A New Readahead Framework for SSD-based Caching Storage in IoT Systems

Hsung-Pin Chang *, Wei-Ming Su , Da-Wei Chang

Posted Date: 23 January 2024

doi: 10.20944/preprints202401.1695.v1

Keywords: data retrieval; prefetching; readahead; SSD-based caching storage systems; IoTs; Linux

A New Readahead Framework for SSD-based Caching Storage in IoT Systems

Hsung-Pin Chang [a,*,†], Wei-Ming Su [a] and Da-Wei Chang [b]

[a]Department of Computer Science and Engineering, National Chung Hsing University, Taichung, Taiwan, R.O.C.; fff2456e2@gmail.com
[b]Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan, R.O.C.; david.oslab@gmail.com
† This paper is an extended version of our paper published in "A Cross-Layered Readahead Architecture for Multi-Tiered Storage Systems" presented at the 6th IEEE Symposium on Computers and Communications (ISCC), Athens, Greece, 5-8 Sept. 2021.
* Correspondence: hpchang@cs.nchu.edu.tw

Abstract: In an IoT system, the sheer volume of data generated by numerous sensing devices necessitates a well-designed scheme for storing and retrieving data efficiently, enabling streamlined data processing and analytics. One promising storage architecture involves utilizing solid-state drives (SSDs) to cache data from hard disk drives (HDDs), thereby creating an SSD-based caching storage system. To further enhance access performance, it is possible to employ readahead techniques to minimize data access latency. However, the existing Linux readahead scheme falls short in fully leveraging SSD-based caching storage systems. We address this limitation by introducing a novel cross-layered readahead architecture that effectively communicates with the VFS layer, the file system layer, and the block I/O layer. This communication facilitates the acquisition of readahead timing, readahead data continuity, and readahead data location, respectively. To guide prefetching decisions, our architecture analyzes the degree of data access sequentiality, the performance model of the target storage device, and the access patterns of the I/O workload on the corresponding storage device. The implementation of this new architecture in the Linux kernel yields promising experimental results, demonstrating its robustness by consistently outperforming the stock Linux kernel. Notably, our architecture reduces the total execution time of the stock Linux kernel by up to 49%, except in cases of random workloads where both the stock Linux kernel and our architecture exhibit similar performance.

Keywords: data retrieval; prefetching; readahead; SSD-based caching storage systems; IoTs; Linux.

1. Introduction

The Internet of Things (IoT) represents a novel paradigm that has significantly changed the way individuals live, work, and interact with the physical world. Through the deployment of a myriad of devices, an IoT system can establish a dynamic ecosystem, facilitating the delivery of innovative services and enabling intelligent decision-making [1–4]. Figure 1 illustrates a typical deployment scenario for IoT.

Central to IoT, as depicted in Figure 1, are numerous devices identified as "things." These devices are equipped with sensors that continuously collect data, transmitting the gathered information to a central server. Subsequently, in the central server, a domain-specific algorithm or tool is employed to process and analyze the data, extracting meaningful insights to foster informed decision-making. Finally, the central server issues specific commands back to the devices, creating a feedback loop that translates derived insights or decision-making results into tangible actions.

 doi:10.20944/preprints202401.1695.v1

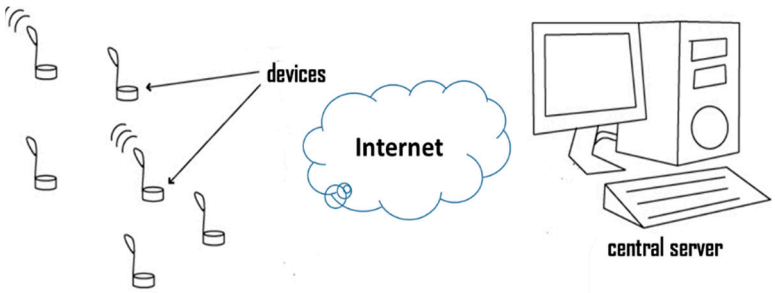


Figure 1. A simple IoT deployment example.

Nevertheless, the storage subsystem has long been the primary bottleneck in computer system performance. This problem is getting more serious due to the vast amount of data generated by IoT devices. To provide efficient data processing and analytics, an effective data storing and retrieving scheme is needed. One promising scheme is to extend the storage hierarchy to include SSDs as an intermediate tier between main memory and HDDs, forming a SSD-based caching storage system [5]. In such a SSD-based caching storage system, SSDs are treated as the second-level HDD cache, where the first-level cache is the main memory. The aim of such approaches exploit the advantages of both storage devices, i.e., the high access performance of SSDs and the low price and large capacity of HDDs, seeking comparable performance to SSDs for a cost and capacity similar to HDDs.

To further boost the access performance, prefetching, also known as *readahead*[1] in Linux, that proactively read more data that is expected to be used in the near future can be applied to shorten the data access latencies [6]. Prefetching not only improves storage utilization because a larger amount of data is accessed at a time, but it also reduces the number of disk accesses by aggregating a sequence of tiny sequential reads into a large request. It also enhances system efficiency by overlapping storage device activity with central processing unit (CPU) computations. Finally, prefetching hides disk access latency from applications. User read requests are satisfied from main memory instead of from HDDs, thus transforming millisecond-based disk access into microsecond-based memory access.

Given the limited capacity of an SSD, a cache manager is used to manage the SSD caching space. In an SSD-based caching storage system, HDDs and SSDs have different performance models. Also, since HDD sequential reads are much faster than random reads, SSD cache managers tend to cache only random accessed data on SSDs; the workloads placed on HDDs and SSDs are correspondingly different. Consequently, in an SSD-based caching storage system, prefetching policies should reflect whether the data is cached in the SSD or stored in the HDD.

Nevertheless, in Linux, readahead logic is implemented at the virtual file system (VFS) layer, as user applications access storage devices by file abstraction. Thus, to predict which data will be requested and to prefetch it before the arrival of the corresponding requests, the Linux kernel intercepts file read requests, analyzes the request access patterns, and makes prefetch decisions at the file abstraction level. Since the mapping of a file onto the corresponding physical storage device occurs below the VFS layer, the Linux readahead scheme is designed for unitary storage devices. As a result, in SSD-based caching storage systems, Linux readahead logic cannot tell whether the accessed data is cached in the SSD or stored in the HDD. That is, the current Linux readahead scheme does not fully support SSD-based caching storage systems.

We address this by proposing a Linux readahead architecture for SSD-based caching storage systems. The contributions of this paper are as follows:

- We identify the limitations of the current Linux readahead scheme for SSD-based caching storage systems, which allows us to design a new readahead architecture for SSD-based caching storage systems.
- We propose a novel architecture that addresses the limitations of the current Linux readahead scheme. The new readahead architecture is cross-layered: it communicates with VFS layer, file system layer, and block I/O layer to obtain information for prefetching. First, it communicates with the VFS layer to determine when to trigger a readahead, i.e., *readahead timing*. It also

 doi:10.20944/preprints202401.1695.v1

cooperates with the file system to determine whether the readahead data is continuous or fragmented, i.e., *readahead data continuity*. Finally, it queries the SSD cache manager to determine whether the accessed data is cached in the SSD or stored in the HDD, i.e., *readahead data location.* Then, according to the degree of data access sequentiality, the performance model of the corresponding storage device, and the data access patterns applied on the corresponding storage device, the proposed architecture determines whether to invoke prefetching and, if so, calculates an appropriate prefetch depth (also known as prefetch degree or prefetch size).

- We present a comprehensive design and implementation of our new architecture in Linux. The experimental results reveal that the architecture improves the performance of the current Linux prefetching scheme. In particular, it reduces the total execution time of the stock Linux kernel by up to 49%.

The remainder of this paper is organized as follows. Section 2 reviews SSD-based caching storage systems, the current Linux readahead scheme, and the Linux I/O stack. Section 3 presents the limitations of the current Linux readahead scheme for SSD-based caching storage systems, and Section 4 details the proposed readahead architecture. Section 5 presents the experimental results. Section 6 presents related work and Section 7 concludes.

2. Background

2.1. SSD-based Caching Storage Systems

SSDs are increasingly being incorporated into storage systems. To meet the demand for performance, capacity, and cost, one promising solution is to stack SSDs on top of HDDs, forming an SSD-based caching storage system.

Given the limited capacity of an SSD, an SSD cache manager is needed to manage the SSD caching space. The cache manager recognizes the underlying storage devices, and upon receiving a user request to the HDD, decides whether the requested data block should be cached in the SSD. In Linux, the management of the block devices and their associated requests is handled at the block I/O layer. Therefore, modern Linux SSD cache managers such as dm-cache [7], bcache [8], and EnhanceIO [9] all operate at the block I/O layer, acting as block-based second-level caches for HDDs.

Besides, current page cache managers adopt an on-demand caching policy [10] in an attempt to absorb all read/write traffic to achieve a high page cache hit ratio, thus resulting in frequent data updates to the page cache. Since SSDs have lifetime constraints, such frequent data updates to the caching space are not suitable for SSDs. Thus, current SSD cache managers adopt a selective caching policy, in which only blocks that are identified as valuable enough are cached into the SSD. That is, they treat the SSD as a by-passable cache to minimize unnecessary writes to the SSD [11–13]. Since HDDs exhibit poor performance with random accesses, many SSD cache managers differentiate randomly accessed data from sequentially accessed data and only cache randomly accessed blocks in the SSD.

For example, bcache maintains a sequential cutoff parameter: for each process, upon receiving the request, if the time interval between it and the previous request is less than 4 milliseconds and the logical block numbers (LBNs) of these two requests are continuous, their request sizes are accumulated. When the accumulated value exceeds the *sequential cutoff value*, indicating that the input workload is sequential, bcache by-passes the following requested data and serves these requests directly from the HDD. However, if the workload is not identified as sequential, bcache serves the requests via the SSD.

Furthermore, given the lifetime limitation of the SSD, to strike a balance between write performance and data consistency, bcache implements three cache writing policies: write-through, write-back, and write-around. In the write-through policy, a user write is completed only when the data are written to both the SSD and the HDD. For the write-back policy, the user write is completed after the data is written to the SSD. The written data is propagated to the HDD later in the background. For the write-around policy, the user-written data is written directly to the HDD. That is, data is loaded into the SSD cache only upon read misses.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 January 2024 doi:10.20944/preprints202401.1695.v1

Finally, once the capacity of the SSD cache space has been filled, the cache manager decides which blocks should be evicted from the cache. Bcache implements both first-in-first-out (FIFO) and least recently used (LRU) replacement policies. In this paper, we use bcache as the SSD cache manager. We set the sequential cutoff value to 4 MB, use the write-back policy, and select LRU as the replacement scheme; note that these are all default values or default policies of bcache.

2.2. Linux Readahead Scheme

To facilitate the specification of the location of blocks of data, storage devices provide a linearized data layout by exporting a sequence of logical block numbers (LBNs) to operating systems. Disk manufacturers seek to ensure that blocks with consecutive LBNs are physically contiguous on HDDs [14]. Furthermore, file systems attempt to allocate logical blocks of a file in an adjacent way. Therefore, the performance of sequentially accessing a file is similar to that of accessing contiguous physical blocks on HDDs.

Based on this observation, sequential prefetching was proposed to optimize I/O performance, and has long been used in operating systems [15]. UNIX-based operating systems [16], database servers [17,18], distributed file systems [19], big data processing [20], cloud storage systems [21], mobile devices [22], and high-end storage controllers [23] also employ sequential prefetching. Linux has used sequential prefetching since 2002 [24–26]. The readahead scheme in Linux operates at the VFS layer. Thus, sequential prefetching in Linux is, in fact, sequential file prefetching; it intercepts file read requests at the VFS layer, tracks access patterns at the level of file abstraction, and makes file-level prefetching decisions.

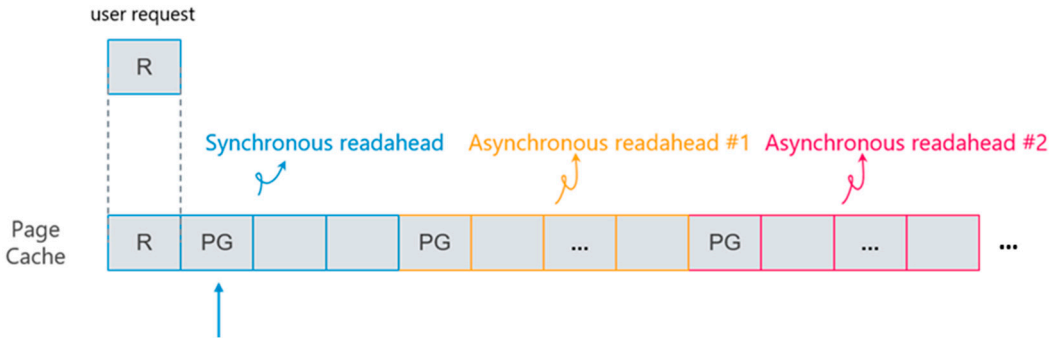


Figure 2. Synchronous and asynchronous readaheads. Pages marked PG are PG_readahead pages.

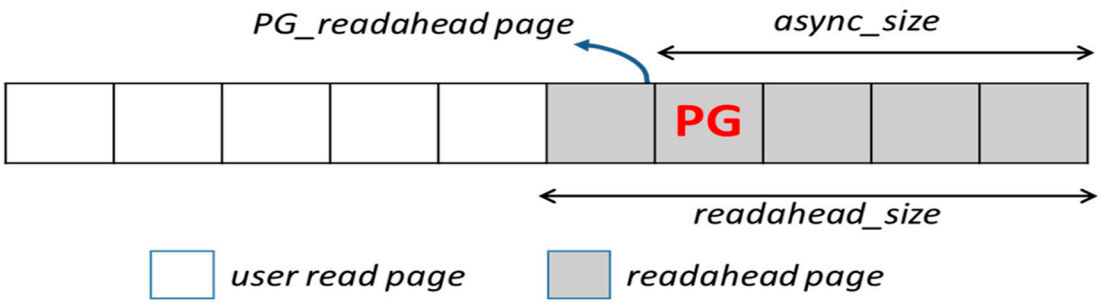


Figure 3. Linux readahead scheme.

Usually, a prefetching algorithm must answer two main questions: when to trigger a prefetch, and how much data to prefetch. For the first question, Linux issues a prefetch when the following two types of pages are read respectively:

- *Cache miss page*. When a user process issues a read request, Linux first searches the page cache. If the requested data is not in the page cache, i.e., this is a cache miss, the request is forwarded to the underlying storage device. Simultaneously, Linux performs a synchronous readahead that

 doi:10.20944/preprints202401.1695.v1

issues a much larger request than the original request to pull in more pages from the storage device.

- *PG_readahead page*. It is inefficient to trigger a prefetch only when a page cache miss occurs because the corresponding user process must be blocked until the requested data has been read, cached in the page cache, and copied to the user buffer. Based on the concept of asynchronous I/O, Linux addresses this using asynchronous readahead by proactively launching a readahead to the storage devices. Figure 2 shows that, in a sequential access stream, a synchronous readahead is followed by a number of asynchronous readaheads. Specifically, Linux introduces the *async_size* threshold. When the number of not-yet-referenced readahead pages falls below *async_size*, which means that the user process has consumed a sufficient number of readahead pages, an asynchronous readahead commences. A page is called a readahead page if it was fetched from the storage devices by the readahead algorithm. Thus, as shown in Figure 3, in a set of readahead pages, Linux sets a trigger page whose location is at a trigger distance of *async_size* from the end of the set of readahead pages. Linux marks the trigger page with the *PG_readahead* flag. When a user process accesses the page marked with PG_readahead, indicating that the readahead pages will soon be used up, Linux proactively issues an asynchronous readahead to pull in more pages. Thus, when users issue requests to read these pages, the requests are read immediately from the page cache instead of visiting the storage devices.

For the second question, depending on whether a readahead is synchronous or asynchronous, Linux determines the readahead size and the value of *async_size* for each readahead as follows:

- *Synchronous readahead*. Depending on the size of the user read request, the readahead size is two or four times the user read size. Equation (1) calculates *async_size* :

$$async\_size = readahead\_size - user\_read\_size, \quad (1)$$

where *readahead_size* and *user_read_size* respectively denote the readahead size and the size of the user read request. That is, the PG_readahead page is the first page of the set of readahead pages. For example, as shown in Figure 4(a), if the user read size is 1 page and the readahead size is four times the user read size, i.e., 4 pages, then *async_size* = 3. Thus, if the upcoming read request is continuous with the current one, an asynchronous readahead is triggered upon accessing the first readahead page.

- *Asynchronous readahead.* Depending on the size of the previous readahead size, the readahead size is also set to two or four times the previous readahead size. Thus, the readahead size grows exponentially until it reaches the maximum readahead size, denoted by *max_readahead*. Besides, Equation (2) calculates *async_size* size. From Equation (2), *async_size* is set to equal the readahead size. For example, as shown in Figure 4(b), if the readahead size is 4 pages, then *async_size* = 4. Thus, if the upcoming read request is sequential, the next asynchronous readahead starts upon accessing the first readahead page.

$$async\_size = readahead\_size. \quad (2)$$

In summary, Linux uses prefetching only for strictly sequential reads, due to its simplicity and robustness. In contrast to sophisticated prediction methods, detecting sequentiality is easy and yields very high prediction accuracy [27].

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 January 2024 doi:10.20944/preprints202401.1695.v1

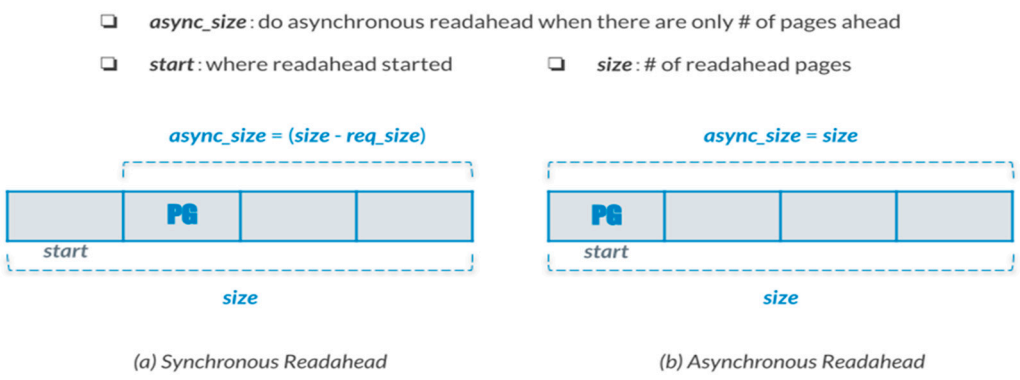


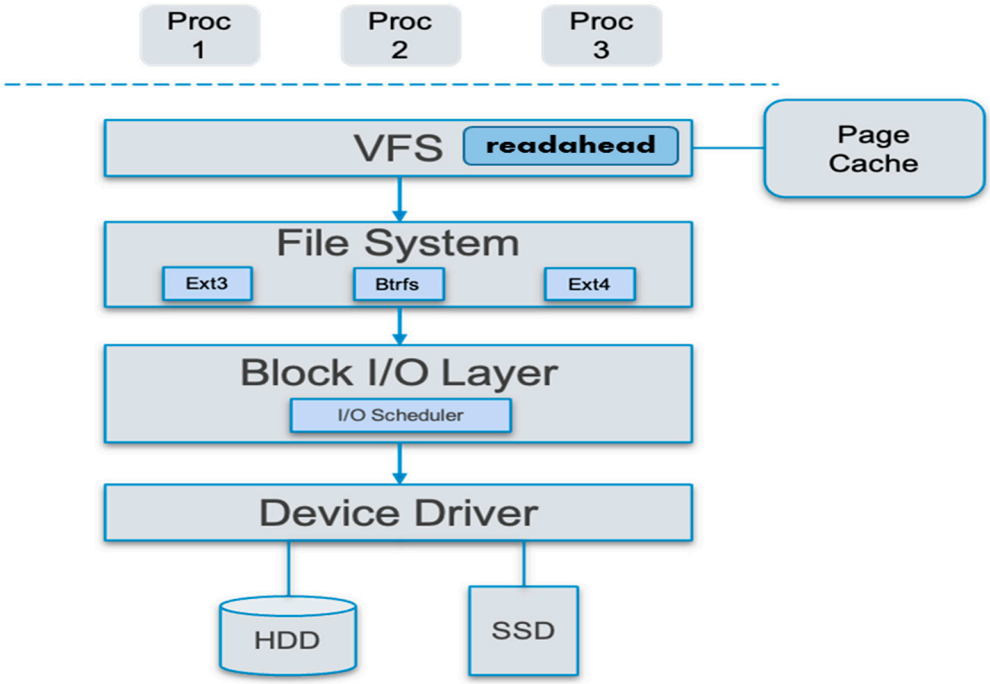
Figure 4. *async_size* in synchronous and asynchronous readaheads.

2.3. Linux I/O Stack

Figure 5 shows the Linux I/O stack diagram. When a process issues a disk I/O request via file I/O-based system calls, the request first arrives at the VFS layer. In Linux, VFS provides a standard interface so that user processes can uniformly access files located in different concrete file systems. Then, the VFS layer looks up the page cache to determine whether the requested data is cached in the page cache. If it has been cached in the page cache, i.e., it is a page cache hit, the cached data is copied to the user buffer and the request is completed. Otherwise, the request is passed to the file system layer.

In the VFS layer, a file is considered logically subdivided into a number of pages. Thus, reading a file is conducted in a page-based manner and the page numbers of a file are numbered continuously in the VFS layer. By contrast, the locations of blocks of data stored on storage devices are identified by logical block numbers (LBNs); a file's LBNs may not be continuous due to fragmentation. Thus, the file system is responsible for converting page-based file accesses to LBN-based I/O requests, i.e., *bio* objects in Linux.

After file system processing, the request is sent to the block I/O layer. In the block I/O layer, block-oriented I/O requests are placed in request queues, where I/O merging and scheduling are performed for possible performance optimization. Finally, the request reaches the device driver layer and is eventually issued to the storage devices to access that particular block.



 doi:10.20944/preprints202401.1695.v1

Figure 5. Linux I/O stack diagram.

3. Motivations

Figure 5 also shows the current Linux readahead architecture. As shown in the figure, readahead logic is implemented at the VFS layer, which reflects the fact that user applications access storage devices by file abstraction. If the request is a read request and is a page cache miss, the request is handed over to the readahead module, which launches a synchronous or asynchronous readahead, as stated in Section 2.2.

Nevertheless, HDDs and SSDs are in fact block-based storage devices. As stated in Section 2.3, in Linux, the management of the block devices and their associated requests is handled at the block I/O layer. Therefore, current Linux SSD cache managers are all implemented at the block I/O layer.

In an SSD-based caching storage system, the following two characteristics greatly impact readahead performance:

- The performance models of HDDs and SSDs are drastically different. Since SSDs have no mechanical moving parts compared to HDDs, they provide much better performance, especially for read operations. In addition, HDD sequential reads are much faster than HDD random reads. By contrast, in SSDs the performance of sequential reads and sequential writes are similar.
- The I/O workloads applied to HDDs and SSDs are also different. Since SSDs and HDDs have different performance models, an SSD cache manager should exploit the complementary properties of SSDs and HDDs. For example, bcache and profit caching differentiate random accesses from sequential accesses and only cache randomly accessed data in SSDs [8,11]. Therefore, data cached in SSDs exhibits different access patterns compared to data stored in HDDs.

Consequently, in an SSD-based caching storage system, a well-behaved prefetching algorithm should adopt different prefetching policies for HDDs and SSDs. Nevertheless, in the current Linux system, since the readahead scheme is located at the VFS layer and the SSD cache manager is implemented at the block I/O layer, the readahead logic has no idea on which physical storage device the accessed data blocks are stored. As a result, the current Linux readahead scheme lacks good support for data prefetching on SSD-based caching storage systems.

4. Design and Implementation

4.1. System Architecture

To address the limitations of the Linux readahead scheme, we propose a new readahead architecture for Linux to properly support SSD-based caching storage systems, thus using different prefetching policies for SSDs and HDDs.

In the current SSD-based caching storage solution, the designated physical storage device is unknown until a user I/O request is processed by the SSD cache manager. Consequently, one possible solution is to move the prefetching logic from the VFS layer to the block I/O layer, i.e., at the same layer of the SSD cache manager [28]. The readahead logic queries the SSD cache manager to determine whether a given data block is cached in the SSD or stored in the HDD, and then adopts the corresponding prefetching policy.

However, in current file systems, files may be fragmented, which occurs when the storage device is nearly out of free space or the user repeatedly appends or edits files. As a result, the LBNs of a file may not be numbered continuously. If the readahead logic was implemented at the block I/O layer, making readahead decisions using LBNs may readahead data blocks belonging to other files. For example, as shown in Figure 6, File A is fragmented and is allocated in two segments of continuous logical blocks. Upon receiving a user request reading block 201, a block-layer readahead solution may query the SSD cache manager, knowing that blocks 201–204 are all stored in the HDD, and decide to readahead the subsequent three blocks, i.e., blocks 202–204. As a result, it reads blocks 203–204 which in fact belong to other files, i.e., File B. In practice, if knowledge from the file systems were available,

 doi:10.20944/preprints202401.1695.v1

a readahead solution would know that File A is fragmented and the readahead request would be revised to read blocks 201–202 and 301–302.

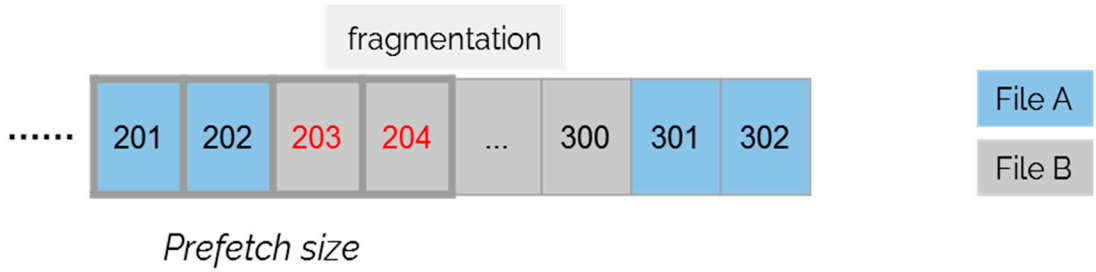


Figure 6. Fragmented file: File A.

Furthermore, as stated in Section 2.2, a readahead solution must catch both the page cache miss and PG_readahead page hit events, so as to trigger synchronous and asynchronous readaheads, respectively. As stated in Section 2.2, both events are maintained at the VFS layer. Consequently, a readahead solution also needs help from the VFS.

From the above discussion, in an SSD-based caching storage system, a well-behaved readahead solution must have access to the following information.

- *Readahead Timing*. A readahead solution must know when to trigger a prefetch. As stated above, a readahead should be invoked when cache miss pages or PG_readahead pages are accessed.
- *Readahead Data Continuity.* A readahead solution must know whether the data blocks of a file are located continuously or fragmented. If a file is fragmented, it must also know the LBNs of each file segments, so as to skip LBNs that belong to other files.
- *Readahead Data Location*. To apply different prefetching policies for SSDs and HDDs, a readahead solution must know whether an accessed data block is cached in the SSD or stored in the HDD.

As stated in Section 2.3, in the current Linux I/O stack, these three types of information are maintained in the VFS, file systems, and block I/O layers, respectively. Thus, as shown in Figure 7, we propose a cross-layered optimized readahead solution. We implemented a new readahead kernel module consisting of the ReqInterceptor, ReqAnalyzer, and RaHandler components. ReqInterceptor communicates with the VFS layer to derive the readahead timing and then invokes ReqAnalyzer, which communicates with the file system layer and block I/O layer. ReqAnalyzer derives the readahead data continuity from the concrete file system and retrieves the readahead data location from the SSD cache manager, i.e., bcache, respectively. Finally, RaHandler determines whether to prefetch and, if so, the prefetch size.

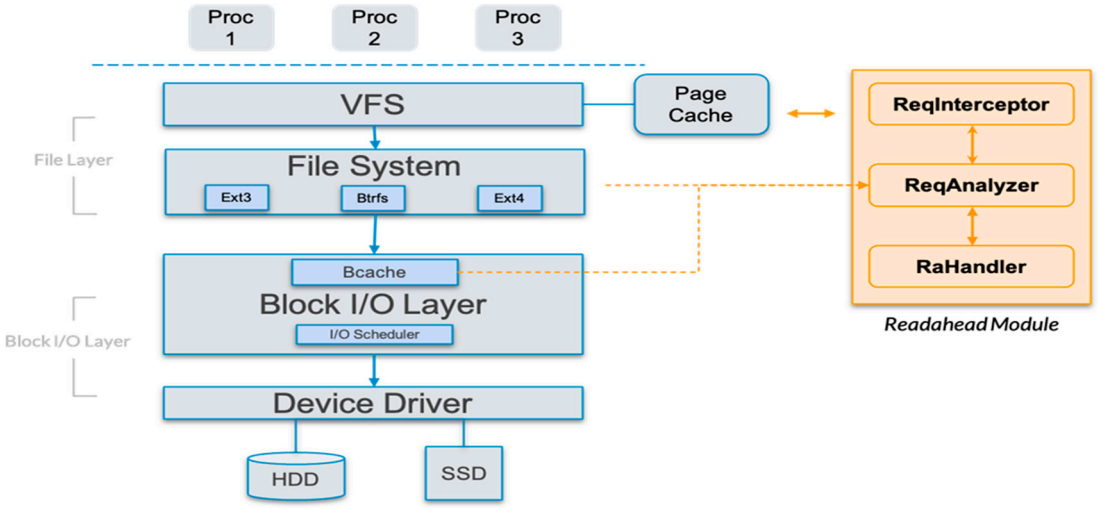


Figure 7. Proposed readahead architecture.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 January 2024 doi:10.20944/preprints202401.1695.v1

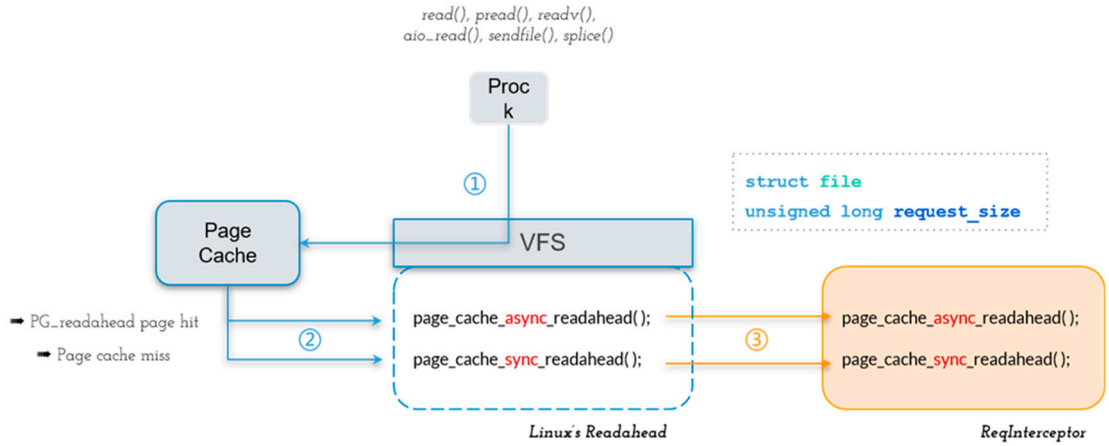


Figure 8. Schematic diagram of read request interception.

4.2. ReqInterceptor

ReqInterceptor interfaces with the VFS layer and is activated when either a cache miss page or a PG_readahead page is accessed. As shown in the step ① of Figure 8, in Linux, file read operations such as *read()*, *pread()*, *readv()*, *aio_read()*, *sendfile()*, and *splice()* are served by the VFS layer. Then, as shown in the step ② of Figure 8, VFS queries the page cache to determine whether a page cache miss or a PG_readahead page hit has occurred and invokes *page_cache_sync_readahead()* to perform synchronous readahead or *page_cache_async_readahead()* to perform asynchronous readahead accordingly.

To intercept these two events, we modify the Linux kernel to add two function pointers. Normally, these point to the above two kernel functions. When our readahead kernel module is added into the kernel, as shown in step ③ of Figure 8, we modify these two function pointers to instead point to the proposed *page_cache_sync_readahead()* or *page_cache_async_readahead()* functions.

When ReqInterceptor is invoked, it intercepts the corresponding read request and derives the metadata of the request, including the page number of the first page of the request, request size, the process identifier (process ID or PID) of the process issuing the request, and the inode number of the accessed file. Finally, it sends this metadata to ReqAnalyzer.

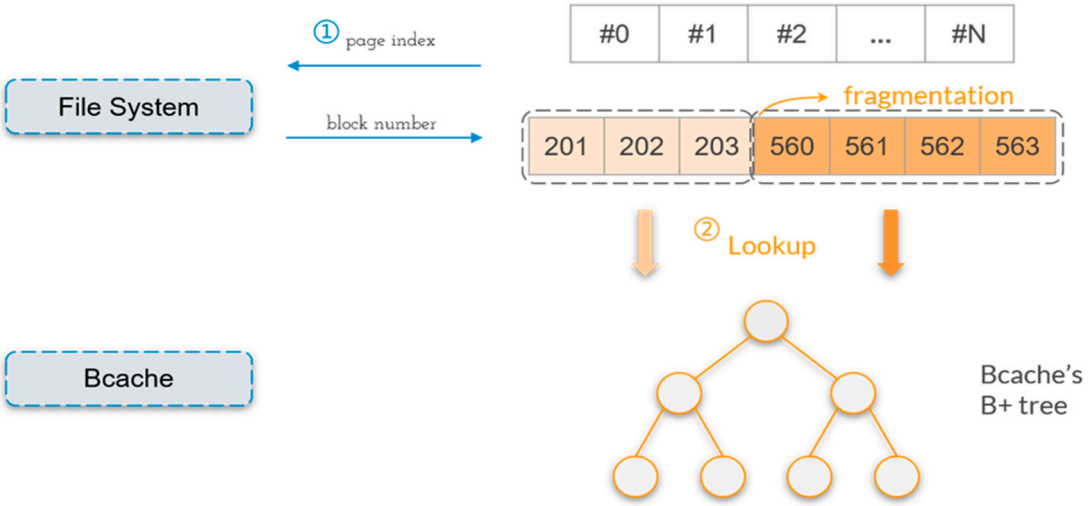


Figure 9. Main tasks performed by ReqAnalyzer.

4.3. ReqAnalyzer

Taking data locations into consideration in the key idea of our readahead solution. ReqAnalyzer is responsible for obtaining this key information. Figure 9 shows the two main tasks performed by

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 January 2024 doi:10.20944/preprints202401.1695.v1

ReqAnalyzer: querying the file system to determine data continuity, and obtaining the data location from the SSD cache manager, i.e., bcache.

After receiving the page index of the request sent from ReqInterceptor, ReqAnalyzer must know the location of each page to be prefetched, i.e., the pages following the user requested data if a synchronous readahead is invoked, or the pages following the previous readahead if an asynchronous readahead is activated.

As shown in the step ① of Figure 9, ReqAnalyzer first queries the file system to derive the corresponding LBNs of these pages. There are two reasons for performing this task. Firstly, ReqInterceptor runs at the VFS layer and follows the page-level abstraction. By contrast, bcache is a block-layer SSD caching solution and thus maintains data cached in SSDs by the block-level abstraction. Thus, ReqAnalyzer must know the LBNs of these pages so that it can query the SSD cache manager to know the location of each page. As stated in Section 2.3, the concrete file systems are responsible for converting page-based file accesses to LBN-based I/O requests. For example, Figure 10(a) shows a file that is allocated in continuous logical blocks. In the figure, the numbers in each upper rectangle and each lower rounded rectangle are the page number and logical block number, respectively. Here, we make a simplifying assumption that the sizes of a page and a logical number are equal. If a user request accesses the first page index, i.e., 0, of the file, after querying the file system, ReqAnalyzer then knows that its corresponding LBN is 35. After knowing the LBN, ReqAnalyzer then queries the SSD cache manager to determine the location of the corresponding block.

Secondly, and much more importantly, the file may be fragmented. If it is not fragmented, we need only know the LBN of any of the pages in the file. Then, other pages' LBNs are easily derived without asking the file system. For example, assume that the sizes of a page and a logical block are 4 KB and 512 bytes, respectively. Further, assume that the page number of the first accessed page of a file is p and its corresponding logical block number is q. Then, given any page number x, its corresponding logical block number y is calculated as

$$y = q + (x - p) \times (4\ \text{KB}/512\ \text{bytes}) \tag{3}$$

However, as shown in Figure 10(b), if the file is fragmented, the LBNs are not numbered continuously. In this case, determining the corresponding LBN of a page is only possible by querying the file system.

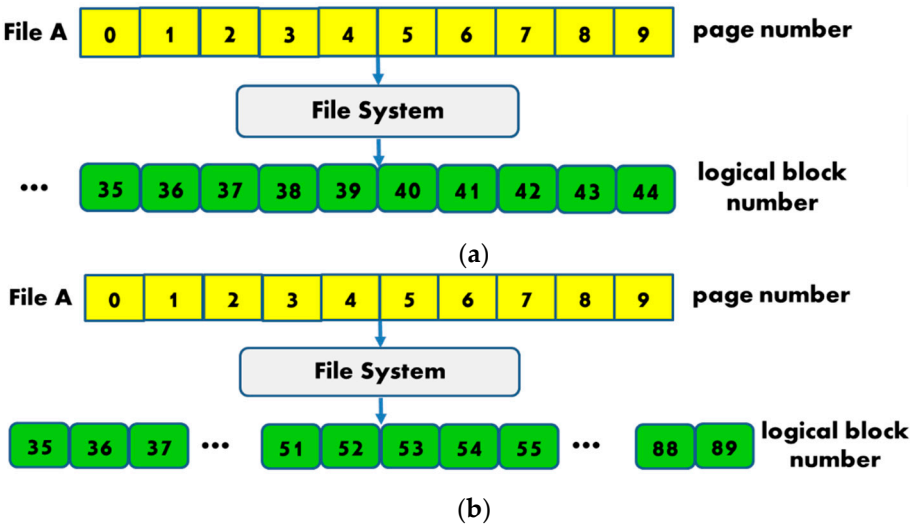


Figure 10. (a) A file allocated in continuous logical blocks. (b) A file allocated in three segments of continuous logical blocks.

After determining the LBNs, as shown in the step ② of Figure 9, ReqAnalyzer queries bcache, which uses a b+ tree to maintain data cached in the SSD, to determine whether the data to be prefetched is cached in the SSD or stored in the HDD.

 doi:10.20944/preprints202401.1695.v1

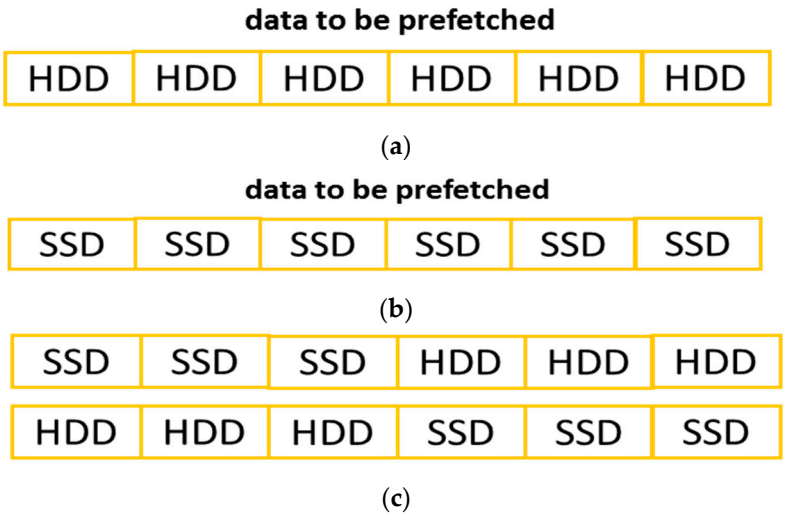


Figure 11. (a) Data to be prefetched all stored in the HDD. (b) Data to be prefetched all stored in the SSD. (c) Data to be prefetched located in both the SSD and the HDD.

4.4. RaHandler

RaHandler makes the actual prefetching decisions. As stated in Section 3, in an SSD-based caching storage system, HDDs and SSDs have different access performance models and their associated I/O workloads exhibit different access regularities. Thus, RaHandler is both device-aware and workload-aware, and adopts different prefetching policies to prefetch data from SSDs and HDDs.

As shown in Figure 11, after querying bcache, there are three possible cases. In the following, we analyze each case in terms of device-level access characteristics and workload-level access regularities, and we suggest the most appropriate prefetching policy:

- **Case 1.** Data to be prefetched are all stored in the HDD. For the HDD, the additional readahead cost is smaller since the seek time, the dominating time in serving a disk request, can be eliminated. Furthermore, as stated in Section 2.1, in bcache, if data to be prefetched is located in the HDD, it may have a high degree of sequentiality. In this case, RaHandler adopts an aggressive prefetching policy.
- **Case 2.** Data to be prefetched is all cached in the SSD. In this case, a read from flash memory can be completed in a few microseconds, compared to the HDD's latency of several milliseconds. Thus, there is little the prefetching benefit with SSDs due to the small performance gap between the SSD and main memory [29]. Furthermore, in bcache, SSD workloads tend to exhibit random access behavior. For these two reasons, RaHandler adopts a conservative prefetching policy.
- **Case 3.** Data to be prefetched is located in both the SSD and the HDD. This case also exhibits random access behavior. Thus, RaHandler adopts a conservative prefetching policy.

Assume that a sequence of consecutive readahead requests performed for a stream is $\boldsymbol{S} = \{R_0, R_2, \ldots, R_n\}$ and that R_i denotes the i-th readahead request. As stated in Section 2.2, a readahead is activated when a page cache miss or a PG_readahead page hit occurs. Thus, R_0 is invoked by a page cache miss and is a synchronous readahead. Then R_1, R_2, …, and R_n are invoked due to PG_readahead page hits and are asynchronous readaheads. Since these two types of readahead represent different degrees of sequentiality, RaHandler acts differently for each type.

4.4.1. Synchronous Readahead

Since a page cache miss occurs because the corresponding stream is new or is considered to exhibit random access regularity, for synchronous readahead, RaHandler adopts a conservative prefetching policy and calculates *readahead_size*, the readahead request size, as

$$readahead\_size = user\_read\_size * scale\_factor_1, \quad (4)$$

 doi:10.20944/preprints202401.1695.v1

where *scale_factor*$_1$ depends on the data distribution situation and is calculated as

$$scale\_factor_1 = \begin{cases} 2, & if\ case\ 1\ is\ true \\ 1, & if\ case\ 2\ is\ true \\ \min\left(1, {first\_seg\_length}/{prev\_readahead\_size}\right), & if\ case\ 3\ is\ true \end{cases} \quad (5)$$

where *first_seg_length* is the length of the first segment, located either in the SSD or the HDD for Case 3. From Equation (5), if data to be prefetched is stored in the HDD, a less conservative prefetching policy is adopted that sets the prefetch depth to two times the requested data size. By contrast, if the data to be prefetched is cached in the SSD, the prefetch depth is set to one times the requested data size, indicating a conservative prefetching policy. Finally, if the data to be prefetched is located in both the SSD and the HDD in an interleaved manner, RaHandler adopts a conservative prefetching policy and the original I/O size is inflated to readahead a minimum of *user_read_size* and *first_seg_length* data.

4.4.2. Asynchronous Readahead

An asynchronous readahead is activated by a PG_readahead page hit, indicating that the previous readahead data is needed by the corresponding stream. Consequently, the stream is considered to exhibit a high degree of a sequential access pattern. Thus, RaHandler adopts an aggressive prefetching policy that increases the readahead request size exponentially. Specifically, RaHandler calculates *readahead_size*, the readahead size, as

$$readahead\_size = prev\_readahead\_size * scale\_factor_2, \quad (6)$$

where *prev_readahead_size* is the previous readahead size, and *scale_factor*$_2$ depends on the data distribution situations and is calculated as

$$scale\_factor_2 = \begin{cases} 4, & if\ case\ 1\ is\ true \\ 2, & if\ case\ 2\ is\ true \\ \min\left(2, {first\_seg\_length}/{prev\_readahead\_size}\right), & if\ case\ 3\ is\ true \end{cases} \quad (7)$$

If the data to be prefetched is all stored in the HDD, RaHandler adopts an aggressive prefetching policy that sets the readahead size to four times the previous readahead size [30]. By contrast, if the data to be prefetched is all cached in the SSD, RaHandler adopts a less aggressive prefetching policy that only doubles the readahead size. Finally, if data to be prefetched is located in both the SSD and the HDD in an interleaved manner, RaHandler adopts a less aggressive prefetching policy that multiplies the previous readahead size by a minimum of 2 and the ratio of *user_read_size/first_seg_length*.

4.4.3. Maximum Readahead Size

However, the readahead size cannot be infinitely increased. Thus, Linux has a *max_readahead* threshold—32 pages by default—that represents the maximum readahead size. In an SSD-based caching storage system, it would not be appropriate to use a single *max_readahead* threshold for both data cached in SSDs and stored in HDDs. A *max_readahead* value suitable for HDDs would not be suitable for SSDs since SSD workloads may exhibit random access behavior, and vice versa.

Consequently, our readahead solution not only sets the scale factor but also the maximum readahead size differently according to whether the data is cached in the SSD or stored in the HDD. Specifically, RaHandler has *HDDthresh* and *SSDthresh*, thresholds which respectively set the upper bounds of readahead size on an HDD and an SSD. In this paper, we set the values of SSDthresh and HDDthresh to 128 and 8 pages, respectively, which are four times and one-quarter of the Linux default value, i.e., *max_readahead*, respectively.

 doi:10.20944/preprints202401.1695.v1

5. Experimental Study

5.1. Experimental Environment

In this section, we evaluate the performance of the proposed readahead architecture. The experiments were conducted on a machine with a 3.4 GHz Intel Core i7-3770 processor, 16 GB DDR4 DRAM, a Seagate SATA III Barracuda 1TB HDD, and a Transcend SATA III SSD360S 128GB SSD. The OS was Ubuntu 20.04 with Linux kernel version 5.5.10. We compared application performance under the stock Linux kernel and the kernel with our new prefetching solution. Both kernels ran bcache to support SSD-based caching storage systems, and we set the sequential cutoff value to 4 MB, used the write-back policy, and selected LRU as the replacement scheme, which are all default values or default policies of bcache. Finally, to ensure each experiment had the same initial cache states for both the page cache and SSD cache, before each experiment, we re-formatted the SSD cache and uploaded the input data set to the SSD cache again. Then, the Linux page cache was emptied via the vmtouch –e command to ensure the same initial page cache content in each experiment.

We used the following user applications and benchmarks for the performance evaluations. First, we used fio to generate two different access patterns, one a pure sequential and the other a pure random access pattern. Then, real-world applications grep, diff, and gcc were used for the performance comparisons. In addition, to compare the impact of input data sets on readahead performance, in both grep and diff, two different kinds of data sets were used. Finally, since grep, diff, and gcc are mainly desktop applications, we selected another two popular server-based workloads, i.e., postmark and tpc-h, for performance comparisons. Below, we briefly describe the benchmarks and the applications.

- flexible IO tester **(**fio**)** is a workload simulation tool that simulates various types of I/O workloads [31].
- grep is a command-line utility that searches plain-text data sets for lines containing a match to a given regular expression. It visits files essentially in the order of the disk layout. We ran grep under two different datasets: the source-code tree of Linux kernel 5.5.10, and a 100 MB text file. The latter was chosen because the Linux source-code tree mostly consists of small files. This utility can be beneficial in IoT systems for searching specific patterns.
- diff is a data comparison utility that calculates and displays character-wise differences between two files. Instead of visiting files in the order of the disk layout, it scans files in alphabetical order of file names. The datasets used to generate grep workloads were also used for diff. This utility can simulate data comparison workloads in IoT systems.
- gcc is the official compiler of Unix-based operating systems. We ran the gcc benchmark by compiling and building the 5.5.10 Linux kernel. This workload is adopted to simulate application development on central servers within IoT systems
- postmark is a representative and popular storage-related benchmarking tool [32] that simulates a workload composed of many short-lived, related small files typically seen in Internet applications such as web-based transaction servers.
- tpc-h is also a popular benchmark [33]. It is a decision support benchmark that supports 22 business-oriented ad-hoc queries with each query having a high degree of complexity with various I/O characteristics. This benchmark is chosen to simulate workloads that involve examining large volumes of data, executing highly complex queries, and delivering answers in IoT systems.

5.2. Experimental Results

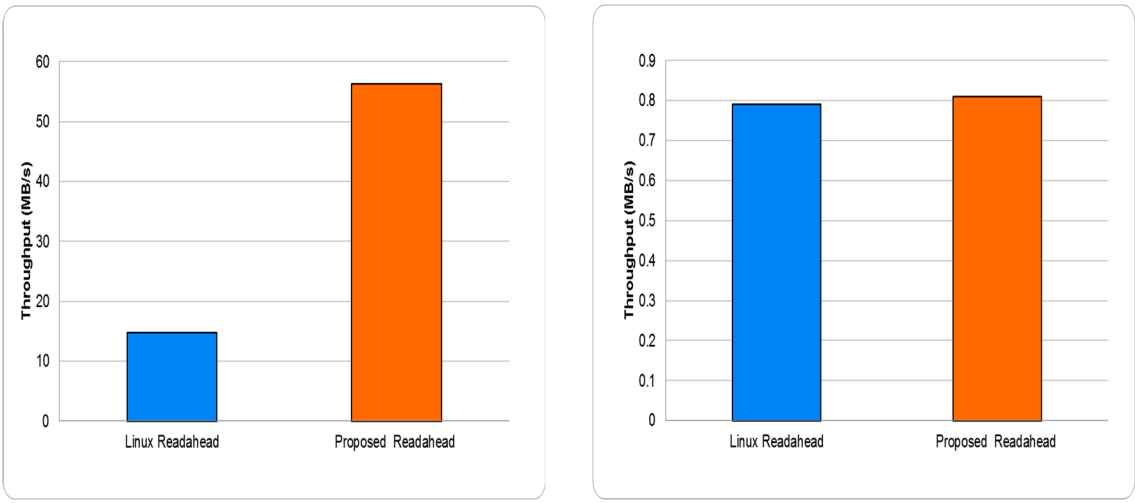
Figures 12–17 show the total execution times or throughput of the stock Linux kernel and our proposed readahead architecture under different workloads. In the following paragraphs, we analyze the performance results in detail under each benchmark:

fio. To demonstrate the effectiveness of our aggressive and conservative readahead policies, we used fio to generate two completely different workloads: a pure sequential and a pure random workload. First, Figure 12(a) shows the throughput under the stock Linux kernel and our readahead

 doi:10.20944/preprints202401.1695.v1

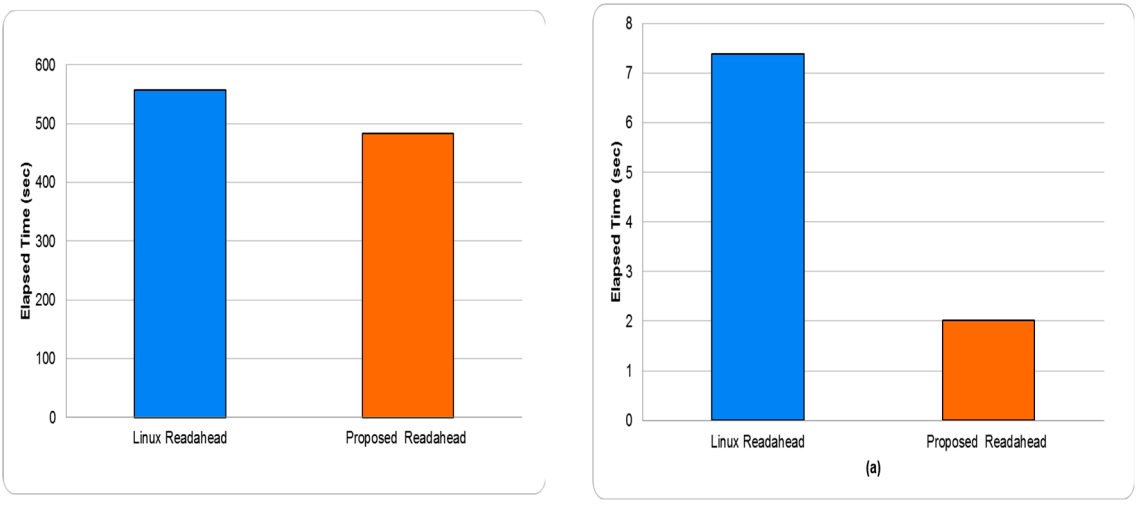
architecture in a pure sequential workload. In the experiment, we used fio to create a process that sequentially accesses a 100 MB file to simulate a fully sequential workload. The figure shows that the throughput of the proposed readahead architecture is 3.8 times that of the Linux readahead scheme, because the workload is completely sequential; according to the bcache cache policy, most accesses are targeted at the HDD. Since our readahead scheme adopts an aggressive readahead policy for data stored in the HDD, with a larger scale factor and a larger maximum readahead size, our readahead solution significantly outperforms the Linux readahead scheme.

Figure 12(b) then shows the performance under a pure random workload. In this experiment, we used fio to create a process that randomly read a 10 MB file. The figure shows that the performance of our readahead architecture and the Linux readahead scheme are similar; this is because the workload is completely random and thus readahead is useless. However, the performance of our scheme is still lightly better that that of the Linux readahead scheme since we adopt a conservative readahead policy for random workloads to avoid unnecessary prefetches, reducing the I/O bandwidth associated with fetches of unnecessary data and the memory resources needed to store the data.



(a) (b)

Figure 12. fio throughput using stock Linux kernel and proposed architecture. (a) Pure sequential workload; (b) pure random workload.

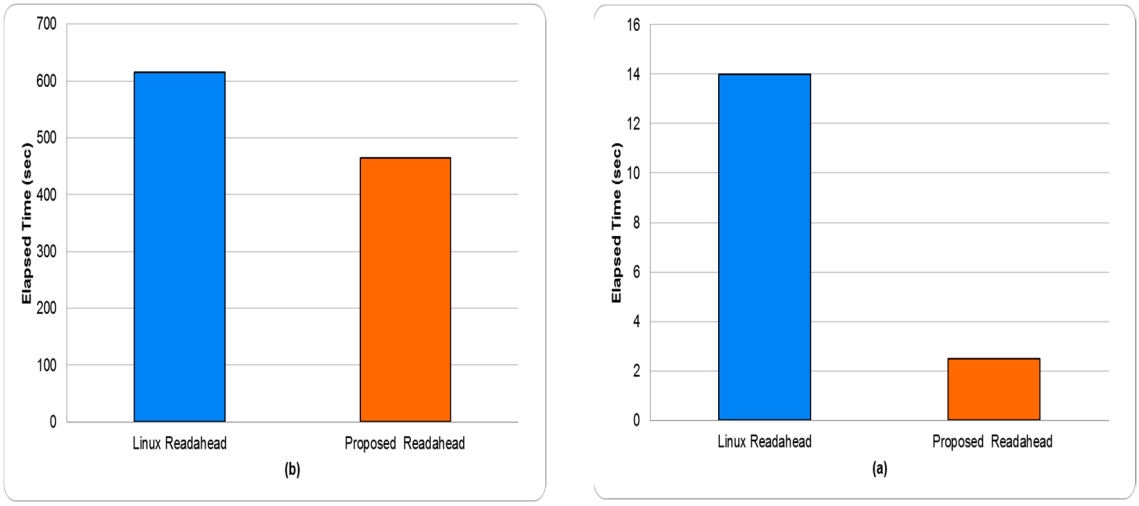


(a) (b)

Figure 13. (a) Total grep execution time using stock Linux kernel and proposed architecture. (a) Linux source-code tree, (b) a 100 MB text file.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 January 2024 doi:10.20944/preprints202401.1695.v1

grep. Figure 13(a) shows the total grep execution time using the stock Linux kernel and the proposed architecture with the Linux source-code tree and a 100 MB text file. As shown in Figure 13(a), the proposed architecture improves the total execution time by 13.3% compared with the stock Linux kernel. Although grep is a sequential-read dominated workload, the Linux source-code tree consists of a large amount of small files. Thus this improvement is because the advantages of readahead are limited, regardless of the approach. By contrast, as shown in Figure 13(b), the grep execution time with the proposed architecture is 27% of that with the stock Linux kernel if the 100 MB file is used as the input data set. This improvement is due to the cache policy of bcache, as the 100 MB file is mostly located in the HDD. Since the proposed architecture adopts an aggressive prefetching policy for HDDs, the readahead performance is significantly improved.



(a) (b)

Figure 14. (a) Total diff execution time using stock Linux kernel and proposed architecture. (a) Linux source-code tree, (b) a 100 MB text file.

diff. Figure 14 shows the corresponding experimental results when running diff on the Linux source-code tree and on the 100 MB text file. The figure shows that the proposed architecture outperforms the total execution time of the stock Linux kernel by 25% for the Linux source-code tree data set, and it outperforms the stock Linux kernel by 82% if the 100 MB file is used as the dataset. Similar to grep, this is because when the Linux source-code tree is used as input data set, the prefetch benefit is limited. However, for the 100 MB text file, the proposed architecture significantly improves readahead performance. Specifically, the total execution time of the new architecture is only 19% of that of the stock Linux kernel. This is because for the 100 MB text file, the data to be prefetched is mostly located in the HDD. With the prefetching policy of the proposed readahead architecture, the readahead size increases rapidly, yielding a larger maximum readahead size. Thus, the proposed architecture reduces the total execution time.

gcc. Compiling the kernel is a typical use of storage devices during software development. Figure 15 shows the total execution times of the proposed architecture and the Linux kernel under a gcc workload. The figure shows that the performance of the proposed architecture and the Linux readahead scheme are similar, and our architecture improves on the performance of the Linux kernel by only 4.5%. This is because in contrast to grep and diff, which consist of only read operations, in addition to reads, kernel compilation involves a significant number of write operations to generate the thousands of object files. Since readahead is only effective for reads, performance improvements are thus limited.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 January 2024 doi:10.20944/preprints202401.1695.v1

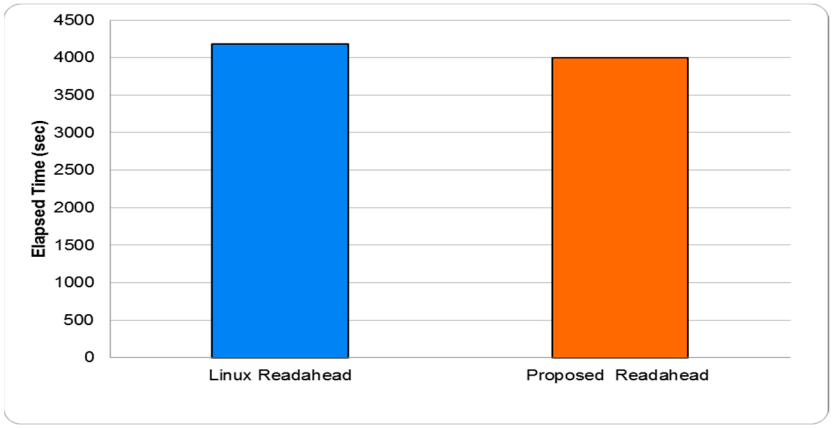


Figure 15. Total gcc execution times using stock Linux kernel and proposed architecture.

Postmark. Postmark first creates an initial pool of files of varying size, after which it executes two types of transactions on the files in the pool: (1) read from and append to a file, and (2) create and delete files. Since readahead is only effective for reads, to fully verify the effectiveness of the proposed readahead architecture and to avoid the interference of writes, in the experiment, we set the read/append ratio to 100. Then, other parameters, including create/delete ratio, file size range, number of files, and number of transactions, are set to default values. Figure 16 shows the total execution times of the stock Linux kernel and the proposed readahead architecture. The figure shows similar performance for the proposed readahead architecture and the Linux readahead scheme (within one second). This is because postmark simulates email server-like workloads with small files corresponding to user mails. Prefetching opportunities are thus limited.

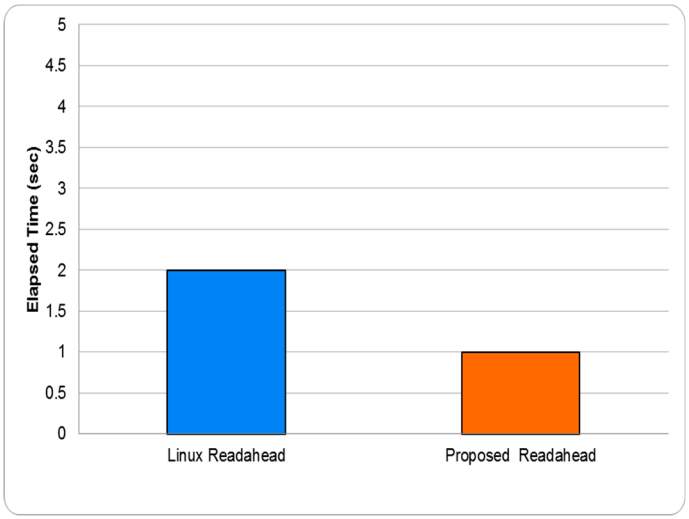


Figure 16. Total postmark execution times using stock Linux kernel and proposed architecture.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 January 2024 doi:10.20944/preprints202401.1695.v1

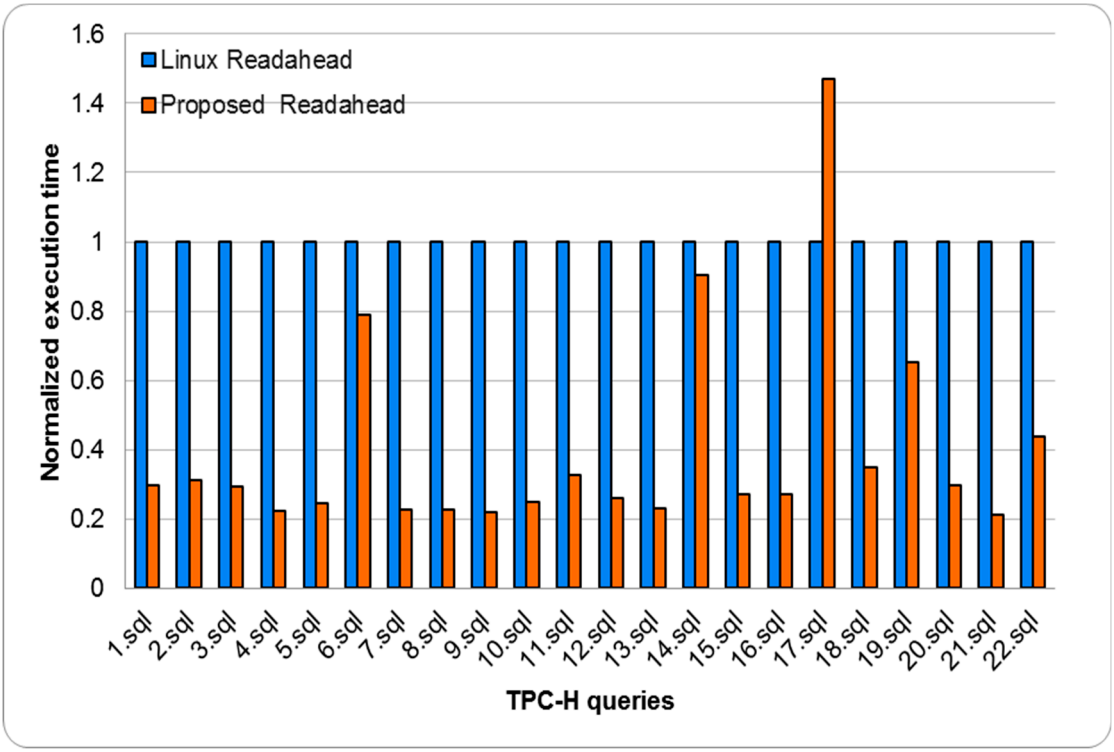


Figure 17. Total tpc-h execution times using stock Linux kernel and proposed architecture.

tpc-h. To run tpc-h, we installed MonetDB 11.39.17 [34] as the database and populated it with 10 GB of data. Then, we ran 22 tpc-h queries, each of which was issued to MonetDB to measure the elapsed time for each query. Figure 17 shows the execution times of each tpc-h query normalized to the Linux readahead scheme. The figure shows that the proposed readahead architecture reduces the execution time below that of the Linux readahead scheme for all queries except 17.sql. For 17.sql, its access pattern might sometimes exhibit sequential access behavior but soon changed to random access pattern. As a result, the readahead may be not only useless but also waste system resources, including disk I/O bandwidth and page cache space. Our method is more affected since we adopt an aggressive prefetching policy for data stored in HDDs. However, compared with the Linux readahead scheme, our architecture reduces the execution time of queries by 70% on average, up to 78%.

6. Related Work

Much work has been done to improve the performance of data prefetching algorithms. We classify existing solutions into three categories: (1) HDD-based prefetching algorithms, (2) SSD-based prefetching algorithms, and (3) prefetching algorithms based on hints disclosed in user-level applications. We survey the representative algorithms in each category.

6.1. HDD-based Prefetching Algorithms

Sequential prefetching has been long employed in operating systems to leverage the low cost of sequential accesses on HDDs. Sequential prefetching must answer two important questions: when to prefetch, and how much to prefetch. Competitive prefetching [35], for example, controls the prefetch degree to balance the overhead of disk I/O switches and unnecessary prefetching. Due to the growing capacity of main memory, Sequentiality and Thrashing dEtection based Prefetching scheme (STEP) is inclined to perform aggressive prefetching [36]. Additionally, a cost-benefit model considering access history, prefetching cost, and data access locality is built to determine the prefetch length. Adaptive Multi-stream Prefetching scheme (AMP) is concerned with cache pollution and wasted prefetch [37]. In the context of prefetching, cache pollution occurs when prefetched data causes more useful data to be evicted from the cache; wasted prefetch means that prefetched data is not

 doi:10.20944/preprints202401.1695.v1

subsequently used. To solve these two problems, AMP dynamically adapts the prefetch degree and trigger distance to achieve the highest I/O efficiency.

Pages that have been prefetched but not-yet-referenced are stored in the page cache. The management of these pages is also important. Many studies focus on how to efficiently manage prefetch memory [38]. For example, Sequential Prefetching in Adaptive Replacement Cache (SARC) [27] divides the cache space into prefetch and random cache lists and adapts the size of the lists dynamically to reduce read misses. Table-based Prefetching (TaP) dynamically determines the optimal prefetch cache size by determining the degree of sequentiality of the current I/O workload [39].

Instead of adopting sequential prefetching, many prefetching algorithms use historical information to predict data to be accessed in the near future. For example, C-Miner [40], QuickMine [41], and ClusterFetch [42] use data mining techniques to discover sequential and non-sequential correlations between data blocks. These discovered correlations are then used to determine prefetchable blocks. Similarly, DiskSeen attempts to identify sequences of blocks where, for each sequence, its blocks have been addressed recently and are located in a spatially bounded region [43]. That is, DiskSeen simultaneously exploits both temporal and spatial correlations of block access events to improve prefetching performance. However, these schemes cannot be applied in environments where I/O patterns of applications change at run time; they also fail when an application is run for the first time.

In addition to prefetching the next page within a file, there exists a rich set of prior research that allows prefetching across files. Representative techniques include graph-based prefetching and compression-based prefetching. Graph-based prefetching techniques track the sequence of file access events and capture correlations between file references to build semantic data structures, e.g., a probability graph [44], an access tree [45], or a partitioned trie [46]. Based on the data structure, a probability-based prediction algorithm is used to predict future file accesses to make prefetching decisions [47]. Compression-based prefetching uses data compression techniques for prefetching [48,49]. A data compressor tracks strings of characters to compress data by postulating a dynamic probability distribution on the data. Similarly, a prefetching algorithm must track file system events and the sequence in which they occur. In this way, data compression modeling techniques can be used by a prefetching algorithm to determine which files should be prefetched. However, because the prefetching units are files, these schemes are more suitable to web proxy or web server environments.

However, such sophisticated prefetching schemes are too complicated to implement. Thus, general-purpose operating systems, including Linux, still provide only strict sequential prefetching. Our architecture also adheres to strict sequential prefetching policy.

6.2. SSD-based Prefetching Algorithms

Because of the widely different performance models of SSDs and HDDs, recent studies have proposed prefetching schemes for SSDs. Fast Application Starter (FAST) [50] is a new I/O prefetching technique to shorten the application launch time on SSDs, leveraging the fact that the block requests generated when an application is launched are nearly identical from run to run. Thus, when an application is launched, FAST simultaneously prefetches requested blocks of data according to an a priori monitored application I/O request sequence.

Flashy prefetching [29] is a sophisticated and complicated feedback-based prefetching scheme on SSDs which consists of four stages: (1) trace collection, which collects I/O events using the blktrace utility; (2) pattern recognition, which builds state machines to identify the access patterns of I/O requests; (3) block prefetching, which issues the actual prefetch requests; and (4) feedback monitoring, which monitors the performance of the prefetch operation and adjusts the prefetch degree accordingly. However, Flashy prefetching is a user-level solution and is not fully integrated into the Linux kernel. Lynx is a prediction-based prefetching scheme on SSDs which uses a Markov chain as a probabilistic machine to learn and predict I/O access patterns, including sequential and random patterns [51].

 doi:10.20944/preprints202401.1695.v1

Our proposed readahead architecture is orthogonal to the above SSD prefetching techniques since the above techniques could be applied to prefetch data cached on SSDs. However, the proposed architecture adheres to the strict sequential file-level prefetching policy adopted in the Linux kernel and does not implement complicated prefetching schemes for HDDs or SSDs.

6.3. Hint-based Prefetching Algorithms

Application-controlled file prefetching [52] and application-informed prefetching [53] rely on user-level hints for prefetching. Application programmers must insert hints to indicate to operating systems what data will be visited in the near feature and when. The operating system then utilizes these hints to make prefetching decisions. To provide these hints, programmers must fully understand the execution scenarios of the applications. To relieve programmers of this burden, Chang and Gibson propose speculatively pre-executing an application's code to discover future read accesses and to generate hints accordingly [54]. Prefetching hints can also be automatically inserted by compilers [55–57]. Although these works are based on HDDs, the underlying ideas can also be applied to SSDs. However, this group of prefetching schemes would perform poorly if the workload characteristics were not known until run time.

7. Conclusion

A proficient data storage and retrieval scheme is essential for IoT systems. One promising approach involves leveraging SSDs as HDD caches to enhance I/O performance while providing extensive storage capacity. Additionally, the implementation of data prefetching can further amplify I/O performance. However, the current Linux readahead scheme falls short in fully harnessing the potential of SSD-based caching storage systems. In response to this limitation, we present a novel Linux readahead architecture. In contrast to prior schemes that exclusively focus on workload-aware prefetching, our proposed architecture is both device-aware and workload-aware. It determines prefetching decisions by exploiting the complementary properties of HDDs and SSDs, as well as the distinct workload characteristics applicable to various storage devices.

A prototype of the proposed readahead architecture is implemented on the Linux kernel. We conduct detailed experiments under a variety of workloads, with experimental results that show that the proposed architecture successfully supports SSD-based caching storage systems. In fact, with little modification, the architecture can be extended to support heterogeneous storage systems which use both HDDs and SSDs as storage devices, or homogeneous storage systems in which each storage device has different performance models.

References

1. Al-Ali AR, Beheiry S, Alnabulsi A, Obaid S, Mansoor N, Odeh N, Mostafa A. An IoT-Based Road Bridge Health Monitoring and Warning System. *Sensors*. 2024; 24(2):469. https://doi.org/10.3390/s24020469
2. Barros N, Sobral P, Moreira RS, Vargas J, Fonseca A, Abreu I, Guerreiro MS. SchoolAIR: A Citizen Science IoT Framework Using Low-Cost Sensing for Indoor Air Quality Management. *Sensors*. 2024; 24(1):148. https://doi.org/10.3390/s24010148
3. Shaheen A, Kazim H, Eltawil M, Aburukba R. IoT-Based Solution for Detecting and Monitoring Upper Crossed Syndrome. *Sensors*. 2024; 24(1):135. https://doi.org/10.3390/s24010135
4. Elfaki AO, Messoudi W, Bushnag A, Abuzneid S, Alhmiedat T. A Smart Real-Time Parking Control and Monitoring System. *Sensors*. 2023; 23(24):9741. https://doi.org/10.3390/s23249741
5. R. Appuswamy, D. C. Moolenbroek, A. S. Tanenbaum, "Integrating flash-based ssds into the storage stack. *in Proc. IEEE Mass Storage Syst. and Technol. Conf. (MSST)*, April 2012. https://doi.org/10.1109/MSST.2012.6232365.
6. E. Shriver, C. Small, and K. A. Smith, "Why does file system prefetching work?" in *Proc. USENIX Annu. Tech. Conf.*, 1999, pp. 71-84.
7. dm-cache. https://web.archive.org/web/20140718083340/
8. http://visa.cs.fiu.edu/tiki/dm-cache
9. bcache. https://bcache.evilpiepirate.org/
10. EnhanceIO. https://github.com/stec-inc/EnhanceIO
11. H. P. Chang and C. P. Chiang, "PARC: a novel OS cache manager," *Softw Pract Exper*, Vol. 48, no. 12, pp. 2193-2222, 2018. https://doi.org/10.1002/spe.2633.

12. H. P. Chang, S. Y. Liao, D. W. Chang, G. W. Chen, "Profit data caching and hybrid disk-aware completely fair queuing scheduling algorithms for hybrid disks," *Softw Pract Exper*, vol. 45, no. 9, pp. 1229-1249, 2015. https://doi.org/10.1002/spe.2279.
13. S. Ahmadian, R. Salkhordeh, and H. Asadi, "Lbica: A load balancer for i/o cache architectures," in *Proc. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, 1196–1201. https://doi.org/10.23919/DATE.2019.8714805.
14. L. Yin, L. Wang, Y. Zhang, and Y. Peng, "NUDTMapperX: adaptive metadata maintenance for fast crash recovery of dm-cache based hybrid storage devices," in *Proc. Usenix Annual Technical Conference (ATC)*, 2021.
15. S. W. Schlosse, J. Schindler, S. Papadomanolakis, et al., "On multidimensional data and modern disks," in *Proc. the USENIX Conf. on File Storage Technol.*, 2005, pp. 225-238.
16. R. J. Feiertag and E. I. Organick EI, "The multics input/output system," in *Proc. the 3rd ACM Symp. on Operating Systems Principles*, Oct. 1971, pp. 35-41. https://doi.org/10.1145/850614.850622.
17. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix," *ACM Trans. on Comput. Syst.*, vol. 2, no. 3, pp. 181-197, 1984. https://doi.org/10.1145/989.990.
18. T. Z. Teng and R. A. Gumaer, "Managing ibm database 2 buffers to maximize performance," *IBM Sys. J.*, vol. 23, no. 2, pp. 211-218, 1984. https://doi.org/10.1147/sj.232.0211
19. P. Mead, "Oracle Rdb buffer management,www.oracle.com/technology/products/rdb/pdf/2002tech forums/rdbtf 2002buffer.pdf," 2002.
20. Jianwei Liao, "Server-side prefetching in distributed file systems," *Concurrency Computat: Pract Exper.*, vol.28, pp. 294-310, 2016. https://doi.org/10.1002/cpe.3432.
21. Liao J, Gerofi B, Lien G-Y, et al., "A flexible I/O arbitration framework for netCDF-based big data processing workflows on high-end supercomputers," *Concurrency Computat: Pract Exper.*, vol.29, e4161, 2017. https://doi.org/10.1002/cpe.4161.
22. Dong F, Zhou P, Liu Z, Shen D,XuZ, LuoJ, "Towards a fast and secure design for enterprise-oriented cloud storage systems," *Concurrency Computat: Pract Exper.*, vol. 29, e4177, 2017. https://doi.org/10.1002/cpe.4177.
23. Y. Liang, R. Pan, Y. Du, C. Fu, L. Shi, T. W. Kuo, and C. J. Xue, "Read-Ahead Efficiency on Mobile Devices: Observation, Characterization, and Optimization," *IEEE Transactions on Computers*, Vol. 70, No. 1, Jan. 2021. https://doi.org/10.1109/TC.2020.2984755
24. G. Castets, P. Crowhurst, S. Garraway, and G. Rebmann, "IBM total storage enterprise storage server model 800," *IBM Redbook*, 2002.
25. F. Wu, H. Xi, J. Li, "Linux readahead: less tricks for more," *in Proc. the Linux Symp.*, 2007, pp. 273–284.
26. P. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 performance improvement through readahead optimization," in *Proc. the Linux Symp.*, 2004, pp. 391-402.
27. F. Wu, H. Xi, and C. Xu, "On the design of a new linux readahead framework," *ACM SIGOPS Operat. Syst. Review*, vol. 42, no. 5, pp. 75-84, 2008. https://doi.org/10.1145/1400097.1400106.
28. B. S. Gill and D. S. Modha, "SARC: sequential prefetching in adaptive replacement cache," in *Proc. the USENIX Annu. Tech. Conf.*, 2005, pp. 292-308.
29. H.-P. Chang, C.-Y. Chen, C.-Y. Liu, "A prefetching scheme for multi-tiered storage systems," *in Proc. 15th IEEE International Conference on Advanced and Trusted Computing (ATC)*, 2018. https://doi.org/10.1109/SmartWorld.2018.00271
30. A. J. Uppal, R. C. Chiang, H. H. Huang, "Flashy prefetching for high-performance flash drives," in *Proc. IEEE 28th Symp. on Mass Storage Syst. and Technol. (MSST)*, 2012. https://doi.org/10.1109/MSST.2012.6232367.
31. A. E. Papathanasiou and M. L. Scott, "Aggressive prefetching: an idea whose time has come," in *Proc. Conf. on Hot Topics in Operating Syst (HotOS)*, 2005, Art. no. 6.
32. FIO benchmark. https://github.com/axboe/fio
33. Postmark, http://postmarkapp.com/
34. TPC-H. http://www.tpc.org/tpch/
35. Monetdb. https://www.monetdb.org/Home
36. C. Li, K. Shen, and A. E. Papathanasiou, "Competitive prefetching for concurrent sequential I/O," in *Proc. the ACM European Conf. on Comput. Syst. (EuroSys)*, March 2007, pp. 189-202. https://doi.org/10.1145/1272996.1273017.
37. S. Liang, S. Jiang, Z. Zhang, "STEP: sequentiality and thrashing detection based prefetching to improve performance of networked storage servers," in *Proc. Conf. Distributed Computing Syst.* (*ICDCS)*, July 2007. https://doi.org/10.1109/ICDCS.2007.141.
38. A. D. Bathen and B. S. Gill, "AMP: adaptive multi-stream prefetching in a shared cache," in *Proc. 5th USENIX Conf. File and Storage Technol. (FAST)*, 2007, Art. no.. 26.
39. C. Li and K. Shen, "Managing prefetch memory for data-intensive online servers," in *Proc. the 4th USENIX Conf. File and Storage Technol.*; 2005, Art. no. 19.
40. M. Li, E. Varki, S. Bhatia, and A. Merchant, "TaP: table-based prefetching for storage caches," in *Proc. USENIX Conf. File and Storage Technol.*, 2008, Art. no. 6.

41. Z. Li, Z. Chen, and Y. Zhou, "Mining block correlations to improve storage performance," *ACM Trans. on Storage*, vol. 1, no. 2, 2005, pp. 213-245.. https://doi.org/10.1145/1063786.1063790.
42. G. Soundararajan, M. Mihailescu, C. Amza, "Context-aware prefetching at the storage server," in *Proc. USENIX Annu. Tech. Conf.*; 2008, pp. 377-390.
43. J. Ryu, D. Lee, K. G. Shin, and K. Kang, "ClusterFetch: A lightweight prefetcher for intensive disk reads," *IEEE Trans. Comput.*, vol. 67, no. 2, pp. 284-290, Feb. 2018. https://doi.org/10.1109/TC.2017.2748939
44. S. Jiang, X. Ding, Y. Xu, and Davis K, "A prefetching scheme exploiting both data layout and access history on disk," *ACM Trans. on Storage*, vol. 9, no. 3, 2013, Art. no. 10. https://doi.org/10.1145/2508010.
45. G. Griffioen, "Performance measurements of automatic prefetching," in *Proc. Conf. on Parallel and Distributed Computing Syst.*, 1995, pp. 165-170.
46. H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *Proc. the USENIX Annu. Tech. Conf.*, Jan. 1997, pp. 21-32.
47. T. M. Kroeger and D. E. Long, "Design and implementation of a predictive file prefetching algorithm," in *Proc. the USENIX Annu. Tech. Conf.*, 2001, pp. 105-118.
48. G. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *Proc. the USENIX Summer Tech. Conf.*, Jun. 1994, pp. 13-23.
49. K. M. Curewitz, P. Krishnan and J. S. Vitter, "Practical prefetching via data compression," in *Proc. ACM SIGMOD International Conf. on Management of Data (SIGMOD)*, 1993, pp. 257-266. https://doi.org/10.1145/170036.170077.
50. T. M. Kroeger and D. D. Long, "Predicting file system actions from prior events," in *Proc. USENIX Annu. Tech. Conf.*, Jan. 1996, pp. 26-35.
51. Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: quick application launch on solid-state drives," in *Proc. USENIX Conf. on File and Storage Technol.*, Feb. 2011, Art. no. 19.
52. A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff, "Lynx: a learning linux prefetching mechanism for SSD performance model," in *Proc. Non-Volatile Memory Syst. and Applications Symp. (NVMSA)*, Aug. 2016, pp. 1-6.. https://doi.org/10.1109/NVMSA.2016.7547186
53. P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Trans. on Comput. Syst.*, vol. 14, pp. 311–343, 1996. https://doi.org/10.1145/235543.235544.
54. R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, " ," in *Proc. the 15th ACM Symp. on Operating Syst. Principles (SOSP)*, Dec. 1995, pp. 79-95. https://doi.org/10.1145/224056.224064.
55. F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proc. 3rd USENIX Symp. on Operating Syst. Design and Implementation (OSDI)*, 1999, pp. 1-14.
56. T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic compiler-inserted i/o prefetching for out-ofcore applications," in *Proc. the USENIX Symp. on Operating Syst. Design and Implementation (OSDI)*; Oct. 1996, pp. 3-17.
57. S. W. Son, S. P. Muralidhara, O. Ozturk, M. Kandemir, I. Kolcu, M. Karakoy, "Profiler and compiler assisted adaptive I/O prefetching for shared storage caches," in *Proc. Conf. Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 112-121. https://doi.org/10.1145/1454115.1454133