetch detects these files by checking whether their path contains a reference to 'icons' (e.g., `/opt/Savage2/s2icon.png` and `/usr/share/icons/hicolor/16x16/apps/evince.png`), 'desktop' (e.g., `/usr/share/applications/gnubg.desktop`), or 'font' (e.g., `/opt/Savage2/game/core/fonts/system.ttf` and `/usr/share/fonts/DejaVuSans.ttf`).

Shared library files are used by many applications. In the 15 test scenarios, a total of 535 shared library files were opened, among which 327 were opened by more than two scenarios and 125 were opened by more than five. This is shown in Fig. 2. ClusterFetch excludes this type of file by checking whether a file path contains `.so.` or ends with the `.so` extension. However, `/etc/ld.so.cache` file, which provides a reference from the name of a shared object to its full path, is not a shared library file, although it has `.so.` in its name. It was opened in all of the test scenarios, and thus should not be regarded as a trigger file candidate.

2.2 Log Management

The core data structure of ClusterFetch, as shown in Fig. 3, is a circular queue which is used to log all disk I/Os and file openings. When a disk I/O occurs *or* a file opens, ClusterFetch calls the `add_log` function described in Algorithm 1 to create a new entry in the log. This function first tries to merge the new log entry with the previous entry (Line 2), if they both refer to disk I/O for the same file and their block chunks are adjacent. If it succeeds, then

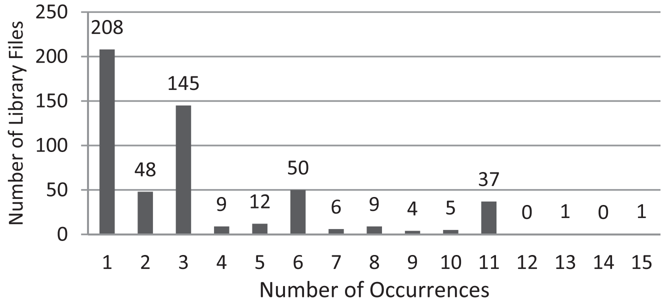

Fig. 2. Histogram of the number of occurrences of shared library files over different test scenarios of application launch and loading.

`add_log` returns immediately. This was successful for an average of 37 percent of disk I/O entries in the 15 benchmark scenarios.

Then ClusterFetch invokes `handle_IO_burst` at Line 6 to detect I/O burst when the circular queue is full. (See Algorithm 2.) When ClusterFetch detects a burst of disk I/O, it logs the disk blocks that are being read.

Finally, a new entry is appended to the queue (Lines 8 $\sim$ 30). However, a disk I/O entry is discarded if there is no preceding file-opening entry (Lines 15 $\sim$ 17). ClusterFetch only inserts an entry for file opening into the queue when the file path meets the criteria for trigger file candidates. ClusterFetch guarantees that the IOPS (I/O operations per second) between the first two file open logs meets a predefined threshold `iops_threshold` used to check I/O burstness (Lines 24 $\sim$ 28). The idea behind this is to prevent a file opened during non-burst periods from being used as a trigger file for the following I/O burst.

2.3 I/O Burst Detection

`add_log` function invokes `handle_IO_burst` function in Algorithm 2 to detect two kinds of I/O burst when the circular

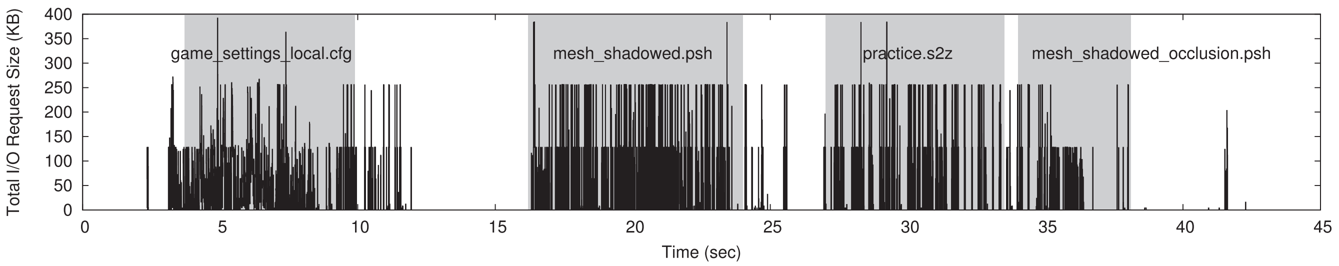

Fig. 1. Monitoring bursts of disk I/O for the game savage 2. The y-axis represents the size of I/O requests, and the x-axis is the time at which they are dispatched to the device driver. Each burst of I/O is labeled with the name of a file that was opened just before that burst took place.

queue is full: 1) *Full I/O burst*: whole range of I/O logs in the circular queue that meets the requirement of `iops_threshold`, 2) *Partial I/O burst*: the longest partial set of I/O logs that meets the requirements of `length_threshold` and `iops_threshold`. In our experimental settings using HDDs, the minimal number of I/O logs during 15 benchmark scenarios was 192 (loading of 0 A.D.), and the minimal IOPS was 106 (loading of BFW). Thus we empirically set these two thresholds to 100. We believe that an appropriate `iops_threshold` value depends on the performance of CPU and storage, and this value could be guided by an additional benchmark program that simulates a set of launch and loading scenarios.

Algorithm 1. ClusterFetch I/O mining algorithm

```
1:  function add_log (new_log, log_type)
2:    if try_to_merge_with_the_last(new_log) = true then
3:      return
4:    end if
5:    if queue_is_full() = true then
6:      handle_IO_burst()
7:    end if
8:    if log_type = TYPE_DISK_IO then
9:      if front_idx = rear_idx then
10:       return
11:     end if
12:     rear_idx ← (rear_idx + 1) mod queue_size
13:     insert_io_log_into_queue(new_log)
14:   else if log_type = TYPE_FOPEN then
15:     if is_trigger_candidate(new_log) = false then
16:       return
17:     end if
18:     rear_idx ← (rear_idx + 1) mod queue_size
19:     insert_fileopen_log_into_queue(new_log)
20:     queue[rear_idx].next_open_idx ← INVALID
21:     if last_open_idx ≠ INVALID then
22:       queue[last_open_idx].next_open_idx = rear_idx
23:     end if
24:     if last_open_idx = front_idx then
25:       if meet_iops(front_idx,rear_idx) = false then
26:         front_idx ← rear_idx
27:       end if
28:     end if
29:     last_open_idx ← rear_idx
30:   end if
31: end function
32: function meet_iopsstart_idx, end_idx
33:   IO_count ← item_count(start_idx, end_idx)
34:   duration ← queue[end_idx].ts − queue[start_idx].ts
35:   if (IO_count / duration) ≥ iops_threshold then
36:     return true
37:   else
38:     return false
39:   end if
40: end function
41: function item_countstart_idx, end_idx
42:   if end_idx > start_idx then
43:     return end_idx − start_idx + 1
44:   else
45:     return queue_size − start_idx + end_idx + 1
46:   end if
47: end function
```

At Line 10, ClusterFetch checks if the difference between the times when the first and the last entries in the circular queue were logged is shorter than the threshold `burst_threshold` computed at Line 8. If that is the case, ClusterFetch decides that these entries correspond to a full burst of disk I/O, and invokes `write_full_logs_to_disk` function (Line 11) to store the IDs of the corresponding disk blocks in a *prefetch information file* that is later used to prefetch these blocks.

Algorithm 2. I/O Burst Detection Algorithm

```
1:  function handle_IO_burst
2:    if handle_full_burst() = false then
3:      if handle_partial_burst() = false then
4:        front_idx ← queue[front_idx].next_open_idx
5:      end if
6:    end if
7:  end function
8:  burst_threshold ← queue_size / iops_threshold
9:  function handle_full_burst
10:   if (queue[rear_idx].ts − queue[front_idx].ts) ≤ burst_-
      threshold then
11:     write_full_logs_to_disk()
12:     front_idx ← 0
13:     rear_idx ← 0
14:     last_open_idx ← INVALID
15:     return true
16:   else
17:     return false
18:   end if
19: end function
20: function handle_partial_burst
21:   start_idx ← front_idx
22:   end_idx ← queue[start_idx].next_open_idx
23:   burst_found ← false
24:   while end_idx ≠ INVALID do
25:     if meet_length(start_idx, end_idx) = true then
26:       if meet_iops(start_idx, end_idx) = true then
27:         burst_found ← true
28:         break
29:       else
30:         start_idx ← queue[start_idx].next_open_idx
31:         continue
32:       end if
33:     end if
34:     end_idx ← queue[end_idx].next_open_idx
35:   end while
36:   if burst_found = false then
37:     front_idx ← 0
38:     rear_idx ← 0
39:     last_open_idx ← INVALID
40:     return false
41:   end if
42:   longest_burst_end_idx ← end_idx
43:   end_idx ← queue[end_idx].next_open_idx
44:   while end_idx ≠ INVALID do
45:     if meet_iops(start_idx, end_idx) = true then
46:       longest_burst_end_idx ← end_idx
47:       end_idx ← queue[end_idx].next_open_idx
48:     else
49:       break
50:     end if
51:   end while
52:   write_partial_logs_to_disk_and_reset_idx(start_idx,
      longest_burst_end_idx)
53:   return true
54: end function
55: functionmeet_lengthstart_idx, end_idx
56:   if item_count(start_idx, end_idx) ≥ length_threshold
      then
57:     return true
58:   else
59:     return false
60:   end if
61: end function
```

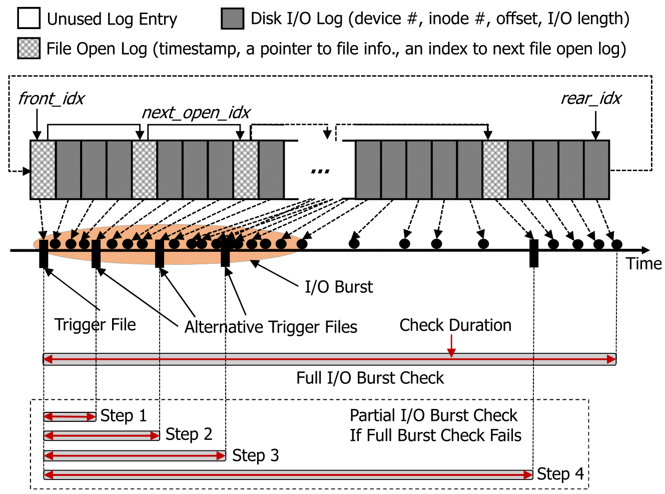
Unused Log Entry
Disk I/O Log (device #, inode #, offset, I/O length)
File Open Log (timestamp, a pointer to file info., an index to next file open log)
front_idx
next_open_idx
rear_idx
...
Time
I/O Burst
Trigger File
Alternative Trigger Files
Check Duration
Full I/O Burst Check
Step 1
Step 2
Step 3
Step 4
Partial I/O Burst Check
If Full Burst Check Fails

Fig. 3. Data structure for I/O burst detection based on I/O intensiveness.

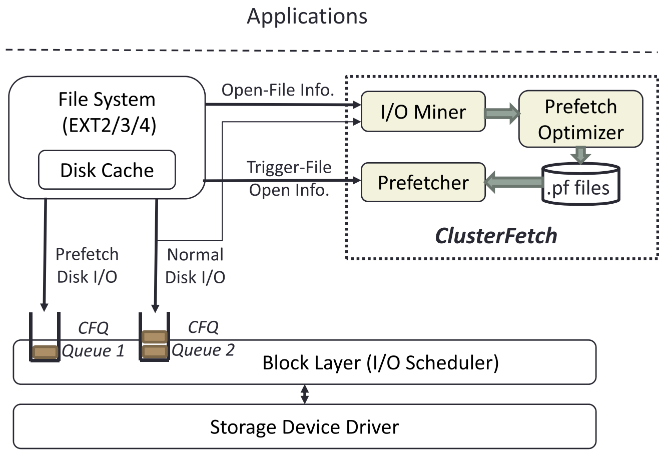

Fig. 4. Interface between ClusterFetch and the Linux kernel.

Analyzing IOPS averaged over the queue might lose intensive bursts of I/O that occur within a short interval. The seriousness of this problem depends on the length of the queue. ClusterFetch, therefore, checks for I/O bursts occurring between adjacent logs of file openings, using the thresholds mentioned above, and finds the longest period of block I/O that meets both `iops_threshold` and `length_threshold`. To allow file-opening entries to be traversed quickly, we use `next_open_idx` to construct a linked list of these entries. As described in Algorithm 2, `handle_partial_burst` function consists of two steps to detect the partial I/O burst: the first step is to find the shortest I/O burst that meets both thresholds (Lines 21 $\sim$ 35), and the second step is to find the longest duration that still meets both thresholds (Lines 42 $\sim$ 51).

Functions `write_full_logs_to_disk` (Line 11) and `write_partial_logs_to_disk_and_reset_idx` (Line 52) record I/O bursts to the corresponding prefetch information files. The latter function additionally adjust `front_idx`, `rear_idx`, and `last_open_idx` to meet the `iops_threshold` requirement between the first two file open logs in the queue. At the same time, ClusterFetch selects a trigger file (the first file opened in the circular queue) and three alternatives (three files opened next to the trigger file). When that trigger file is opened again, ClusterFetch prefetches the same disk blocks into the disk cache.

By limiting the accuracy with which it analyzes the correlations among disk blocks, we limit the space required by ClusterFetch. It only requires 144 KB of main memory to store a circular queue with 6,000 entries, and its total memory requirement is less than 200 KB.

2.4 Discussion

The correlation of chosen trigger files with subsequent disk reads determines the effectiveness of ClusterFetch. ClusterFetch basically considers four trigger file candidates explained in Section 2.1 and selects one among them that is the most likely to be the trigger file associated with the burst.

In a PC environment, the input log of ClusterFetch can be corrupted by data relating to multiple applications. To address this issue, ClusterFetch waits 10 seconds after prefetching disk blocks, and checks if the prefetched blocks are actually read by any application. If the proportion of the prefetched blocks which are accessed after this period is less than 90 percent, ClusterFetch deletes a corresponding prefetch information file. This can be easily done because the Linux kernel traces the reference state of disk cache pages. The time required to check this state was 132 μs for the Eclipse launch, during which 3,812 I/Os were performed.

As mentioned earlier, games are one of the primary target applications of ClusterFetch. During a game play, we believe that the aforementioned issue rarely occurs: users may not run a game concurrently with other applications, or those applications would be mostly in idle states even if they do.

Oftentimes, the I/O pattern of an application could be random. For example, the Linux copy program (`cp`) and the file search program (`find`) issue disk reads to read from a source file, which varies from run to run. However, checking the access of prefetched blocks after 10 seconds cannot be a fundamental solution in this case. A possible workaround for this could be an adoption of the blacklist [12] that specifies applications where monitoring and prefetching of disk I/Os should be disabled.

3 ClusterFetch Framework

ClusterFetch consists of three main components: an I/O miner, a prefetch optimizer, and a prefetcher. Fig. 4 shows how ClusterFetch is interfaced with the Linux kernel.

As discussed in the previous section, ClusterFetch monitors bursts of reads and logs the corresponding disk blocks. Then, it traces files that were opened before the burst. When the same file is opened again, ClusterFetch prefetches the associated disk blocks into the disk cache to reduce access times.

ClusterFetch logs disk reads when misses occur in either of two distinct disk caches used by the Linux kernel.[1] ClusterFetch is currently implemented with the Ext4 file-system [15], in which the buffer cache contains metadata blocks such as inode table blocks and extent blocks. The disk blocks involved in intense periods of reading can be identified using `ext4_readpage(s)` for regular files and `submit_bh` for metadata blocks [8], [15].

When a burst of I/O is identified, ClusterFetch links the trigger file to the entries in the prefetch information file, and sets the *sticky permission bit* of the trigger file. In current versions of Linux, the sticky permission bit is only meaningful in directory files, allowing it to be used to label regular files as trigger files. This avoids the need for lookup. The `sys_open` system call is modified to check the sticky permission bit of each file being opened. Thus, when the Linux kernel opens a trigger file, the corresponding disk blocks recorded in the prefetch information file are brought into the disk cache.

Each entry for the I/O log contains the following data: a device number (4 bytes), an inode number (8 bytes), an offset (8 bytes), and the amount of I/O (4 bytes). The inode number of a metadata block is set to 0. The size of each entry in the prefetching table is 24 bytes (on a 64-bit machine). Each entry in the file-open log contains a pointer to the next entry (8 bytes), a timestamp for the entry (8 bytes), a pointer (8 bytes) to additional information that includes the file name (dynamically allocated), and the inode number

1. The *page cache* contains data blocks for regular files, and the *buffer cache* contains disk blocks for block devices.

(8 bytes) of the opened file. The use of pointers allows the static allocation of memory space for the log regardless of the types of files that will be logged. In addition, a prefetch entry in the prefetch information file stores information for prefetching disk blocks.

3.1 Prefetch Optimizer

3.1.1 Optimization on HDDs

When writing logs of an I/O burst into disks, ClusterFetch optimizes the order in which block numbers are recorded to reduce the head movement required from an HDD. Reads are sorted by device number and LBN, and adjacent reads are merged. Reordering processes of this sort are not unusual: existing file-level prefetchers [12], [16] for HDDs use file IDs and offsets as sort keys. However, file-level ordering is not ideal for prefetching because the result is not identical to the LBN order [17]. Further, in Linux, metadata blocks in Linux are not related to a file ID (represented as an inode number), so file-level I/O sorting cannot order metadata blocks correctly at the same time as file blocks (also known as data blocks). To address this issue, ClusterFetch extracts the LBNs from read requests, and sorts the requests by LBN. In our experiments, LBN sorting outperformed file-level sorting by 14.2 percent in terms of prefetching time.

Merging also reduces the log size, and can allow a prefetcher to issue more efficient read requests. In our experiments, we found numerous periods during which there were multiple independent reads with sequential LBNs, but they were often wrongly ordered and mixed up with other blocks. To handle this issue, ClusterFetch merges reads with adjacent LBNs after they have been sorted by LBN. This reduced the size of the log by 11.5 percent on average, over 15 test scenarios. After scheduling, the entries in the log file are stored as a prefetch information (`.pf`) file in the `/clusterfetch` directory; the filename includes both the device and inode number, making it unique within the system.

3.1.2 Optimization on SSDs

ClusterFetch does not perform I/O sorting on SSDs, where I/O sorting has little impact on throughput, thus preserving original sequence of I/O logs. When logging each disk I/O to the mining queue, ClusterFetch tries to merge the I/O log with the previous one. This reduced the size of the log by 2.8 percent and prefetching time by 0.3 percent on average.

3.2 Prefetcher

The native command queue (NCQ) was introduced in the Serial ATA (SATA) II interface specification to improve I/O throughput [18], [19], [20], [21]. NCQ allows HDDs to accept up to 32 commands in advance, and to reorder them to reduce disk head movements. In SSDs, NCQ enables the controller to parallelize commands [18], [19]. ClusterFetch uses the NCQ for asynchronous prefetching of both metadata and data blocks, by filling NCQ without waiting for the completion of previous reads. In the 15 benchmarks on HDDs, using NCQ reduced the launching/loading times by 4.7 percent on average.

In Linux, each metadata block is separately dispatched to the device driver without explicit I/O plugging [22]. However, ClusterFetch uses I/O plugging to merge contiguous metadata requests into a single request that is then delivered to the dispatch queue. This reduced launching/loading times further by 2.7 percent on average in the 15 test scenarios. To prevent a burst of I/O which occurs during the prefetching process itself from being detected as a burst of application I/O, ClusterFetch does not monitor disk I/O while it is actually prefetching blocks.

Notably, duplicate I/O bursts can be detected by the prefetcher, and therefore trigger files can also conflict. In order to prevent this, ClusterFetch do not log file openings and disk I/Os during its prefetching operations.

3.3 I/O Control

To reduce the impact of ClusterFetch on other system I/O, the I/O control processes shown in Fig. 4 balance the priority of prefetching against other disk block requests. It is possible to preset an upper limit on the proportion of the disk bandwidth that ClusterFetch is allowed use for prefetching, thus preventing significant delays to the disk I/O of other processes.

As long as general disk I/O is not hindered, ClusterFetch need not delay prefetching. The I/O control process examines the depth of the NCQ to see if there is room for prefetching. If the queue is short, ClusterFetch can perform prefetching immediately without impacting other I/O.

The I/O control process is based on the *CFQ (Completely Fair Queuing) I/O scheduler* [23] and also employs the Linux *blkio* controller, which controls the time-slots allocated to disk I/O using weights attached to user-defined process groups of processes (*cgroups*). Basically, disk I/O requests related to prefetching by ClusterFetch are held in an I/O scheduling queue in the block layer; this is different from the queue of general disk I/O requests, as shown in Fig. 4. However, if the disk load from normal processes is low, the disk I/O requests related to prefetching can be queued together with other disk I/O requests to maximize throughput. The I/O control process eliminates prefetching requests from the queue after a timeout period, since there is no point in 'prefetching' blocks that have already been read directly by the application in response to a cache miss.

4 Experimental Results

4.1 Evaluation on HDDs

We evaluated ClusterFetch on a desktop computer with an Intel Core i3-2100 CPU, 4 GB of RAM, and a Seagate 3.5-inch, 500 GB, 7200 RPM HDD, running Fedora 17 64-bit Linux, with an Ext4 filesystem.

No prefetching method can outperform a warm start, when all the disk blocks required by an application already reside in the disk cache. Conversely, any prefetcher should be able to speed up a cold start, when all the disk blocks have to be accessed directly from the disk.

We compared the application launch and loading times of nine popular free applications: Eclipse (development tool), Firefox (web browser), OOWriter (LibreOffice word processor), OOImpress (LibreOffice presentation tool), FlightGear (flight simulator), Savage 2 (game), 0 A.D. (game), Battle for Wesnoth (game), and The Open Racing Car Simulator (motorsport simulator).

Table 3 shows the effect of ClusterFetch on the launch and loading times of these applications, relative to a cold start with no prefetcher: the greatest reductions were 41.3 percent for launching and 38.2 percent for loading Battle for Wesnoth. Overall, the performance of ClusterFetch is around the midpoint between a warm start and a cold start with no prefetching.

Fig. 5 shows how the number of commands in the NCQ varies during the launch of FlightGear, without and with ClusterFetch, which reduces the cold-start launch time of 27.5 to 23.3 seconds. We can see from this figure that ClusterFetch is making a significant use of the NCQ: 1) there are more outstanding commands in the NCQ because read requests have been issued asynchronously; and 2) the total amount of data represented by the commands in the NCQ has been increased significantly by merging adjacent I/O requests into a single large request.

Fig. 6 shows the amount of data requested during the same launch: without a prefetcher, the sum of outstanding data requests never exceeds 500 KB; when ClusterFetch merges I/O requests, this rises to 10 MB. ClusterFetch detected three partial I/O bursts during the launch of FlightGear, consisting of 1624, 683, and 1029 read requests; these figures drop to 744, 221, and 644 after merging.

Loading produces similar results: during the period in which Savage 2 is loading its game map, ClusterFetch detected three

TABLE 3
Cold and Warm Start Application Launch and Loading Times in Comparison with ClusterFetch

Applications (stage)	Cold start on HDD (s)	Warm start (s)	ClusterFetch on HDD (s)	Reduction on HDD (%)	Cold start on SSD (s)	ClusterFetch on SSD (s)	Reduction on SSD (%)
Eclipse (launch)	16.5	6.1	11.4	30.9	8.8	6.4	27.3
Firefox (launch)	7.4	1.9	5.3	28.4	2.7	2.2	18.5
OOWriter (launch)	8.2	2.1	5.7	30.5	3.1	2.4	22.6
OOImpress (launch)	7.8	1.5	5.4	30.8	2.8	2.3	17.9
FlightGear (launch)	27.5	18.9	23.3	15.2	22.4	20.2	9.8
Savage 2 (launch)	20.1	13.6	16.4	18.4	16.1	15.2	5.6
Savage 2 (loading 1)	22.0	17.0	18.5	15.9	19.1	18.2	4.7
Savage 2 (loading 2)	16.3	10.2	12.8	21.5	12.5	11.6	7.2
0 A.D. (launch)	6.0	1.6	4.8	19.6	3.0	2.5	16.7
0 A.D. (loading 1)	8.3	3.6	6.2	23.6	4.4	3.8	13.6
0 A.D. (loading 2)	7.8	4.2	6.6	15.4	5.1	4.6	9.8
BFW (launch)	7.6	1.3	4.3	41.3	2.9	2.2	24.1
BFW (loading 1)	5.5	1.5	3.4	38.2	2.3	2.3	0
BFW (loading 2)	6.1	1.6	4.2	31.1	2.2	2.1	4.5
TORCS (launch)	6.2	1.1	5.0	19.4	1.6	1.5	6.2
TORCS (loading)	6.7	2.7	4.8	25.9	3.0	2.9	3.3

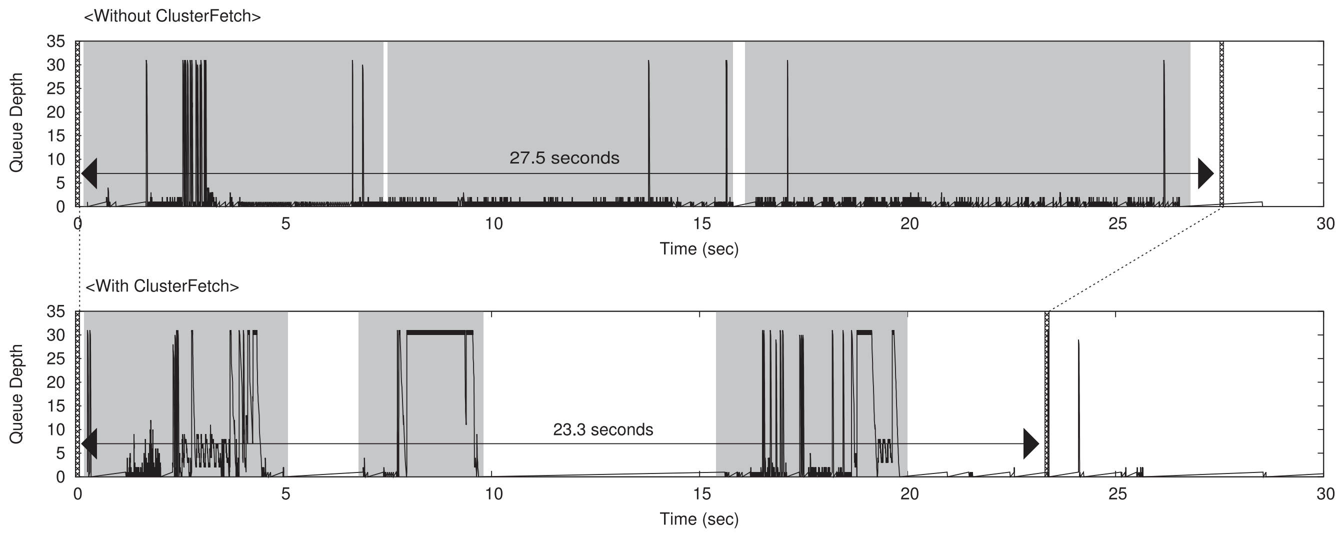

Fig. 5. Variation in the number of outstanding commands in the NCQ during the launch of FlightGear. The maximum number of outstanding I/O requests rises from 3 to 31 when ClusterFetch is used. Launch time is reduced from 27.5 to 23.3 seconds.

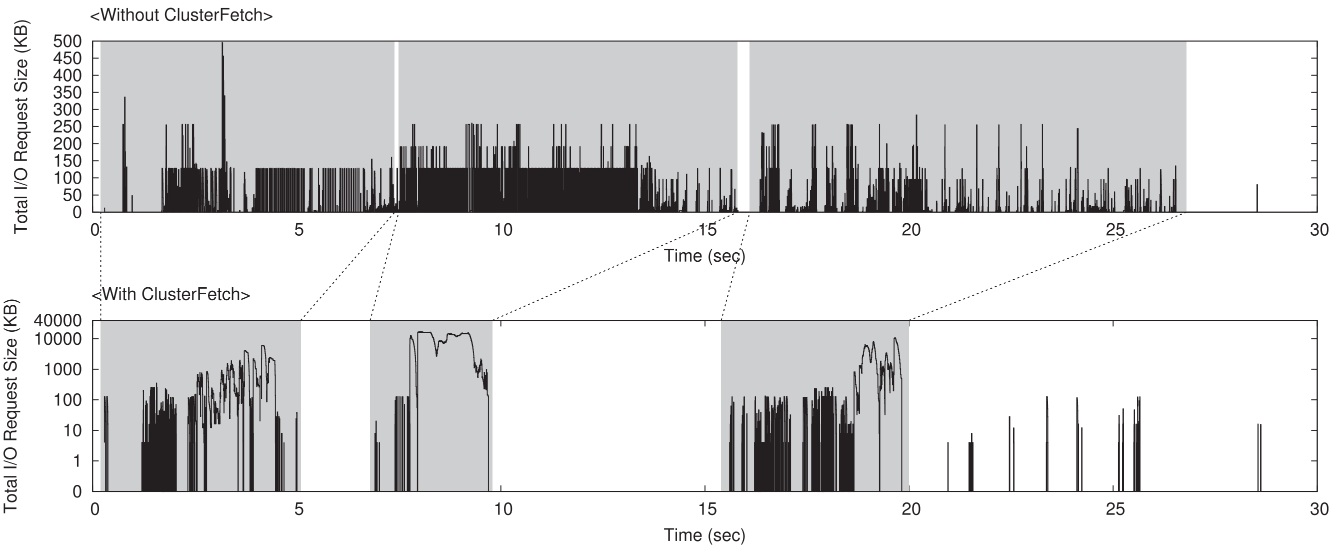

Fig. 6. Variation in amount of data being requested by FlightGear during launch. Many requests for less than 500 KB are combined into large requests for around 10 MB. Note that the y-axis in the lower plot is logarithmic.

partial I/O bursts and generated three trigger files. The loading time was reduced from 22.0 to 18.5 seconds. Again, disk block requests were issued asynchronously, which increases the maximum number of commands in the NCQ from 3 to 31. In addition, by combining adjacent I/O requests into a single large request, the maximum amount of data represented by outstanding I/O requests increased from less than 400 KB to around 10 MB. The number of I/O requests issued during each of the three partial I/O bursts was 381, 257, and 148; these figures drop to 141, 96, and 49, respectively, after merging requests.

Remarkably, launch and loading of an application under heavy CPU and disk I/O loads benefit from disk prefetching [1], [7]. This is mainly because desktop workload tends to activate only one CPU core or a single disk I/O command exclusively. In this situation, the prefetcher thread reduces the CPU time for disk I/O processing, and it also exploits the parallelism of disk I/Os and reduces prefetching time by LBN sorting for disk I/Os. We also measured the launch time of FlightGear while copying a large file, which issues 128 KB reads and writes during the launch. The cold start without and with ClusterFetch was 39.4 seconds and 36.5 seconds, respectively.

4.2 Evaluation on SSDs

We replaced the HDD in our PC with an Intel 520 series 120 GB SSD, which significantly reduced both launch and loading times, as shown in Table 3. This leaves little room for optimization through prefetching. ClusterFetch reduced launch and loading times on the SSD by 12.0 percent on average; but the average time saving in time was only 0.7 seconds.

5 Conclusion

We have developed a lightweight prefetcher called *ClusterFetch* that reduces both application launch and loading times on a Linux PC. ClusterFetch uses a disk block mining algorithm based on a circular queue, and only requires 200 KB of memory. Our experiments on HDD with nine popular applications showed reductions in launch times of between 15.2 percent and 41.3 percent, and in application loading times of between 15.4 and 38.2 percent.

Acknowledgment

This research was supported by the Basic Science Research Program and the Information Computing Development Program, through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2016R1D1A1B03935831; NRF-2017R1A2B4007970; NRF-2017M3C4A7065980). This research was also supported by an Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean government (MSIT; No. B0101-15-0557, Resilient Cyber-Physical Systems Research).

References

[1] 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.*, Feb. 2011, pp. 259–272.

[2] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. 10th USENIX Conf. File Storage Technol.*, Feb. 2012, pp. 209–222.

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

[4] 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. Ser.*, Jun. 2012, pp. 113–126.

[5] A. Parate, M. Bőhmer, D. Chu, D. Ganesan, and B. Marlin, "Practical prediction and prefetch for faster access to applications on mobile phones," in *Proc. 2013 ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, Sep. 2013, pp. 275–284.

[6] A. Bovenzi, J. Alonso, H. Yamada, S. Russo, and K. S. Trivedi, "Towards fast OS rejuvenation: An experimental evaluation of fast OS reboot techniques," in *Proc. 24th IEEE Int. Symp. Softw. Reliability Eng.*, Nov. 2013, pp. 61–70.

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

[8] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. Newton, MA, USA: O'Reilly & Associates, 2005.

[9] 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.*, Mar./Apr. 2004, pp. 173–186.

[10] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining closed sequential patterns in large datasets," in *Proc. 3rd SIAM Int. Conf. Data Mining*, May 2003, pp. 166–177.

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

[12] K. Lichota. Prefetch: Linux solution for prefetching necessary data during application and system startup. [Online]. Available: https://code.google.com/archive/p/prefetch/

[13] B. Esfahbod, "Preload—An Adaptive Prefetching Daemon," Master's thesis, Graduate Department of Computer Science, University of Toronto, Toronto, Canada, 2006.

[14] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing seek overhead with application-directed prefetching," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2009, pp. 1–14.

[15] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new Ext4 filesystem: Current status and future plans," in *Proc. Ottawa Linux Symp.*, Jun. 2007, pp. 21–33.

[16] M. Russinovich, D. Solomon, and A. Ionescu, *Windows Internals*. Redmond, WA, USA: Microsoft Press, 2012, pp. 324–328.

[17] 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.

[18] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. 11th Int. Joint Conf. Meas. Modeling Comput. Syst.*, Jun. 2009, pp. 181–192.

[19] 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 Int. Conf. High-Performance Comput. Archit.*, Feb. 2011, pp. 266–277.

[20] C. Han, J. Ryu, D. Lee, J. Lee, K. Kang, and H. Shin, "File-system-level flash caching for improving application launch time on logical hybrid disks," in *Proc. 33rd IEEE Int. Performance Comput. Commun. Conf.*, Dec. 2014, pp. 1–2.

[21] Y. Deng, "What is the future of disk drives, death or rebirth?" *ACM Comput. Surveys*, vol. 43, no. 3, Apr. 2011, Art. no. 23.

[22] J. Axboe. Explicit block device plugging. [Online]. Available: https://lwn.net/Articles/438256/

[23] S. Pratt and D. A. Heger, "Workload dependent performance evaluation of the Linux 2.6 I/O schedulers," in *Proc. Ottawa Linux Symp.*, Jul. 2004, pp. 139–162.