Matrix Stripe Cache-Based Contiguity Transform for Fragmented Writes in RAID-5

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

***Abstract*— Given that contiguous reads and writes between a cache and a disk outperform fragmented reads and writes, fragmented reads and writes are forcefully transformed into contiguous reads and writes via a proposed matrix stripe cache-based contiguity transform (MSC-CT) method, which employs a rule of consistency for data integrity at the block level, and a rule of performance that ensures no performance degradation. MSC-CT performs for reads and writes, both of which are produced by write requests from a host, as a write request from a host employs reads for parity-update and writes to disks in a RAID-5 array. MSC-CT is compatible with existing disk technologies. The proposed implementation in a Linux kernel delivers a peak throughput that is 3.2 times higher than a case without MSC-CT on representative workloads. The results demonstrate that MSC-CT is extremely simple to implement, has low overhead, and is ideally suited for RAID controllers not only for random writes but also for sequential writes in various realistic scenarios.**

***Index Terms*— Storage Management, Parallel I/O, RAID**

I. Introduction

THE performance disparity between processor speed and the disk transfer rate can be compensated for via the disk parallelism capability of disk arrays. Patterson and Chen [1] described six types of disk arrays and termed them RAID-1 through RAID-6. RAID-6 protects against double disk failures using P+Q redundancy, and requires six disk accesses in order to perform a small write operation. RAID-5 requires four disk accesses for a small write operation and protects against single-disk failure. RAID-5 arrays are widely used due to their reasonable overhead and moderate reliability.

RAID-5 requires four disk accesses to update a data block; two to read old data and parity, and two to write new data and parity. To reduce the overhead of small writes in a RAID-5 array, Menon posited the *fast write* process that utilizes non-volatile storage (NVS) as a write cache in a disk array controller [2]. The NVS is commonly built using battery-backed volatile RAM. A block received from a host system is initially written to the NVS in a disk array controller and the disk array controller sends a completion message to the host system at that time. Both the data and the parity blocks on the disks can then be written (destaged) asynchronously at a later time, thus hiding the write latency of the disk.

A. Destage

Due to the asynchronous nature of the write cache, the contents of the write cache may be destaged in a desired order to reduce the average seek time or improve the peak write throughput of the system. Some disk schedulers (see Section I-E.2), least recently written (LRW) [3] and wise ordering for writes (WOW) [4], focus on the question of the order in which the writes are destaged from the NVS. In particular, WOW exploits both temporal locality and spatial locality by considering both data recency and disk seek latency.

Another problem in the use of the write cache is to determine when to destage data from the cache to the disk to guarantee continued high response time for writes. In contrast, if new data that were written to the cache are destaged too aggressively, it is not then possible to fully exploit the benefits of write caching. Varma et al. [5] introduced linear threshold (LT) scheduling, which adaptively varies the rate of destages based on the instantaneous occupancy of the write cache. A simpler form of scheduling is known as high/low mark (HLM) [6] scheduling, which enables destages to the disk when the cache occupancy drops below a “low mark”, and disables destages when it increases over a “high mark”. Gill and Modha [4] introduced a destage scheduler combining high/low mark scheduling with linear threshold scheduling.

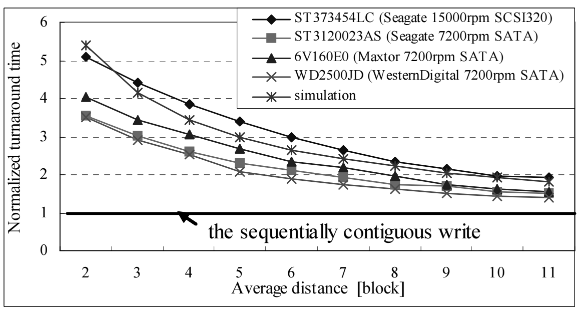
B. How to Destage Fragmented Writes

The destage starts by using a decision algorithm such as LT or HLM, that determines when to destage. When the start time of the destage is decided, it is then necessary to decide which block or stripe should be destaged by using a destage algorithm such as LRW and WOW, that determines what to destage. Our work contends with what comes after the selecting of a block or stripe that is to be destaged. The focus here is on how to efficiently destage fragmented writes from the NVS to the disk in a RAID controller. This work is compatible with the existing destage algorithms in terms of what to destage and when to destage it. Fragmented writes are produced in a number of cases, as follows:

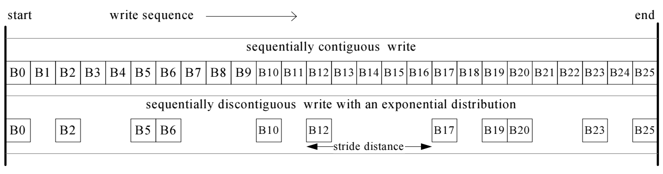
1) Random writes with destage: All random writes from the host can be converted into fragmented writes to the disk by destage algorithms that re-order write-requests in an increasing block index order. CSCAN [7], CLOCK [8], and WOW destages dirty blocks to disks only in an increasing block index order. WOW is a variant of CLOCK with a stripe cache that is managed in terms of stripe groups. If a destage algorithm that does not destage blocks in an ascending order manages their cache in terms of stripe groups, all writes within the stripe are sequentially contiguous or sequentially discontiguous. Section III-A describes the stripe cache in detail.

2) Multiple concurrent writes: In the local storage of a server and in storage servers such as network attached storage [9], multiple users read or write concurrently. Especially if multiple users create a large amount of files simultaneously, these files may be interleaved with each other. Thus, fragmented writes are produced due to the in-place update of the files.

The authors are with the Division of Electrical Engineering, Korea Advanced Institute of Science and Technology, 335 Gwahangno Yuseong-gu, Daejeon, Korea, 305-701. E-mail: shbaek@core.kaist.ac.kr, kpark@ee.kaist.ac.kr



(a) The normalized turnaround time of the write versus the average stride distance with an exponential distribution for various disks.


start
write sequence
end
sequentially contiguous write
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16 B17 B18 B19 B20 B21 B22 B23 B24 B25
sequentially discontiguous write with an exponential distribution
B0 B2 B5 B6 B10 B12 B17 B19 B20 B23 B25
stride distance

(b) A sequentially contiguous write and a sequentially discontiguous write with an exponential distribution.

Fig. 1. Writes to the disk perform in an increasing block index order. The distribution of the stride distance is exponential. The write starts at the same preset block and stops when it reaches the same preset destination. Each turnaround time is normalized with the sequentially contiguous write, of which the normalized value is one.

3) Aged filesystems: On real filesystems, files are divided into pieces scattered around the disk, thus sequential writes at the filesystem level may appear as fragmented writes at the block level. Fragmentation occurs naturally when we use a disk frequently, creating, deleting, and modifying files.

4) Sequential writes of multiple small files with pre-allocation: When a new disk data block is allocated, filesystems such as ext2 and ext3 internally pre-allocate a small number of disk blocks adjacent to the just allocated block, in order to expand the files in the future [10]. Hence, there exist pre-allocated, unused blocks between files, which cause fragmented writes when we copy a directory that contains small files. As the size of the files decreases, the portion of fragmentation increases.

C. Motivation of Our Work

It is evident that a sequentially discontiguous (fragmented) write to a disk is slower and employs a longer seek distance compared to a sequentially contiguous write if the two types of writes contain the same amount of data. However, we found another characteristic about a fragmented write. By referring to the experimental results shown in Fig. 1, we discovered that a sequentially contiguous write outperforms a sequentially discontiguous write even if the two types of writes employ the equivalent seek distance with different amounts of data as shown in Fig. 1(b). The equivalent seek distance indicates that identical start and end positions are applied, and signifies that the maximum stride distance does not exceed a threshold value. The stride distance is defined by the number of blocks between two discontiguous I/Os.

We presume that the result shown in Fig. 1 is induced by the following process: after finishing a block write, a host sends the next data to the disk for the subsequent discontiguous block in the same disk track as the previous write. If sending the subsequent data transfer requires more time than the rotational latency that is required for the disk head to reach to the subsequent block, then the disk head misses the block and requires another revolution.

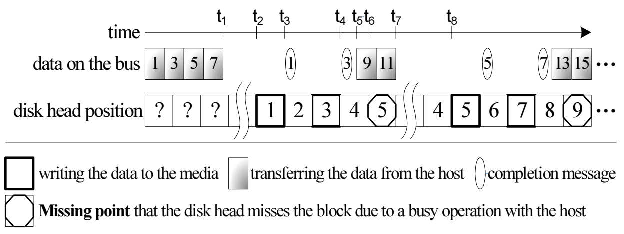

time
t1 t2 t3 t4 t5 t6 t7 t8
data on the bus
1 3 5 7
1
3 9 11
5
7 13 15 ···
disk head position
? ? ?
1 2 3 4 5
4 5 6 7 8 9 ···
writing the data to the media
transferring the data from the host
completion message
Missing point that the disk head misses the block due to a busy operation with the host

Fig. 2. An example that exhibits the unexpected disk rotation.

It is important to reduce not only the seek time but also unexpected disk rotation. For example,the revolution time of the disk ST373454LC is 4 ms, which is as long as the average write-seek time (i.e. 4.1 ms).

The workloads used in the experiment of Fig. 1 are sequentially discontiguous writes, the stride distance of which has an exponential distribution as shown in Fig. 1(b), where the maximum stride distance is limited up to 32 blocks, and the block size is set to 4 KiB (KiB is the abbreviation of kibibyte, as defined by the International Electrotechnical Commission. 1 KiB refers to 2^{10} bytes, instead 1 kB refers to 10^3 bytes. [11]). Fig. 1(a) shows the normalized turnaround time of a write for various disks by varying the average stride distance with an exponential distribution. Write starts at the same preset block and stops when it reaches the same preset destination. Each turnaround time of the writes shown in Fig. 1(a) is normalized with that of the sequentially contiguous write.

The normalized turnaround time of the contiguous write is 1. All of the normalized turnaround times of the fragmented writes shown in Fig. 1(a) are always higher than 1. Therefore a sequentially contiguous write outperforms sequentially discontiguous writes with an equivalent seek distance. Sequentially discontiguous writes with a small stride distance significantly degrade disk performance. In other words, fragmented writes of finer granularity give rise to a much higher probability of the additional disk rotation. Some disks also show such a characteristic for reads as shown in Fig. 9.

To obtain quantitative results of the unexpected disk rotation, we implemented a benchmark as a kernel driver of Linux (version 2.6) in order to eliminate the effects of cache, prefetching, synchronous I/O, and disk scheduling. The benchmark uses the function *generic_make_request()* of the Linux kernel for a disk interface to avoid cache and prefetching. The benchmark produces from 32 to 64 outstanding I/Os, processes all I/Os asynchronously, and periodically calls the function *unplug()* every position of 32 blocks to mitigate the unexpected waiting time caused by the anticipatory disk scheduler [12], which is the default disk scheduler of Linux.

Figure 2 illustrates an example where unexpected disk rotations occur. In Fig. 2, the host transfers sequentially discontiguous data (blocks 1, 3, 5, 7, 9, 11, and so on) to the disk in a command queuing fashion where the host queues multiple commands to the disk. In Fig. 2, the host transfers blocks 1, 3, 5, and 7 to the disk at the first time (t_1), and the disk buffers the blocks in its cache and seeks the block 1 ($t_1 \sim t_2$). After writing blocks 1 and 3 to the magnetic media, the disk sends completion messages for blocks 1 and 3 (t_3, t_4). The host transfers the subsequent blocks 9 and 11 ($t_5 \sim t_7$) after it receives the completion messages. While the disk receives the subsequent blocks 9 and 11 from the host, the disk head misses the media location that corresponds to the queued block 5 (t_6, missing point). In order to write the missed block to the media, the disk must fully rotate ($t_6 \sim t_8$). In this scenario, missing points occur every two written blocks if the host sends two data blocks simultaneously in order to sequentially write odd blocks. The frequency of the missing point depends on the cache policy of the disk, the stride distance, the command queuing policy of the host, the internal/external transfer rate of the disk, and atypical operation of the disk.

The following equation provides an analytical model of this problem.

$$t_d = t_c + t_r \times N_r, \tag{1}$$

where t_d and t_c are the turnaround time of the discontiguous write and the contiguous write, respectively. t_r is one revolution time of the disk. N_r is the number of occurrences of unexpected disk rotation, which occurs if the sum of the data transfer time and the computational overhead requires more time than the rotational latency that corresponds to the interval between two discontiguous blocks.

The turnaround time t_c consists of only the transfer time without any overhead. The sum of seek time and transfer time of t_d equals t_c due to the equivalent distance, but t_d requires additional rotations ($t_r \times N_r$). The simulation result for this analytical model with performance parameters of ST373454LC is shown in Fig. 1. The simulation result is similar to the experimental results.

The write-back policy of the disk aggregates written data into its on-board cache and transfers the data from the on-board cache into the disk media at a later time, thereby mitigating the missing points shown in Fig. 2. However, the write-back policy is unreliable against power-failure. Commercial disks, shown in Fig. 1, widely use write-on-arrival [13] capability, which begins writing sectors to the disk media as soon as they are transferred to the on-board cache, thereby reducing rotational latency. The closely-discontiguous read can be enhanced by a prefetching scheme that reads ahead up to the end of the track or cylinder containing the last sector of the read request [13]. However, some disks, shown in Fig. 9(a) and 9(d), do not use this prefetching scheme in order to reduce response time.

D. Our Contributions

There are a large number of algorithms for fragmentation reduction at the filesystem level, or spatial locality and temporal locality of the write cache at the block level. However, in contrast to the purpose of traditional schemes, our proposal, known as the matrix stripe cache-based contiguity transform (MSC-CT) as explained in Section III, mitigates the unnecessary rotational latency caused by fragmented writes at the block level. MSC-CT is compatible with existing algorithms such as destage, disk scheduling, prefetching, read cache management, and filesystem; it cooperates with the existing algorithms for better performance as it solves an entirely novel problem.

We make several contributions to the design and implementation of a low overhead, self-tuning algorithm for an actual RAID system that exploits a sophisticated redundancy scheme such as the distributed-parity and P+Q redundancy of RAID-5 and RAID-6 [1]. To efficiently destage fragmented writes, MSC-CT employs a stripe cache management scheme and a contiguity transform scheme. By using a stripe cache that is managed in terms of stripe groups, MSC converts all reads (for parity-update) and writes into sequentially contiguous or sequentially discontiguous reads and writes within the stripe at the destage. CT then transforms discontiguous reads and writes into contiguous reads and writes by inserting additional reads and writes. Such transform is performed by a proposed rule for consistency, thereby not affecting the filesystem and not changing the data and block locations. Furthermore, MSC-CT does not involve performance degradation by using our proposed rule for performance in a self-tuning fashion.

We implemented a kernel module in Linux version 2.6 as a RAID-5 driver with MSC-CT, which significantly outperforms the MD (Multi-Device) driver that is a traditional RAID-5 driver of Linux. Our experiments are performed on random writes, concurrent sequential writes using existing benchmarks, artificially aged filesystems, and various types of realistic sequential copies.In summary, MSC is a practical algorithm that does not conflict with existing algorithms and enhances a RAID controller to process more fragmented I/Os and sequential I/Os.

E. Prior Works for the Fragmented I/O

To improve random or fragmented I/O, which is the leading cause of performance degradation, researchers have investigated a variety of new technologies that are related to filesystems, disk scheduling, prefetching, read cache management, RAID, destage, and etc. However, our scheme is independent of and compatible with the prior technologies. This section briefly describes these technologies.

1) Filesystems: Fragmented I/Os can be produced by a highly utilized filesystem, in which files are fragmented into multiple segments [14]. Sequential access to a fragmented file causes fragmented I/Os at the block level. A log-structured filesystem (LFS) [15] always sequentially writes data without in-place updates. A de-fragmented filesystem (DFS) [16] dynamically relocates the fragmented data of a file. Howerver, such filesystems involve garbage collection and degradation of read performance.

2) Disk Scheduling: The disk scheduler is responsible for dynamically re-ordering the pending requests in a block device's request queue into a dispatching order that results in minimal seeks or instantaneous access time for random I/Os. Such algorithms include FCFS [17], SSTF [18], SCAN [18], CSCAN (it is also known as C-LOOK) [7], VSCAN [19], FSCAN [17], SPTF [20], GSTP [21]. In addition, disk schedulers are also responsible for load balancing, quality of service, and accumulating pending requests. The complete fairness queuing, the earliest deadline first scheduler, and the anticipatory scheduler [12] are adopted to Linux for these purposes.

3) Prefetching: By speculatively prefetching or prestaging pages even before they are requested, multiple sequential reads may be made of a single read request, thus the prefetching reduces a deep read latency [22]. Fragmented reads within prefetched pages lead to cache hits, thereby improving read performance by an order of magnitude. One of the latest and most outstanding investigations of prefetching involves sequential prefetching in adaptive replacement cache (SARC) [23]. SARC combines and adaptively balances both the cache and prefetch.

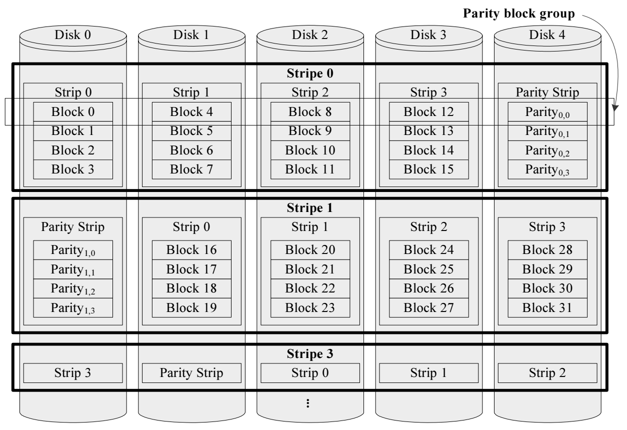


Fig. 3. The data organization and terminologies of a RAID-5 array.

4) Read Cache Management: Fragmented or random reads can be mitigated by a read cache. A large number of cache replacement algorithms have been well studied. Examples are LRU, CLOCK, LRU-2, LRFU, 2Q, LIRS, FBR, MQ, CAR [24], ARC [25], and DULO [26]. Most of the read cache algorithms focus on the hit ratio by statically or adaptively combining recency and frequency. Meanwhile, DULO gives randomly-placed blocks higher durability than sequentially-accessed blocks in the read cache, thereby improving fragmented or random reads.

5) RAID: A large number of researchers have investigated RAID technologies. Among these studies, AutoRAID [27], AFRAID [28], and DMP [29] focus on the small write that also produces a fragmented write. To improve the response time related to the small write that produces two reads and two writes in RAID-5, AutoRAID writes data to the mirrored storage class in a log-structured fashion, and then migrates the data to the RAID-5 storage class at a later time. AFRAID does not wait for the parity to be updated; it updates the parity when disks are idle, thereby improving the instantaneous response time, however it loses guaranteed reliability. DMP exploits more than one (R) number of parity strips per stripe for single-disk-failure tolerance, thereby enabling R number of writes to acquire the stripe lock simultaneously.

II. Background of RAID-5

RAID-5 employs a coarse-grained striped parity technique so that small requests can benefit from the concurrency during the servicing of multiple requests. The RAID Advisory Board [30] describes RAID-5 using the terminology of both strip and stripe in more detail, as shown in Fig. 3. A RAID-5 array is organized by stripes, each of which consists of a parity strip and data strips. Each strip comprises a set of blocks that are partitioned by disks. The parity block group (PBG[1]) is defined by the group of blocks that are located at the same offset of the member disks. The parity block in the parity strip stores the result of the bitwise eXclusive OR (XOR) of the data blocks that are in the same PBG. The parity strip comprises the parity blocks in the stripe.

A. Reconstruct-write and Read-modify-write

All actions of RAID-5 can be categorized by six operations: r, t, w, x, xx, and _ that are described in Table I. We use these mnemonics to conveniently describe our work.

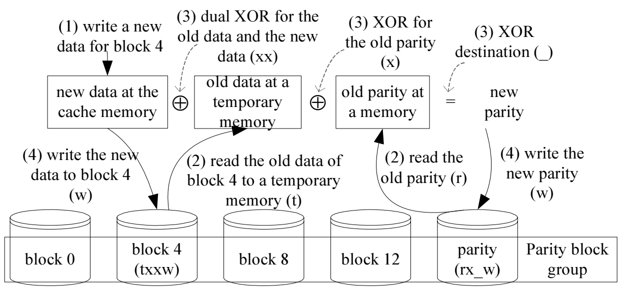

(1) write a new data for block 4
(3) dual XOR for the old data and the new data (xx)
(3) XOR for the old parity (x)
(3) XOR destination (_)
new data at the cache memory
old data at a temporary memory
old parity at a memory
new parity
(4) write the new data to block 4 (w)
(2) read the old data of block 4 to a temporary memory (t)
(2) read the old parity (r)
(4) write the new parity (w)
block 0
block 4 (txxw)
block 8
block 12
parity (rx_w)
Parity block group

Fig. 4. A read-modify-write cycle in a RAID-5 array.

TABLE I
OPERATION MNEMONICS

Mnemonic	Description
r	Read the block from the disk to the block cache memory
t	Read the block from the disk to a temporary memory
w	Write the block cache memory to the disk
x	XOR the block cache memory
xx	XOR the block cache memory and the temporary memory
_	The destination of the XOR result

To update one block with new data, it is necessary to (1) read all other blocks of the PBG to which the updated block belongs, unless it is cached; (2) XOR all data blocks; and (3) write the parity block and the new block. This operation requires $(N - 1 - d - c)$ reads and $(d + 1)$ writes, both of which comprise $(N - c)$ I/Os, where N is the number of disks, c is the number of cached clean blocks, and d is the number of dirty blocks to be updated. This process is known as a *reconstruct-write* cycle. When $d = N - 1$, it is unnecessary to read any block; this case is known as a *full-parity-block-group-write*[2].

A *read-modify-write* cycle can be used to reduce the number of I/Os when $N-c > 2(1+d)$, as the *reconstruct-write* cycle requires $(N - c)$ I/Os while the *read-modify-write* cycle requires $2(1 + d)$ I/Os. This process does the following: (1) it copies the new data to the cache memory; (2) it reads the old parity block (r) and reads the old block to a temporary memory (t) simultaneously; (3) it XORs the new block with the old block (xx), and XORs the result with the old parity block (x) to generate the new parity block (_); and (4) it writes the new block (w) and writes the new parity block (w) simultaneously, as shown in Fig. 4. The *read-modify-write* cycle requires $(1 + d)$ reads and $(1 + d)$ writes, both of which comprise $2(1 + d)$ I/Os.

For various cases of cache status, Fig. 5 shows operations for PBGs to be destaged by the mnemonics. If two blocks are cached (clean) and another block is written (dirty) as shown in Case 2 of Fig. 5, we choose the *reconstruct-write* cycle to destage the PBG, as $N - c < 2(1 + d)$, where $N = 5$, $c = 2$, and $d = 1$. Hence it is necessary to read the empty block (r), XOR all data blocks to update the parity block (x), and write the dirty block and the new parity (w). Therefore, the clean blocks only involve x, the dirty block requires xw, the empty block requires rx, and the parity block requires _w. If all blocks are dirty as in Case 4,

[1]Many people confuse stripe with PBG, as there is no widely-used terminology for PBG.

[2]The terminology full-stripe-write is usually used instead of it because there is no general terminology for PBG.

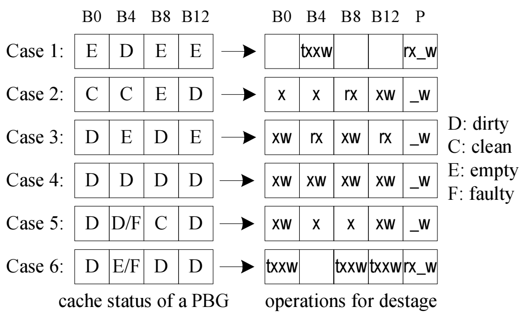
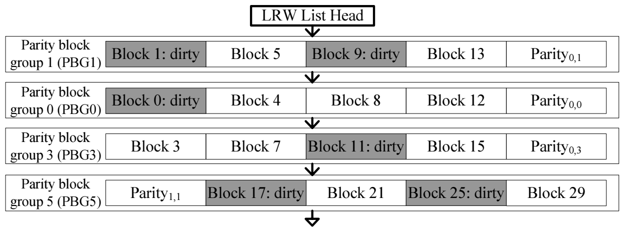
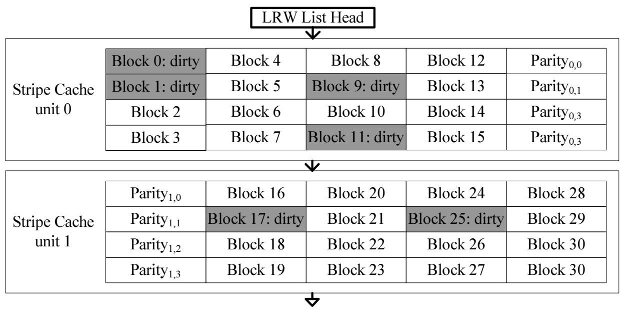

B0 B4 B8 B12
B0 B4 B8 B12 P
Case 1: E D E E → txxw rx_w
Case 2: C C E D → x x rx xw _w
Case 3: D E D E → xw rx xw rx _w
Case 4: D D D D → xw xw xw xw _w
Case 5: D D/F C D → xw x x xw _w
Case 6: D E/F D D → txxw txxw txxw rx_w
D: dirty
C: clean
E: empty
F: faulty
cache status of a PBG
operations for destage

Fig. 5. Destage operations of RAID-5 in various cases of PBG



(a) A LRW list in the Parity Block Group Cache (the traditional method)



(b) A LRW list in the Stripe Cache

Fig. 6. The LRW list examples for the conventional parity block group and the stripe cache when the sequence of block numbers for writes is ⟨17, 1, 9, 0, 11, 25⟩

it is necessary to XOR all data blocks without a read, and write all blocks and the parity. In other words, all data blocks and the parity block require xw and _w, respectively. Case 1 and 6 in Fig. 5 show the cases that use the *read-modify-write* cycle.

III. Matrix Stripe Cache-Based Contiguity Transform

We now describe our scheme for how to efficiently destage fragmented writes, namely, matrix stripe cache-based contiguity transform (MSC-CT). Most of the writes from the host can be converted into sequentially contiguous or sequentially discontiguous (fragmented) writes to the disk during the destage by a stripe cache (SC) that is managed in terms of stripe groups. MSC is a stripe cache managed by the rxw-matrix (explained in Section III-B) for the contiguity transform (CT) of fragmented writes. CT transforms fragmented I/Os into a contiguous I/O by inserting unnecessary I/Os into the discontiguous region using the rules for consistency and performance.

A. Stripe Cache

Figure 6(a) shows the traditional parity block group cache (PBGC) of RAID-5. Traditional cache replacements and destages are managed in terms of parity block groups (PBG) (see Section II for PBG). For example, the software RAID-5 of the Multi-device (MD) of Linux manages the LRW cache in terms of PBG. Fig. 6(b) shows an example of the LRW list of a stripe cache (SC) for RAID-5. Each SC unit corresponds to each stripe, which is the management unit for cache replacement and for destages. Figure 6 shows the LRW list examples when the sequence of the block numbers for writes is ⟨17, 1, 9, 0, 11, 25⟩.

In a PBGC that is destaged by LRW, the LRW list head points to the least recently written PBG. The blocks to be destaged are selected from the head of the LRW list. When a block in a PBG becomes dirty, the updated block and all of the other blocks that belong to the same PBG move to the end of the LRW list. For example, because block 25, which belongs to PBG5, is the most recently written block, PBG5 is in the end of the LRW list even though the block 17, which belongs to the PBG5, is the least recently written block. Similarly, in an SC that is destaged by LRW, the head of the LRW list points to the least recently written stripe cache unit. When a block in a stripe cache unit becomes dirty, the updated block and all of the other blocks that belong to the same stripe cache unit move to the end of the LRW list. Therefore, SC exploits spatial locality by coalescing writes that are in the stripe.

When a stripe cache unit destages, all blocks in the stripe are destaged sequentially, thereby converting all types of writes into sequentially contiguous or sequentially discontiguous writes within the stripe. SC also allows a better exploitation of spatial locality by coalescing writes together. Disk schedulers and destage algorithms that determine what to destage also save disk head seeks. However, MSC-CT can coexist with such schemes and helps such schemes to exploit much spatial locality. WOW also manages the cache in terms of stripe groups to reduce the sorting overhead of CLOCK and to exploit spatial locality. However MSC-CT treats the stripe cache unit as a matrix and enhances the performance of fragmented writes using the CT scheme.

B. Contiguity Transform

If a disk head misses the subsequent fragmented block due to the missing point of Fig. 2, it requires a full disk spin (see Section I-C). To alleviate this problem, we propose a contiguity transform (CT) process at the destage. CT transforms the discontiguous writes into the contiguous writes by inserting additional writes in between the discontiguous writes. CT is also applied to the read operations that occur in the reconstruct-write and read-modify-write cycles. Fig. 7 shows an example of the CT process in a RAID-5 array when a stripe is destaged. However CT is applicable to the other types of RAID, such as RAID-6.

Figure 7(a) shows an example of the block status in a stripe, where D denotes a dirty block in which new data is in the cache but not yet updated to a disk, C denotes a clean block in which consistent data with the disk is in the cache, and E denotes an empty block in which valid data is not in the cache. Let u be the number of blocks per strip, and let v be the number of disks consisting of a RAID-5 array. The cache status of the blocks in a stripe that is shown in Fig. 7(a) can be represented by the following $u \times (v-1)$ matrix:

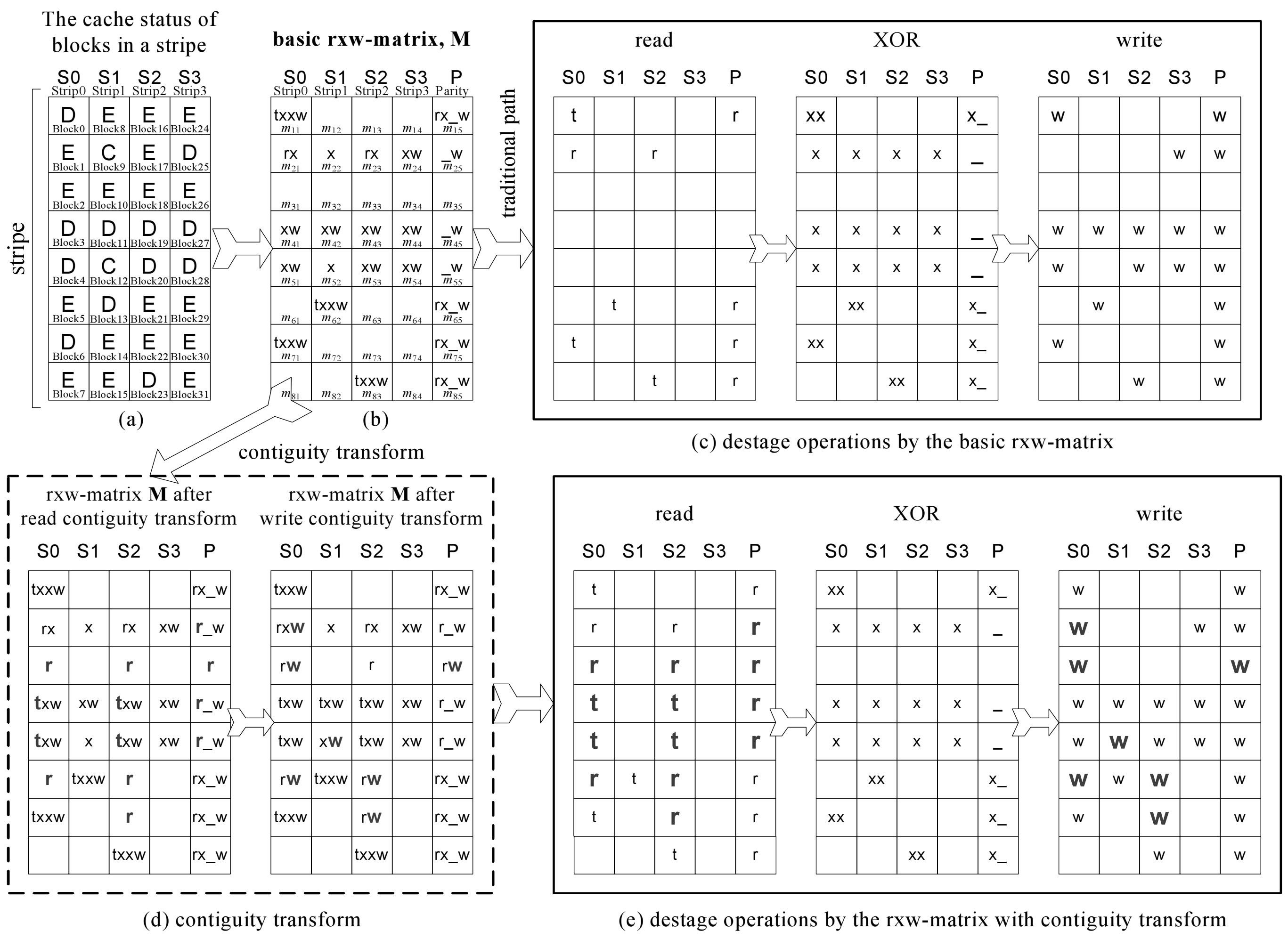

The cache status of blocks in a stripe
basic rxw-matrix, M
stripe
traditional path
read
XOR
write
(a)
(b)
(c) destage operations by the basic rxw-matrix
contiguity transform
rxw-matrix M after read contiguity transform
rxw-matrix M after write contiguity transform
(d) contiguity transform
(e) destage operations by the rxw-matrix with contiguity transform

Fig. 7. MSC-based contiguity transform: When a RAID driver destages a stripe, it uses the reconstruct-write cycle or the read-modify-write cycle for each row to generate the basic rxw-matrix. It is possible to successfully destage the stripe by executing reads, XORs, and writes based on the basic rxw-matrix (c). However the proposed scheme inserts the contiguity transform process (d) between the generation of the basic rxw-matrix (b) and the actual execution of the rxw operations (e).

$$\mathbf{Z} = [z_{ij}]_{u\times(v-1)} = \begin{bmatrix} \mathsf{D} & \mathsf{E} & \mathsf{E} & \mathsf{E} \\ \mathsf{E} & \mathsf{C} & \mathsf{E} & \mathsf{D} \\ \mathsf{E} & \mathsf{E} & \mathsf{E} & \mathsf{E} \\ \mathsf{D} & \mathsf{D} & \mathsf{D} & \mathsf{D} \\ \mathsf{D} & \mathsf{C} & \mathsf{D} & \mathsf{D} \\ \mathsf{E} & \mathsf{D} & \mathsf{E} & \mathsf{E} \\ \mathsf{D} & \mathsf{E} & \mathsf{E} & \mathsf{E} \\ \mathsf{E} & \mathsf{E} & \mathsf{D} & \mathsf{E} \end{bmatrix}. \quad (2)$$

Before the actual execution of the read, XOR, and write for all blocks in a stripe, it is necessary to determine which blocks should be read, how the parity blocks should be made, and which blocks should be written, by generating a basic rxw-matrix, as shown in Fig. 7(b). By choosing one of the reconstruct-write cycle and the read-modify-write cycle shown in Fig. 5 for each row of the matrix $\mathbf{Z}$, we determines the basic rxw-matrix, $\mathbf{M}$, whose element, m_{ij}, is a subset of {t, r, x, xx, _,w}. The basic rxw-matrix represents all operations that destage all blocks in the stripe. We can express the basic rxw-matrix shown in Fig. 7(b) as the following equation:

$$\mathbf{M} = [m_{ij}]_{u\times v} = \begin{bmatrix} \{\mathsf{t,xx,w}\} & \{\} & \{\} & \{\} & \{\mathsf{r,x,\_,w}\} \\ \{\mathsf{r,x}\} & \{\mathsf{x}\} & \{\mathsf{r,x}\} & \{\mathsf{x,w}\} & \{\mathsf{\_,w}\} \\ \{\} & \{\} & \{\} & \{\} & \{\} \\ \{\mathsf{x,w}\} & \{\mathsf{x,w}\} & \{\mathsf{x,w}\} & \{\mathsf{x,w}\} & \{\mathsf{\_,w}\} \\ \{\mathsf{x,w}\} & \{\mathsf{x}\} & \{\mathsf{x,w}\} & \{\mathsf{x,w}\} & \{\mathsf{\_,w}\} \\ \{\} & \{\mathsf{t,xx,w}\} & \{\} & \{\} & \{\mathsf{r,x,\_,w}\} \\ \{\mathsf{t,xx,w}\} & \{\} & \{\} & \{\} & \{\mathsf{r,x,\_,w}\} \\ \{\} & \{\} & \{\mathsf{t,xx,w}\} & \{\} & \{\mathsf{r,x,\_,w}\} \end{bmatrix}. \quad (3)$$

It is possible to successfully destage the stripe by executing reads, XORs, and writes that are based on the basic rxw-matrix as shown in Fig. 7(c). However, the proposed scheme inserts the contiguity transform process between the generation of the basic rxw-matrix and the actual execution of the rxw operations as shown in Fig. 7(d). The contiguity transform consists of a read contiguity transform and a write contiguity transform. The read contiguity transform must precede the write contiguity transform to increase the possibility of the write contiguity transform. Section III-C describes the reasons in detail.

The read contiguity transform inserts read operations, which appear as the bold-faced r in Fig. 7(d) and 7(e), between two discontiguous reads for each column in the basic rxw-matrix. For the example in Fig. 7(b), there exists the read operation r in m_{21}, {r,x}; the read operation t belongs to m_{71}, {t,xx,w}. However, there is no read operation between the two elements m_{21} and m_{71}. Hence, r or t is inserted into all of the elements between m_{21} and m_{71}. The read operations, r and t, must be properly chosen for consistency. The choice of r and t is explained in Section III-C. The left matrix of Fig. 7(d) shows the rxw-matrix after applying the read contiguity transform.

If we formally describe the above constraints, i and j of the rxw element m_{ij}, to which we can add the read operation, must satisfy at least the following predicate:

$$\begin{aligned} &\forall i \forall j \forall a \forall b[\ (a < i < b) \\ &\wedge\ ((m_{aj} \cap \{\mathsf{r}, \mathsf{t}\}) \neq \emptyset)\ \wedge\ ((m_{bj} \cap \{\mathsf{r}, \mathsf{t}\}) \neq \emptyset) \\ &\wedge\ (\neg \exists h \forall h[((m_{hj} \cap \{\mathsf{r}, \mathsf{t}\}) \neq \emptyset)\ \wedge\ (a < h < b)])\ ]. \end{aligned} \tag{4}$$

The rxw element m_{ij}, to which the read operation can be added, is in the discontiguous region between two discontiguous blocks, m_{aj} and m_{bj}, both of which include r or t. In the discontiguous region, there exists no element (m_{hj}) that includes r or t. In the case of the strip 0 shown in Fig. 7(b), it is possible to add r or t between m_{21} and m_{71} by predicate (4) with $j = 1$, $a = 2$, and $b = 7$.

The write contiguity transform is processed in the same manner as the read contiguity transform. For example, in the basic rxw-matrix, M, shown in Fig. 7(b), there is the write operation, w, in the two separated elements, m_{11} and m_{41} but there is no w in m_{21} and m_{31}, to which, therefore, w is added. It is now feasible to transform two discontiguous writes into a single write command.

The indices i and j of the rxw element m_{ij}, to which the write operation can be added, must satisfy at least the following predicate that is similar to predicate (4):

$$\begin{aligned} &\forall i \forall j \forall a \forall b[\ (a < i < b) \\ &\wedge\ (\mathsf{w} \in m_{aj})\ \wedge\ (\mathsf{w} \in m_{bj}) \\ &\wedge\ (\neg \exists h \forall h[(\mathsf{w} \in m_{hj})\ \wedge\ (a < h < b)])\ ]. \end{aligned} \tag{5}$$

In this way, two separated requests can be merged into a single request, thereby reducing the number of fragmented requests that show poorer performance than the contiguous I/O that employs the same start and end block. This section describes the contiguity transform and the several rules required for consistency and performance. The following sections present the rules.

C. Rules for Consistency

We must distinguish r and t for data consistency in the read contiguity transform. If a cache status z_{ij} is dirty, reading old data from the disk to the cache through r spoils the latest data in the cache. Hence, t should be used to preserve the newest data in the cache. t reads data to a temporary memory that will be released after the destage. If the block status is not dirty, r is used for a cache hit in the future. For example, t is added to m_{51} in Fig. 7, as z_{51} is dirty (D). However r is used for m_{61}, as z_{61} is not dirty. This process can be formally written as the following equation:

$$m_{ij} \leftarrow \begin{cases} m_{ij} \cup \{\mathsf{t}\} & \text{if } z_{ij} = \mathsf{D}, \\ m_{ij} \cup \{\mathsf{r}\} & \text{otherwise.} \end{cases} \tag{6}$$

We add t to m_{ij} if the cache status z_{ij} is dirty, otherwise r is added to m_{ij}.

If we implement a MSC driver that a physically contiguous memory is allocated to each strip, a read group that is composed only of r can be made for a single request without a scatter-gather list, by which a single procedure call sequentially reads data form a single data stream to multiple buffers. A read group that contains t may be merged into a single request using a scatter-gather list.

In contrast to the read contiguity transform, the write contiguity transform can not be processed in the case that at least one cache block, which is empty (E) and contains no r, exists in the discontiguous region that is located between the two separated blocks. By r, the empty cache block becomes clean (C) before the write execution to the disk. It is possible, therefore, to add w into an rxw element containing r, although the corresponding block cache is empty. Therefore, the read contiguity transform must precede the write contiguity transform.

In other words, the write contiguity transform between m_{aj} and m_{bj} must satisfy the following predicate:

$$\neg \exists l \forall l[\ (a < l < b)\ \wedge\ (z_{lj} = \mathsf{E})\ \wedge\ (\mathsf{r} \notin m_{lj})\ ]. \tag{7}$$

For all m_{lj} that are in the discontiguous region between m_{aj} and m_{bj}, there must be no m_{lj} such that its cache status, z_{lj}, is empty and it does not contain r.

For example, m_{24} and m_{44} in Fig. 7(b) form a discontiguous write operation but m_{34} of the final rxw-matrix shown in Fig. 7(d) does not contain r and z_{34} is empty. Hence it is impossible to make a write contiguity transform in this case. In the other case, blocks between m_{21} and m_{71}, z_{31} and z_{61} are empty, and m_{31} and m_{61} of the basic rxw-matrix do not contain r. However, m_{31} and m_{61} will contain r after the read contiguity transform (see Fig. 7(d)), thereby containing the latest data in the cache before the write execution and enabling the write contiguity transform of $\langle m_{11}, m_{41} \rangle$ and $\langle m_{51}, m_{71} \rangle$. The read contiguity transforms of m_{63}, m_{73}, and m_{35} also enable the write contiguity transforms.

D. Rules for Performance

To guarantee the performance of CT without degradation, we introduce a terminology, stride distance, which is defined by the number of blocks between two discontiguous writes. In other words, if two discontiguous writes are located at block position a and b and if a write operation between a and b does not exist, then the stride distance is $(b - a)$. In addition, the stride distance for a read is applied in the same way.

The write contiguity transform is restricted by automatically determined parameters, $S_{\mathrm{w}}^{\max}$ and $\mathbb{S}_{\mathrm{w}}^{\mathrm{except}}$, that are shown in Fig. 9. $S_{\mathrm{w}}^{\max}$ is the threshold value of the stride distance S in order to limit a write contiguity transform process. A write contiguity transform is permitted when its stride distance, S, is less than $S_{\mathrm{w}}^{\max}$ and not included in any excluded regions that are determined by $\mathbb{S}_{\mathrm{w}}^{\mathrm{except}}$ $(= \{S_{\mathrm{w},1}^{\mathrm{except}}, S_{\mathrm{w},2}^{\mathrm{except}}, \dots\})$. Fig. 9(d)

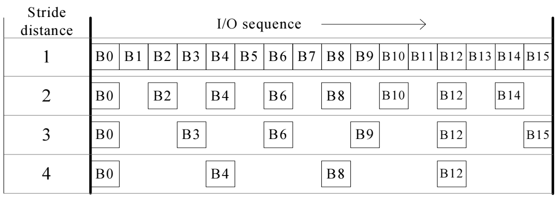
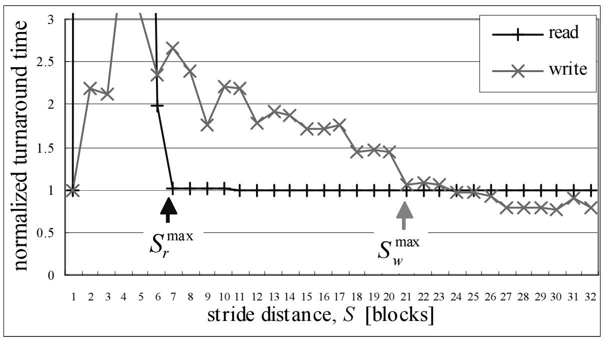
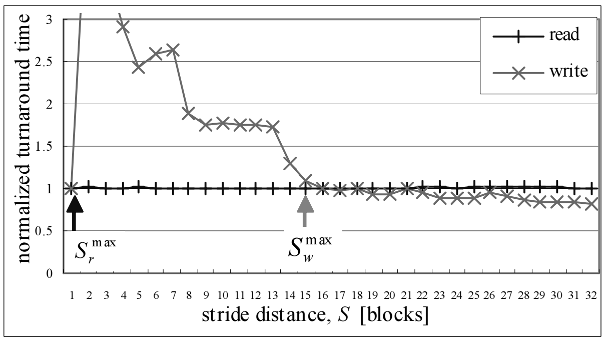

Stride distance
I/O sequence
1
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15
2
B0 B2 B4 B6 B8 B10 B12 B14
3
B0 B3 B6 B9 B12 B15
4
B0 B4 B8 B12

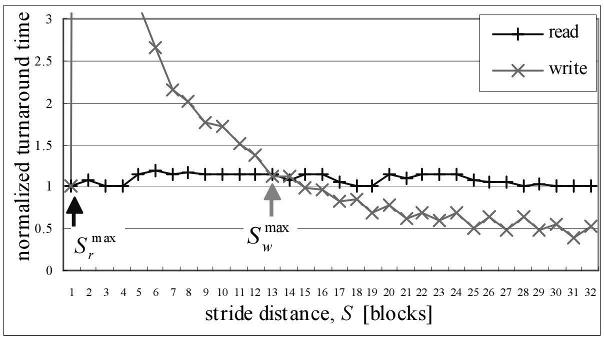
Fig. 8. The stride benchmark generates the workload in stride pattern by varying the stride distance. All workloads employ the same start and end positions. The stride distance of 1 denotes the contiguous I/O. When the stride distance is i, after one block is read or written, the consecutive $(i - 1)$ blocks are skipped and this cycle repeats.



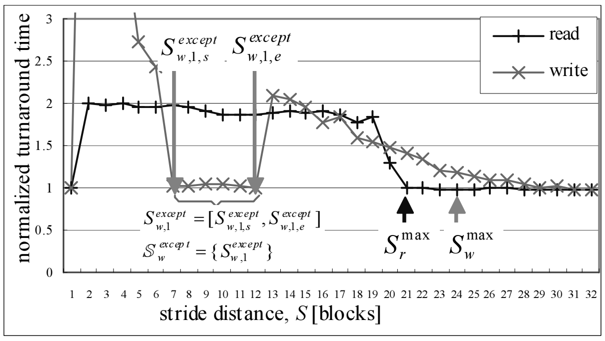
(a) ST373454LC (Seagate 15000rpm, SCSI320)



(b) ST3120023AS (Seagate 7200rpm, SATA)



(c) 6V160E0 (Maxtor 7200rpm, SATA)



(d) WD2500JD (WesternDigital 7200rpm, SATA)

Fig. 9. $S_{\mathrm{w}}^{\max}$, $S_{\mathrm{r}}^{\max}$, $\mathbb{S}_{\mathrm{w}}^{\mathrm{except}}$, and the normalized turnaround time versus the stride distance for various disks. A fragmented I/O whose normalized turnaround time is below one is faster than the contiguous I/O.

shows an excluded region, $S_{\mathrm{w},1}^{\mathrm{except}}$, that ranges from $S_{\mathrm{w},1,\mathrm{s}}^{\mathrm{except}}$ to $S_{\mathrm{w},1,\mathrm{e}}^{\mathrm{except}}$. If there exists a second excluded region, it can be expressed as $S_{\mathrm{w},2}^{\mathrm{except}}$. For the read contiguity transform, $S_{\mathrm{r}}^{\max}$ and $\mathbb{S}_{\mathrm{r}}^{\mathrm{except}}$ are applied in the same manner.

$S_{\mathrm{w}}^{\max}$, $\mathbb{S}_{\mathrm{w}}^{\mathrm{except}}$, $S_{\mathrm{r}}^{\max}$, and $\mathbb{S}_{\mathrm{r}}^{\mathrm{except}}$ are obtained from a member disk of a disk array by a stride benchmark, which is automatically performs when an administrator creates the disk array. The obtained parameters are saved in a non-volatile storage, on which the other information of the disk array is also stored. Hence, the stride benchmark that obtains the parameters performs only once. It is possible to retrieve the parameters from the non-volatile storage without executing the stride benchmark any more.

To find the statistical and relative costs of discontiguous I/Os over the contiguous I/O shown in Fig. 9, the stride benchmark generates the workload in a stride pattern by varying the stride distance as shown in Fig. 8. The stride distance S of 1 indicates the contiguous I/O. If S is i, after one block is read or written, the next consecutive $(i-1)$ blocks are skipped, and this cycle repeats. In this way, the stride benchmark repeats such a pattern to statistically evaluate a fragmented I/O with a specific stride distance. All workloads start at the same source block and stop at the same destination block.

Figure 9 displays the parameters and the normalized turnaround time versus the stride distance for various disks. A fragmented I/O is slower than the contiguous I/O if its normalized turnaround time is larger than 1. In all of the disks shown in Fig. 9, a fragmented I/O such that its stride distance S is less than a threshold shows degraded performance. The threshold value can be the maximum stride distance, $S_{\mathrm{w}}^{\max}$ for write or $S_{\mathrm{r}}^{\max}$ for read. Therefore the write contiguity transform performs when its S is less than $S_{\mathrm{w}}^{\max}$. $S_{\mathrm{w}}^{\max}$ is chosen as the smallest S such that the normalized turnaround time is less than 1.2 for all S. 1.2 is used instead of 1.0 because it is necessary consider the tolerance of the benchmark results and the performance degradation caused by additional I/Os that occupy additional bus resources such as the SCSI bus and the memory bus. The threshold value of 1.2 was obtained by some empirical results, hence it may not be optimal to apply the value to all systems. $S_{\mathrm{r}}^{\max}$ is also determined in the same way. This selection method of $S_{\mathrm{w}}^{\max}$ and $S_{\mathrm{r}}^{\max}$ is validated through an experiment, which is described in Section IV-F.

We can formally describe the performance rules for the read contiguity transform, whose stride distance is denoted by $S(= b-a)$, as the following predicate:

$$[\ (S < S_{\mathrm{r}}^{\max})\ \wedge\ (\neg\exists k \forall k [S_{\mathrm{r},k,\mathrm{s}}^{\mathrm{except}} \leq S \leq S_{\mathrm{r},k,\mathrm{e}}^{\mathrm{except}}])\ ]. \quad (8)$$

S must be less than $S_{\mathrm{r}}^{\max}$ and not be included in $[S_{\mathrm{w,k,s}}^{\mathrm{except}}, S_{\mathrm{r},k,\mathrm{e}}^{\mathrm{except}}]$ for all k. Similarly, the write contiguity transform must satisfy the following predicate:

$$[\ (S < S_{\mathrm{w}}^{\max})\ \wedge\ (\neg\exists k \forall k [S_{\mathrm{w},k,\mathrm{s}}^{\mathrm{except}} \leq S \leq S_{\mathrm{w},k,\mathrm{e}}^{\mathrm{except}}])\ ]. \quad (9)$$

In summary, the total rule for the read contiguity transform is constructed by aggregating (6), (4), and (8). The total rule for the write contiguity transform is constructed by aggregating (5), (7), and (9).

Some disks, as shown in Fig. 9(d), may have excluded regions, $\mathbb{S}_{\mathrm{w}}^{\mathrm{except}}$ or $\mathbb{S}_{\mathrm{r}}^{\mathrm{except}}$. CT does not perform if S is included in such excluded regions because it is impossible to achieve a gain through CT. Unlike $S_{\mathrm{w}}^{\max}$, some disks have a $S_{\mathrm{r}}^{\max}$ value of 1, as shown in Figs. 9(b) and 9(c).

```
ReadContiguityTransform($\mathbf{M}$, $\mathbf{Z}$, $S_r^{max}$, $\mathbb{S}_\mathrm{r}^\mathrm{except}$)
1    for ($j \leftarrow 1$; $j \le v$; $j$++) { // for each column
2      $p \leftarrow -1$;
3      for ($i \leftarrow 1$; $i \le u$; $i$++) { // for each row
4        if ($m_{ij} \cap \{\mathsf{r},\mathsf{t}\}) \neq \emptyset$ ) {
5          $S \leftarrow i - p$;
6          if ($p \ge 0$ and $S < S_r^{max}$
              and ($S < S_{\mathrm{r},k,\mathrm{s}}^\mathrm{except}$ or $S > S_{\mathrm{r},k,\mathrm{e}}^\mathrm{except}$, for all $k$)) {
7            for ($h \leftarrow p + 1$; $h < i$; $h$++) {
8              if ($j = v$) $m_{hj} \leftarrow m_{hj} \cup \{\mathsf{r}\}$;
9              else if ($z_{hj} = \mathsf{D}$) $m_{hj} \leftarrow m_{hj} \cup \{\mathsf{t}\}$;
10             else $m_{hj} \leftarrow m_{hj} \cup \{\mathsf{r}\}$;
11           }
12         }
13         $p \leftarrow i$;
14   } } }
WriteContiguityTransform($\mathbf{M}$, $\mathbf{Z}$, $S_w^{max}$, $\mathbb{S}_\mathrm{w}^\mathrm{except}$)
16   for ($j \leftarrow 1$; $j \le v$; $j$++) { // for each column
17     $p \leftarrow -1$;
18     for ($i \leftarrow 1$; $i \le u$; $i$++) { // for each row
19       if ( $\mathsf{w} \in m_{ij}$ ) {
20         $S \leftarrow i - p$;
21         if ($p \ge 0$ and $S < S_w^{max}$
              and ($S < S_{\mathrm{w},k,\mathrm{s}}^\mathrm{except}$ or $S > S_{\mathrm{w},k,\mathrm{e}}^\mathrm{except}$, for all $k$)) {
22           for ($h \leftarrow p + 1$; $h < i$; $h$++) {
23             if ($j = v$) {
24               if ($\mathsf{r} \notin m_{hj}$) goto Next;
25             } else if ($\mathsf{r} \notin m_{hj}$ and $z_{hj} = \mathsf{E}$) goto Next;
26           }
27           for ($h \leftarrow p + 1$; $h < i$; $h$++)
28             $m_{hj} \leftarrow m_{hj} \cup \{\mathsf{w}\}$;
29         }
30   Next: $p \leftarrow i$;
31   } } }
```

Fig. 10. The procedures of CT

If $S_\mathrm{r}^\mathrm{max}$ is 1, there is no write contiguity transform that is assisted by the read contiguity transform (see Section III-C). All reads shown in Fig. 9 are never below one. In other words, the throughput may be lightly affected even if we do not obey the rule of the normalized turnaround time of 1.2 to determine $S_\mathrm{r}^\mathrm{max}$. Therefore, if we aggressively increase $S_\mathrm{r}^\mathrm{max}$ without obeying the threshold value of 1.2 in order to increase the possibility of the write contiguity transform, we may achieve better performance. So we plan to research a better scheme of an aggressive transform.

E. Procedure

Figure 10 shows the pseudo code that represents the procedures of CT. Lines 4 and 13 resolve two discontiguous reads, m_{pj} and m_{ij}. Line 6 determines that the condition of the stride distance satisfies the performance rule. If the condition is true, lines 7-11 add r or t into all elements between m_{pj} and m_{ij} by the consistency rule. If column j is the parity column, r is added without checking the cache status (line 8) because the cache for parity blocks is not employed.

In the write contiguity transform, if the condition of the stripe distance satisfies the performance rule (line 21), lines 22-26 determine that it is possible to perform the write contiguity transform between m_{pj} and m_{ij} by the consistency rule (Eq. (7)). The discontiguous write region is skipped by jumping to line 30 if there is at least one element that does not satisfy the consistency rule. If column j is the parity column (line 23), we do not check the cache status (line 24). Finally, w is added to all elements between m_{pj} and m_{ij} (lines 27-28).

F. Performance Issues

The proposed scheme causes a negligible amount of performance degradation because CT requires very inexpensive computational power that investigates which blocks are applicable to CT. The overhead of CT is that additional reads and writes consume additional bus resources, such as those of the SCSI bus and memory bus. However, fragmented writes significantly deteriorate write performance, thereby rarely saturating those buses. Moreover, our experimental results show that CT enhances performance to a much greater extent than the degradation that is caused by the additional bus occupation.

The amount of performance enhancement depends on the amount of the fragmented writes and on the distribution of the stride distance. The proposed CT scheme employs no gain for the purely contiguous write, but affects various types of writes that include (1) concurrent sequential writes, (2) sequential writes on an aged filesystem, (3) sequential writes of multiple small files, (4) random writes, and so on.

The benefit from MSC-CT depends on the internal characteristics of the disk drives and on the portion of the closely discontiguous writes that depend on the write cache size and the workspace, to which requested I/O locations are bounded in terms of storage capacity. The average stride distance increases as the workspace increases, and the average stride distance decreases as the write cache size increases. CT has greater benefits for shorter distributions of the stride distance.

CT performs for writes and reads for parity-update but not read requests from the host. A read request cannot be delayed to the disk unlike a write request. Hence a read request must immediately respond to the host with data from the disk. However CT requires multiple pending I/Os that form as sequentially discontiguous. Therefore, it is impossible to apply CT to the read requests.

MSC is a stripe cache managed by the rxw-matrix for the contiguity transform (CT) of fragmented writes. By coalescing I/Os together, SC allows for a better exploitation of spatial locality than parity block group cache (PBGC). SC simplifies cache management and control, thereby enhancing bulky or sequential I/Os. CT depends highly on the strip size of SC because CT performs within the strip. The maximum stride distance for CT is restricted by the strip size if the number of blocks per strip is less than $S_\mathrm{r}^\mathrm{max}$ or $S_\mathrm{w}^\mathrm{max}$.

G. Implementation

The RAID can be implemented either using a dedicated computing device, called hardware RAID, or using a host processor, called software RAID. Teigland [31] discussed several software RAIDs that include MutliDevice of Linux, Veritas Volume Manager, Sun Solstice DiskSuite, and FreeBSD Vinum.

In Linux 2.6, we have implemented a software RAID-5 with the MSC-based CT (MSC-CT) scheme as a block device driver. To be precise, this is the MSC driver, which includes a stripe prefetching scheme [32], a configuration tool for administrators, and a status notification interface using the */proc* filesystem. For complete implementation of the MSC driver, we exploit several design techniques as follows:

1) When and What to Destage: An important decision for destage is "When to Destage". We chose a destage scheduler as the well-known scheduler, the high/low mark scheduler [6]. Destages to disks are disabled when the cache occupancy is below a "low mark" (85%). If the cache occupancy is at or above a "high mark" (95%), then the destage is continued. Better destage schedulers such as the linear threshold (LT) scheduler [5], variants of LT [33], and a combination of LT and HLM [4] may be considered in the design, but destage schedulers are not in the scope of this paper. Upon starting the destage, we must determine which block or stripe is selected to be destaged. For a fair comparison with the existing RAID-5 driver of Linux, the simplest scheme, LRW, was used in order to decide what to destage.

2) Lock: A lock management of our implementation for parity access performs in terms of stripe groups, thereby resulting in a coarser granularity compared to a lock that is managed in terms of parity block groups. However our lock management is simple, thereby providing easy implementation and efficient performance for sequential or bulky I/Os. The coarse-grained lock is somewhat acceptable, except for a write lock while reading. The read lock can enter into the critical section that was previously granted with another read lock. The read lock can not be granted if the section is locked with a write mode. However, the granted duration of the write lock is very short because the write from the host to the NVS performs merely by copying requested data to the NVS. Destaging a stripe acquires the write lock, which delays read requests and write requests for the stripe until the destage of the stripe finishes. However, the stripe selected for destage may be the least frequently or least recently used stripe, thus read or write requests for the stripe, which is under destage, are rarely claimed.

3) Memory allocation: Two alternative memory allocation schemes can exist for SC. The first is that cache memories are only assigned to blocks that contain valid data, thereby increasing memory utilization. The second is that a physically-contiguous memory of the stripe size is assigned to the stripe cache unit, which, although, holds only a single clean or dirty block. The latter scheme decreases memory utilization for widely-spread random small I/Os; however, it can reduce the number of memory allocation processes by an order of magnitude, and is efficient for sequential or bulky I/Os. We adopted the latter allocation scheme.

IV. Evaluation

A. Experimental Set-up

This section describes a system that we built to measure the performance of MSC-CT and SC, and to compare them with the existing RAID-5 driver of Linux, MultiDevice (MD), the cache of which is managed in terms of PBG. In the source code of MD, the *stripe_head* structure represents the PBG. Hsieh et al. [34] evaluated the software RAID of Linux, MD, which has a comparable performance as a hardware RAID for most of test cases that are performed by them.

The system in the experiments uses dual 3.0 GHz Xeon processors, 1 GB of main memory, two Adaptec SCSI320 host bus adapters, and five ST373454LC disks, each of which has a speed of 15000 rpm and a 74 GB capacity. A Linux kernel (version 2.6.11) runs on this machine hosting benchmark programs, the ext3 filesystem, and the proposed MSC driver or the MD driver as a RAID-5 driver. Apart from the page cache of Linux, both the MD driver and the MSC driver have their own cache memory of 512 MiB. The block size is set to 4 KiB.

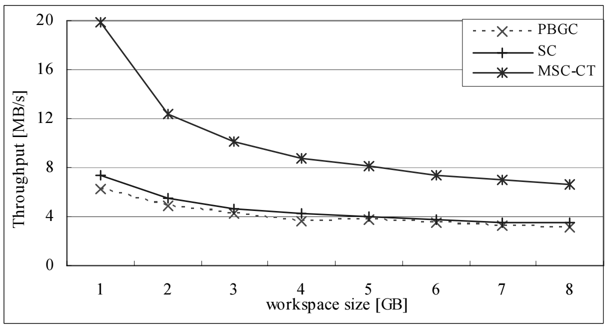


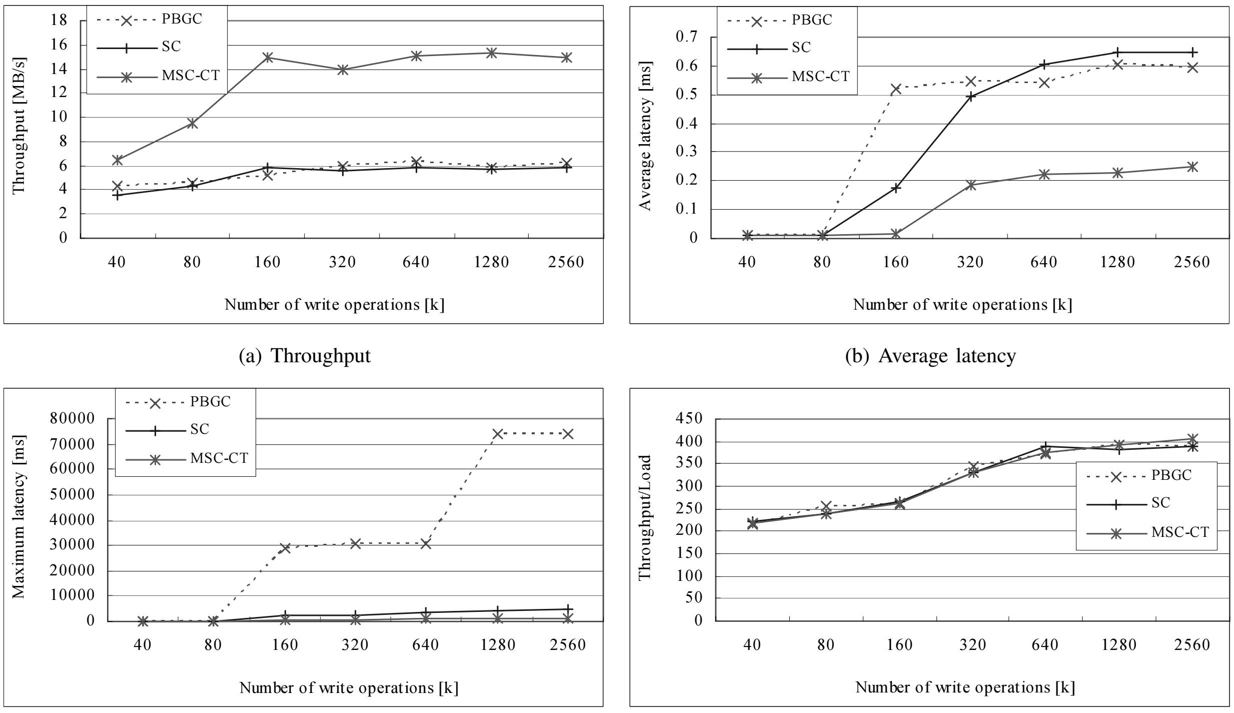
Fig. 11. A throughput comparison for random write workloads by varying the workspace size using Tiobench. As the workspace size increases, the gain of MSC-CT over SC decreases but never become negative.

All through the figures and tables of this paper, PBGC (parity block group cache, Fig. 6(a)) denotes the PBGC scheme using the MD driver, SC (stripe cache, Fig. 6(b)) when using the MSC driver without CT, and MSC-CT when using the MSC driver with CT.

B. Random Write Workload

We use a random write workload using Tiobench [35], which generates multiple threads that uniformly write 4 KiB pages in their own file. In Fig. 11, we compare PBGC, SC, and MSC-CT using random write workloads with a fixed number of writes (160k) on a RAID-5 array by varying the size of the workspace. A 1 GB file is allocated for each thread, and we vary the number of threads. Hence the workspace increases as the number of threads increases. At the lowest workspace size of 1 GB, MSC-CT outperforms SC by 170%, and MSC-CT outperforms PBGC by 220%. With a workspace size of 8 GB, MSC-CT outperforms SC by 108%, and SC outperforms PBGC by 10%. As the workspace size increases, the gain of MSC-CT over SC decreases but will not be negative because the increased workspace size with the fixed cache size causes for the average stride distance to increase and CT only performs when a stride distance is less than $S_{\mathrm{r}}^{\max}$ or $S_{\mathrm{w}}^{\max}$. Theoretically, if the workspace size is infinite, the throughput of MSC-CT is nearly identical to that of SC because there is no CT with an infinite stride distance.

Figure 12 shows various metrics such as throughput, average latency, maximum latency, and throughput over CPU load by varying the number of writes from a host with a fixed workspace size of 1.5 GB using Tiobench. In Fig. 12(a), the increase in the throughput is saturated as the number of I/Os increases. Meanwhile, too small a number of writes cannot consume the entire cache before the destaging of the cache and decreases the number of dirty blocks in a stripe, thereby increasing the average stride distance and reducing the transform opportunity. With the smallest number of writes (40k), the throughput gain of MSC-CT over SC is 53%. However, with the largest number of writes (2560k), MSC-CT outperforms SC by 141%. SC shows slightly less throughput than PBGC due to the memory allocation scheme of our implementation but not SC itself. The memory allocation of a full stripe size provides less memory utilization than the MD driver.



(a) Throughput (b) Average latency

(c) Maximum latency (d) Throughput / CPU load

Fig. 12. Various metrics by varying the number of I/Os using Tiobench

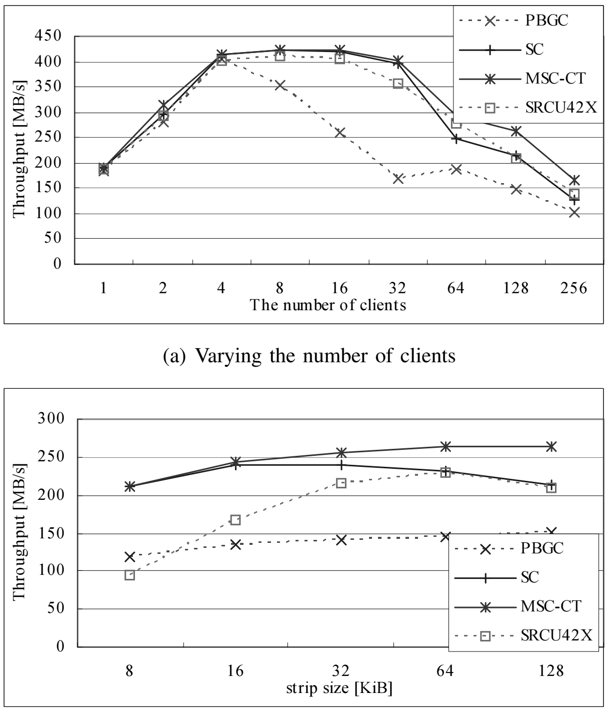
Figure 12(b) shows the average latency in this experiment. The average latency shows a similar trend to the throughput. When the number of writes is 2560k, MSC-CT delivers the maximum latency that is 260% less than SC. PBGC delivers severely poorer maximum latency than SC; nearly 56 times worse. This significant difference in the maximum latency may depend on the different implementation techniques of the MD driver for PBGC and the MSC driver.

It is necessary to measure the amount of CPU load degradation by the additional I/Os caused by CT. Throughput over CPU load may be an adequate metric for this measurement. According to Fig. 12(d) that shows this metrics, it becomes clear that PBGC, SC, and MSC-CT have similar values of throughput over CPU load. Therefore, CT rarely has a CPU load degradation.

C. Concurrent Sequential Write

To generate concurrent sequential I/Os, we used the benchmark DBench [36], which produces the local filesystem load. It does all the same I/O calls that the *smbd* server in Samba [37] would produce when confronted with a Netbench [38] run. DBench generates multiple threads, each of which produces various I/O patterns. However, a large portion of its writes form sequential or bulky accesses.

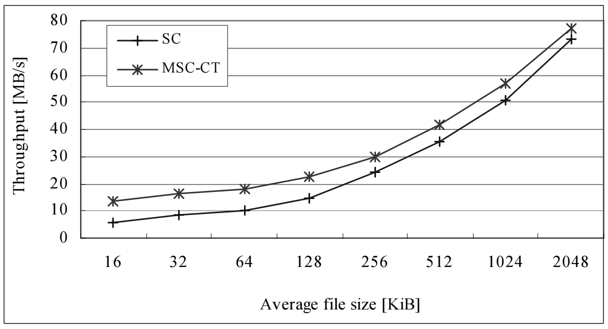
Figure 13 compares PBGC, SC, MSC-CT, and a hardware-based RAID using DBench. The hardware-based RAID is an Intel SRCU42X [39] with 512MB memory and write-back/cached-IO capability. The benchmarks used in this paper, except for DBench, cannot fairly compare SRCU42X with the software RAID drivers because there is no interface to fully flush the cache of SRCU42X. This can only be accomplished by shutting down its device driver. However, DBench excludes the results of the cleanup phase and thus it is unnecessary to fully flush the cache.



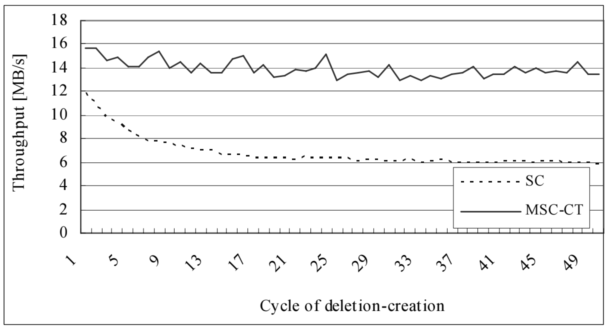
(a) Varying the number of clients

(b) Varying the strip size

Fig. 13. A comparison of PBGC, SC, and MSC-CT using DBench



(a) Varying the average file size



(b) Aging cycle

Fig. 14. A comparison of SC and MSC-CT on a synthetically aged filesystem

We vary the number of clients from 1 to 256 in Fig. 13(a). When the number of clients is 1, there is no gain by CT because most of the writes that each client produces are sequential or bulky. However, MSC-CT significantly outperforms SC, with the exception of a single thread and the range that the performance is saturated. In the best case, MSC-CT outperforms SC by 30%, and PBGC by 59%. Through this experiment, it becomes clear that concurrent sequential writes at the filesystem level may produce fragmented I/Os at the block level.

In Fig. 13(b), we vary the strip size from 8 to 256 with 128 clients. When the strip size is 8, there is no discontiguous region in a strip because one 8 KiB strip consists of only two 4 KiB blocks. As the strip size increases, the gain by CT increases. In the best case, MSC-CT outperforms PBGC by 80%. The throughput of SC decreases as the strip size increases as a result of the memory allocation scheme of our implementation. However, SC significantly outperforms PBGC in all of the cases shown in Fig. 13.

D. Aged Filesystem

In actual filesystems, files are divided into pieces scattered around the disk, thus sequential writes at the filesystem level may appear as fragmented writes at the block level. We made a benchmark to artificially age a filesystem. Up to 90% of the storage capacity, the benchmark sequentially writes files, the sizes of which have a normal distribution with a mean value μ and standard deviation $\mu/4$. The benchmark deletes 20% of the files, and then creates such files up to 90% of the storage capacity. To increase the fragmentation of the filesystem, the deletion and creation cycles were repeated several times.

This experiment uses an ext3 [40] filesystem of 10 GB. As shown in Fig. 14(a), the gain by MSC-CT in an aged filesystem is evaluated by repeating the process of deletion and creation 50 times. For an average file size of 16 KiB, MSC-CT is 2.3 times faster than SC. For an average file size of 2048 KiB, MSC-CT outperforms SC by 6%. The smaller the file, the better the throughput. The average file size of the */usr* folder in a Linux filesystem is approximately 12 KiB. The size of photo files ranges from 1 MiB from 2 MiB. Therefore the experimental results help infer the enhancement of MSC-CT in terms of widely used files.

Figure 14(b) shows the trend of the gain of MSC-CT on the aging amount of a filesystem with an average file size of 16 KiB. For the twentieth aging cycle over the first aging cycle, SC is degraded by 86% but MSC-CT is degraded by only 17%.

TABLE II
EXPERIMENTAL RESULTS OF BULKY I/OS

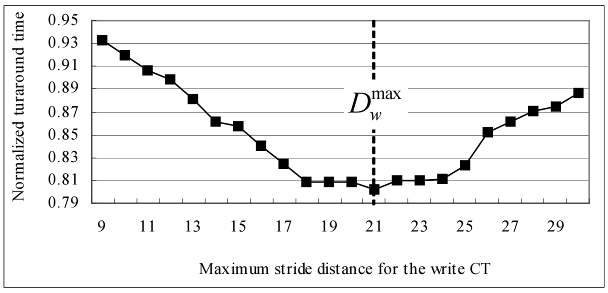
Benchmarks	PBGC (MD)	SC	MSC-CT
video self-copy	31.4 s	21.6 s	21.6 s
music self-copy	20.6 s	10.5 s	10.5 s
src self-copy	29.3 s	15.2 s	13.5 s
mkfs (ext3)	42.0 s	34.2 s	34.0 s
sequential read (fs)	190 MB/s	252 MB/s	252 MB/s
sequential write (fs)	166 MB/s	189 MB/s	189 MB/s
bulky random read (fs)	57.5 MB/s	67.4 MB/s	67.4 MB/s
bulky random write (fs)	58.8 MB/s	68.0 MB/s	68.0 MB/s
sequential read (block)	106 MB/s	381 MB/s	381 MB/s
sequential write (block)	168 MB/s	289 MB/s	289 MB/s
bulky random read (block)	81 MB/s	85 MB/s	85 MB/s
bulky random write (block)	80 MB/s	94 MB/s	94 MB/s

E. Sequential or Bulky I/O

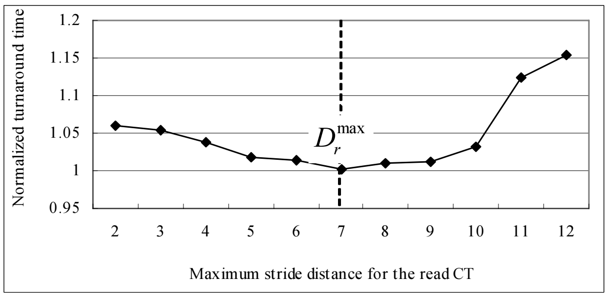
To investigate the effect of sequential or bulky I/Os on MSC-CT, we performed various experiments, as shown in Table II, where *self-copy* denotes the writing of files, that are read from a storage device, to a different directory on the same storage device. The *video*, *music*, and *src* benchmarks were performed by a single file of 2.13 GB, 154 mp3 files of 682 MB, and 25628 Linux kernel source and object files of 458 MB. We also performed sequential-random read-write tests both at the filesystem level using Tiobench and at the block level without a filesystem, where the request size and the boundary of the random I/O are aligned in the stripe so that there is no contiguity transform. The stripe size was set to 512 KiB in this experiment.

MSC-CT shows the same throughput as SC at *video self-copy*, *music self-copy*, and *sequential/random read/write*, while SC significantly outperforms PBGC in all of the cases, as shown in Table II because SC exploits more spatial locality than PBGC. SC delivers 45% better throughput than PBGC for *video self-copy*, and 117% higher throughput than PBGC for *src self-copy*. In addition, the MSC driver significantly outperforms the MD driver. In particular, the best increase in speed of 117% was achieved for the *src self-copy* benchmark. Therefore, MSC-CT employs no disadvantage for sequential or bulky I/Os, and SC delivers a more efficient performance than PBGC for sequential or bulky random I/Os.

MSC-CT outperforms SC by 13% for the *src self-copy* benchmark, which sequentially writes a large number of small files. The *src self-copy* benchmark produces fragmented writes that are caused by the pre-allocation scheme used in the ext3 filesystem:



(a) write



(b) read

Fig. 15. By varying $S_{\mathrm{w}}^{\max}$ and $S_{\mathrm{r}}^{\max}$, CT is applied to a fragmented read and write with an exponential distribution. The optimal $S_{\mathrm{r}}^{\max}$ and $S_{\mathrm{w}}^{\max}$ should produce the fastest turnaround time.

when a new disk data block is allocated, a filesystem such as ext2 or ext3 internally pre-allocates a small number of disk blocks adjacent to the just allocated block in order to expand files in the future [10]. Hence, there exists pre-allocated, unused, blocks between files.

The block-level throughput of the MD driver (PBGC) shows severe degradation because the anticipatory disk scheduler [12], which is the default disk scheduler of Linux 2.6, does not function well with the MD driver due to the write-after-read pattern. The anticipatory disk scheduler expects the read-after-read pattern. Our MSC driver shows the full throughput of the physical devices by repairing the problem that the scheduling strategy of the MD driver interferes with the anticipatory disk scheduler, where we neither modified nor disabled the anticipatory disk scheduler. The bulky random write of MSC-CT is better than the bulky random read because the anticipatory disk scheduler significantly reduces the average seek distance of writes.

F. Experimental Verification of $S_{\mathrm{r}}^{\max}$ and $S_{\mathrm{w}}^{\max}$

This section verifies the selection method of $S_{\mathrm{r}}^{\max}$ and $S_{\mathrm{w}}^{\max}$ that is performed by the repeating stride pattern described in Section III-D. The method selects $S_{\mathrm{r}}^{\max}$ and $S_{\mathrm{w}}^{\max}$ as the minimum stride distance, S, with a normalized turnaround time of less than 1.2.

Figure 15 shows the normalized turnaround time of the disk ST373454LC, which is shown in Fig. 9(a). The block interval between sequentially discontiguous blocks is exponentially distributed, where we use the mean value of 10. We assume that the stripe size is 128 KiB (= 64 blocks with the 4 KiB block size), hence there is no stride distance that exceeds 64 blocks within the 128 KiB stripe. Therefore, we limit the maximum stride distance up to 64 blocks. CT was performed on the workload by varying $S_{\mathrm{r}}^{\max}$ and $S_{\mathrm{w}}^{\max}$. Fig. 9 shows $S_{\mathrm{r}}^{\max}$ and $S_{\mathrm{w}}^{\max}$ that were selected by the selection method described in Section III-D. Fig. 9(a) with a stride pattern shows the same $S_{\mathrm{r}}^{\max}$ and $S_{\mathrm{w}}^{\max}$ as the fastest cases shown in Fig. 15 with an exponential distribution. The parameters, $S_{\mathrm{w}}^{\max}$ and $S_{\mathrm{r}}^{\max}$, that are obtained by the stride benchmark by using the stride pattern, are applicable to realistic workloads.

V. CONCLUSION

Conventional destage algorithms typically focus on when and what to destage. However, the proposed scheme, MSC-CT, solves how to efficiently destage fragmented writes. It was found that a sequentially contiguous write outperforms a sequentially discontiguous write, even if the two types of writes employ the equivalent seek distance. The stripe cache that manages the cache in terms of stripe groups converts all writes from the host into sequentially contiguous or sequentially discontiguous reads for parity-updates and writes to the disk. MSC-CT generates the rxw-matrix to destage a stripe and transforms two discontiguous reads or writes into a contiguous read or write by inserting additional reads or writes, thereby reducing fine-grained fragmented reads and writes. MSC-CT exploits the rules for consistency and performance, which enable data to be consistent without filesystem dependency, data modification, and performance degradation.

Additional reads of the read contiguity transform can assist the write contiguity transform by the rxw-matrix that separates the read stage from the write stage in term of stripe. Therefore MSC-CT is effective to disk arrays with sophisticated fault-tolerant schemes such as RAID-5, RAID-6, and Hierarchical RAID [41], which require additional reads to update redundant information. In such disk arrays, a fragmented write produces a fragmented read, thereby also obtaining the gain by the read contiguity transform and the higher possibility of the write contiguity transform.

We have implemented the MSC driver for RAID-5 as a block device driver of Linux, and compared it with the MD driver that is a software RAID driver of Linux. We used various types of workloads that include random writes using Tiobench, multiple concurrent sequential writes using DBench, sequential writes on an artificially-aged filesystem, and sequential writes of multiple small files on an ext3 filesystem. MSC-CT delivers a peak throughput that is 3.2 times higher than the traditional scheme in a random workload. In a setup-up for concurrent sequential I/O using DBench, MSC-CT delivers a peak gain that is 80% higher than MD. In a sequential write on an aged filesystem with an average file size of 16 KiB, MSC-CT delivers 180% throughput gain. Furthermore, in a sequential copy of small files, MSC-CT shows 2.7 times better throughput compared to MD. MSC-CT outperforms MD by 45% in a video file copy.

In summary, MSC-CT is a destage algorithm that destages fragmented I/Os in a RAID controller. MSC-CT is extremely simple to implement, has low overhead, and is ideally suited for RAID controllers for random I/Os as well as sequential I/Os. It was demonstrated that the RAID-5 implementation of MSC-CT significantly outperforms the existing RAID-5 driver of Linux in a variety of realistic scenarios.

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] J. Menon and J. Cortney, "The architecture of a fault-tolerant cached raid controller," in *20th Annual International Symposium on Computer Architecture*, May 1993, pp. 76–86.
[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," in *International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 27, no. 9, 1992, pp. 10–22.
[4] B. S. Gill and D. S. Modha, "WOW: Wise ordering for wrties - combining spatial and temporal locality in non-volatile caches," in *USENIX Conference on File and Storage Technologies*, 2005, pp. 129–142.
[5] A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with non-volatile caches," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE Computer Society Press and Wiley, 2001, pp. 129–146.
[6] P. Biswas and K. K. Ramakrishnan, "Trace driven analysis of write caching policies for disks," in *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 13–23.
[7] P. H. Seaman, R. A. Lind, and T. L. Wilson, "An analysis of auxiliary-storage activity," *IBM Systems Journal*, vol. 5, no. 3, pp. 158–170, 1966.
[8] F. J. Corbatò, "A paging experiment with the multics system," in *In Honor of P. M. Morse*. MIT Press, 1969, pp. 217–228.
[9] G. A. Gibson and R. V. Meter, "Network attached storage architecture," *Communications of the ACM*, vol. 43, no. 11, pp. 37–45, Nov. 2000.
[10] T. Y. Ts'o and S. Tweedie, "Planned extensions to the linux ext2/ext3 filesystem," in *USENIX Annual Technical Conference, FREENIX track*, June 2002, pp. 235–244.
[11] "Prefixes for binary multiples," International Electrotechnical Vocabulary (IEC 60027-2). [Online]. Available: http://www.iec.ch/zone/si-test/si_bytes.htm
[12] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o," in *USENIX Annual Technical Conference*, 2001, pp. 117–130.
[13] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes, "On-line extraction of SCSI disk drive parameters," The University of Michigan, Tech. Rep. CSE-TR-323-96, 1996.
[14] K. A. Smith and M. I. Seltzer, "File system aging - increasing the relevance of file system benchmarks," in *Measurement and Modeling of Computer Systems*, 1997, pp. 203–213.
[15] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.
[16] W. H. Ahn, K. Kim, Y. Choi, and D. Park, "DFS: A de-fragmented file system," in *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, 2002, pp. 71–80.
[17] E. Coffman, L. Klimko, and B. Ryan, "Analysis of scanning policies for reducing disk seek times," *SIAM Journal of Computing*, vol. 1, no. 3, pp. 269–279, 1972.
[18] P. J. Denning, "Effects of scheduling on file memory operations," in *AFIPS Spring Joint Computer Conference*, Apr. 1967, pp. 9–21.
[19] R. Geist and S. Daniel, "A continuum of disk scheduling algorithms," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 77–92, Feb. 1987.
[20] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 16–20 1994, pp. 241–251.
[21] M. Seltzer, P. Chen, and J. Ousterhout, "Disk scheduling revisited," in *USENIX Winter Tech. Conf.*, 1990, pp. 313–324.
[22] S. P. VanderWiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174–199, 2000.
[23] N. Megiddo and D. S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," in *USENIX Annual Technical Conference*, 2005, pp. 293–308.
[24] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *USENIX Conference on File and Storage Technologies*, Mar 2004, pp. 187–200.
[25] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," *IEEE Computer*, vol. 37, no. 4, pp. 58–65, April 2004.
[26] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities." in *USENIX FAST*, 2005.
[27] J. Wikes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," *ACM Trans. on Computer Systems*, vol. 14, no. 1, pp. 108–136, Feb 1996.
[28] S. Savage and J. Wilkes, "AFRAID— A frequently redundant array of independent disks," in *Proceedings of the 1996 USENIX Technical Conference*, 1996, pp. 27–39.
[29] K. Yeung and T. Yum, "Dynamic multiple parity (DMP) disk array for serial transaction processing," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 949–959, Sept. 2001.
[30] *The RAIDBook: A Source Book for RAID Technology sixth edition*. The RAID Advisory Board, Lino Lakes MN, 1999.
[31] D. Teigland and H. Mauelshagen, "Volume managers in linux," in *USENIX Annual Technical Conference*, June 2001, pp. 185–198.
[32] S. H. Baek and K. H. Park, "Massive stripe cache and prefetching for massive file I/O," in *IEEE International Conference on Consumer Electronics*, Jan 2007, pp. 5.3–5.
[33] C. P. Young Jin Nam, "An adaptive high-low water mark destage algorithm for cached RAID5," in *2002 Pacific Rim Internation Symposium on Dependable Computing (PRDC'02)*. IEEE Computer Society Press and Wiley, 2002, pp. 177–184.
[34] J. Hsieh, C. Stanton, and R. Ali, "Performance evaluation of software raid vs. hardware raid for parallel virtual file system," in *ICPADS '02: Proceedings of the 9th International Conference on Parallel and Distributed Systems*, 2002, p. 307.
[35] "Tiobench - Threaded I/O bench for Linux." [Online]. Available: http://directory.fsf.org/sysadmin/monitor/tiobench.html
[36] M. Vieira and H. Madeira, "A dependability benchmark for oltp application environments," in *Proceedings of the 29th VLDB Conference*, 2003.
[37] R. Sharpe, "Just what is SMB?" 2002. [Online]. Available: http://samba.org/cifs/docs/what-is-smb.html
[38] G. Memik, W. H. Mangione-Smith, and W. Hu, "NetBench: A benchmarking suite for network processors," in *ICCAD*, 2001, pp. 39–43.
[39] Intel Corporation, "Intel RAID controller SRCU42X overview," 2006. [Online]. Available: http://www.intel.com/design/servers/raid/srcu42x/index.htm
[40] S. Tweedie, "Journaling the Linux ext2fs filesystem," in *LinuxExpo '98*, 1998.
[41] S. H. Baek, B. W. Kim, E. J. Joung, and C. W. Park, "Reliability and performance of hierarchical RAID with multiple controllers," in *Twentieth ACM Symposium on Principles of Distributed Computing*, Aug 2001, pp. 246–254.

Sung Hoon Baek received the B.S. degree in electronics engineering from Kyungpook National University, Korea in 1997 and the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1999. He worked for Electronics Telecommunication Research Institution (ETRI) as a engineering staff from 1999 to 2005. He is currently pursuing a Ph.D. degree in the division of electrical engineering at KAIST. His research interests include storage systems, flash file systems, and embedded systems.

Kyu Ho Park received the B.S. degree in electronics engineering from Seoul National University, Korea in 1973, the M.S degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1975, and the Dr.Ing. degree in electrical engineering from the University de Paris XI, France in 1983. He has been a professor in the division of electrical engineering of KAIST since 1983. He was a president of Korea Institute of Next Generation Computing for the period of 2005-2006. His research interests include computer architectures, file systems, storage systems, ubiquitous computing, and parallel processing. Dr. Park is a member of KISS, KITE, Korea Institute of Next Generation Computing, IEEE, and ACM.