Striping-Aware Sequential Prefetching for Independency and Parallelism in Disk Arrays with Concurrent Accesses

Sung Hoon Baek and Kyu Ho Park, *Member*, *IEEE*

Abstract—Traditional studies on disk arrays have focused on parallelism or load balance of disks. However, this paper reveals that the independency of disks is more important than parallelism for concurrent reads of large numbers of processes in striped disk arrays, whereas parallelism is significant only for concurrent reads of small numbers of processes. We investigate and evaluate how much aligning sequential prefetch in strip or stripe boundaries affects the I/O performance by comparing with the proposed two types of sequential prefetching schemes, both of which are implemented in Linux kernel 2.6.18. In the experiments in this study, the combination of our schemes outperforms the original sequential prefetching of Linux by 3.2 times for 128 clients and 2.4 times for a single sequential read.

Index Terms—Storage management, operating systems.

1 Introduction

PREFETCHING is necessary to reduce or hide the latency between a processor and a main memory, as well as between a main memory and a storage subsystem that consists of disks. Some prefetching schemes for processors can be applied to prefetching for disks after slight modifications, whereas many prefetching techniques that are dedicated to disks have been studied. Especially, this study focuses on disk prefetching for striped disk arrays.

The frequently addressed goal of disk prefetching is to make data available in a cache before the data are consumed; in this way, the computational operations overlap the transfer of data from the disk. Another goal is to enhance disk throughput by aggregating multiple contiguous blocks as a single request. Prefetching schemes for a single disk may cause some problems in striped disk arrays. There is a need for a special scheme for multiple disks, in which the characteristics of striped disk arrays [1] are considered.

The performance disparity between the processor speed and the disk transfer rate can be compensated for via the disk parallelism of disk arrays. Chen et al. [1] described six types of disk arrays and termed them the redundant arrays of independent disks (RAID). In these arrays, blocks are striped across the disks and the striped blocks provide the parallelism of multiple disks, thereby improving the access bandwidth. Many RAID technologies have focused on the following: the reliability of RAID [2], the write performance [3], multimedia streaming with a disk array [4], RAID management [5], and so on.

However, prefetching schemes for disk arrays have seldom been studied in comparison with prefetching for a single disk. Some offline prefetching schemes [6], [7] and application-hint-based prefetching schemes [8], [9] account for parallelism and load balancing among disks. However, these schemes failed to consider independency loss of multiple disks.

The striping scheme of RAID improves the parallelism of disks [1]. The greater number of concurrent I/Os implies more evenly distributed I/Os across the striped blocks. As a result, researchers are forced to address new problems that arise in the striped disk arrays. Accordingly, we reveal that the independency of disks is more important than parallelism for a larger number of concurrent accesses in striped disk arrays, whereas parallelism is significant only for a small number of concurrent accesses. The following sections describe the independency and parallelism loss in striped disk arrays in more detail.

1.1 Independency

A strip is defined by the RAID Advisory Board [10], as shown in Fig. 1, which illustrates an RAID-5 array consisting of five disks. The stripe is divided by the strips. Each strip is comprised a set of contiguous blocks.

Traditional prefetching schemes fail to consider the data placement of the striped disk arrays and therefore suffer independency loss for multiple concurrent reads. Fig. 2a shows an example of independency loss. In traditional prefetching schemes, a prefetch request consisting of multiple blocks is not aligned in a strip, and thus, it is split across several disks. In Fig. 2a, three processes request sequential blocks for their own files in terms of a block. A preset amount of sequential blocks are aggregated as a single prefetch request by prefetcher. Each prefetch request is not aligned in a strip; therefore, each request is split across two disks and each disk requires two accesses. For example, if a single prefetch request is for Block 2 to Block 5, the single prefetch request generates two disk commands that correspond to Blocks 2 and 3 belonging to Disk 0 and Blocks 4 and 5 belonging to Disk 1. This problem is called independency loss. In contrast, if each prefetch request is dedicated to only one disk, as shown in Fig. 2b, independency loss is resolved.

Independency loss frequently arises if the number of concurrent accesses is greater than roughly the number of disks in a striped disk array. For a larger number of concurrent accesses, the disk parallelism is less important than independency because parallelism is achieved by the striping scheme of RAID.

Traditional sequential prefetching schemes exhibit severe independency loss. In the worst case in our evaluations, sequential prefetching was 3.2 times poorer than the proposed scheme that excludes independency loss. If the prefetch size of the sequential prefetching is larger than the strip size, independency loss is inevitable. In addition, sequential prefetching is based on files rather than blocks, but the files can be fragmented at the block level and stored unaligned in strips.

Sequential prefetching has been widely deployed without cognition of independency loss in real operating systems. Hence, we propose a strip-aligned sequential prefetching (SASEQP), which inspects the physical data placement of files for every prefetch request to stop prefetching at the boundaries of strips.

However, the prefetch size of SASEQP is much less than the stripe size; as a result, it suffers parallelism loss for concurrent reads of small numbers of processes, which are roughly less than the number of member disks. These two problems, independency loss and parallelism loss, conflict with one other: if one problem is resolved, the other arises. Thus, we propose a massive stripe prefetching (MSP) to eliminate parallelism loss. Combining SASEQP with MSP resolves the conflicting problems of independency and parallelism without relying on the number of concurrent reads.

1.2 Parallelism

If the prefetch size is much less than the stripe size and the number of concurrent accesses is much less than the number of member

- S.H. Baek is with Corporate Technology Operations SAIT, Samsung Electronics Co., Ltd., San #14-1 Nongseog-dong, Giheung-gu, Yongin-si, Gyeonggi-do, 446-712, Republic of Korea.
E-mail: sung.baek@samsung.com.
- K.H. Park is with the Division of Electrical Engineering, Department of Electrical Engineering & Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 335 Gwahangno Yuseong-gu, Daejeon 305-701, Republic of Korea. E-mail: kpark@ee.kaist.ac.kr.

Manuscript received 26 Feb. 2008; revised 23 Sept. 2008; accepted 4 Dec. 2008; published online 8 Jan. 2009.
Recommended for acceptance by A. George.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-02-0091.
Digital Object Identifier no. 10.1109/TC.2009.14.

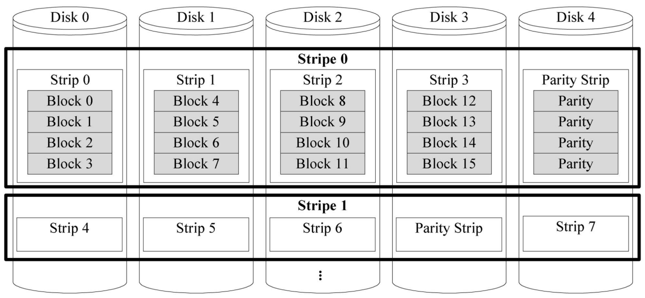

Fig. 1. The data organization and terminologies of an RAID-5 array.

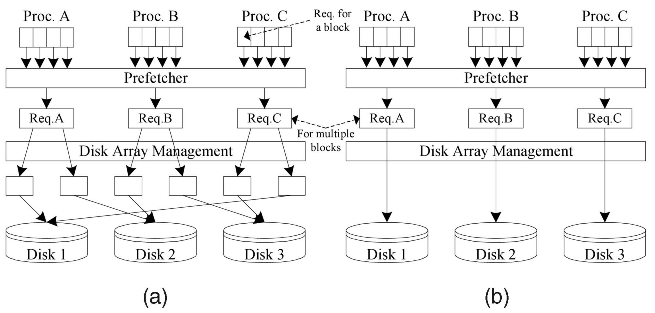

Fig. 2. (a) Independency loss: prefetch requests are split across multiple disks, so each disk requires two accesses. (b) No independency loss: each prefetch request is dedicated to one disk.

disks that compose a striped disk array, some disks become idle, thereby losing parallelism. This case exemplifies what is termed parallelism loss in this paper.

A single stream suffers parallelism loss if the prefetch size is smaller than the stripe size. However, a large prefetch size that is laid across multiple disks can prevent parallelism loss. For a single stream, the prefetch size must be equal to or larger than the stripe size. However, this method may result in thrashing and prefetching wastage [11] for multiple concurrent reads. Furthermore, this method for preventing parallelism loss causes independency loss for multiple concurrent reads.

Parallelism is more important in a single or small number of processes than a large number of processes due to the striping scheme. The proposed MSP maximizes the parallelism of striped disk arrays only for a single or small number of sequential reads, and is deactivated in other types of workloads. Consequently, combining MSP with SASEQP achieves a solution that resolves both independency loss and parallelism loss.

Our major contribution is the unveiling of independency loss and the performance evaluation, which, by comparing with the two resolved variants of sequential prefetching, demonstrates that the yet unveiled but very important problem on independency significantly affects the performance of striped disk arrays. In our experiments, the default sequential prefetching function of Linux was evaluated and also modified as SASEQP; IOzone and FileBench benchmarks were exploited to investigate how much independency loss and parallelism loss affect the storage performance of a streaming server and a file download server.

2 Prior Work

2.1 History-Based Prefetching

History-based prefetching, which predicts future accesses by learning the stationary past accesses, has been proposed in various forms [12], [13], [14]. Recording and analyzing past accesses require a significant amount of memory; hence, data compression techniques have also been used to predict future access patterns [15], [16]. History-based prefetching, which records, mines, and maintains an extensive history of past accesses, is cumbersome and expensive to maintain in practical systems. Furthermore, it is ineffective for nonstationary reads.

2.2 Application-Hint-Based Prefetching

When a small number of processes or a single process generate a nonsequential access, a small number of concurrent I/Os may not fully exploit disk parallelism in the disk array. In order to solve this problem, Patterson and Gibson suggested a disclosure hint interface [8]. This interface must be exploited by an application programmer so that information about future accesses can be given through an I/O-control (ioctl) system call. The disclosure hint forces programmers to modify applications so that the applications issue hints. Some applications involve significant code restructuring to include a disclosure hint. A speculative execution provides application hints without modifying the code [9]. A copy of the original thread is speculatively executed without affecting the original execution in order to predict future accesses. However, this type of prefetching regards parallelism as the only important factor and ignores independency.

2.3 Offline Optimal Prefetching

Traditional buffer management algorithms that minimize cache misses are substantially suboptimal in parallel I/O systems where multiple I/Os can proceed simultaneously [6]. Analytically, optimal prefetching and caching schemes have been studied with respect to situations in which future accesses are given [6], [7]. These schemes are optimal in terms of cache hit rates and disk parallelism. As a metric value, however, the cache hit rate may not accurately reflect the real performance because a sequential read for tens of blocks can achieve a much higher disk throughput than random reads for two blocks [17]. Furthermore, off-line prefetching does not resolve the two conflicting problems of parallelism loss and independency loss.

2.4 Sequential Prefetching

The above prefetching schemes are not widely used in practical systems due to their complexity and expense. The most common form of prefetching is sequential prefetching (SEQP), which is widely used in a variety of operating systems because sequential accesses are common in practical systems. The simplest form of sequential prefetching is the one block lookahead (OBL) method, which initiates a prefetch for block $b+1$ when block b is requested [18]. There are many variations of OBL [11]. P-block lookahead is a variation that involves the prefetching of p blocks instead of one block; various other schemes dynamically adjust the degree of prefetching p [19], [20].

The most popular SEQP schemes are synchronous SEQP [20] and asynchronous SEQP [18]. If a sequential miss occurs in block b, synchronous SEQP reads blocks b to $b+p$ in advance. The degree of sequential read-ahead, p, starts from 1 and exponentially or linearly increases up to a predetermined maximum value when block $(b+p+1)$ is requested. Asynchronous SEQP synchronously conducts a read-ahead of the first prefetch group; after that when a preset fraction g of the prefetched group is requested, it asynchronously conducts a read-ahead of the next group, and so on [21].

The primary concept of SEQP has been extended and varied in many studies. Dynamics aware prefetching [22] adaptively adjusts the maximum prefetch size p in the dynamic changes of the bandwidth and latency of TCP connections. For multiple streams, Gill and Bathen provided an extension of the asynchronous SEQP: their method adaptively changes the values p and g in order to maximize the throughput and minimize cache wastage and cache pollution [11].

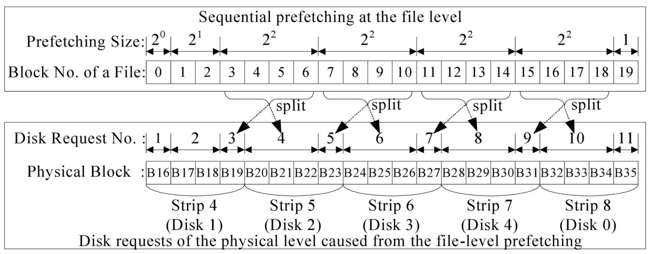

Fig. 3. SEQP: a sequential prefetching is performed for a file consisting of 20 blocks that correspond to the physical blocks [B16, B35]. Several prefetch requests are laid across two strips at the disk level. Consequently, seven prefetch requests generate 11 disk requests.

Table-based prefetching (TaP) [23] detects these sequential patterns in a storage cache without any help of file systems, and dynamically adjusts the cache size for sequentially prefetched data, namely prefetch cache size, which is adjusted to an efficient size that obtains no more prefetch hit rate above the preset level even if the prefetch cache size is increased.

Although sequential prefetching is the most popular prefetching scheme in practical systems, sequential prefetching and its variations have not yet considered striped disk arrays. Sequential prefetching and conventional prefetching schemes suffer from parallelism loss, independency loss, or both.

3 Independency: Strip-Aligned Sequential Prefetching

The conventional sequential prefetching (SEQP) scheme, which is based on sequential block accesses at the file level, fails to consider the fragmentation and strip-unaligned deployment of files. Hence, a disk request resulting from its prefetching is laid across multiple disks, thereby causing independency loss for concurrent reads produced by multiple processes.

Although a file is aligned in strips and not fragmented, SEQP suffers independency loss. In Fig. 3, the file is not fragmented, consists of 20 blocks, and is perfectly aligned in five strips; the file is sequentially stored in the physical blocks [B16, B35]; and the disk array exemplified in Fig. 3 is the same as the disk array shown in Fig. 1. Fig. 3 exemplifies an SEQP operation that sequentially reads the first block of the file B16 to the last block of the file B35, where its prefetch size increases exponentially from one block to the maximum four blocks. At first, B16 is read, and then B17 and B18 are requested by the second prefetch. The third prefetch includes the four blocks [B19, B22]. However, the third prefetch is laid across two strips (Strip 4 and Strip 5) and split into two disk requests for Disk 1 and Disk 2, thereby causing independency loss. The subsequent prefetch requests after the third prefetch are also unaligned in strips.

To resolve the independency loss of SEQP, we propose SASEQP, which provides an additional feature to SEQP to align the prefetch requests in strips. Fig. 4 exemplifies an operation of SASEQP, which resizes the ahead window [24] (which is termed a read-ahead group in [25]) determined by SEQP to assign it to only one strip. In Fig. 4, each prefetch request is dedicated to only one strip. The original prefetch size of the third prefetch generated by SEQP was four blocks, but was shrunk to one block for all blocks of the ahead window to be assigned to the strip to which the first block of the current request belongs. As shown in Fig. 3, SEQP generates 11 disk requests to sequentially read the file, while SASEQP requires seven disk requests, as shown in Fig. 4, thereby outperforming SEQP.

Fig. 5 shows the pseudocode of SASEQP in a style of C language. After an ahead window is created by SEQP (Line 6), SASEQP modifies the size of the ahead window to resolve the independency loss (7$\sim$12) before performing the actual read-ahead (Line 13). All blocks in the ahead window must be assigned to only one strip. Hence, SASEQP traverses from the second block ($start + 1$) to the last block (end) of the ahead window to search for the first block that physically splits the ahead window into two disk requests (Lines 9 and 10). GetPhysicalPosition() returns the physical block number for the specified block offset in *File*. STRIPWIDTH is the number of blocks per strip. For easy comprehension, a linear search (Line 8) is shown in Fig. 5, but a binary search can replace it to reduce the overhead.

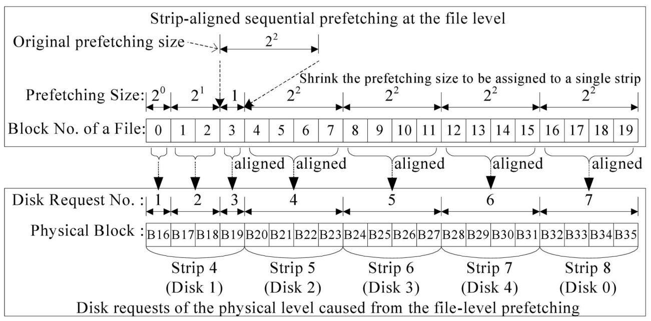
Strip-aligned sequential prefetching at the file level
Original prefetching size
Shrink the prefetching size to be assigned to a single strip
Prefetching Size:
Block No. of a File:
aligned
Disk Request No. :
Physical Block :
Strip 4 (Disk 1)
Strip 5 (Disk 2)
Strip 6 (Disk 3)
Strip 7 (Disk 4)
Strip 8 (Disk 0)
Disk requests of the physical level caused from the file-level prefetching

Fig. 4. SASEQP: to align the prefetch requests in strips, SASEQP dynamically shrinks the prefetch size from the original prefetch size that is determined by SEQP. Each prefetch request generates only one disk request. Consequently, seven prefetch requests generate seven disk requests.

4 Parallelism: Massive Stripe Prefetching

SASEQP suffers parallelism loss for concurrent reads of a small number of processes, usually when there are fewer processes than the number of disks in the disk array. In other words, because blocks of each prefetch request of SASEQP are dedicated to only one disk, the disks are serialized for a single stream. To resolve parallelism loss in SASEQP, we propose MSP as a new type of sequential prefetching. SASEQP and MSP can be used separately, but they can also be combined. MSP resolves the parallelism loss of SASEQP without losing the advantages of SASEQP.

Because parallelism loss occurs in a single read or concurrent reads of a small number of processes, MSP is designed to resolve this situation only. If a process or a small number of processes access long sequential blocks, MSP aligns the prefetch requests in stripes and sets the prefetch size to a multiple of the stripe size. As a result, MSP maximizes the read performance by performing a perfect parallelism of the disks. MSP is automatically deactivated for concurrent reads generated by a large number of processes that is roughly greater than the number of member disks, thus SASEQP dominates the prefetching performance without independency loss for this workload.

```
StripAlignedSequentialPrefetching(struct file *$File$)
1:  unsigned int $start$; //the prefetch start in the $File$ domain
2:  unsigned int $end$; // the prefetch end in the $File$ domain
3:  unsigned int $pos$; // a block number in the $File$ domain
4:  unsigned int $stripstart$; // the strip number of $start$
5:  unsigned int $strippos$; // a strip number
6:  GetAheadWindow($File$, &$start$, &$end$);
7:  $stripstart \Leftarrow \lfloor$GetPhysicalPosition($File$, $start$) / STRIPWIDTH$\rfloor$;
8:  for ($pos \Leftarrow start + 1$; $pos \leq end$; $pos$++) {
9:        $strippos \Leftarrow \lfloor$GetPhysicalPosition($File$, $pos$) / STRIPWIDTH$\rfloor$;
10:      if ($stripstart \neq strippos$)
11:            { $end \Leftarrow pos - 1$; /*shrink the ahead window/* break; }
12: }
13: Prefetch blocks in the range of [$start$, $end$] of the $File$
```

Fig. 5. SASEQP: algorithm.

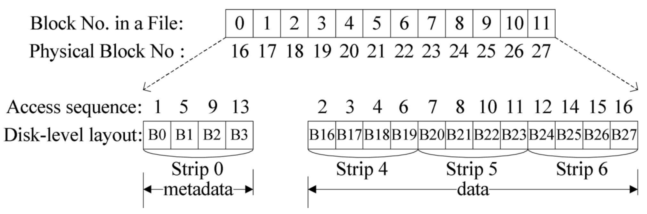

Fig. 6. Semisequentiality: a sequential read at the file level forms a semisequential read, which includes both metadata access and data access. The metadata blocks of the file are in [B0,B3]. The data blocks of the file are located in blocks [B16, B27].

Although MSP can be activated for concurrent sequential block reads of a small number of processes, the basic concept of MSP is designed for *a single sequential read at the file level*, which forms *a semisequential read at the block level,* because it requires both metadata access and data access.

Fig. 6 shows a simplified example of a semisequential read for a file of 12 data blocks. Composing a file requires metadata blocks in addition to data blocks. At first, the semisequential read accesses metadata block B0 to find the physical location of the first block of the file, and then reads the first block of the file B16 and the subsequent blocks (B17 and B18). To locate the subsequent blocks followed by B18, another metadata (B1) of the file must be read. The sequence of the accessed block numbers to read 12 blocks of the file is $\langle 0, 16, 17, 18, 1, 19, 20, 21, 2, 22, 23, 24, 3, 25, 26, 27\rangle$, which resembles a sequential read but is not perfectly sequential at the block level. Hence, this access pattern is termed a semisequential read.

Fig. 7 shows the proposed MSP algorithm. Lines 15-28 are designed to detect a semisequential access at the block level. As the semisequential read progresses, the sequential counter SC increases up to a predetermined value SC^{max} (Lines 19 and 22). If SC is equal to or larger than the predetermined threshold value SC^{thresh}, the prefetch size becomes a multiple of the stripe size and the prefetch requests are aligned in the stripe; thus, MSP performs the perfect parallelism of disks.

MSP inspects the sequence of the accessed strip numbers instead of the block numbers. If the sequence of the accessed block numbers $\langle 0, 16, 17, 18, 1, 19, 20, 21, 2, 22, 23, 24, 3, 25, 26, 27\rangle$ shown in Fig. 6 is assigned to the algorithm shown in Fig. 7, the sequence of the accessed strip number SN_c becomes $\langle 0, 4, \mathbf{0}, 4, 5, 0, 5, 6, 0, 6\rangle$. Two strip numbers before and after the strip number zeros (metadata access) are the same. For example, the third strip number **0** of this sequence is between two accesses to Strip 4. To detect this type of semisequential read, MSP increases the sequential counter SC (Line 19) when the current strip number SN_c is equal to SN_{pp}, which is the value prior to the previous value of SN_c (Line 21), as well as when SN_c is sequentially increased (Line 18).

Therefore, the transitional sequence of SC becomes $\langle 1, 0, 1, 2, 3, 2, 3, 4, 3, 4\rangle$, which tends to increase. Thus, as the semisequential read progresses, the sequential counter SC increases. If SC is greater than the threshold value, MSP infers that a long sequential access is detected.

Several mechanisms that detect semisequentiality at the block level have been proposed [11], [23], [26]. Although prior works may be more robust than the detection method of MSP, MSP is more lightweight and can mitigate the disadvantage of SASEQP.

Parallelism loss occurs not only in a single stream, but also in a small number of streams. Although the semisequentiality detection algorithm is originally designed for a single stream, a small number of streams can be detected as a semisequentiality by adjusting the threshold value SC^{thresh} and the maximum value SC^{max}. Section 5.2 shows experimental results with various values

```
int $SN_p \Leftarrow 0$; // the previous value of the current strip number
int $SN_{pp} \Leftarrow 0$; // the previous value of $SN_p$
int $SC \Leftarrow 0$; // the sequential counter
int $SPS \Leftarrow 0$; // the stripe prefetch size
#define $SC^{max}$ 42
#define $SC^{thresh}$ 38
#define $SPS^{max}$ 8
**MassiveStripePrefetching**(off_t physical_block)
14: int $SN_c$; //the current strip number
15: bool sequentiality $\Leftarrow$ *false*;
16: $SN_c \Leftarrow$ physical_block/(STRIP_SIZE/BLOCK_SIZE)
17: if ($SN_c = SN_p$) return;
18: if ($SN_p + 1 = SN_c$) {
19:     $SC \Leftarrow \min(SC^{max}, SC + 1)$;
20:     sequentiality $\Leftarrow$ *true*;
21: } else if ($SN_{pp} = SN_c$)
22:     $SC \Leftarrow \min(SC^{max}, SC + 1)$;
23: else $SC \Leftarrow \max(1, SC - 1)$;
24: $SN_{pp} \Leftarrow SN_p$;
25: $SN_p \Leftarrow SN_c$;
26: if ($SC < SC^{thresh}$) {
27:     $SPS \Leftarrow \min(SPS - 1, 0)$; return;
28: } else if (sequentiality=*false*) return;
29: for (int $i \Leftarrow 0$; $i < SPS$; $i \Leftarrow i + 1$) {
30:     if ($SN_c + i \geq$ "storage capacity in stripe size" ) break;
31:     if (($SN_c + i$)-th stripe is under prefetching by MSP) continue;
32:     Queue the read commands for all blocks of ($SN_c+i$)-th stripe except
for blocks that are clean or under reading or prefetching
33: }
34: $SPS \Leftarrow \min(SPS^{max}, SPS + 1)$;
```

Fig. 7. MSP algorithm.

of SC^{thresh}. These values that were chosen in the evaluation of this paper are shown in Fig. 7.

Just tuning SC^{thresh} may not be an optimal approach. In some cases, if the cache is large enough, prefetching multiple stripes may be beneficial even for multiple processes whose number is much greater than the number of member disks. A sophisticated algorithm replacing MSP may optimize the performance except for either a single or very many numbers of streams. However, MSP can mitigate the important disadvantage of SASEQP with the lightweight manner.

The prefetch size of MSP is large enough for parallelism, but may cause prefetching wastage. Hence, prefetch initiation must require a very long sequential access, which is determined by SC^{thresh} and the stripe size. MSP is suspended for much longer sequential accesses than either SASEQP or SEQP. Hence, MSP must be used as a combination with other prefetching schemes.

5 Performance Evaluation

5.1 Experimental Setup

We implemented the function of SASEQP in Linux kernel 2.6.18 $\times$ 86_64 by modifying the original read-ahead feature of Linux. The function of MSP is implemented in the RAID driver that was introduced in our previous works [3], [27], which shows that our RAID driver outperforms the software-based RAID of Linux (MultiDevice) and a hardware-based RAID. We disabled all the features of our previous works in all experiments. The Linux kernel and the RAID driver have their own cache memory. SASEQP stores data into the page cache of the Linux kernel at the file level, while MSP stores data at the block level in the RAID cache managed in the RAID driver.

The system in the experiments uses dual 3.0-Hz 64-bits Xeon processors, two Adaptec Ultra320 SCSI host bus adapters, and five ST373454LC disks, each of which has a speed of 15,000 revolutions per minute (rpm) and a 75-GB capacity. The five disks comprise an RAID-0 array with a strip size of 128 KB. A Linux kernel (version 2.6.18) for the $\times$86_64 architecture runs on this machine;

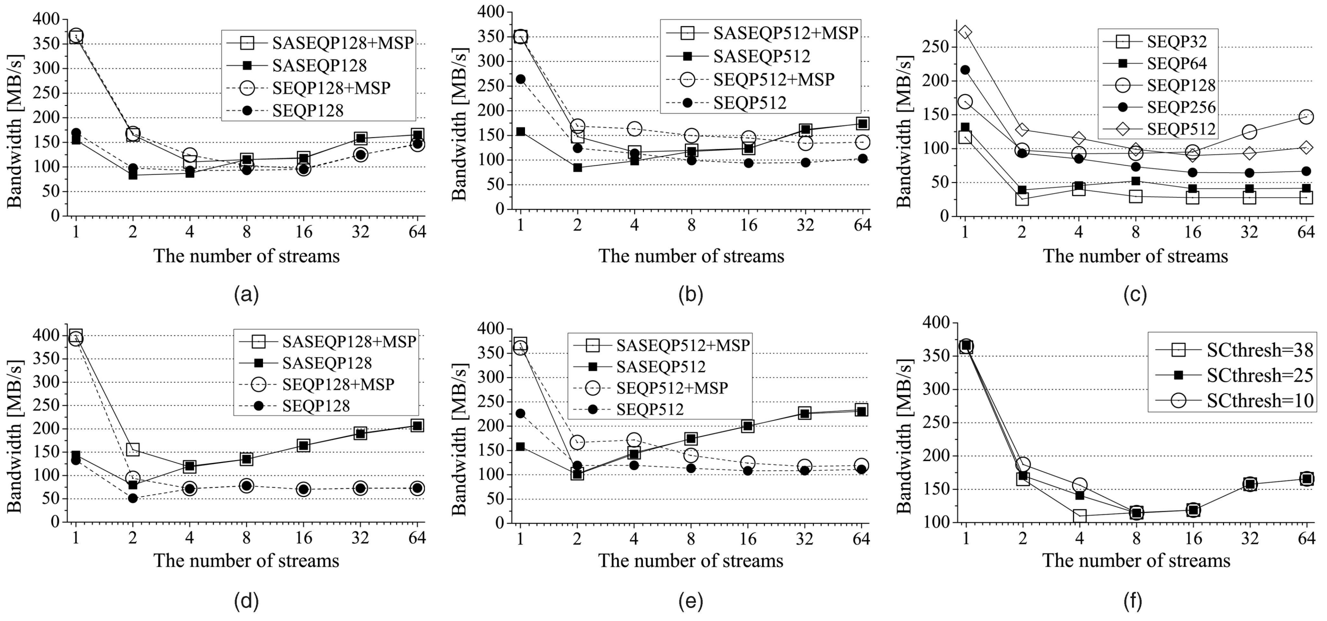

Fig. 8. IOZone: bandwidth of concurrent sequential reads in relation with various numbers of streams. (a) maximum prefetch $\text{size} = \mathbf{128}$ KB, and strip $\text{size} = \mathbf{128}$ KB. (b) maximum prefetch $\text{size} = \mathbf{512}$ KB, and strip $\text{size} = \mathbf{128}$ KB. (c) The bandwidth of SEQP by varying the maximum prefetch size. (d) maximum prefetch $\text{size} = \mathbf{128}$ KB, and strip $\text{size} = \mathbf{256}$ KB. (e) maximum prefetch $\text{size} = \mathbf{512}$ KB, and strip $\text{size} = \mathbf{256}$ KB. (f) The bandwidth of MSP+SEQP128 for various values of SC^{thresh}.

the kernel also hosts the ext3 file system and the anticipatory disk scheduler. The block size is set to 4 KB. 1 GB and 512 MB were allotted to the system memory of Linux and the RAID cache, respectively.

In these experiments, we compare four combinations: MSP+SASEQP, SASEQP, MSP+SEQP, and SEQP. In all figures in this paper, SASEQPX denotes a SASEQP with a maximum prefetch size of X KB. For example, SASEQP128 indicates that the maximum prefetch size is 128 KB.

5.2 IOZone: A Microbenchmark

In Fig. 8, we used the benchmark IOzone (version 3.283) [28] for concurrent sequential reads by varying the number of processes from 1 to 64 with a fixed aggregate file size of 8 GB. All experiments were repeated 10 times in cold cache. Each standard deviation was below 2 percent of the average value.

Evidence of parallelism loss and independence loss appears in Figs. 8a, 8b, 8d, and 8e, which show the average aggregate bandwidth of the concurrent sequential reads in relation to various numbers of streams. In these figures, SASEQP and SEQP are significantly inferior to the combinations with MSP for a single stream due to parallelism loss. MSP+SASEQP128 outperforms SASEQP128 by 2.4 times for a single stream in Fig. 8a.

For eight or more streams, SEQP is notably inferior to SASEQP due to independency loss. With 64 streams, SASEQP outperforms SEQP by 12 percent in Fig. 8a, 69 percent in Fig. 8b, 108 percent in Fig. 8d, and 82 percent in Fig. 8e. When the maximum prefetch size is equal to the strip size, the performance gap between SASEQP and SEQP is minimized because SEQP has the best performance in this matching condition, as shown in Fig. 8c, which shows the bandwidth of SEQP in relation to various prefetch sizes. As shown in Fig. 8c, generally, the greater the maximum prefetch size, the higher the bandwidth except for this matching condition.

The greater the prefetch size, the higher the parallelism for a small number of streams. By comparing Fig. 8a with Fig. 8b, SEQP512 outperforms SEQP128 by 56 percent for a single stream but it is still inferior to any combination with MSP and suffers much independency loss for a large number of streams; SEQP512 is inferior to SEQP128 by 43 percent.

Because the prefetch size of SASEQP is limited to the strip size of 128 KB, SASEQP512 and SASEQP128 show the very similar throughput, but are strictly different. The greater the maximum prefetch size, the faster the incremental speed of the prefetch size: In Linux, when the first block of a file is accessed at first, the initial prefetch size is determined by a function of the maximum prefetch size (*max*) and the request size (see function *get_init_ra_size()* of mm/readahead.c); the greater the *max*, the greater the initial prefetch size. In addition, the incremental speed of the prefetch size depends on the *max*. If the prefetch size is less than $max/16$, the prefetch size is increased by four times. Otherwise, the prefetch size doubles up to the maximum value.

Fig. 8f shows the effect of the threshold value of the sequential counter SC^{thresh}. The basic concept of MSP is based on a single sequential read, but parallelism loss arises in a small number of streams as well as a single stream. Lowering SC^{thresh} forces MSP to be activated for a small number streams. As shown in Fig. 8f, the bandwidth for four streams improves by lowering the SC^{thresh}. However, if the SC^{thresh} is too low, it may cause prefetching wastage in other types of workloads.

5.3 Aligning in Multiple Strips with Concurrent Accesses

Prefetching two or more strips has no disadvantage in terms of request split if the cache is large enough and prefetches are aligned in strip boundaries. Fig. 9 compares single-strip alignment and multiple-strip alignment. In this experiment, each of **256** threads

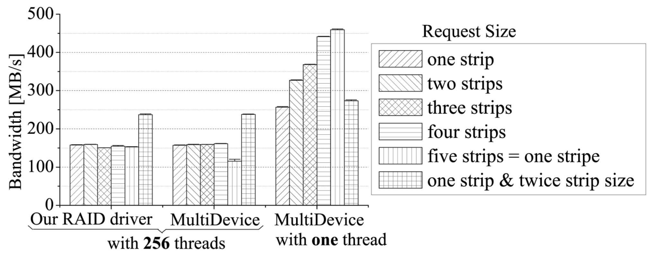

Fig. 9. Aligning reads in multiple strip boundaries with multiple concurrent accesses or a single access.

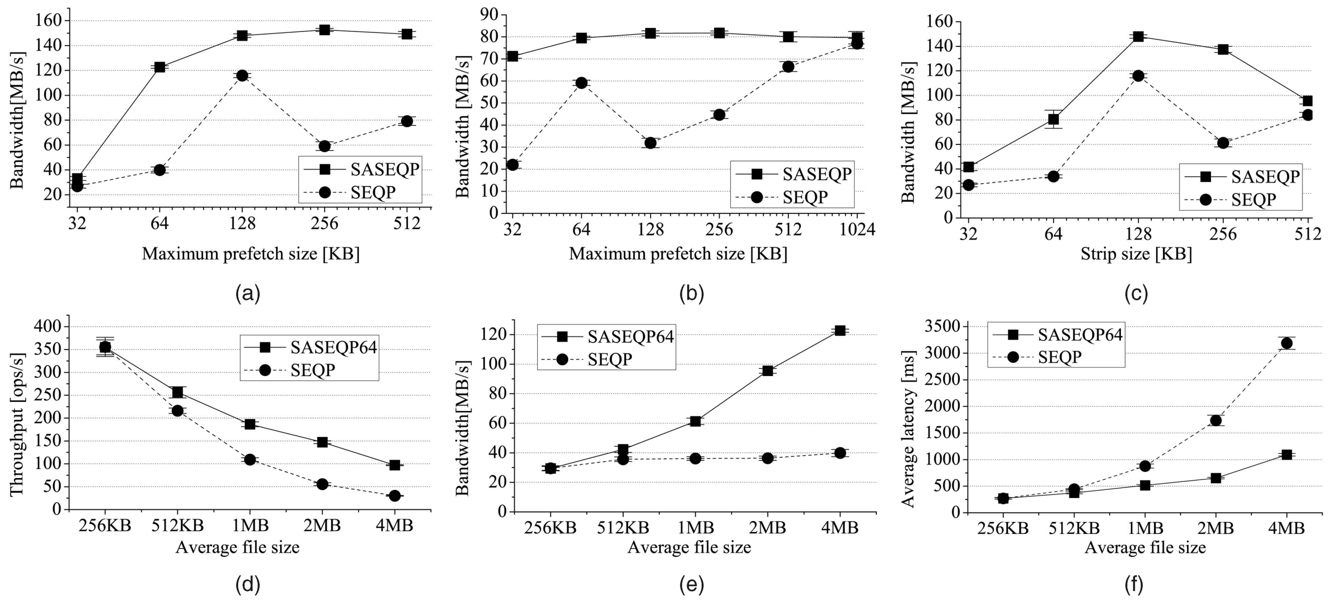

Fig. 10. Filebench with the Web server profile. In default, the average file size ($filesize) is 4 MB, the number of files ($nfiles) is 8 GB / $filesize, the IO size ($iosize) is 4 KB, the number of threads ($nthreads) is 32, the strip size is 128 KB, and the maximum prefetch size is 64 KB. (a) A strip size of 128 KB, (b) a strip size of 64 KB, (c) a maximum prefetch size of 128 KB, (d) average throughput, (e) average bandwidth, and (f) average latency.

performs a sequential read at the block level with direct I/O (which disables the readahead feature of Linux), the strip size is 128 KB, read offsets are aligned in strips, and each read is perfectly fit to one or more strips. The bars in this figure are distinguished by the request sizes. The last bar indicates that the strip size is twice of the other bars (256 KB).

As shown in the right group of Fig. 9, as more strips are prefetched with only one thread that performs a sequential read, the disk parallelism and performance increases. However, when multiple concurrent reads contend for disks of a disk array as shown in the left and center groups of Fig. 9, the two software RAID drivers exhibit the common characteristic that the performance of prefetching two or more strips is nearly same as that of prefetching one strip; all of these prefetching types spend the same access time and transfer time. However, prefetching two or more strips may evict more cached data by consuming cache memory for prefetched data. In addition, prefetching only one strip with the twice strip size (the sixth bar) outperforms the others. Therefore, if the cache is large enough to serve required clients, we had better increase both the strip size and the prefetch size and just stop prefetching at the single strip boundaries.

5.4 FileBench: A Macrobenchmark

For more realistic workloads, we chose the "Web server" workload personality of FileBench version 1.1.0. Fig. 10 shows the results of FileBench with the Web server profile that creates multiple processes, each of which reads whole data of an arbitrary file.

In Fig. 10, the average size of files is set to 4 MB to imitate a Web server that serves music files. Figs. 10a, 10b, 10c show the relationship between the strip size and the maximum prefetch size with 32 threads and a workspace of 8 GB. All results were obtained from the average values of 10 performance snapshots. The standard deviation of each result is illustrated in the figures.

In Fig. 10a for a strip size of 128 KB, SASEQP outperforms SEQP by 3.1 times, 1.3 times, 2.6 times, and 1.9 times for a maximum prefetch size of 64, 128, 256, and 512 KB, respectively. In Fig. 10b for a strip size of 64 KB, SASEQP outperforms SEQP by 3.2 times, 1.3 times, 2.6 times, 1.8 times, and 1.2 times for a maximum prefetch size of 32, 64, 128, 256, and 512 KB, respectively.

When the strip size is equal to the prefetch size, as shown in Figs. 10a, 10b, 10c, the performance of SEQP is maximized. This is the same conclusion discussed in Fig. 8c. As shown in Fig. 10b, when the prefetch size covers multiple stripes (1,024 KB of prefetch size) sacrificing a large amount of prefetch memory, SEQP reaches up to the performance of SASEQP.

Figs. 10d, 10e, 10f illustrate the relationship with the file sizes. Both SASEQP and SEQP perform at the file level, thereby depending on the average size of files. For example, if a file shown in Fig. 3 consists of seven blocks instead of 20 blocks, there is no difference between SASEQP and SEQP. From Figs. 10d, 10e, 10f, we can conclude that: 1) the file size have to be four times larger than the strip size for SASEQP to provide a noticeable gain and 2) SASEQP outperforms SEQP in terms of throughput and latency as well as bandwidth.

6 Conclusion

Traditional SEQP schemes have ignored the fact that independency of disks significantly affects the throughput of servers that exhibit multiple concurrent I/Os. Our experiments have shown that SEQP suffers independency loss that has been resolved by the proposed SASEQP. In addition, to take parallelism as well as independency into account, we evaluated the combination of SASEQP and MSP, which outperforms the SEQP of Linux by 3.2 times for 32 threads and 2.4 times for a single sequential read in our experiments. Our proposal resolves both independency loss and parallelism loss regardless of the amount of concurrency.

References

[1] 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.

[2] A. Thomasian, G. Gu, and C. Han, "Performance of Two-Disk Failure-Tolerant Disk Arrays," *IEEE Trans. Computers*, vol. 56, no. 6, pp. 799-814, June 2007.

[3] S.H. Baek and K.H. Park, "Matrix-Stripe-Cache-Based Contiguity Transform for Fragmented Writes in RAID-5," *IEEE Trans. Computers*, vol. 56, no. 8, pp. 1040-1054, Aug. 2007.

[4] S.H. Kim, H. Zhu, and R. Zimmermann, "Zoned-RAID," *ACM Trans. Storage*, vol. 3, no. 1, Mar. 2007.

[5] D. Kenchammana-Hosekote, D. He, and J.L. Hafner, "REO: A Generic RAID Engine and Optimizer," *Proc. Fifth USENIX Conf. File and Storage Technologies,* 2007.
[6] M. Kallahalla and P.J. Varman, "PC-OPT: Optimal Offline Prefetching and Caching for Parallel I/O Systems," *IEEE Trans. Computers,* vol. 51, no. 11, pp. 1333-1344, Nov. 2002.
[7] T. Kimbrel, A. Tomkins, R. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li, "A Trace-Driven Comparison of Algroithms for Parallel Prefetching and Caching," *Proc. Second Symp. Operating Systems Design and Implementation,* Oct. 1996.
[8] R.H. Patterson and G.A. Gibson, "Exposing I/O Concurrency with Informed Prefetching," *Proc. Third Int'l Conf. Parallel and Distributed Information Systems,* pp. 7-16, Aug. 1994.
[9] F. Chang and G.A. Gibson, "Automatic I/O Hint Generation through Speculative Execution," *Proc. Third Symp. Operating Systems and Design and Implementation,* pp. 1-14, Feb. 1999.
[10] The RAID Advisory Board, *The RAID Book: A Source Book for RAID Technology,* sixth ed. ENDL Publications, 1999.
[11] B.S. Gill and L.A.D. Bathen, "AMP: Adaptive Multi-Stream Prefetching in a Shared Cache," *Proc. Fifth USENIX Conf. File and Storage Technologies,* 2007.
[12] K.S. Grimsrud, J.K. Archibald, and B.E. Nelson, "Multiple Prefetch Adaptive Disk Caching," *IEEE Trans. Knowledge and Data Eng.,* vol. 5, no. 1, pp. 88-103, Feb. 1993.
[13] J. Griffioen and R. Appleton, "Reducing File System Latency Using a Predictive Approach," *Proc. USENIX Summer Technical Conf.,* pp. 197-208, June 1994.
[14] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *IEEE Trans. Computers,* vol. 48, no. 2, pp. 121-133, Feb. 1999.
[15] T.M. Kroeger and D.D.E. Long, "Design and Implementation of a Predictive File Prefetching Algorithm," *Proc. 2001 USENIX Ann. Technical Conf.,* pp. 105-118, June 2001.
[16] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching," *Proc. 1997 USENIX Ann. Technical Conf.,* Jan. 1997.
[17] X. Ding, S. Jiang, and F. Chen, "A Buffer Cache Management Scheme Exploiting Both Temporal and Spatial Localities," *ACM Trans. Storage,* vol. 3, no. 2, June 2007.
[18] A.J. Smith, "Cache Memories," *ACM Computing Surveys,* vol. 14, no. 3, pp. 473-530, 1982.
[19] M.K. Dahlgren, M. Dubois, and P. Stenstöm, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," *Proc. Int'l Conf. Parallel Processing,* 1993.
[20] M.K. Tcheum, H. Yoon, and S.R. Maeng, "An Adaptive Sequential Prefetching Scheme in Shared-Memory Multiprocessors," *Proc. Int'l Conf. Parallel Processing,* 1997.
[21] B.S. Gill and D.S. Modha, "SARC: Sequential Prefetching in Adaptive Replacement Cache," *Proc. USENIX Ann. Technical Conf.,* pp. 293-308, Dec. 2005.
[22] S.S. Lim and K.H. Park, "TPF: TCP Plugged File System for Efficient Data Delivery over TCP," *IEEE Trans. Computers,* vol. 56, no. 4, pp. 459-473, Apr. 2007.
[23] M. Li, E. Varki, S. Bhatia, and A. Merchant, "TaP: Table-Based Prefetching for Storage Caches," *Proc. Sixth USENIX Conf. File and Storage Technologies,* pp. 81-96, Feb. 2008.
[24] D.P. Bovet and M. Cesati, *Understanding the Linux Kernel,* third ed. O'Reilly, Nov. 2005.
[25] B. Ali R, C. Gniady, and Y.C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms," *IEEE Trans. Computers,* vol. 56, no. 7, pp. 889-904, July 2007.
[26] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," *Proc. 2007 USENIX Ann. Technical Conf.,* pp. 261-274, June 2007.
[27] 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.
[28] W. Norcutt, "The IOzone Filesystem Benchmark," http://www.iozone.org/, 2007.