Information Systems 54 (2015) 28–42

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Parity Resynchronization using a Block-level Journaling for Software RAID

Sung Hoon Baek [a], Ki-Woong Park [b,*]

[a] Department of Computer System Engineering, Jungwon University, Republic of Korea
[b] Department of Computer Hacking and Information Security, Daejeon University, Republic of Korea

ARTICLE INFO

Article history:
Received 7 November 2014
Received in revised form
14 May 2015
Accepted 26 May 2015
Recommended by: F. Carino Jr.
Available online 5 June 2015

Keywords:
Secondary storage
Fault tolerance

ABSTRACT

Software redundant arrays of independent disks (RAID) suffer from several hours of resynchronization time after a sudden power-off. Data blocks and a parity block in a stripe must be updated in a consistent manner. However, a data block may be updated without a parity update if power goes off. Such a partially modified stripe must be updated with a correct parity block after a reboot. It is difficult, however, to find which stripe is partially updated. The widely used traditional parity resynchronization approach entails a very long process that scans the entire volume to find and fix partially updated stripes. As a remedy to this problem, this paper presents a parity resynchronization scheme that exhibits a small overhead for a wide range of workloads, finishes parity resynchronization within several minutes, and is transparent to file systems, thanks to a new seamless block-level journaling. The proposed scheme is integrated into a software RAID driver in a Linux system. A performance evaluation demonstrates that the proposed scheme shortens the resynchronization process from 200 min to 30 s with 1% overhead, compared to 51% overhead for the prior scheme.

1. Introduction

Various types of redundant arrays of independent disks (RAID) [1] have been introduced for large capacity and high throughput while protecting data against one or more disk failures. Research topics on RAID include access method [2,3], reconstruction [4–7], scrubbing [8–10], scaling [11–15], data layout [16–21], erasure code [22–26], and resynchronization [27–29]. This paper is focused on resynchronization for software RAID systems.

High-end RAID systems utilize uninterruptible power supply (UPS) or battery-backed RAM to achieve both reliability and performance [30]. However, such power-fail-safe devices are not used in software RAID systems.

When a write is issued to a RAID-5 array, two or more disks for data and parity must be updated in a consistent manner. If a sudden power-off occurs after a data block is written but before a parity block is updated, the stripe is left in an inconsistent state. In this case, the system cannot simply detect which stripe is in an inconsistent state after the crash.

Finding and fixing inconsistent stripes is called parity resynchronization. Because inconsistent stripes are not recoverable if a disk fails, the software RAID suffers from several hours of a parity resynchronization process that entails scanning the entire volume to search for inconsistent stripes [27]. Scanning the entire volume prolongs downtime and increases maintenance cost.

The software RAID has a strong possibility to experience parity resynchronization processes. The software RAID system

* Corresponding author. Tel.: +82 10 9165 1624; fax: +82 42 280 2404.
E-mail addresses: shbaek@jwu.ac.kr (S.H. Baek), woongbak@dju.kr (K.-W. Park).

http://dx.doi.org/10.1016/j.is.2015.05.004
0306-4379/

is widely deployed to workstations and home RAID devices. In a home environment, users may frequently turn off their RAID box as it makes a harsh noise. Some consumers do not properly shutdown the box, and thus they experience inconvenience of a long parity resynchronization process in the next power-on. In addition, software RAID may be run on an unreliable computer, without a reliable power source, or with buggy programs. Consequently, the software RAID will not be free from parity resynchronization.

As a RAID driver to reduce the expensive resynchronization time, the multi-device (MD) in the Linux kernel provides an intent bitmap scheme [28] that sacrifices ordinary write performance.

An alternate resynchronization approach eliminates the time-exhaustive resynchronization process with a small overhead by modifying the journaling scheme of file systems [29], but it must change the file systems and their tools such as *fsck*. This approach cannot serve a block device. Furthermore, more development costs will be incurred to apply the concept of the declared mode to various new file systems.

The proposed scheme of this paper aims at software RAIDs and provides a fast resynchronization process with a negligible overhead for ordinary I/Os while not changing the existing file systems. In addition, it is resistant to frequent *sync* and synchronous writes. Therefore, this technology effectively removes the need for costly power-fail-safe devices, thereby bringing a breakthrough to lower the cost of RAID systems.

2. Problem and prior solutions

2.1. Problem

Data blocks and a parity block that comprise a stripe must be updated in a consistent manner. If a data block is modified, the corresponding parity block must be updated at the same time.

On the other hand, the disk platters are not synchronized, and thus rotational latency time for each disk head to reach the same target sector varies from disk to disk. Therefore, the time that each disk spends on recording a block is different even if multiple disks receive the same write request at the same time.

When the RAID system crashes, the system cannot avoid an inconsistent stripe where its parity block has no relationship with its data blocks. If the power is turned off while the member disks of a RAID-5 array record a stripe on their magnetic medium, the stripe could be inconsistent because some parts of the stripe may not be written.

Not only RAID-5 but also all RAID levels (1, 6, etc.) that tolerate a disk failure suffer from inconsistent stripes after a power failure.

In the example shown in Fig. 1, a data block in disk3 is updated but the parity and the other data blocks in disk1 and disk2 are not updated due to a sudden power-off. The old parity of the stripe is inconsistent with the updated data of disk3.

If a disk fails while the RAID system has an inconsistent stripe, the RAID software performs a recovery process for the data blocks of the failed disk [23]. It recovers a wrong

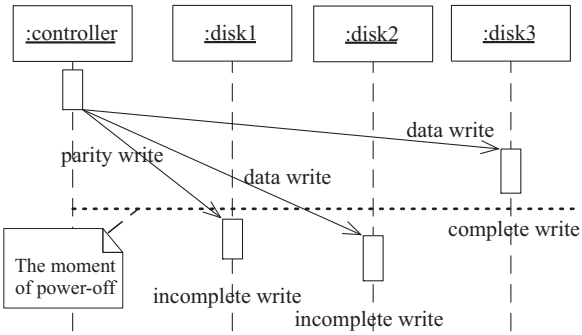


Fig. 1. A scenario for an inconsistent stripe: a data block in disk3 is updated but Disk1 and Disk2 fail to update their parity block and data block due to a sudden power-off. The old parity of the stripe is inconsistent with the updated data of disk3.

data block for inconsistent stripes because the parity block of the inconsistent stripe has no relationship with the data block and the system cannot recognize which stripe is inconsistent if a disk fails.

When a storage system reboots after a crash, there is no information on write activities that are pending or completed at the moment of the crash. Consequently, there is no indication of where stripe inconsistencies exist. Hence, during the next booting process after a crash, a long parity resynchronization process must be performed to find and fix inconsistent stripes.

2.2. Prior solutions

In terms of performance and cost, various approaches have been introduced to avoid stripe inconsistency when power goes off unexpectedly.

2.2.1. Hardware solution

To achieve both reliability and performance, hardware RAID employs non-volatile memory, which is implemented by battery-backed memory or an uninterruptible power supply (UPS). Updated data that are buffered in non-volatile memory are safely retained regardless of a crash [30].

After a system reboots after a crash, not-yet-destaged data blocks in non-volatile memory are destaged to disks and the corresponding parities are rebuilt. Therefore, the hardware solution achieves the best performance without any software overhead.

2.2.2. Full scanning

A widely used approach to find and fix inconsistent stripes is to read and inspect the entire volume [27]. Fig. 2 shows the time required for full scanning. Full scanning takes several hours or even days.

If I/Os are requested during full scanning, the storage system suffers from poor performance with high latency to serve both scanning and user requests.

2.2.3. Intent bitmap

Clements and Bottomley [28] introduced an intent bitmap for a short resynchronization process sacrificing ordinary write performance. An intent bitmap is used to

keep track of which data blocks are out of sync between the nonvolatile storage (e.g., RAID) and the volatile buffer. Multi-device (MD) of the Linux kernel employs the intent bitmap scheme [28].

A bit in the bitmap corresponds to a segment of the volume. The segment can be a stripe or a group of blocks. A bit of the intent bitmap is set to one if the corresponding segment in the buffer is not yet updated to the disk, and the bit value is cleared when the corresponding segment is synchronized with the disk.

The intent bitmap scheme inspects only the stripes that correspond to the set bits of the intent bitmap during the resynchronization process. Hence, this scheme can significantly shorten the resynchronization time.

Fig. 3 presents the state diagram for each bit of the bitmap. The bit must be set before updating the corresponding block. A daemon asynchronously clears the bit after finishing its update.

The intent bitmap boots up the resynchronization processes, but it comes at the cost of considerable overhead in a normal process.

2.2.4. Journaling

The proposed scheme inserts a journaling into a block level driver to avoid scanning the entire disk after a crash. Journaling has been proposed in the field of file systems.

The file system must be recovered after a system crash to bring the file system to a consistent state again by a tool such as fsck, which takes a long time to scan the entire disk. The journaling file system [31–34] allocates a dedicated area – the journal – in which it atomically records the changes in a form of circular queue before it updates the changes in the main file system. This approach is known as journaling (also known as write-ahead logging). The log keeps track of the changes of metadata or data before committing them to the main file system. In the event of crashes or power losses, such file systems quickly recover their consistency by analyzing only that log [35,32]. The recovery simply involves reading the last journal and updates changes from this journal to home.

Our scheme utilizes a journaling technique that makes the parity synchronization process quick and simple. However, we introduces a dual write cache that enables journaling at a block level and a new scheme that can reduce the frequency of journal updates by merging intent logs.

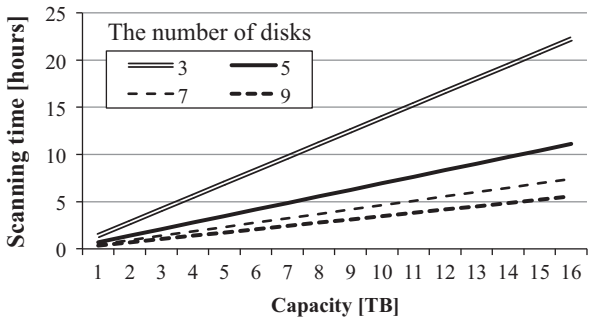


Fig. 2. Scanning time: full scanning takes tens of hours.

2.2.5. Ext3 ordered mode

Because of the high cost of writing every data block to both the journal and the main file system twice, people have tried a few different things in order to speed up performance. A simpler form of journaling is sometimes called ordered journaling (or just metadata journaling), and it is nearly the same, except that user data is not written to the journal [35]. In the event of a crash with a completed transaction that has a commit block, the file system copies the metadata in the journal to the home. If the last transaction has no commit block after a crash, the data and metadata blocks in the journal are discarded.

For example, ext3 ordered mode proceeds as follows:

1. It writes all dirty data blocks to the home, and issues a flush.
2. It appends descriptor block and metadata blocks to the journal.
3. It appends a commit block to the journal with the write barrier.
4. it writes metadata blocks to the home.

2.2.6. Ext3 declared mode

The ext3 declared mode [29] is a variant of the ext3 ordered mode to enhance the parity resynchronization process. This file system has a special interface with a software RAID to transfer an intent data from the file system to the software RAID. It additionally records intents of outstanding writes (called declare block) in its journal. The intents written at the file system level are transferred to the software RAID after a crash to resynchronize parity blocks.

Fig. 4 shows the journal layout of the ext3 declared mode. The ext3 declared mode is quite similar to the ext3

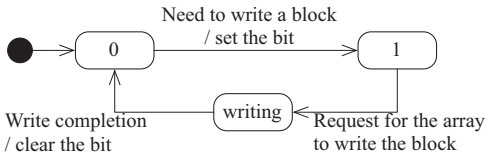
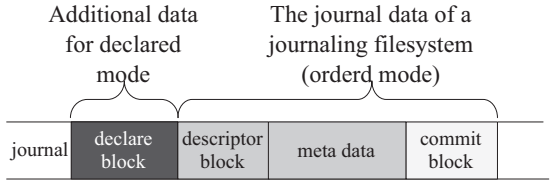


Fig. 3. The state diagram for each bit of the bitmap in the intent bitmap scheme: a bit corresponds to a stripe, a block, or a chunk of the entire volume.



(1) appends declare block to journal with the write barrier.
(2) writes data blocks to the home, and issues a flush.
(3) appends a descriptor block and metadata to the journal, and issues a flush.
(4) appends a commit block to the journal.
(5) write metadata blocks to the home.

Fig. 4. The journal layout of the ext3 declared mode.

ordered mode except for that a declared block is written prior to writing data blocks. The declare block includes the locations of data blocks.

The ext3 declared mode involves small overhead and provides fast resynchronization time. However, the RAID with the ext3 declared mode depends on the file system and therefore cannot support various other file systems such as ext4, Btrfs, and xfs. Furthermore, this approach cannot serve a block device.

2.3. Our contributions

Most systems do not sacrifice the performance of normal I/Os, and hence they use full scanning resynchronization, which requires several hours or even days. The full scanning resynchronization severely degrades the performance of the software RAID after a crash or a power-failure.

Unlike the ext3 declared mode, our scheme inserts the journal structure into the block level driver so that file systems do not need to be changed.

This paper presents a sophisticated scheme that

1. includes dual write caches that aggregate the locations of multiple writes into a single a write-ahead log (called the intent log in this paper) and enable both buffering and destaging to be processed simultaneously;
2. employs an intent log that describes the locations of future multiple writes with an additional I/O;
3. can reduce the frequency of updating intent logs by a merging scheme;
4. dramatically reduces the resynchronization time with much less overhead than that of prior works at normal I/Os; and
5. is transparent to file systems.

The proposed scheme is implemented in a software RAID as a kernel module in Linux. It shows a very small overhead for random writes, sequential writes, a file server workload, a mail server workload, a TPC-C trace, synchronous writes, and general usage. For example, it has 1% slowdown for a TPC-C trace whereas an other scheme has 51% slowdown. It reduces the resynchronization time to 30 s from 200 min. The proposed scheme provides a major breakthrough to solve the major problem of the software RAID .

3. Dual write caches and merged intent log

This chapter introduces the proposed parity resynchronization scheme, which has a low overhead for both synchronous and asynchronous writes, and does not employ any change of file systems, or a power-fail-safe device.

The proposed scheme consists of a merged intent log and dual write caches. The dual write caches aggregates multiple writes into an intent log and process both buffering and destaging simultaneously. The merged intent log scheme is optimized to the dual write caches and significantly mitigates a logging overhead.

The dual write caches utilize a volatile cache but guarantee file system consistency. The consistency of volatile write cache is described in Section 3.5.

3.1. Dual write caches

3.1.1. Problems

Our scheme is based on a block-level journaling that describes the locations of future writes. After a crash, a system can inspect a bounded number of stripes that are described by the journal (log) rather than scanning the entire volume.

A write cache can make the block-level journaling efficient by aggregating the information of write requests into a single log. A write cache may aggregate multiple writes, record a log, and destage the buffered data to disks.

During the destaging state, the write cache must reject new write requests. If not, newly buffered data after the last log is recorded may be destaged while the last log does not describe the information of the new data.

No data must be destaged before being registered in the last log. Hence, the buffering state and destaging state must be separated thereby sacrificing performance.

3.1.2. Continuous buffering and destaging

The discrete three steps – buffering, logging after buffering, and destaging after logging – are inefficient. However, we propose dual write caches that process both buffering and destaging simultaneously.

The basic idea is as follows: the system stores multiple write requests in a write cache, records the intent log in a nonvolatile storage before stripes in the write cache are destaged to disks, and destages all data in the write cache while new write requests are delivered to *another write* cache that does not destage its data to disks.

The dual write caches aggregate the intents of future multiple writes into a single intent log, which lists the locations of multiple blocks. The log gives hints about which stripes were being written when the power went off.

Fig. 5 shows how to process the intent log, which is produced by two write caches, the buffering write cache (WC_B) and the destaging write cache (WC_D).

The process consists of three steps. In Step 1, Stripe D, Stripe E, and Stripe F are buffered in the buffering write cache (WC_B). Stripe A, Stripe B, and Stripe C in the destaging write cache (WC_D) are destaged to disks. In Step 2, WC_B is swapped with WC_D if WC_B is not empty and WC_D becomes empty by destaging all its buffered data. In Step 3, the locations of Stripe D, Stripe E, and Stripe F are journaled as an intent log (IL). In the second iteration of Step 1, Stripe D, Stripe E, and Stripe F in WC_D are destaged to disks while new write requests for Stripe G, Stripe H, and Stripe I are buffered in WC_B. Thus both buffering and destaging are processed simultaneously.

Data blocks that are requested from the host are buffered only to WC_B. The only way to insert data blocks to WC_D is to swap WC_D with WC_B. Therefore, no data are destaged until their location is journaled in the intent log.

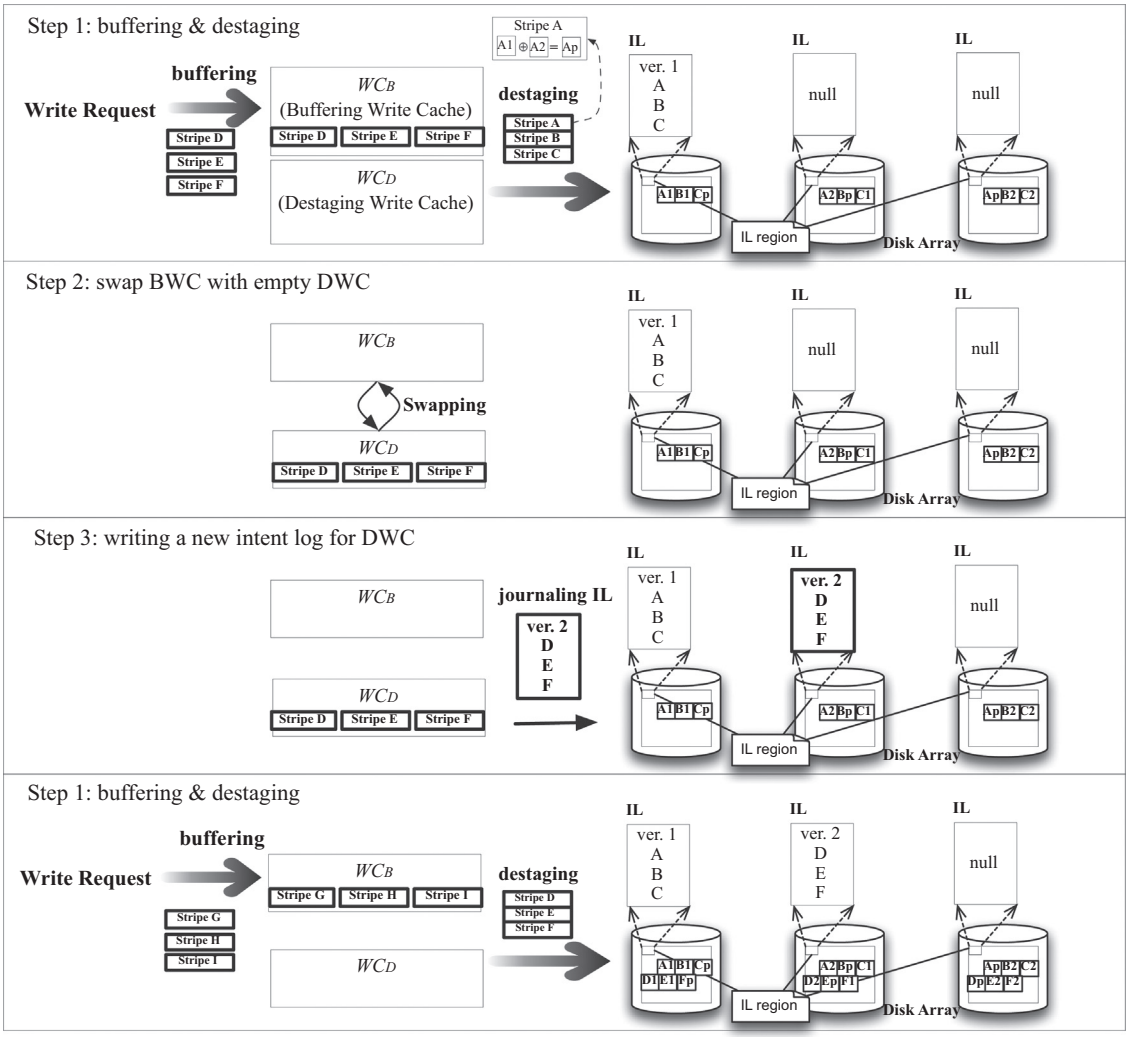


Fig. 5. The dual write caches process both buffering and destaging simultaneously, while no data are destaged until their location is journaled in the intent log region. (Step 1) All blocks that are requested from the host are buffered to WC_B. (Step2) If WC_D becomes empty and WC_B is not empty, WC_B is swapped with empty WC_D. (Step 3) An intent log containing the locations of all stripes that are buffered in the WC_D is logged in a designated region. In the second Step 1, Stripe D, Stripe E, and Stripe F in the WC_D are destaged to disks while new requests for Stripe G, Stripe H, and Stripe I are buffered in WC_B.

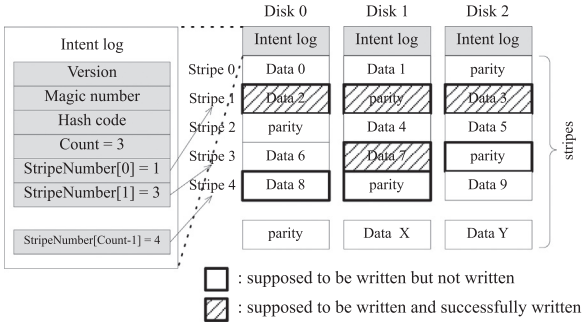


Fig. 6. The structure of an intent log.

3.1.3. Intent log

An intent log data contains stripe numbers or segment numbers as shown in Fig. 6. All stripes in WC_D are registered in the intent log before they are destaged to disks. Therefore, the last intent log can be a superset for the locations of inconsistent stripes that were being written at the time of a crash.

The parity resynchronization process can be restricted to the stripes that are indicated by the indent log, thereby dramatically reducing the resynchronization time.

The IL region where intent logs are journaled can be in the disk array or a small nonvolatile memory such as flash memory. In the implementation of this paper, the IL region is located at the disk array. Each intent log with an increasing version is recorded across the member disks in a round-robin fashion.

A software RAID in Linux, MD, allocates a region for an intent bitmap in a dedicated file or an internal area of the disk array. In our experiment, MD uses the disk array to record an intent bitmap by the option '-binternal'.

3.1.4. The varying sizes of WC_D and WC_B

Fig. 7 shows the size of WC_D and the size of WC_B for each step. Let the first write cache be WC_1 and the second write cache be WC_2. The solid line and dotted line indicate WC_1 and WC_2, respectively. First, let WC_1 be WC_B and WC_2 be WC_D. The size of WC_B increases by buffering (Step 1). WC_B is swapped with empty WC_D, so that WC_1 becomes WC_D and WC_2 becomes WC_B (Step 2). An intent log is logged in a designated region (Step 3).

WC_B and WC_D share the same memory resource. The size of WC_B can increase by buffering while the size of WC_D decreases by destaging because memory that is evicted from WC_D by destaging can be allocated to WC_B.

3.1.5. Miscellaneous operations

Cache policy: Stripes in WC_D can move to WC_B anytime. If a stripe in WC_D hits the cache, it moves to WC_B by a destaging scheme such as the least recently written (LRW) policy.

Forced cache synchronization: A cache synchronization enforces an IL update. If a cache synchronization is requested (in the middle of the third Step 1 in Fig. 7), no further buffering to WC_B is allowed and a new IL should be recorded for a small amount of data. However, Section 3.3 presents a merged intent log scheme that mitigates the IL overhead incurred by frequent cache synchronizations.

Invisible IL region: The RAID reserves the first blocks for the IL region, reports a slightly smaller capacity, and makes it invisible to the upper layer such as the file system. To make the region invisible, the logical block address from the upper layer is translated to a physical block address that is added by the size of the IL region. Hence, file systems can dynamically change their size over time without causing any problems.

3.2. Resynchronization

If the system detects an unclean shutdown in the booting stage, it performs the resynchronization process. First, it reads all intent log blocks that are stored across all member disks and finds the latest one that has the highest version number. The latest intent log indicates the stripes that were being updated at the moment of power-off.

For each stripe that is described in the latest intent log, the system reads data blocks and a parity block, and investigates whether the parity block is correct for the data blocks. If it is incorrect, a newly rebuilt parity block is updated to the stripe.

The latest intent log contains the stripe numbers for all stripes of the current WC_D, which is a superset of the stripes that were being written at the moment of the power-off.

Fig. 6 shows an example of the intent log, which indicates that some of the stripes are inconsistent (Stripe 3), some of them are consistently updated (Stripe 1), and some of them are not yet updated (Stripe 4). Stripe 1, Stripe 3, and Stripe 4 are candidates for inconsistent stripes and hence they are read and inspected to determine whether they are inconsistent. Stripe 3 is detected as inconsistent, and then a new parity block is rebuilt and recorded back to the disk.

The number of inspected stripes for resynchronization is limited by the maximum size of the intent log. Hence, the synchronization process can be completed within an estimated time.

The intent log requires an additional write access per hundreds of stripes; however the intent bitmap without the dual write cache causes two writes to set and clear the bit for every block in the worst case. The intent log scheme employs much less overhead than the intent bitmap scheme.

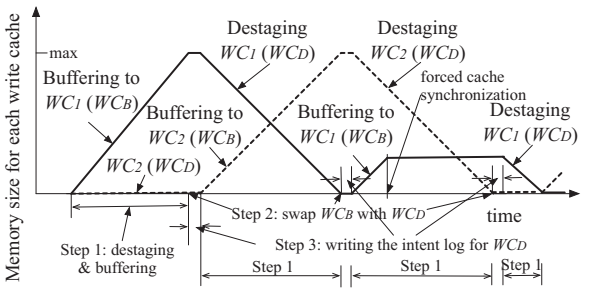


Fig. 7. The size of WC_D and the size of WC_B are shown for each step. The solid line and dotted line indicate WC_1 and WC_2, respectively.

3.3. Merged intent log

3.3.1. A problem by synchronous writes

In some cases, a forced write barrier or a synchronous write frequently updates an intent log (IL) block with a small amount of buffered data before the write cache becomes full. Frequent updates of the intent log with a small amount of data degrade the write performance.

The command *sync* by a user or writing a commit block in a journaling file system forces a new intent log to be recorded even though the write cache is not full.

A write with the O_DIRECT option or updating the superblock of a file system generates a synchronous write, which must finish recording data to a reliable medium with a short latency without a write-back policy.

A database transaction is a typical application that produces frequent synchronous writes. A synchronous write involves a new IL update, thereby severely degrading performance.

3.3.2. Solution: skip logging

This paper includes a scheme that significantly reduces the frequency of IL updates. The main idea of the scheme is that the current IL for WC_D can be exempt from being logged to a disk if the current IL is a subset of the previously recorded IL.

If the previous IL is a superset of many subsequent ILs, it can dramatically reduce the frequency of IL records. This paper introduces a merged IL that can be a superset of the future ILs.

Fig. 8 illustrates an example of merged ILs that reduce IL records. (1) Immediately after swapping WC_D and WC_B, the system creates a current IL that includes the positions of the stripes buffered in WC_D. (2) It creates a merged IL by merging the current IL and the previous merged IL. The previous merged IL is the merged IL of the previous stage.

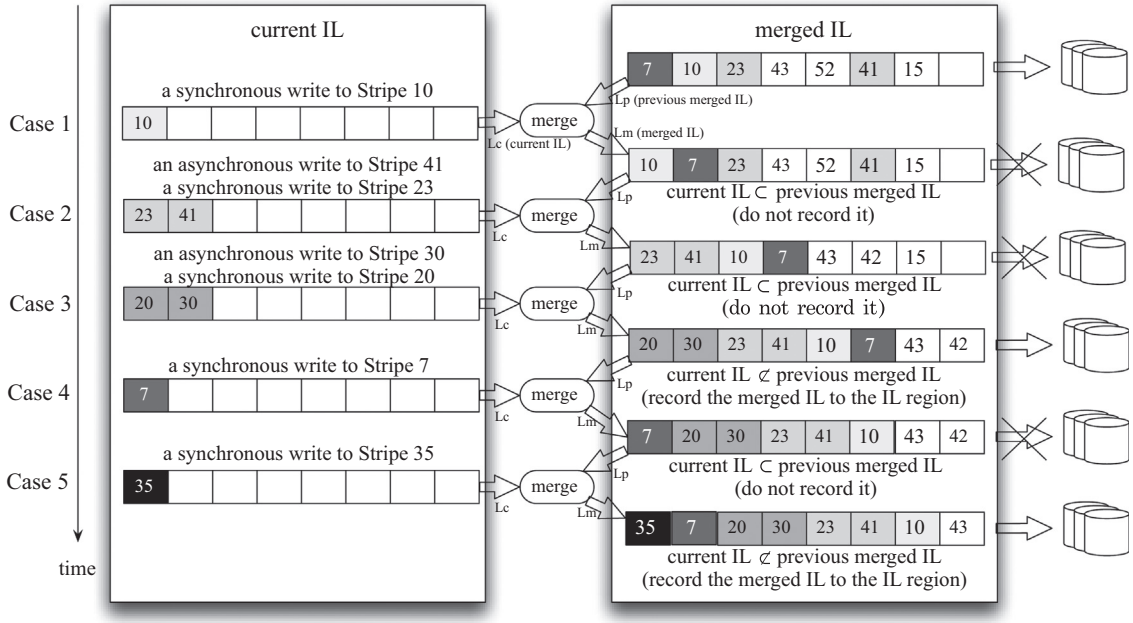


Fig. 8. The union of the previous IL and the current IL creates the merged IL. If the current IL is a subset of the previous IL, no IL is recorded so as to provide performance improvement. Finally, the merged IL replaces the previous IBL for the next stage.

Hence, the merged IL is a union of the past ILs and can be a superset of the future data.

If the current IL is a subset of the previous merged IL, we can skip recording the current IL (Case 1, Case 2, and Case 4). Otherwise, the merged IL is journaled to a disk (Case 3 and Case 5). Finally, the merged IL is used as the previous merged IL in the next stage.

Because the capacity of the IL is limited, the union of the previous merged IL and the current IL may exceed the limited capacity. We therefore have to decide which elements should be excluded from the merged IL. Our scheme adopts the LRU-like policy, which chooses the least recent stripe as a victim.

The procedure to make the merged IL is as follows: (Step 1) the current IL is created from WC_D. (Step 2) The merged IL is initialized from the current IL, which informs the data location of the current WC_D. (Step 3) For each element of the previous merged IL, the element is appended to the merged IL if the element is not in the current IL. If the merged IL is full, this iteration stops. (Step 4) If the current IL is not a subset of the previous merged IL, the merged IL is recorded in a disk. (Step 5) The merged IL is used as the previous merged IL in the next procedure.

In Case 1 of Fig. 8, the storage system receives a synchronous write to Stripe 10, which forces the system to immediately deliver the write data to the disks. If only Stripe 10 is in the write cache, the current IL includes only '10'. The stripe number '10' is in the previous merged IL. Hence, the system skips recording the IL.

In Case 2, Stripe 41 is buffered to the write cache and a synchronous write to Stripe 23 is delivered. The system decides to flush Stripe 41 and Stripe 23 to disks and makes the current IL with '41' and '23', which are in the previous merged IL. Hence, the system skips recording the IL.

In Case 3, the current IL includes '20' and '30', which are not in the previous merged IL. Hence, the system must record the merged IL. The size of the union of the current IL and the previous merged IL exceeds the maximum size of the IL. The system then evicts the least recently used stripe number '15' from the merged IL.

3.3.3. *Properties*

Temporal locality: The merged IL includes the locations of many past writes, and as such it may include future writes if the writes exhibit temporal locality. Therefore, it can dramatically reduce the overhead of the scheme.

Reading intent logs: There is no read operation from disks in normal operations. We maintain a copy of the previous merged IL in a main memory. Reading intent logs from disks is required only in the case of an unclean restart after a crash.

3.4. *Pseudo code*

Fig. 9 shows the pseudo code of the proposed scheme. The destaging function, raid_destage() as a thread, evicts dirty data in WC_D to disks (Line 19). If WC_D is empty and WC_B is not empty (Line 15), WC_D is swapped with WC_B (Line 16) and a new intent log for the new WC_D is built and recorded to disks (Line 17).

The function, raid_write() processes a new write request that is delivered from the host (Line 25), buffers it to the dual write caches (Lines 27–35), and carries out the remaining write operation (Line 36).

When a host delivers a write request to the storage system, the storage system determines if the requested block is found in WC_D (Line 29), removes it from WC_D (Line 30), and adds it to WC_B (Line 35). Otherwise, the system

```
01: write_cache *WC_B, *WC_D, WC_1, WC_2
02: boolean run ← true
03: ordered_list 𝕃_p // the previous IL

04: swap_write_caches()
05: begin
06:   write_cache *temp
07:   temp ← WC_B
08:   WC_B ← WC_D
09:   WC_D ← temp
10: end

11: // A destage thread.
12: raid_destage()
13: begin
14:   while (run)
15:     if WC_D is empty and WC_B is not empty then
16:       swap_write_caches ()
17:       make_record_il()
18:     end if
19:     Destage a stripe cache entry in WC_D
20:     Wait for a new request or a completion of a destaging
21:   end while
22: end

24: // It processes a write request
25: raid_write(write_request *wr)
26: begin
27:   find a stripe cache entry in the write caches for the request
    wr.
28:   if the stripe cache entry is found in a cache then
29:     if the stripe cache entry is in WC_D then
30:       Remove the stripe cache entry from WC_D //promotion hit
31:     end if
32:   else
33:     Allocate a new stripe cache entry. Where, if no free memory
    is left, wake up the destage kernel thread to get a free stripe cache
    entry, and wait until there is a free memory.
34:   end if
35:   Add the stripe cache entry to WC_B
36:   ...
37: end

38: make_record_il()
39: begin
40:   ordered_list 𝕃_c, 𝕃_m // the current IL and the merged IL
41:   for each stripe s ∈ WC_D
42:     append the location of s into 𝕃_c
43:   end for
44:   copy all elements of 𝕃_c into 𝕃_m
45:   for each stripe location p ∈ 𝕃_p
46:     if p ∈ 𝕃_c
47:       continue
48:     if |𝕃_m| ≥ MAX
49:       break
50:     append p into 𝕃_m
51:   end for
52:   if |𝕃_c| ⊈ |𝕃_p|
53:     record 𝕃_m
54:   set 𝕃_p to 𝕃_m
55: end
```

Fig. 9. The Pseudo code.

allocates a new strip cache entry for the write request (Line 33) and adds it to WC_B.

The function make_record_il() builds a merged IL and records it under a condition. The current IL $\mathbb{L}_c$ is initialized with WC_D (Lines 41 and 42). The merged IL $\mathbb{L}_m$ is initialized from $\mathbb{L}_c$ (Line 44). For each element in the previous IL $\mathbb{L}_p$ (Line 45), the element is appended to $\mathbb{L}_m$ (Line 50) if the element is not in $\mathbb{L}_c$ (Line 46) and the size of $\mathbb{L}_m$ does not reach the maximum size (Line 48). Under the condition that $\mathbb{L}_c$ is not a subset of $\mathbb{L}_p$ (Line 52), the system journals $\mathbb{L}_m$ to a designated IL region. Finally, $\mathbb{L}_p$ is replaced with $\mathbb{L}_m$ for the next processing.

3.5. Consistency in the volatile dual write caches

The proposed dual write caches (WC_B and WC_D) are a kind of volatile memory, which may lose data at a power failure. However, a write barrier can lend the reliability to the volatile dual write caches. Journaling file systems (EXT3 [33], EXT4 [36], NTFS [37], and XFS [34]) support the write barrier [38].

A storage device that employs a volatile write cache must properly process write barriers for the file system to maintain its consistency.

The journaled mode of a journaling file system proceeds as follows:

1. It writes a descriptor block, metadata blocks, and data blocks to a journaling region
2. It issues a flush.
3. It appends a commit block to a journal with the write barrier.
4. It writes metadata blocks and data blocks to the home.
5. It issues a flush.

Journaling file systems guarantee transactions that have a commit block. If a descriptor block or metadata is found without a commit block, journaling file systems abandon the last incomplete transaction and roll back to the previous complete transaction.

The write barrier guarantees that all buffered blocks of a transaction are flushed to a magnetic medium before the transaction records its commit block. In other words, a transaction with a commit block ensures that all of the data in the transaction are safely recorded even if the storage system has a volatile write cache.

Linux employs a write barrier to record a commit block [39]. The write barrier with an I/O request forces the write cache to synchronize with disks. The I/O request is then delivered to a disk immediately following the cache synchronization. All subsequent requests are blocked until the I/O request is completed.

Recording a commit block with the write barrier enables journaling systems to employ a volatile write cache without inconsistency of the file systems. Hence, the proposed dual write cache can be implemented without any reliable power source.

3.6. Recovery in a journaling file system

Generally, volatile write caches may cause data losses. However, a system that follows the technique shown in Fig. 10 can employs a volatile write cache while retaining file system consistency. Fig. 10 illustrates the steps of a

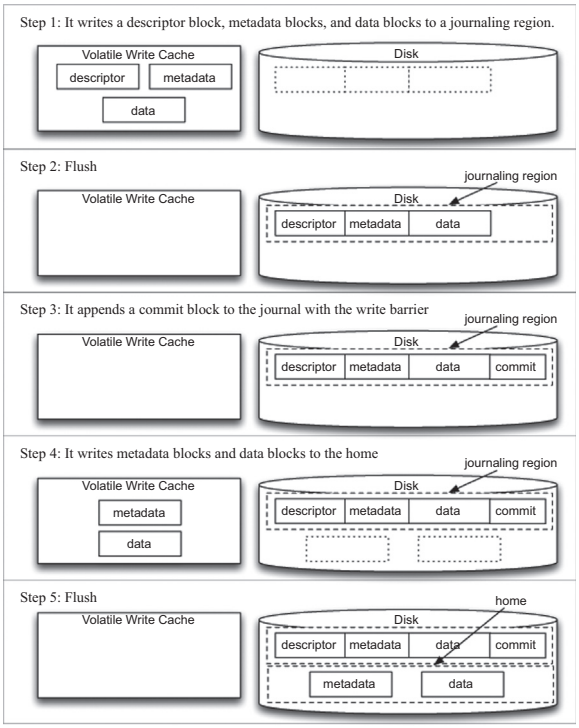


Fig. 10. The steps of a journaling file system to consistently save data blocks and metadata blocks into a storage system that employs a volatile write cache.

journaling file system to consistently save data blocks and metadata blocks into a storage system that employs a volatile write cache.

If a crash occurs during Step 1 of Fig. 10, the storage system will lose the blocks that are destined for a journal. However, we can discard the blocks because the blocks are not for the home.

If a crash occurs during Step 2, the file system discovers an incomplete transaction that has no commit block in the next restart, and it discards the incomplete journal. If a crash occurs during Step 3, Step 4, or Step 5, the file system discover a complete transaction in the next restart and it copies metadata blocks and data blocks in the journal to their home. Therefore, the file system guarantees that the blocks written to their home are not lost.

The ext3 ordered mode is designed not to guarantee data consistency but it guarantees metadata consistency and the order between metadata blocks and data blocks. Even if the ext3 ordered mode is applied to a volatile write cache, the system can ensure the requirements of the ext3 ordered mode.

3.7. Accumulated intent bitmap: applying the intent bitmap to the dual write caches

The intent bitmap scheme (discussed in Section 2.2.3) is implemented in MD (a software RAID in Linux), which cannot employ the intent log scheme because it does not support any write cache. However, our new software RAID with a dual write cache can include both the intent bitmap and the intent log.

Both the intent log and the intent bitmap can describe the locations of data. Each bit of the intent bitmap represents a stripe or a group of blocks while each entry of the intent log indicates a data address. Hence, the intent log can be replaced with an intent bitmap in the dual write cache system.

We can update the changed parts of the intent bitmap instead of recording an intent log. Thanks to the dual write cache, a variation of the intent bitmap, the accumulated intent bitmap scheme, accumulates set bits and updates the changed bits at once instead of updating a bit at every write. The task of clearing bits is postponed until the number of set bits exceeds a predefined number.

The accumulated intent bitmap that is combined in the dual write cache has the following rules: The accumulated intent bitmap

1. does not clear any bit to minimize the changed parts of the bitmap until the number of set bits exceeds a threshold value;
2. clears all bits if the number of set bits exceeds the threshold value;
3. sets bits that correspond to the cached stripes in WC_B, and updates only the changed sectors in the bitmap area before swapping WC_D and WC_B; and
4. thereby, accumulates set bits to reduce the changes of the bitmap.

However, the accumulated intent bitmap has different performance issues compared to the merged intent log:

- Changed blocks in the intent bitmap are scattered and produce multiple discontiguous writes in the intent bitmap area, whereas the intent log requires only a single contiguous write.
- The intent bitmap requires bigger space than the intent log. The bitmap size of the intent bitmap scheme is proportional to the storage capacity while the intent log requires a fixed area size. For a 10TiB storage system with a 512KiB stripe size, the intent bitmap requires 2.4MiB for each bit to represent a stripe, whereas the intent log requires 128KiB for the best performance in an experiment (described in Section 4.8). For the worst case, the intent bitmap may update the entire bitmap space, thereby taking longer time. (Even though a single bit is changed, a sector that can represent tens of thousands of bits should be updated.)

In the intent bitmap that is implemented in a write-through cache, bits are frequently set and cleared, whereas the accumulated intent bitmap that is applied to the dual write cache aggregates bit changes and updates changed sectors at once. The accumulated intent bitmap, however, suffers from discontiguous writes on a much bigger space, and thereby is slower than the merged intent log.

4. Experimental results

The experiments exclude the declared mode, discussed in Section 2.2.6 among prior works, because it cannot

serve a volume drive and requires a dedicated file system. For a fair comparison, we compare the merged intent log (IL) scheme with the intent bitmap (IB) because neither modifies file systems.

In addition, we evaluated the performance improvement of the dual write cache (DWC) when IB is applied to DWC in comparison with IB in a write-through policy.

4.1. Experimental setup

We implemented the proposed merged IL scheme with the dual write caches in a RAID driver, LORE, which has been introduced in several previous articles [40–42]. The RAID driver which is implemented as a kernel module in Linux kernel 2.6.35.11 x86_64, similar to Multiple Devices (MDs) [43], which is another RAID driver in Linux.

The testbed uses a 3.2 GHz i7 processor, 2 GBytes of main memory, and five 7200 rpm SATA3 2 TB hard disk drives (ST2000DM001). The five HDDs are configured as a RAID level 5 array using LORE or MD, which hosts the ext4 file system.

LORE includes DWC, basic IL, merged IL, and accumulated IB. MD employs IB with a write-through (WT) policy. In all figures of this section, 'DWC', 'DWC + accum.IB', 'DWC + basic IL', and 'DWC + merged IL' are evaluated by LORE . 'WT' and 'WT + IB' are evaluated by MD. All graphs plot the average of six experiments.

4.2. Dual write cache vs. write through (LORE vs. MD)

The proposed merged IL scheme requires the dual write cache that is implemented in LORE, whereas MD has no write cache and IL cannot be applied to MD. We implemented the accumulated IB with DWC in LORE, and evaluated the accumulated IB with DWC and IB with a write-through policy using MD.

LORE records an intent log or an intent bitmap in the disk array. In our experiments, MD use the disk array to record IB by option '-binternal'.

LORE and MD shows different performance in various workloads. However, for synchronous writes, LORE and MD show similar performance because the write cache has no beneficial effect on synchronous writes. Fig. 11 compares LORE and MD with synchronous writes.

Both LORE and MD are efficient implementations. Fig. 11 compares the basic performance of LORE and MD. The *y*-axis is the relative performance of LORE over MD (the bandwidth of LORE over the bandwidth of MD). In Fig. 11, Financial2 is the financial2 trace of SPC [44]. PC1, PC2, PC3, and PC4 are four difference traces that are retrieved from personal computers using the XPerf tool. The left bars (synchronous write) show the relative performance when we artificially call fsync() for all writes of the traces. The right bars (async. & sync & flush) show the relative performance when we replay writes of the traces as they are (without any artificial flush call, the trace includes actual flush calls).

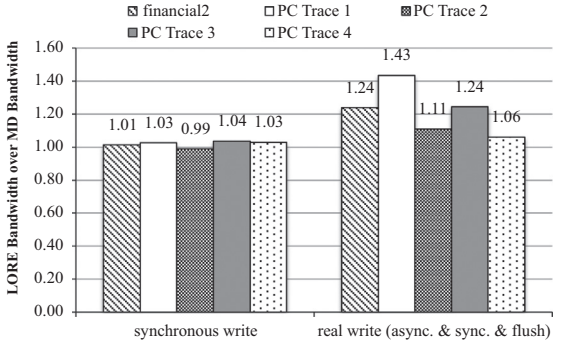


Fig. 11. Performance comparison between LORE (DWC) and MD (write-through).

The left bars of Fig. 11 are close to 1.0. It means that DWC (LORE) and WT (MD) have similar performance for synchronous writes that have no effect on write caches. The right bars indicate that DWC shows better performance than WT for the mixed I/Os (asynchronous writes + synchronous writes + flushes) because DWC of LORE produces more reconstruct writes and full stripe writes than the write through policy of MD.

DWC as a write cache is beneficial to RAID systems. In addition, DWC improves the performance of IB. The following evaluations show the performance improvement of IB by comparing 'DWC + accum. IB' (LORE) with 'WT + IB' (MD).

4.3. Micro benchmarks

This section evaluates the proposed schemes with random write and sequential write. First, we test the performance of random writes on a 500GB storage, as

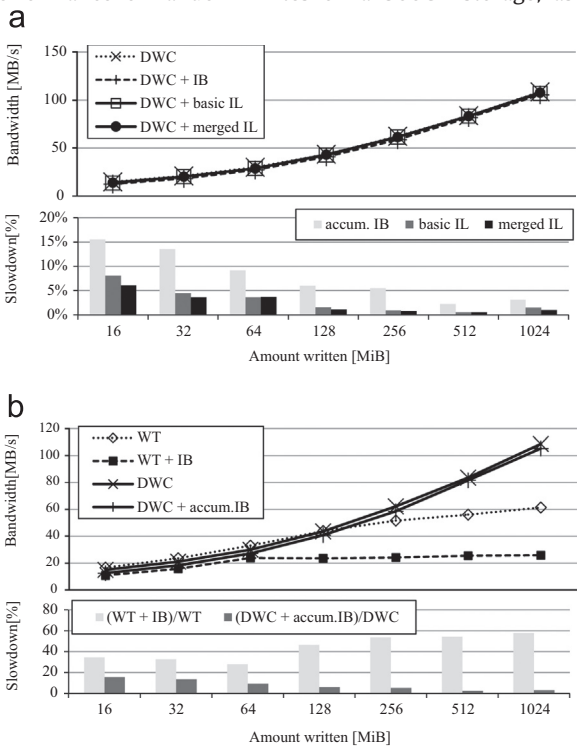


Fig. 12. Random write: the top graph plots random write performance as the amount of data written is increased along the *x*-axis. The bottom graph shows the slowdown. (a) Accumulated IB vs. basic IL vs. merged IL with DWC and (b) WT vs. DWC.

shown in Fig. 12. All blocks are flushed out to disks by sync () at the end of each experiment.

The top graph plots the random write performance as the amount of data written increases along the *x*-axis. The bottom graph shows each relative slowdown.

In the bottom graph of Fig. 12(a), the labels 'accum. IB', 'basic IL', and 'merged IL' are obtained from'DWC + accum. IB' against 'DWC', 'DWC + basic IL' against 'DWC', and 'DWC + merged IL' against 'DWC', respectively.

The random writes cause the accumulated IB to produce scattered bit changes and discontiguous writes in the bitmap, whereas the merged IL always produces a single log regardless of I/O patterns. For the random writes, the merged IL shows the best performance (1–6% slowdown) and the accumulated IB shows the worst performance (3–16% slowdown).

Fig. 12(b) compares WT and DWC with random writes. In the bottom graph of Fig. 12(b), the labels 'WT+IB/WT' and 'DWC+accum.IB/DWC' denote the relative degradations of 'IB' and 'accum. IB', respectively. The slowdowns of 'IB', and 'accum. IB' range from 47% to 59% and 4% to 11%, respectively. WT makes IB frequently update the bitmap, whereas DWC makes IB accumulate the bit changes, thereby dramatically reducing the frequency of the bitmap update.

Fig. 13 shows the sequential write performance on an ext4 file system as the amount of data written increases along the *x*-axis. The page cache is synchronized at the end of the experiment by a sync() call.

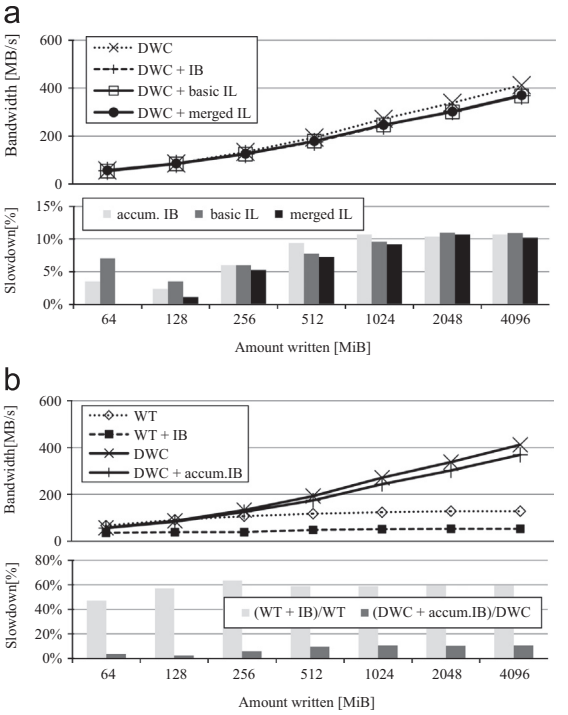


Fig. 13. Sequential write: the top graph plots sequential write performance as the amount of data written is increased along the *x*-axis. The bottom graph shows the slowdown of IB and IL. (a) Accumulated IB vs. basic IL vs. merged IL with DWC and (b) WT vs. DWC.

Fig. 13(a) compares 'accum. IB', 'basic IL', and 'merged IL' with 'DWC'. The three schemes show nearly equivalent performance for the sequential write.

Fig. 13(b) compares WT and DWC with sequential writes. DWC significantly outperforms WT by 3.2 times in the best case. The accumulated IB exhibits a much smaller slowdown (10%) than IB (60% slowdown).

4.4. Macro benchmarks: Filebench

For more realistic workloads, we chose the 'File server' workload of Filebench version 1.4.8, which creates, appends, reads, and deletes files with multiple threads [45]. Figs. 14 and 15 show the results of Filebench with the File server workload as the average file size increases along the *x*-axis.

4.4.1. Accumulated IB vs. basic IL vs. merged IL

Figs. 14(a) and 15(a) show the performance of 'accum. IB', 'basic IL', and 'merged IL' that are integrated with DWC.

The basic IL must record the log at every swap of WC_B and WC_D, whereas the accumulated IB and the merged IL have many opportunities to skip updating a bitmap block or a log block. Therefore, 'basic IL' shows the worst throughput.

The merged IL is better than the accumulated IB from the standpoint of throughput and latency, as shown in

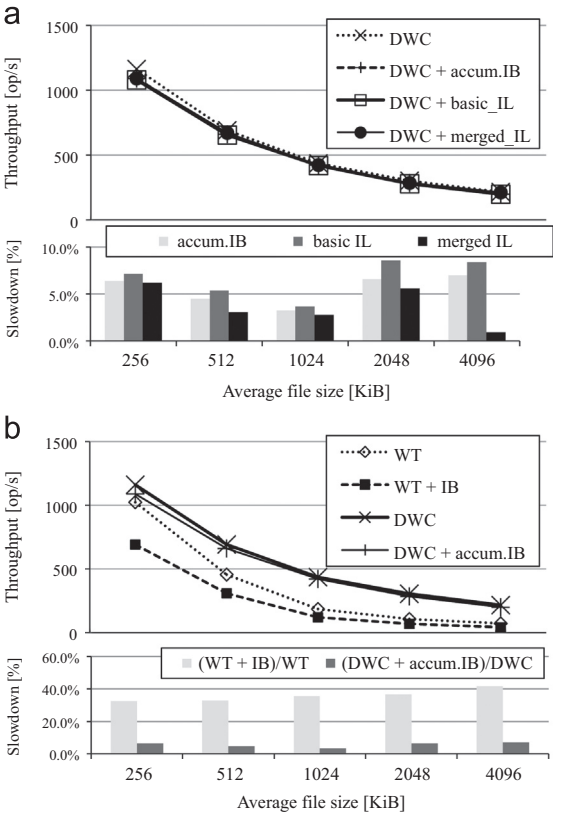


Fig. 14. Filebench throughput: the top graph shows the throughput of Filebench as the size of files served by a file server increases along the *x*-axis. The bottom graph shows the relative throughput. (a) Accumulated IB vs. basic IL vs. merged IL with DWC and (b) WT vs. DWC.

Figs. 14(a) and 15(a). The accumulated IB produces discontiguous writes on a much bigger space than IL, whereas the merged IL produces a contiguous write on a relatively smaller size of the log area.

4.4.2. DWC vs. WT

Figs. 14(b) and 15(b) compare the overhead of IB and accumulated IB. The label '(WT+IB)/WT' denotes the overhead of 'WT+IB' over 'WT'. The label '(DWC+accum. IB)/DWC' is the overhead of 'DWC+accum.IB' over 'DWC'

We observe that the write through cache with IB has a significant slowdown (30–40%), but Filebench utilizes the write cache and enables DWC to aggregate many bit changes of the bitmap. Accumulated IB assisted by DWC shows 2–7% overhead in this experiment.

4.5. Macro benchmarks: Postmark

Fig. 16 shows the execution time of the PostMark benchmark as the number of transactions increases along the *x*-axis. PostMark is a benchmark that creates a large pool of continually changing files and measures the transaction rates for a workload approximating a large Internet electronic mail server [46].

Postmark produces scattered writes over a wide range of space. Hence, the accumulated IB produces scattered bit changes and discontiguous writes to update the bitmap. In this experiment, the basic IL is better than the accumulated IB. The merged IL shows the best performance.

Fig. 16(b) shows that a transactional I/O, Postmark, is beneficial to DWC against WT. 'DWC+accum.IB' is five to six times faster than 'WT+IB'. The accumulated IB has less overhead than IB. The slowdown of 'accum. IB" ranges from 4 to 17% while the slowdown of 'IB' ranges from 22 to 27%. The considerable gap between DWC and WT is due to their different write policies.

4.6. Miscellaneous benchmarks

Fig. 17 compares the normalized execution time of building an ext4 file system (mkfs.ext4), unpacking a Linux kernel source package (unpack), and replaying a TPC-C I/O trace [47] as compared to 'WT', which scans the entire volume for resynchronization.

Adding the IB scheme to WT incurs 200%, 92%, and 51% degradation for mkfs.ext4, unpack, and TPC-C, respectively, whereas the accumulated IB scheme exhibits 11%, 3%, and 1% overhead, respectively.

The bottom graph of Fig. 17 shows the relative degradation of 'DWC+accum IB', 'DWC+basic IL', and 'DWC+merged IL' as compared to 'DWC'. All three schemes show small overheads but the merged IL exhibits the smallest overhead.

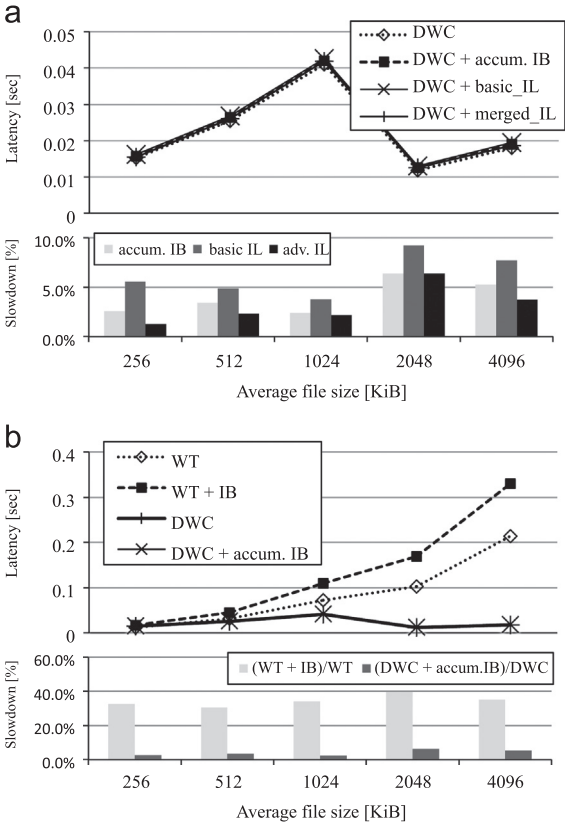


Fig. 15. Filebench latency: the top graph shows the latency of Filebench. The bottom graph shows the relative latency. (a) Accumulated IB vs. basic IL vs. merged IL with DWC and (b) WT vs. DWC.

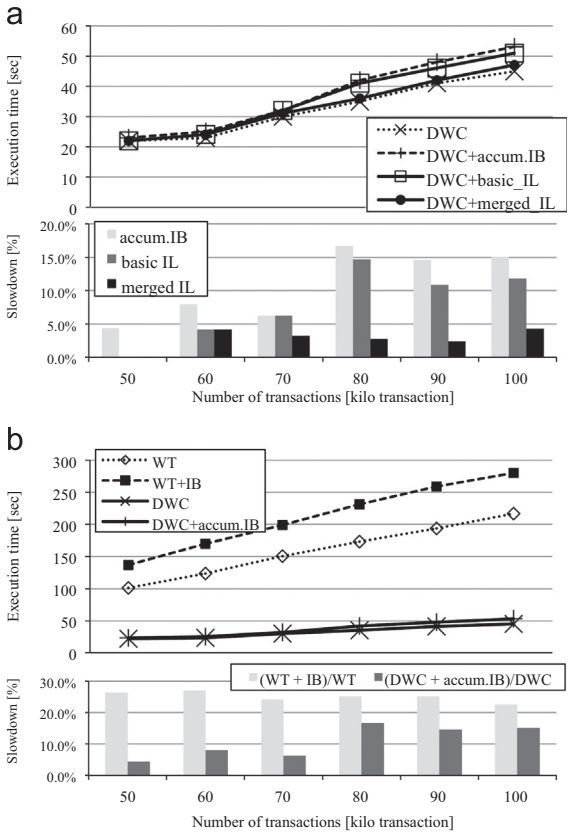


Fig. 16. PostMark benchmark: the top graph shows the execution time of PostMark as the number of transactions increases along the *x*-axis. The bottom graph shows the relative slowdown. (a) Accumulated IB vs. basic IL vs. merged IL with DWC and (b) WT vs. DWC.

Table 1 compares the resynchronization time to find and fix inconsistent stripes after an unclean shutdown with the entire scan and the IL. The conventional method requires 200 min to resynchronize the 4TB volume. It is very inconvenient for consumers to wait for the long resynchronization to be completed. The proposed scheme, the merged IL with DWC, shortens the resynchronization process from 200 min to 30 s in the worst case.

The resynchronization time of the IL scheme is bound to a limited time that is determined by the write cache size and the maximum size of the intent log. The resynchronization time of the IB scheme is comparable to that of the IL scheme.

The resynchronization time of the IB scheme can be bound to a limited time if the number of set bits is restricted. Both the IB and IL schemes exhibit a short resynchronization process and their resynchronization time is tunable according to their configuration.

4.7. Synchronous writes

Synchronous writes with O_DIRECT or sync() calls cause frequent IL updates. It is possible to reduce the IL updates by merging ILs as discussed in Section 3.3. The merging policy can be immune to performance degradation by synchronous writes.

Fig. 18 shows a comparison for excessive sync calls. In this experiment, artificial sync() calls are inserted between the writes of the SPC Financial1 trace [44]. The axis indicates the number of writes per sync() call.

For very frequent cache synchronization (one to two of sync interval), the accumulated IB causes a single or two bit changes, which may require updating of a single sector, thereby resulting in the smallest slowdown.

However, for longer sync intervals (4–16), the accumulated IB may need two or more scattered writes and is outperformed by the merged IL. The basic IL records a log at every cache synchronization and causes severe performance degradation due to frequent synchronization. The merged IL records a single or no log, and thus exhibits much less slowdown.

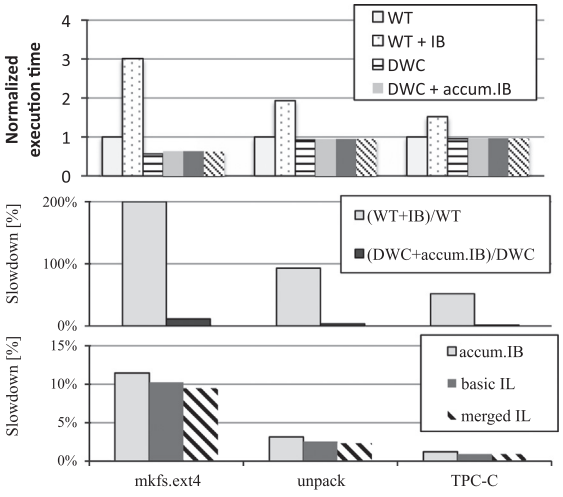


Fig. 17. The figure compares the normalized execution time of building an ext4 file system (mkfs.ext4), unpacking a kernel source package (unpack), and replaying a TPC-C I/O trace (TPC-C) as compared to 'WT'.

Table 1
Resynchronization time.

	Full scan	IL
Resynchronization time	200 min	Less than 30 s

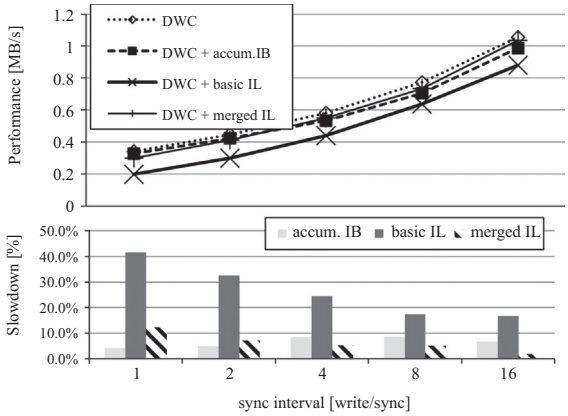


Fig. 18. Synchronization performance: the graph plots the performance for frequent sync() calls when sync() is artificially inserted between the writes of the SPC Financial1 trace. The axis indicates the number of writes per sync() call.

4.8. The size of the IL region

This section evaluates the effect of the size of the IL region. If the IL size becomes larger, the merging efficiency becomes higher and there are more opportunities to skip recording ILs but it takes longer time to save a larger IL.

The minimum number of entries of the IL region must be at least the maximum number of stripes that the write cache can accommodate. LORE is configured to cache at most 3236 512KiB stripes with 256MiB memory in the experiments. The IL consists of a 24B header and 8B entries to represent stripe numbers. The 32KiB IL region can contain 4093 entries and is used in default.

Fig. 19(a) shows the results of Filebench with the File server profile as the size of the IL region increases along the *x*-axis. We performed five trials, which show that 64KiB is optimal for the IL region. This means that there exists a proper size for the IL region.

Fig. 19(b) shows a comparison with synchronous writes for various sizes of the IL region. In this experiment, the SPC Financial2 trace [44] is replayed as synchronous writes. The *x*-axis indicates the size of the IL region, which ranges from 32KiB to 512KiB.

The IL size of 128KiB provides the best performance in this experiment but 64KiB is the best in Filebench. We thus find that the optimal IL size varies from workload to workload.

5. Conclusion

The proposed scheme targets the software RAID, which has no power-fail-safe component. The critical problem of

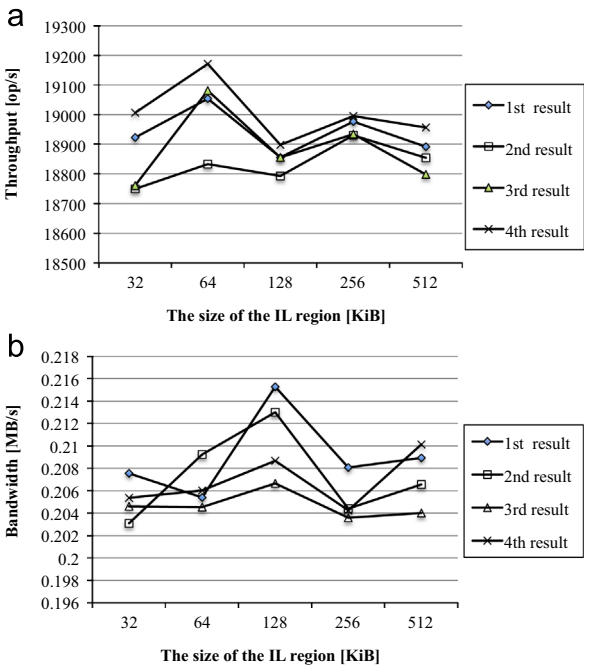


Fig. 19. Performance for various IL region size. (a) Filebench for various sizes of the IL region. (b) Synchronous writes using the SPC Financial2 trace for various sizes of the IL region.

the software RAID is intolerably long resynchronization time or expensive write overhead. The solution of scanning the entire volume exhibits good performance in normal operation but it forces users to wait during several hours of resynchronization time as a penalty for a careless shutdown.

We introduced a new seamless block-level journaling with the dual write cache, which aggregates the intents of multiple writes into an intent log or bit changes of the intent bitmap, thereby reducing overhead. We presented how the dual write cache can be applied to the conventional intent bitmap and evaluated its performance improvement. In addition, we proposed the basic intent log and the merged intent log, which are optimized to the dual write cache.

The merged intent log scheme is the best among the three proposed schemes. The accumulated intent bitmap produces scattered bitmap changes, whereas the merged intent log scheme requires a single contiguous write for a log, can skip recording a log, and reduces performance degradation for frequent synchronization.

The merged IL scheme with the dual write cache is transparent to file systems, has a short resynchronization time, and exhibits negligible performance degradation. The proposed scheme provides a breakthrough to solve the major problem of the software RAID.

Acknowledgments

This work was supported by the Jungwon University Research Grants.

References

[1] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson, RAID: high-performance, reliable secondary storage, ACM Comput. Surv. 26 (2) (1994) 145–185.
[2] S.H. Kim, B. Yu, J.-y. Chang, Zoned-partitioning of tree-like access methods, Inf. Syst. 33 (3) (2008) 315–331.
[3] M.-K. Seo, S.H. Baek, K.H. Park, Arrangement of multi-dimensional scalable video data for heterogeneous clients, Inf. Syst. 35 (2) (2010) 237–259.
[4] M. Holland, On-line data reconstruction in redundant disk arrays (Ph.D. thesis), Carnegie Mellon University, April 1994.
[5] R. Hour, J. Menon, Y. Patt, Balancing i/o response time and disk rebuild time in a raid5 disk array, In: Proceedings of the 26th Hawaii International Conference on System Sciences, vol. 1, 1993, pp. 70–79.
[6] J. Lee, J. Lui, Automatic recovery from disk failure in continuous-media servers, IEEE Trans. Parallel Distrib. Syst. 13 (5) (2002) 499–515.
[7] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, Z. Song, PROC: a popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems, in: Proceedings of the Fifth USENIX Conference on File and Storage Technologies, 2007, pp. 277–290.
[8] I. Iliadis, R. Haas, X.-Y. Hu, E. Eleftheriou, Disk scrubbing versus intra-disk redundancy for high-reliability raid storage system, in: Proceedings of International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS 2008), ACM SIGMETRICS Performance Evaluation Review, 2008, pp. 241–252.
[9] A. Oprea, A. Juels, A clean-slate look at disk scrubbing, in: Proceedings of the Conference on File and Storage Technologies (FAST 2010), 2010, pp. 57–70.
[10] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, S. Ng, Disk scrubbing in large archival storage systems, in: Proceedings of the IEEE International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), 2004, pp. 409–418.
[11] G. Zhang, W. Zheng, K. Li, Rethinking raid-5 data layout for better scalability, IEEE Trans. Comput. 61 (11) (2012) 1–13.
[12] G. Zhang, W. Zheng, J. Shu, ALV: a new data redistribution approach to raid-5 scaling, IEEE Trans. Comput. 59 (3) (2010) 345–357.
[13] Y. Saito, S. Frølund, A. Veitch, A. Merchant, S. Spence, FAB: building distributed enterprise disk arrays from commodity components, in: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004, pp. 48–58.
[14] H. Zhao, Y. Xu, L. Xiang, Scaling up of e-msr codes based distributed storage systems with fixed number of redundancy nodes, in: Proceedings of the International Journal of Distributed and Parallel Systems 3 (2012), 1–12.
[15] W. Zheng, G. Zhang, Fastscale: Accelerate raid scaling by minimizing data migration, in: Proceedings of The Ninth USENIX Conference on File and Storage Technologies, 2011, pp. 149–161.
[16] K. Hwang, H. Jin, R.S. Ho, Orthogonal striping and mirroring in distributed RAID for I/O-centric cluster computing, IEEE Trans. Parallel Distrib. Syst. 13 (1) (2002) 26–44.
[17] C.-I. Park, Efficient placement of parity and data to tolerate two disk failures in disk array systems, IEEE Trans. Parallel Distrib. Syst. 6 (11) (1995) 1177–1184.
[18] M. Blaum, J. Brady, J. Menon, EVENODD: an efficient scheme for tolerating double disk failures in raid architectures, IEEE Trans. Comput. 44 (2) (1995) 192–201.
[19] S.H. Kim, H. Zhu, R. Zimmermann, Zoned-RAID, ACM Trans. Storage 3 (2007) (1).
[20] J. Wikes, R. Golding, C. Staelin, T. Sullivan, The HP AutoRAID hierarchical storage system, ACM Trans. Comput. Syst. 14 (1) (1996) 108–136.
[21] S.-H. Lim, Y.-W. Jeong, K.H. Park, Interactive media server with media synchronized raid storage system (nossdav), in: Proceedings of the International Workshop on Network and Operating System Support for Digital Audio Video, 2005, pp. 177–182.
[22] C. Wu, X. He, G. Wu, S. Wan, X. Liu, Q. Cao, C. Xie, Hdp code: A horizontal-diagonal parity code to optimize i/o load balancing in raid-6, in: 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), IEEE, 2011, pp. 209–220.
[23] Z. Shen, J. Shu, Hv code: An all-around mds code to improve efficiency and reliability of raid-6 systems, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2014, pp. 550–561.

[24] G. Zhang, J. Shu, W. Xue, W. Zheng, SLAS: an efficient approach to scaling round-robin striped volumes, ACM Trans. Storage 3 (2007) (1).
[25] C. Weddle, M. Oldham, J. Qian, A.-I.A. Wang, G. Kuenning, PARAID: A gear-shifting power-aware RAID, in: Proceedings of the Fifth USENIX Conference on File and Storage Technologies, 2007, pp. 245–260.
[26] D. Kenchammana-Hosekote, D. He, J.L. Hafner, REO: A generic RAID engine and optimizer, in: Proceedings of the Fifth USENIX Conference on File and Storage Technologies, 2007, pp. 261–276.
[27] D. Teigland, H. Mauelshagen, Volume managers in Linux, in: Proceedings of the USENIX Annual Technical Conference, 2002, pp. 185–197.
[28] P. Clements, J. Bottomley, High availability data replication, in: Proceedings of the 2003 Linux Symposium, 2003.
[29] T.E. Denehy, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Journal-guided resynchronization for software RAID, in: Proceedings of the Fourth USENIX Conference on File and Storage Technologies, 2005.
[30] J. Menon, J. Cortney, The architecture of a fault-tolerant cached RAID controller, in: Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993, pp. 76–87.
[31] M. Rosenblum, J. Ousterhout, The design and implementation of a log-structured file system, ACM Trans. Comput. Syst. (1992) 10 (1).
[32] J. Piernas, T. Cortes, J.M. Garca, DualFS: a new journaling file system without meta-data duplication, in: The 16th International Conference on Supercomputing, 2002, pp. 137–146.
[33] S. Tweedie, EXT3, journaling filesystem, in: the 2000 Ottawa Linux Symposium, 2000, pp. 24–29.
[34] D. Robbins, Common threads: advanced filesystem implementer's guide. Part 9, introducing XFS, in: Developer Works, IBM, 2002.
[35] R.H. Arpaci-Dusseau, A.C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, 2014.
[36] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, The new EXT4 filesystem: Current status and future plans, in: Proceedings of the 2007 Ottawa Linux Symposium, 2007.
[37] M. Russinvovich, Inside Win2k NTFS, Part 1, Microsoft Developer Network, 2002.
[38] T. Heo, I/O barriers, Linux Kernel Archives, 2005.
[39] J. Bacik, K. Dudka, H. Goede, D. Ledford, D. Novotny, N. Straz, D. Wysochanski, M. Christie, S. Prabhu, R. Evers, D. Howells, D. Lehman, J. Moyer, E. Sandeen, M. Snitzer, Write barriers, in: Red Hat Enterprise Linux 6 Storage Administration Guide, 2011, pp. 149–151.
[40] S.H. Baek, K.H. Park, Striping-aware sequential prefetching for independency and parallelism in disk array with concurrent accesses, IEEE Trans. Comput. (2009) 58 (8).
[41] S.H. Baek, K.H. Park, Matrix-stripe-cache-based contiguity transform for fragmented writes in RAID-5, IEEE Trans. Comput. 56 (8) (2007) 1040–1054.
[42] S.H. Baek, K.H. Park, Prefetching with adaptive cache culling for striped disk arrays, in: Proceedings of the 2008 USENIX Annual Technical Conference, 2008, pp. 363–376.
[43] D. Vadala, Managing RAID on Linux, O'Reilly Media, Inc., 2003.
[44] K. Bates, B. McNutt, S.P. Councile, OLTP application I/O, The UMass Trace Repository.
[45] R. McDougall, J. Mauro, Filebench: File system microbenchmarks, ⟨http://www.opensolaris.org/os/community/performance/filebench⟩, May 2008.
[46] J. Katcher, Postmark: A New File System Benchmark, Technical Report TR3022, Network Appliance. (1997).
[47] SNIA, TPCC Traces 1, SNIA IOTTA Repository (2011).