PAPER

An Asynchronous Striping-Aware Readahead Framework for Disk Arrays in Linux

Sung Hoon BAEK[†a)], ***Member***

SUMMARY Disk arrays and prefetching schemes are used to mitigate the performance gap between main memory and disks. This paper presents a new problem that arises if prefetching schemes that are widely used in operation systems are applied to disk arrays. The key point of the problem is that block address space from the viewpoint of the host is contiguous but from that of the disk array it is discontiguous and thus more disk accesses than expected are required. This paper presents two ways to resolve the problem that arises from the Linux readahead framework. The proposed scheme prevents a readahead window from being split into multiple requests from the viewpoint of the disk array but not from the viewpoint of the host thereby reducing disk head movements. In addition, it outperforms the prior work by adopting an asynchronous solution, improving performance for fragmented files, eliminating readahead size restriction, and improving disk parallelism. We implemented the proposed scheme and integrated it with Linux. Our experiment shows that the solution significantly improved the original Linux readahead framework when a storage server processes multiple concurrent requests.

key words: *storage, parallel I/O, prefetching, disk array*

1. Introduction

The latency gap between main memory and spinning disks has widened to 5-order-of magnitude [1]. To reduce the gap, prefetching is widely used; disk prefetching (1) makes data available in a cache before an application requests the data, thereby overlapping processors and disk activities; and (2) aggregates many small sequential requests into a larger one, thereby reducing costly disk rotations and seeks.

Disk arrays are also used to mitigate the gap with parallelism of multiple disks [2]. Many researchers have designed prefetching schemes combined with disk arrays. These schemes focus on disk parallelism and cache utilization [3]–[6]. The present paper describes an unknown problem that arises if widely used prefetching schemes in practical operating systems are applied to disk arrays. The key point of the problem is that block address space from the viewpoint of the host is contiguous but from that of the disk array is discontiguous. Due to this property, a single read request that is laid across disks splits into multiple disk accesses. This gives disk parallelism for a single I/O but many request splits for multiple concurrent I/Os degrade performance by causing many disk accesses. File servers that process concurrent file services are severely affected by request splits if the prefetching ignores the striped data layout. This is described in detail in Sect. 3.1.

Our prior article unveiled the aforementioned problem for the first time but it has several weak points [7]. This paper illustrates how the asynchronous sequential readahead scheme of Linux gets read requests laid across multiple disks and makes more disk accesses. In comparison with our prior solution, the new scheme presented in this paper was greatly improved with the following advantages: It (1) prevents I/O requests by an asynchronous sequential prefetching, namely, the latest Linux readahead, from making more disk accesses in disk arrays; (2) has no negative effect on file fragmentation; (3) eliminates the restriction of the readahead size; and (4) includes an integrated solution to prevent parallelism loss. We implemented the solutions and integrated them in Linux 2.6.35. Our experiment shows that the proposed approach significantly outperforms the prior works.

2. Prior Works

2.1 Prefetching (readahead)

Many researchers have proposed various forms of history-based prefetching, which analyze the past access history to predict future accesses. Some of them preload data from disks to flash memories or solid state disks (SSD) using the usage pattern over time [8], [9]. Preloaded data in a flash or a SSD can be served promptly with low latency, as opposed to disks. Mining and analyzing past history requires high computing and memory resources, and thus some researchers have proposed interfaces for applications to provide future access hints to kernel [10]. Some schemes load a large amount of data to memory at once anticipating spatial locality and then find and evict uselessly prefetched data at a later time [4].

The most practical prefetching approach is sequential prefetching, which is widely implemented in various operating systems. The basic form of sequential prefetching is the one block lookahead (OBL) method, which initiates a read for block b+1 when block b is requested from an application. Many variations of OBL have been proposed. Among these, p-block lookahead reads in advance additional p blocks followed by the requested block. Other schemes adaptively increase or decrease the prefetching size p [3], [4], [11]–[14].

Recent sequential prefetching schemes are categorized as synchronous sequential prefetching and asynchronous

Manuscript received June 5, 2012.
Manuscript revised August 20, 2012.

[†]The author is with the Dept. of Computer System Engineering, Jungwon University, Goesan-eup, 367–805 Korea.
a) E-mail: shbaek@jwu.ac.kr

DOI: 10.1587/transinf.E96.D.19

prefetching. Synchronous methods double the prefetching size *p* for every subsequent cache miss up to a predetermined value as an application requests a file sequentially. Asynchronous methods conduct prefetching from block *b+p+1* before a cache miss (Prefetching blocks *b* to *b+p* is conducted in the previous prefetching) when an application hits block *b+g*, which is a preset fraction of the previous prefetching group, where *g* is less than *p*.

2.2 Readahead in Linux

Prefetching provides numerous benefits to many applications because sequential accesses are frequent in general usages. Sequential prefetching, known as readahead in Linux, is designed to achieve two goals. (1) One goal is to improve the efficiency of storage devices by converting lots of tiny sequential reads into a large access, which significantly reduces costly disk rotations and seeks. As the request size becomes larger, the costly disk head movement over the total I/O time becomes accordingly smaller. (2) The other goal is to overlap disk operations with processors by means of asynchronous reads that request sequential blocks in advance before they are requested from an application. Hence it hides I/O latency for applications [15], [16].

2.2.1 Basic Operations

The new readahead policy in Linux 2.6.23 is a kind of asynchronous sequential prefetching. Figure 1 shows an example of the Linux readahead framework. The variable *size* indicates the read size for the current readahead. The variable *start* indicates the currently requested page. The variable *async_size* denotes the number of pages that are included in the readahead operation except for the pages requested from the application. Linux marks the page at *start* + *size* - *async_size* with PG_readahead, which is used to trigger the next asynchronous read when an application requests it in the near future.

The read size, *size*, is incremented by four times if the previous size is 1/16 or less of the maximum prefetching size; otherwise, *size* doubles up to the maximum prefetching size. If this request is the first for the file or the current request is considered as random, an initial window is given for the current readahead.

The significantly improved memory capacity and bandwidth makes prefetching misses less relevant to the overall performance. Hence, the new sequentiality detection policy in Linux 2.6.23 or later versions changed from strict page-after-page pattern matching to a more aggressive policy. Instead of the former policy, Linux allows sequential reads mixed with random reads and concurrent multiple streams in a single file descriptor [16].

To detect sequentiality with a small overhead, Linux uses a readahead window, which is contained in each file descriptor and consists of *start*, *size*, and *async_size* as shown in Fig. 1. A single sequential read can be simply detected by the readahead window. In most cases, if the current request offset is the next page of the previous window (current request offset = *start* + *size*), the request is treated as sequential. If a random seek is mixed with a sequential read, the readahead window can be spoiled by the random seek, However, if a random seek intervenes between sequential reads, Linux can quickly recover a sequential readahead window by scanning forward in the page cache from the current page to the first missing page.

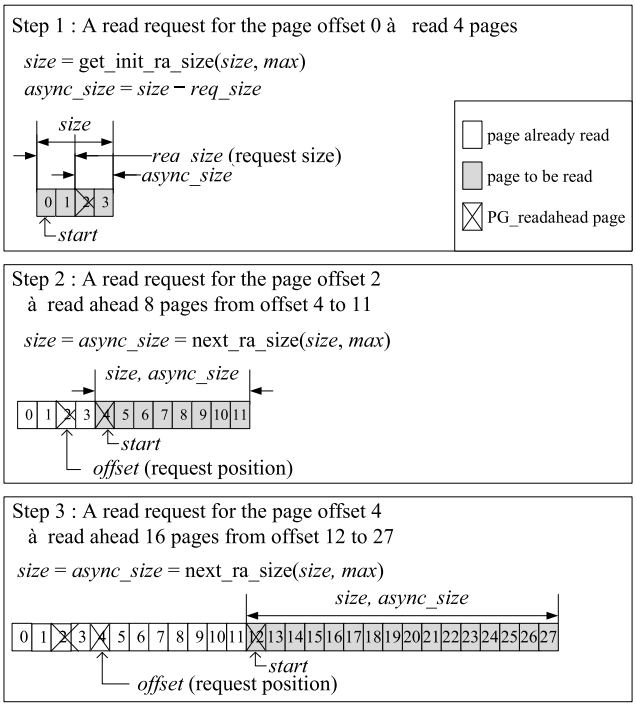

Fig. 1 The figure shows a general example of the Linux readahead framework. The variable, *size*, indicates the read size for each readahead operation. The variable, *async_size*, denotes the number of pages that are included in the read operation except for the pages requested from an application.

2.2.2 An Operation Example

Figure 1 shows an example when an application sequentially reads a file. In the first step, a Linux kernel function get_init_ra_size() determines the initial window as 4 pages if the request size is 2 pages and the maximum readahead size, *max*, is 32 pages. The first page (page 2) that is not requested by the application but included in this readahead window is marked as PG_readahead. PG_readahead marks are maintained in the page cache so they need no additional memory. In the second step, when the application requests the PG_readahead page (Page 2) that is already loaded in the memory by the previous readahead, the readahead size doubles as 8 pages, and 8 pages from offset 4 to 11 are read in advance, where Page 4 is marked as PG_readahead. In the third step, Linux asynchronously reads 16 pages from offset 12 to 27 with twice the size of the previous one when the application hits the latest PG_readahead page (Page 4).

If the storage device is congested, readahead is trig-

gered only by a cache miss but not by a cache hit on a PG_readahead page. In other words, asynchronous reads are not permitted if the storage device is congested. The Linux kernel considers the storage device as congested when the number of I/O requests that are waiting to be delivered to the device in the request queue exceeds 7/8 of the maximum number of requests that are allowed in the request queue. The maximum number is 128 in general.

3. Prefetching Problem in Striped Arrays

Other studies on disk arrays have focused on reliability [17]–[19] of redundant disk arrays of independent disks (RAID), performance by devising variants [20]–[22] of RAID, cache [6], [14], [23], [24], data deduplication [25], [26], and storage virtualization [27]. The present paper deals with the very significant problem that performance degradation occurs if sequential prefetching is incorporated with striped disk arrays such as RAID-0, RAID-5, RAID-6, RAID-10, and RAID-50 that exploit striped data placement.

3.1 Request Split in Disk Arrays

The disk array is composed of multiple disks and serves as a single virtual disk with a logical address space from the viewpoint of the host. Data blocks are striped across the member disks for requests to be evenly distributed over disks. Figure 2 shows strips and stripes in a RAID-0 array consisting of two disks. The minimum contiguous blocks in each disk is called a strip, which is called a chunk in the software RAID module of Linux (Multi-device). Strips from each disk comprise a stripe.

Figure 2 shows that contiguous logical block addresses are not contiguous in the physical components. In the physical layout, the logical block address 3 is adjacent to the logical block address 8 but not to the logical block address 4, which is in the other member disk.

This characteristic can split a single request from the host into multiple requests in the disk arrays. We call this request split. Request splits result in performance degradation when the system serves many concurrent I/Os. Figure 2 (a) shows an example of this problem. If the host requests two concurrent read commands ranging from logical block address 3 to 6 and 14 to 17, respectively, the blocks are divided into four discontiguous regions and thus the disk array handles four independent operations (two operations per disk). Meanwhile, in Fig. 2 (b), there are only two discontiguous regions for the two host requests because each request is within a strip boundary.

In other words, even though the two different cases shown in Fig. 2 (a) and Fig. 2 (b) transfer the same number of blocks, the case of Fig. 2 (a) requires four disk accesses (rotations and seeks) but the case of Fig. 2 (b) requires half of that because each request is aligned in strips. Therefore, aligned requests can reduce mechanical operations, which are the major cause of disk latency.

If a request is placed across strip boundaries, it is divided into two or more disk accesses. We call this a request split, which degrades performance with more disk accesses for concurrent I/Os that have many requests pending in disks. However, current operating systems ignore request splits.

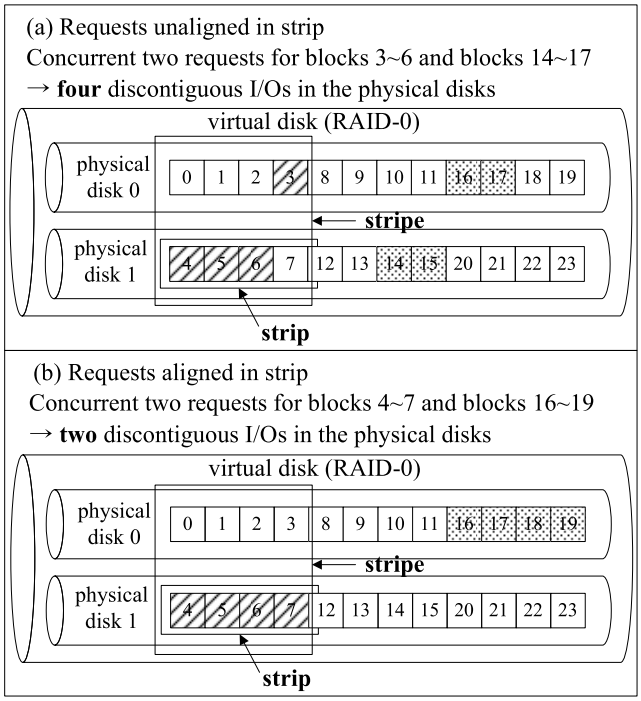

Fig. 2 (a) Two concurrent read requests ranging from logical block address 3 to 6 and 14 to 17 are divided into four discontiguous regions and thus the disk array handles four independent operations. (b) There are two discontiguous regions for two host requests because each request is aligned in strips

A request split has no negative effects on a single stream, for which a request split utilizes disk parallelism. However, because many concurrent I/Os cause every disk to be busy, barring the request split has a positive effect on performance by reducing the total number of requests.

3.2 Request Split in the Linux Readahead

Figure 3 shows request splits caused by the Linux readahead. This example assumes as the following: without file fragmentation, a file is sequentially stored from the logical address 0 in a RAID-0 array with two disks, both the maximum readahead size, *max*, and the strip size are 16 pages, and an application sequentially reads the file with the Linux readahead framework.

Figure 3 omits Steps 1 and 2 because they are the same as those in Fig. 1. In Step 3, the application requested Page 4 that is marked as PG_readahead, and then the request triggers an asynchronous readahead for 16 pages from offset 12 to 27, which are split into two member disks. In Step 4, if an application requests the latest PG_readahead page (Page 12), the next readahead for pages 28 to 43 is issued, but it is also divided into two disks. The next of the next readahead

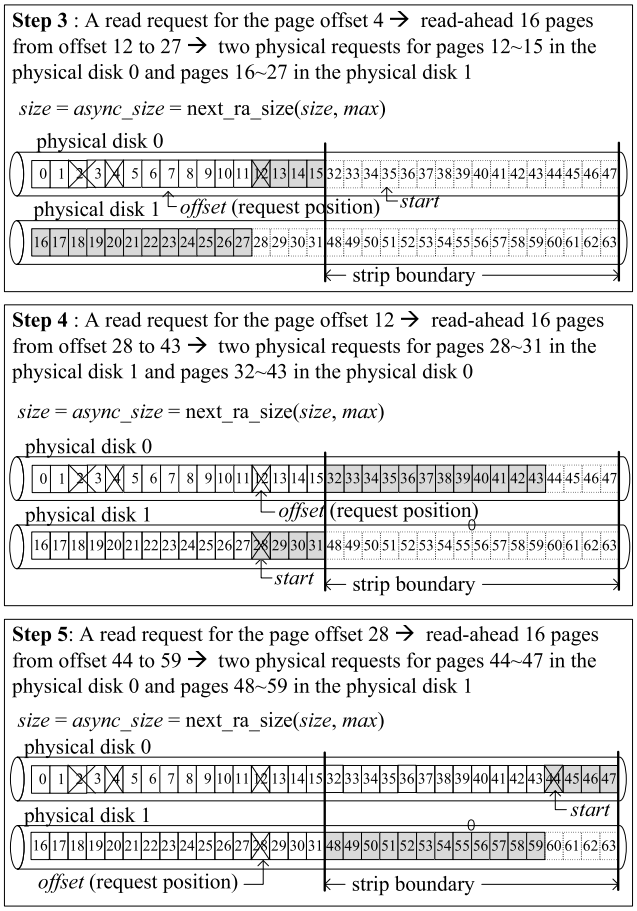

Fig. 3 Readahead windows are unaligned in the original Linux readahead framework. Every readahead window is split into two disks.

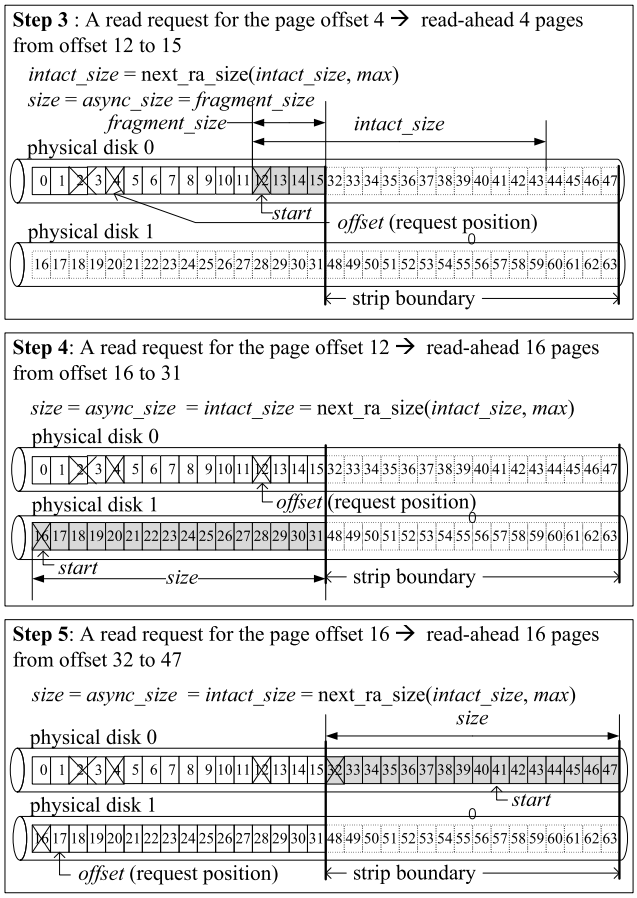

Fig. 4 The basic SARA prevents request splits by shrinking the readahead window.

will also be split into two disk operations as shown in Step 5.

The Linux readahead framework treats a disk array as a single disk ignoring that contiguous logical address space is discontiguous from the viewpoint of the physical address space. Hence, the Linux readahead framework causes request splits and degrades the performance of disk arrays.

4. Solution: Striping-Aware Readahead

4.1 Basic SARA: Asynchronous Approach

The new readahead policies in Linux 2.6.23 adopted an asynchronous approach instead of the synchronous one of the previous version. Hence, we newly designed and implemented SARA as an asynchronous readahead to be integrated with the new Linux readahead framework. Figure 4 shows how to align prefetching requests in strips by shrinking a prefetching request. SARA inspects the physical locations for readahead pages for every readahead to investigate whether a readahead lies across two or more strips,. For every readahead that is unaligned in a strip, SARA modifies the readahead size to prevent request splits.

Steps 1 and 2 are omitted in Fig. 4 because they are the same as those in Fig. 1. In Step 3, when an application requests Page 4 marked as PG_readahead, SARA performs the original readahead framework to make an original readahead window (Pages 12~27) when an application requests the latest PG_readahead page (Page 4). SARA (i) asks the file system where the physical blocks corresponding to the readahead window are, (ii) detects if the original readahead window (Pages 12~27) is fragmented across two disks (see Step 3 of Fig. 3), (iii) inspects which part of the physical blocks for the readahead window are in a single strip, and (iv) shrinks the readahead window as Pages 12~15 to be dedicated to a single strip. Namely, the original readahead window size, *intact_size*, was 16 in Step 3, and SARA reduces the readahead window size, *size*, to 4, where *size* does not shrink below the request size, *req_size*.

In Step 4, when the application requests Page 12 marked as PG_readahead, a readahead for Pages 16 to 31 is issued, where the readahead size, *size*, is calculated from *intact_size* of the previous step instead of the previous *size*. This readahead window is automatically aligned in a strip, thanks to the modification of the previous step. Variable *async_size* is the amount of asynchronous prefetching. It is equal to *size* in most cases.

The current prefetching size doubles or quadruples

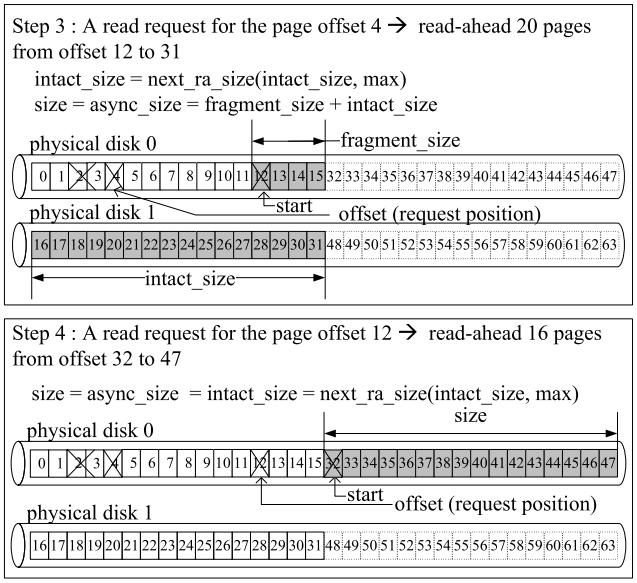

Fig. 5 The extended SARA prevents request splits by extending the readahead size, but the end of the readahead window is aligned in a strip.

from the previous prefetching up to the maximum size (*max*). If the previous prefetching had been shrunk to be aligned in strips, then the current one should not be calculated from the previous reduced one but from the unchanged one (*intact_size*). The variable *intact_size* is the unmodified readahead size before being shrunk.

In SARA, the current *intact_size* is calculated from the previous *intact_size* that was determined in the previous step. The current readahead size, *size*, becomes equal to the current *intact_size* if there is no request split. Otherwise, *size* shrink from the current *intact_size* to fit the readahead window to a single strip.

4.2 Extended SARA

4.2.1 File Fragmentation

The basic SARA shrinking the readahead size degrades the performance when files are too fragmented in the file system level. In the worst case where all blocks of a file are fragmented at the file system level, *size* never becomes greater than *req_size* in the basic SARA even though an application sequentially reads the file.

The basic SARA restricts the window size within the file-fragmented region because fragmented blocks are not aligned in strips, thereby producing many small readaheads. The readahead window that is shrunk by file fragmentation can be resolved by extending the readahead size rather than shrink it for the next readahead to be aligned in strips as shown in Fig. 5. The extending method prevents the readahead size, *size*, from being smaller than the unmodified readahead size, *intact_size*.

Figure 5 exemplifies the extending method, where a file is not fragmented. In Step 3 of Fig. 5, first, SARA asks the file system where the physical blocks corresponding to the readahead window are located. Second, it inspects whether all of the physical blocks for the readahead window are in a single strip. If there is a discontinuous point in the physical blocks, it finds, *fragment_size*, which is the greatest readahead size that lies only in a single strip. The resolved readahead size, *size*, is then extended from *intact_size* to (*intact_size* + *fragment_size*). After this resolved readahead, we can expect that the next readahead will be aligned in a strip without changing size, as shown in Step 4 of Fig. 5.

The extending method works well even if a file is severely fragmented. It bars *size* from being smaller than *intact_size*, because *size* is determined as '*intact_size* + *fragment_size*'; hence it prevents not only request splits but also small readahead regions caused by file fragmentations.

The Linux readahead framework is in the memory manager at the file system level, where we can inspect which block the logical offset in a file corresponds to using the argument *mapping* that represents the address space of the file. A function, *mapping*->a_pos->bmap(*mapping*, logical_sector_in_file), returns the physical sector address for the logical sector of the file.

4.2.2 To Prevent Readahead Size Restriction

If the main memory is large enough to avoid thrashing, a large *max* improves performance. However, the basic SARA prevents the readahead size, *size*, from exceeding the strip size even though we increase the maximum readahead size, *max*. Meanwhile, the extended approach shown in Fig. 5 allows *size* to be extended up to '*max* + *strip size*', because the readahead size is determined by "*fragmented_size* + *intact_size*", where *intact_size* is the unmodified readahead size that can increase up to *max* and *fragmented_size* is less than or equal to the strip size.

If *max* is much greater than the stripe size and the main memory is large enough, the performance can be improved. The basic SARA restricts the readahead size within the strip size, thus limiting the performance even though the sufficient main memory are provided, meanwhile the extended operation allows administrators to increase *max* above the strip size.

4.2.3 Disk Parallelism for a Low Degree of Concurrency

SARA solves the problem caused by request splits when there are many concurrent reads. However, if the degree of concurrency is very low (the number of concurrent I/Os is less than a predetermined threshold), small readahead windows aligned in strips make most of the member disks in a disk array idle. For example, if there is only a single read, only one of the member disks serves but the others become idle.

Because a low degree of concurrent I/O does not suffer from lack of memory, a large readahead size that covers all member disks of a disk array utilizes disk parallelism and provides better performance than the strip-bounded readahead.

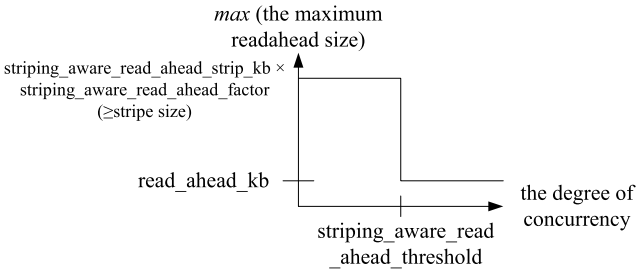

Fig. 6 For a low degree of concurrency, the extended SARA changes *max* to a greater value for disk parallelism. If the degree of concurrency measured by the number of pending I/O per disk is smaller than the threshold value (striping_aware_read_ahead_-threshold), *max* is chosen as a large value that is much greater than the stripe size in general.

We need a consolidated solution in the readahead framework without relying on the unrelated layer. We propose a solution that detects the exact degree of concurrency and solves the dilemma in the SARA framework without an unrelated layer. Figure 6 shows the proposed solution, SARA, dynamically changes *max* based on the degree of concurrency, which is measured by the number of pending I/Os that are waiting to be requested to disks or being served in disks.

At the moment when the number of pending I/Os per disk is greater than the concurrency threshold value, the degree of concurrency is considered to be high. Otherwise, we consider it as low. The concurrency threshold value is exposed to the sys file system as /sys/block/⟨device name⟩/queue/striping_-aware_read_ahead_threshold.

When a high degree of concurrency is detected in real time, SARA selects a small *max*, which is chosen as the strip size in general cases. For a low degree of concurrency, SARA selects a large *max* that is greater than the stripe size. The large *max* is twice the number of member disks in default.

SARA uses a large *max* when the number of pending I/Os drops down for a very short time. When the number of I/Os soars and then goes down abruptly, SARA uses a small *max* for a short time, but this is an undesired operation. To prevent frequent changes of *max* depending on the fluctuating number of pending I/Os, we use an exponential moving average of the degree of concurrency by the following operation that is executed whenever a prefetching is demanded.

$$D \leftarrow C \times D + (1 - C) \times N$$

where, D is the smoothed degree of concurrency, N is the current number of pending I/Os, and C is a constant that is greater than zero and less than one. We chose C as 0.90 in our experiment.

5. Experimental Results

We implemented SARA in Linux kernel 2.6.35. The system in the experiments uses three SATA-II disks (the model name is WD10EVVS), which are configured as a RAID-0 array using the software RAID in Linux, namely Multi-

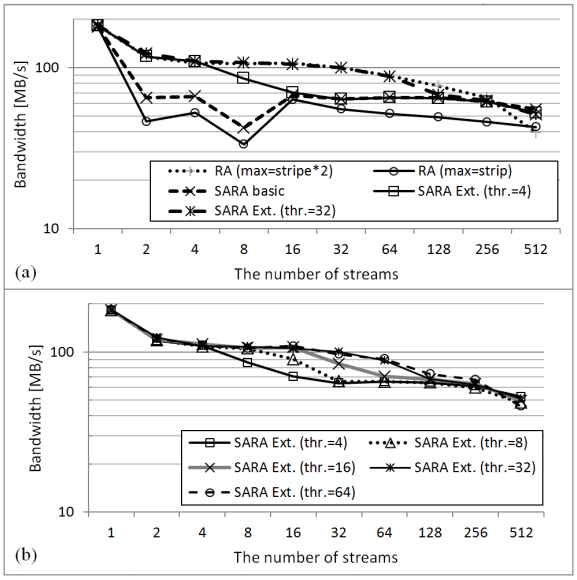

Fig. 7 The figure shows the performance for multiple streams with 2 GiB of main memory and 256 KiB strip size. (a) 'SARA basic' and 'RA (max=strip)' consume the same memory but 'SARA basic' outperforms 'RA (max=strip)'. 'RA (max=stripe*2)' experiences thrashing at 512 streams but 'SARA ext. (thr.=32)' outperforms the others in a wide range of concurrency. (b) Whether the degree of concurrency is high or low depends on the concurrency threshold value. A high degree of concurrency uses the small *max* that is assigned by striping_aware_read_ahead_kb in the sys file system.

device. The kernel was compiled for an x86_64 architecture and hosted an ext4 file system and a completely-fair-queuing disk scheduler.

Our scheme, SARA, is integrated with Linux. Using *sys* file system, we can turned on or off SARA. SARA can be manually turn off by assigning 0 to '/sys/block/⟨device name⟩/queue/striping_aware_read_ahead_strip_kb', which indicates the strip size of the device. The *sys fs* interface, 'read_ahead_kb' and 'striping_aware_read_ahead_strip_kb × striping_aware_read_ahead_factor', specify *max* and a large *max* for a high degree of concurrency, respectively.

Throughout the experiments, we compare the original readahead (RA) feature of Linux with the striping-aware readahead (SARA). 'RA (max=stripe*2)' and 'RA (max=strip)' in the figures indicate RA with the maximum readahead size *max* of twice the stripe size and with *max* of the strip size, respectively. 'SARA basic' corresponds to the basic SARA. 'SARA Ext. (thr.=4)' and 'SARA Ext. (thr.=32)' denotes for the extended SARA with the concurrency threshold of 4 and 32, respectively. The maximum readahead size *max* is equal to the strip size if there is no explicit indication. The software RAID (Multi-device) uses twice the stripe size of *max* in default but the extended SARA prefers a strip size of *max* for a high degree of concurrency.

Figure 7 compares the performance when multiple sequential streams are requested to a single disk array. This experiment evaluates a video-on-demand server. 'SARA basic' outperforms 'RA (max=strip)'; both have the same

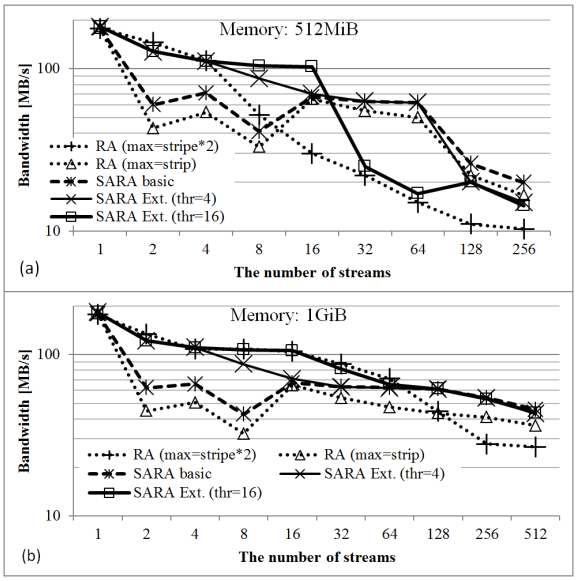

Fig. 8 Performance for multiple streaming with various memory sizes. 'RA (max=stripe*2)' causes thrashing because it requires tremendous memory in comparison with the others. 'SARA basic' outperforms 'RA (max=strip)' even though they consume the same amount of memory. 'SARA ext(thr.=16)' shows the best performance in a wide range of memory sizes.

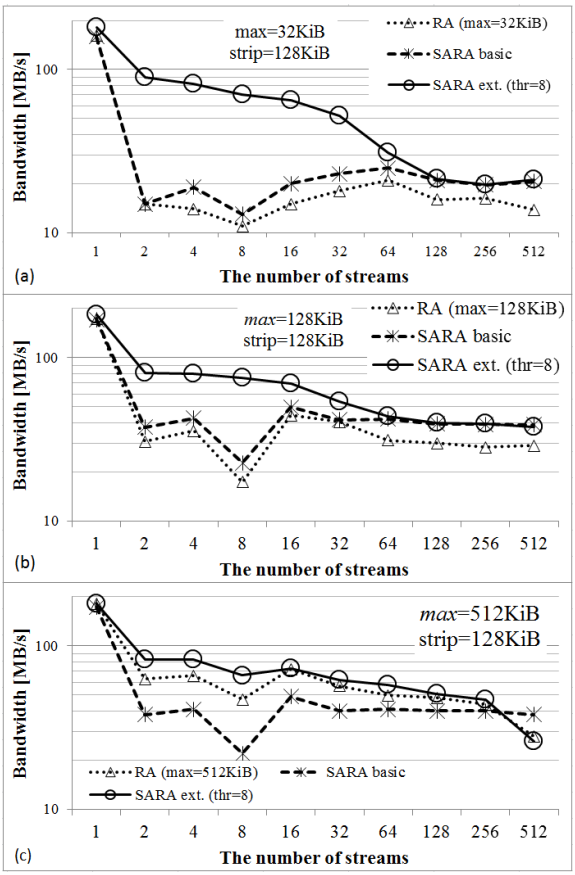

Fig. 9 Streaming performance by varying the maximum readahead size *max*. The maximum readahead size in 'SARA basic' cannot exceed the strip size but SARA ext. can; thus 'SARA ext.' can benefit from a large *max* that is greater than the strip size given that enough memory is provided. The large *max* in 'SARA ext.' was set to twice the stripe size. The small *max* in 'SARA ext.' is equal to the indicated *max* for each subfigure. This experiment was performed with a strip size of 128 KiB and main memory of 1 GiB.

max, but SARA prevents request splits. 'SARA ext.(thr.=4)' outperforms 'SARA basic' at 2 to 8 streams because, with a low degree of concurrency, small *max* by SARA basic and RA does not utilizes all disks in parallel as explained in Sect. 4.2.3. SARA ext. increases *max* to twice the stripe size for a small number of streams. 'RA (max=stripe*2)' has nearly the same curve as 'SARA ext.(thr.=32)' but its performance drops at 512 streams due to a lack of memory unlike SARA ext. Figure 7 shows that SARA ext. is superior to the others for various cases.

Figure 7 (b) shows the performance by varying the concurrency threshold value, which determines how many concurrency I/Os are required to decrease *max* to a small value from a large value. In this experiment, the large *max* is 1.5 MiB, which is twice the stripe size and the small *max* is 256 KiB, which is equal to the strip size. A greater *max* could retain high performance until suffering from lack of memory but it might cause thrashing, because more streams consume much readahead memory. For example, 1024 streams need 256 MiB of readahead memory with 256 KiB of *max* and 1.5 GiB of readahead memory with 1.5 MiB of *max*, respectively.

A large *max* could lend a good performance as long as sufficient memory is provided. However, with a small memory, prefetched data could be evicted before they are used in the near future due to a lack of memory, and they then must be read from disks again. Such performance degradation is called thrashing. Figure 8 shows how the memory size influences each scheme.

As shown in Fig. 8 (a), 'RA(max=stripe*2)', which has the largest *max* shows significantly poor performance due to thrashing with 512 MiB of main memory. It experiences thrashing for 256 or more streams with 1 GiB of memory. 'SARA ext.(thr.=16)' experiences thrashing for 32 and 64 streams in Fig. 8 (a) because it uses a large *max* (twice stripe size) for 64 or less streams. 'SARA ext.(thr.=4)' does not experience thrashing because it uses the large *max* for a smaller number of streams than 'SARA ext.(thr.=16)'. 'SARA basic' and 'RA (max=strip)' consume the same memory but 'SARA basic' always outperforms 'RA (max=strip)' in a wide range of memory.

The maximum readahead size in 'SARA basic' cannot exceed the strip size but 'SARA ext.' can, thus 'SARA ext.' benefits from a large *max* that is greater than the strip size. Figure 9 shows 'SARA ext.' is superior to 'SARA basic' for a wide range of *max*. 'SARA basic' outperforms RA due to request split removal if *max* is equal to or less than the strip size as shown in Fig. 9 (a) and Fig. 9 (b). Otherwise, 'SARA basic' is inferior to both 'SARA ext.' and RA. A small *max* that is restricted by 'SARA basic' can benefit when required readahead memory exceeds the physical memory, as in the

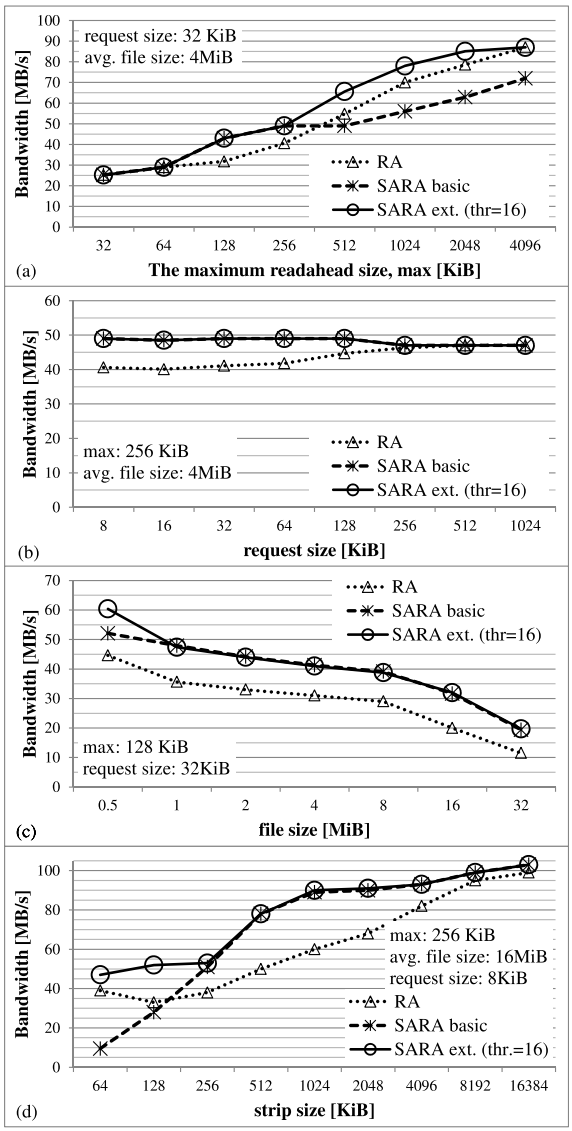

Fig. 10 File benchmark simulates a file server by varying (a) the maximum readahead size *max*, (b) the request size, (c) the average file size, and (d) the strip size. This experiment uses 256 KiB of strip size, 2 GiB of main memory, 256 KiB of *max* and 32 threads by default. 'SARA ext.' outperforms both 'SARA basic' and RA in a variety of configurations. The small *max* of 'SARA ext.' is the same as the specified *max* for each figure. The marge *max* 'SARA ext.' is twice the stripe size.

case of 512 streams in Fig. 9 (c). However, 'SARA ext.' is superior to 'SARA basic' in a wide range of *max*.

Figure 10 shows a more realistic benchmark for file servers using thea benchmark FileBench 1.4.8, which compares RA, 'SARA basic', and 'SARA ext.' with a 'file server' workload personality with 64 threads and 2000 files of 4 MiB on average. This experiment simulates a read-intensive workload, high-performance computing workload, and single-producer many consumer workload of cloud storage servers [28], where we removed write operations from the workload because we only need read operations to compare them.

Figure 10 (a) shows the bandwidth of the system by

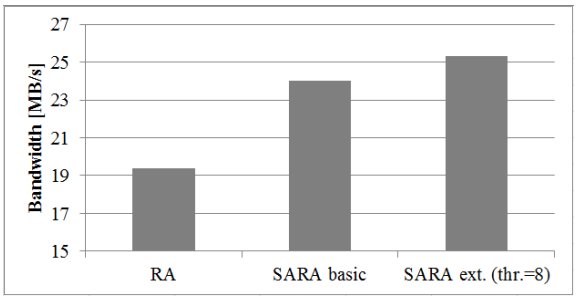

Fig. 11 The extended SARA is better than the basic SARA in an aged file system. This experiment is performed with 64 threads using FileBench. The strip size and *max* are 256 KiB.

varying the maximum prefetching size, *max*, from 32 KiB to 4 MiB with 256 KiB strip size. When *max* is greater than the strip size, 'SARA basic' is inferior to the others. The performance gap between RA and 'SARA ext.' is noticeable when *max* is 128 KiB or greater in Fig. 10 (a). SARA outperforms RA by 32% when *max* is 128 KiB. Even though the actual readahead size does not exceed the strip size (256 KiB) in 'SARA basic', the *max* of 1024 KiB shows better performance than the *max* of 512 KiB in 'SARA basic' because *max* affects the initial readahead size.

Figure 10 (b) shows the effect on the request size. If the request is equal to the strip size, RA rarely exhibits request splits as long as files are aligned in strips. In Fig. 10 (b), when the request size is 256 KiB, which is equal to the strip size, RA shows the same performance as SARA. However, SARA outperforms RA by about 20% with a smaller request size than the strip size. If the request size is much greater than the strip size, there is no difference between RA and SARA. The greater the request size is, the less the request splits appear.

Figure 10 (c) shows the performance by varying the average file size. Although SARA improves the performance in a wide range of file size, larger files tend to give SARA more benefits. When the average file size is 32 MiB and 0.5 MiB, SARA improves performance by 70% and 20%, respectively.

Figure 10 (d) shows the performance by varying the strip size. If the strip size is much greater than *max*, RA employs less request splits. However, RA is never superior to SARA ext. in a wide range of strip size. In addition, to fully utilize all disks in parallel for a low degree of concurrency, a huge strip size requires a huge readahead size that degrees performance for small I/Os.

Figure 11 shows that the extended SARA is better than the basic SARA in an aged file system. To artificially age a file system in this experiment, we repeated creation and deletion of files. The average file size is 256 KB. After aging the file system, we ran the benchmark so that prefetching performs with fragmented files.

6. Conclusion

By integrating the proposed scheme with Linux, we ana-

lyzed that a readahead region across multiple disks of a disk array doubles disk accesses. We call this problem request split. The extended SARA changes the readahead regions to be extended and aligned in strips so that request splits are prevented. In addition, it controls the maximum readahead size based on the number of pending I/Os to improve disk parallelism for a low degree of concurrency. The striping-aware readahead feature is beneficial for file servers or streaming servers, where most I/Os are sequential with concurrent I/Os.

The basic string-aware readahead scheme outperforms the original readahead given that both consume the same memory. The extended SARA is superior to the basic SARA for various numbers of streams and a wide range of the maximum readahead sizes, file sizes, request size, and memory sizes. Also, the extended SARA outperforms the basic SARA in an aged file system.

The proposed scheme was implemented in Linux 2.6 and the experiments were performed in a real system. The suggested scheme can be easily configured or disabled for compatibility by the *sys* file system interface, is applicable to a variety of realistic scenarios, and provides significant performance improvement in cloud storage servers and streaming servers.

References

[1] J. He, J. Bennett, and A. Snavely, "Dash-io: an empirical study of flash-based- io for HPC," Proc. 2010 TeraGrid Conference, Aug. 2010.

[2] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, "RAID: High-performance, reliable secondary storage," ACM Computing Surveys, vol.26, no.2, pp.145–185, June 1994.

[3] B.S. Gill and L.A.D. Bathen, "AMP: Adaptive multi-stream prefetching in a shared cache," Proc. 5th USENIX Conf. on File and Storage Technologies, pp.185–198, 2007.

[4] B.S. Gill and D.S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," Proc. USENIX Annual Technical Conference, pp.293–308, 2005.

[5] S.H. Lim, Y.W. Jeong, and K.H. Park, "Interactive media server with media synchronized raid storage system (nossdav)," Proc. Int'l Workshop on Network and Operating System Support for Digital Audio Video, pp.177–182, June 2005.

[6] S.H. Baek and K.H. Park, "Prefetching with adaptive cache culling for striped disk arrays," Proc. 2008 USENIX Ann. Technical Conf., pp.363–376, June 2008.

[7] S.H. Baek and K.H. Park, "Striping-aware sequential prefetching for independency and parallelism in disk arrays with concurrent accesses," IEEE Trans. Comput., vol.58, no.8, pp.1146–1152, Aug. 2009.

[8] Microsoft, "Windows pc accelerators," Oct. 2010. http://www.microsoft.com/whdc/system/sysperf/perfaccel.mspx.

[9] Y. Joo, J. Ryu, S. Park, and K. Shin, "FAST: Quick application launch on solid-state drives," Proc. USENIX Conf. File and Storage Technologies, Feb. 2011.

[10] S. Bhattacharya, J. Tran, M. Sullivan, and C. Mason, "Linux AIO performance and robustness for enterprise workloads," Proc. Linux Symposium, pp.63–78, July 2004.

[11] A.J. Smith, "Cache memories," ACM Computing Surveys, vol.14, no.3, pp.473–530, 1982.

[12] M.K. Dahlgren, M. Dubois, and P. Stenstöm, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," Proc. Int'l Conf. on Parallel Processing, pp.56–63, 1993.

[13] M.K. Tcheum, H. Yoon, and S.R. Maeng, "An adaptive sequential prefetching scheme in shared-memory multiprocessors," Proc. Int'l Conf. on Parallel Processing, pp.306–313, 1997.

[14] M. Li, E. Varki, S. Bhatia, and A. Merchant, "TaP: Table-based prefetching for storage caches," Proc. USENIX Conf. File and Storage Technologies, pp.81–96, Feb. 2008.

[15] F. Wu, H. Xi, and C. Xu, "On the design of a new linux readahead framework," ACM SIGOPS Operating Systems Review, vol.42, no.5, pp.75–84, July 2008.

[16] Y. Wiseman and S. Jiang, Advanced Operating Systems and Kernel Applications, ch. Sequential File Prefetching in Linux, pp.218–261, Hershey, 2010.

[17] J.S. Plank, "The raid-6 liberation codes," Proc. USENIX Conf. File and Storage Technologies, pp.97–110, Feb. 2008.

[18] A. Krioukpv, L.N. Bairavasundaram, G.R. Goodson, and K. Srinivasan, "Parity lost and parity regained," Proc. USENIX Conf. File and Storage Technologies, pp.127–141, Feb. 2008.

[19] A. Thomasian, G. Gu, and C. Han, "Performance of two-disk failure-tolerant disk arrays," IEEE Trans. Comput., vol.56, no.6, pp.799–814, June 2007.

[20] S.H. Kim, H. Zhu, and R. Zimmermann, "Zoned-RAID," ACM Trans. Storage, vol.3, no.1, pp.1–17, March 2007.

[21] J. Bonwick, "RAID-Z," Nov. 2005. https://blogs.sun.com/bonwick/entry/raid_z.

[22] W. Liu and J. Elerath, "Using netapp raid-dp in exchange server 2007 storage designs," Tech. Rep. TR-3574, Network Appliance, 2007.

[23] S. Wan, Q. Cao, J. Huang, X.L.S. Li, S. Zhan, and L. Yu, "Victim disk first: An asymmetric cache to boost the performance of disk arrays under faulty condition," Proc. 2011 USENIX Ann. Technical Conf., June 2011.

[24] S.H. Baek and K.H. Park, "Maxtrix-stripe-cache-based contiguity transform for fragmented writes in RAID-5," IEEE Trans. Comput., vol.56, no.8, pp.1040–1054, Aug. 2007.

[25] F. Guo and P. Efstathopoulos, "Building a high performance deduplication system," Proc. 2011 USENIX Ann. Technical Conf., June 2011.

[26] D.T. Meyer and W.J. Bolosky, "A study of practical deduplication," Proc. USENIX Conf. File and Storage Technologies, Feb. 2011.

[27] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai, "The design and evolution of live storage migration in VMware ESX," Proc. 2011 USENIX Ann. Technical Conf., June 2011.

[28] M. Maxey, "Real-world use cases: Cloud storage workloads," Cloud Computing Journal, Jan. 2009.

Sung Hoon Baek received the BS degree in electronics engineering from Kyungpook National University, Korea in 1997, the MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1999, and the PhD degree from KAIST in 2008. He worked for Electronics Telecommunication Research Institution (ETRI) as an R&D staff from 1999 to 2005 and Samsung Eletronics as a senior engineer from 2008 to 2011. He is currently with Jungwon University as an assistant professor. His research interests include storage systems, operating system, nonvolatile memory, and embedded systems.