Ph.D. Dissertation

Providing Crash Consistency along with Efficient Page Cache Management on Flash Storage Devices

Dong Hyun Kang

Department of Electrical and Computer Engineering

The Graduate School

Sungkyunkwan University

Providing Crash Consistency along with Efficient Page Cache Management on Flash Storage Devices

Dong Hyun Kang

Department of Electrical and Computer Engineering

The Graduate School

Sungkyunkwan University

Providing Crash Consistency along with Efficient Page Cache Management on Flash Storage Devices

Dong Hyun Kang

A Dissertation Submitted to the Department of
Electrical and Computer Engineering
and the Graduate School of Sungkyunkwan University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

October 2017

Approved By

Professor Young Ik Eom

This certifies that the dissertation
of Dong Hyun Kang is approved.

Sang-won Lee

Dongkun Shin

Hwansoo Han

Jinkyu Jeong

Major Advisor : Young Ik Eom

The Graduate School
Sungkyunkwan University
December 2017

Contents

List of Figures

List of Tables

Abstract

Providing Crash Consistency along with Efficient Page Cache Management on Flash Storage Devices

A long-standing goal of NAND flash storage devices is completely to replace hard disk drives (HDDs) in computing environments, including mobile, desktop, and enterprise server. To achieve the goal, manufacturers of flash storage are driving their technologies and products toward higher capacity and performance. Such a tremendous growth in storage industry leads for storage researchers to opportunities in redesigning the existing storage stacks that have been designed for the HDDs.

In this dissertation, we explore how the existing storage stacks in operating systems can take advantages of the flash storage devices, and present useful three approaches towards optimizing the software techniques inside the stacks. (1) We introduce the *flash-friendly write pattern* and propose a novel page cache replacement algorithm, called TS-CLOCK, that exploits temporal locality to keep the cache hit ratio high and also exploits spatial locality to maintain evicted writes *flash-friendly*. To confirm the effectiveness of TS-CLOCK on the flash storage devices, we have implemented TS-CLOCK and compared it with seven replacement algorithms; TS-CLOCK showed a better performance than other algorithms. (2) We present a new file system, called *AFS*, that can easily and efficiently guarantee application-level crash consistency with the help of the *SHARE* interface supporting atomic address remapping at the flash storage layer. We have prototyped *AFS* by slightly modifying the full data journal mode in ext4 file system, and carried out various experiments by running *AFS* on top of the SSD. Our evaluation results show that *AFS* considerably

outperforms existing journal modes. We also observed the performance of an OLTP benchmark using MySQL/InnoDB engine can be boosted by more than 2–6x. (3) Finally, we revisit the crash consistency mechanism for ext4 from the point of view of the write pattern and propose a useful journaling mode towards exploiting the compression/decompression technique, called *LDJ*. We have prototyped our *LDJ* by slightly modifying the ext4 with jbd2 for journaling and also e2fsck for recovery. Our evaluation results clearly show that *LDJ* outperforms the default journaling mode by up to 9.2x and 5.9x on the standard benchmark and real application, respectively.

Keywords: NAND Flash Storage Devices, Page Cache Replacement Algorithm, File System, Consistency, Journaling Mechanism, High Performance

1. Introduction

NAND flash storage devices are now prevalent in all classes of computing devices including mobile devices, desktops, and enterprise servers. This may be possible because new storage technologies and devices have been relentlessly developed during the last decade [1–7]. Meanwhile, a lot of efforts in the academia are made to redesign the storage stacks inside the operating systems that have been designed for the traditional hard disk drives (HDDs) [8–12]. For example, many storage researchers introduced the attractive features of the flash storage devices and modified the page cache replacement algorithms [13–19], file systems [20–27], and IO schedulers [28, 29] by closely coupling with the storage devices. It is important to make the existing storage stacks flash-friendly because the flash storage devices take three different characteristics compared to HDDs. First, a special software, called flash transaction layer (FTL), is required to support in-place updates [9, 30–32]. Second, I/O latencies among read and write operations are asymmetric; write operations are slower than read operations [13]. Finally, write performance is highly dependent on *write patterns* and eventually determined by the garbage collection cost of the FTL; sequential writes are several times faster than random writes [9, 10, 23, 26, 33, 34].

1.1 Scope of Dissertation

This dissertation aims at improving the overall performance and endurance of the flash storage devices by redesigning the existing storage stacks. To achieve our goals, we first focus on transforming storage write pattern to sequential write. Sequential write pattern is a well-known property to improve the performance and endurance of the flash storage devices because it significantly reduces the write amplification factor (WAF) than random write pattern during the garbage collection (GC) operation; sequential write pattern shows an ideal WAF, 1, with no extra writes. In section 2.2, we explore the benefits of sequential write pattern and describe the differences between sequential writes and random ones in terms of WAF. We also focus on reducing the total amount of data being written to the flash storage devices. Especially, we explore existing consistency mechanisms that always write data twice to guarantee data durability and consistency.

1.2 Dissertation Topics

In this dissertation, we introduce three novel techniques that come from a comprehensive study of the existing storage stacks and present the effectiveness of each technique on top of various flash storage devices, including eMMCs, SSDs, and NVMe SSDs. The followings are the main topics of this dissertation:

- **TS-CLOCK: An Efficient Page Cache Replacement Algorithm for NAND Flash Storage Devices:** Today, flash storage devices have become a standard storage because of their attractive features. However, they suffer from well-known two problems: first, as flash density increases, the lifetime of the storage medium decreases rapidly. Second, random writes significantly decrease performance and lifetime since they generate more *hidden writes* in-

side NAND flash storage devices. We introduce TS-CLOCK page cache algorithm to address those two problems. TS-CLOCK exploits temporal locality to keep the cache hit ratio high and also exploits spatial locality to maintain evicted writes *flash-friendly*. The key idea of our flash-friendly eviction is that, when evicting a dirty page, TS-CLOCK first selects a flash block with the largest number of dirty pages that are least likely to be accessed and then sequentially evicts pages in the block. Since it generates *pseudo sequential writes* for a flash block, it significantly increases performance and lifetime at once by reducing the number of hidden writes. We have implemented TS-CLOCK and compared it with seven replacement algorithms, including traditional and flash-aware ones, for several real workloads. Our experimental results show that TS-CLOCK outperforms the state-of-the-art replacement algorithm, Sp.Clock, by 30% on the NAND flash storage devices and extends the lifetime by 53%.

- **AFS: Making Application-level Crash Consistency Practical on Flash Storage Devices:** We present the design, implementation, and evaluation of a new file system, called *AFS*, supporting application-level crash consistency as its first-class citizen functionality. With *AFS*, application data can be correctly recovered in the event of system crashes without any complex update protocol at the application level. With the help of the *SHARE* interface supporting atomic address remapping at the flash storage layer, *AFS* can easily and efficiently achieve crash consistency as well as single-write journaling. We prototyped *AFS* by slightly modifying the full data journal mode in ext4, implemented the *SHARE* interface as firmware in a commercial SSD available in the market, and carried out various experiments by running *AFS* on top of the SSD. Our preliminary experimental results are very promising. For instance, the performance of an OLTP benchmark using MySQL/InnoDB engine can be

boosted by more than 2–6x by offloading the responsibility of guaranteeing the atomic write of MySQL data pages from the InnoDB engine's own journaling mechanism to *AFS*. This impressive performance gain is in part due to the single-write journaling in *AFS* and in part comes from the fact that the frequent `fsync()` calls caused by the complex update protocol at the application level can be avoided. *AFS* is a practical solution for the crash consistency problem in that (1) the *SHARE* interface can be, like the TRIM command, easily supported by commercial SSDs, (2) it can be embodied with a minor modification of the existing ext4 file system, and (3) the existing applications can be made crash consistent simply by opening files in `O_ATOMIC` mode while the legacy applications can be run without any change.

- **LDJ: Version Consistency Is Almost Free on Flash Storage Devices:** We propose a simple but practical and efficient optimization scheme for journaling in ext4, called lightweight data journaling (*LDJ*). By compressing journaled data prior to writing, *LDJ* can perform comparable to or even faster than the default ordered journaling (OJ) mode in ext4 on top of flash storage devices, while still guaranteeing the *version consistency* of the data journaling (DJ) mode. This surprising result can be explained with three main reasons. First, on modern storage devices, the sequential write pattern dominating in DJ mode is more and more high-performant than the random one in OJ mode. Second, the compression significantly reduces the amount of journal writes, which will in turn make the write completion faster and prolong the lifespan of the storage. Third, the compression also enables the atomicity of each journal write without issuing an intervening `FLUSH` command between journal data blocks and commit block, thus halving the number of costly `FLUSH` calls in *LDJ*. We have prototyped our *LDJ* by slightly modifying the existing ext4 with jbd2 for journaling and also e2fsck for recovery; less than 300 lines of source code were changed. Also, we carried out

a comprehensive evaluation using four standard benchmarks and two real applications. Our evaluation results clearly show that *LDJ* outperforms the OJ mode by up to 9.2x and 5.9x on the standard benchmark and real application, respectively.

The rest of this dissertation is organized as follows. Chapter 2 presents our first effort (*i.e.*, TS-CLOCK) that focuses how to reshape random writes to sequential ones in the page cache layer. In Chapter 3 and Chapter 4, we comprehensively explore crash consistency mechanisms to efficiently take advantages of the flash storage devices; Chapter 3 presents new file system (*i.e.*, *AFS*) that guarantees application-level crash consistency with *SHARE* interface and Chapter 4 introduce new journaling mode (*i.e.*, *LDJ*) for ext4 file system in detail. Finally, Chapter 5 concludes this dissertation.

2. TS-CLOCK: An Efficient Page Cache Replacement Algorithm for NAND Flash Storage Devices

2.1 Introduction

The OS page cache receives I/O requests directly from applications and transforms the requests into a suitable I/O request stream for storage devices. We focus on the page cache replacement algorithm used by the OS page cache due to the algorithm's potential to shape I/O requests. The primary goal of the page cache replacement algorithm is to ensure a high hit-ratio and to reduce the amount of slow I/O operations. The traditional, but flash agnostic, approaches such as Least Recently Used (LRU), CLOCK [35], and Linux2Q [36] exploit *temporal locality* to predict future accesses of the pages and evict the pages which are the most unlikely to be accessed in the future. Though the majority of page cache replacement schemes are based on temporal locality, there are several schemes which exploit *spatial locality* in the context of HDDs [37, 38] and NAND flash storage devices [13–19]. In flash aware schemes, while much attention has been focused on improving performance, lifespans improvement has not been given much attention, unfortunately. CFLRU [13], LRU-WSR [15], and FOR [16] exploit asymmetric read and write operation times and evict clean pages over dirty pages to reduce the more expensive write operations. However, since they do not consider the write patterns of the evicted pages, the generated write patterns

could induce a large number of hidden writes inside NAND flash storage devices depending on workload characteristics. In contrast, FAB [14], BPLRU [18], and LB-CLOCK [19] give more focus to generating flash-friendly write patterns in order to reduce garbage collection (GC) costs and hidden writes inside NAND flash storage devices. However, since they evict multiple pages in a unit of a flash block, they suffer from the *early eviction problem* [39] – their coarse-grained policies evict pages with high locality and such pages are re-accessed and re-evicted after all– and the lower hit ratio results in more I/O operations and decreased performance and lifetime. Recently, Kim et al. [17] proposed Sp.Clock, which is a variant of the CLOCK algorithm and manages page frames by logical sector number rather than recency order, in a manner similar to WOW [38] in HDDs. The sorted eviction scheme could contribute to lower hidden writes in NAND flash storage devices when the I/O range is narrow. However, it suffers from high GC cost for workloads with wide I/O ranges, because the evicted write patterns become random rather than sequential. Moreover, Sp.Clock does not exploit asymmetric read and write operation times to select victim pages.

In this chapter, we propose a novel page cache placement algorithm, which we named TS-CLOCK (Temporal and Spatial locality aware CLOCK), for NAND flash storage devices. We aim to improve performance and expand lifetime by reducing the number of write operations and generating flash-friendly pseudo sequential write patterns, which minimize hidden writes inside NAND flash storage devices. To this end, TS-CLOCK makes use of *asymmetric read and write operation times* as well as *logical flash block utilization*, which is the number of dirty pages in a logical flash block. Throughout this chapter, we use the term *block* for logical flash block, which is calculated from dividing a logical sector number by the number of pages per flash block, for brevity.

Key features of the TS-CLOCK algorithm are derived from the characterization of write performance varying write patterns in NAND flash storage devices, which is our first contribution. Though several replacement schemes consider write patterns, they largely adopt relatively simple policies such as block-level eviction [14, 18, 19] or sorted eviction [17]. In order to characterize write performance to write patterns, we perform trace-driven simulations for various synthetic write patterns. We observe that, when writing dirty pages in a block by the sector number, writing a fewer number of blocks with more dirty pages generates fewer hidden writes. Therefore, *flash-friendly write patterns*, which reduce the GC cost and thus improve performance and lifetime, are *pseudo sequential patterns* that have high block utilization. To our knowledge, this is the first study that experimentally analyzes the relationship between write performance and write patterns in NAND flash storage devices. We expect that our observations can be applicable in other areas for optimizing write performance and lifetime in NAND flash storage devices.

Our other contribution is the extensive evaluation of TS-CLOCK, and its comparison with seven state-of-the-art replacement algorithms including flash agnostic and flash aware schemes. To measure the performance of each algorithm, we performed our experiments on three commercial NAND flash storage devices. In addition, to analyze how each algorithm affects lifetime, we performed our experiments on the two most widely used FTL schemes. As a result, TS-CLOCK outperforms the other algorithms by up to 90% and expands lifetime by up to 53%.

The rest of this chapter is organized as follows. section 2.2 is an overview of the characteristics of NAND flash storage devices and introduces flash-friendly write patterns. section 2.3 describes the TS-CLOCK algorithm in detail. Then, section 2.4 presents the results of our evaluation. In section 2.5, we briefly survey previous related work and then we conclude this chapter in section 2.6.

2.2 What Are Flash-Friendly Write Patterns?

In this section, we investigate flash-friendly write patterns in order to design page cache replacement algorithms that improve performance and lifetime. We first overview NAND flash storage devices and then discuss our experimental results in order to characterize write performance varying write patterns.

2.2.1 NAND Flash Storage Devices

NAND flash storage devices (such as eMMC, microSD, and SSD) are composed of host interface logic, an array of NAND flash memory, and a controller.

In NAND flash memory, *read* and *write* operations are performed at the unit of a *page* (e.g., 4KB or 8KB) and write operations take typically about 10 times longer than read operations. Before overwriting any page, a whole *block* (composed of 64 - 128 pages) must be erased, and erase operations also take about 10 times longer than write operations. Also, each flash memory cell has limited program/erase (P/E) cycles and the number of maximum P/E cycles sharply drops as the density of the cell increases [40, 41].

In order to hide peculiarities of NAND flash memory, a NAND flash controller runs a *flash translation layer* (FTL). Due to having no in-place overwrites allowed in NAND flash memory, a FTL takes a log-structured approach: rather than modifying existing data, the previous data is invalidated and a new copy is written. A FTL manages a *mapping table* from a logical sector number to a physical page address in NAND flash memories and performs *garbage collection* (GC) to recycle invalidated physical pages. According to their mapping granularity, FTLs can be largely

classified into *block-level FTL* [32], *page-level FTL* [42, 43], and *hybrid FTL* [30, 31]. Block-level FTL and page-level FTL are self-explanatory. Hybrid FTL logically partition blocks into data blocks and log blocks and manage data blocks in block-level mapping and log blocks in page-level mapping. There is a trade-off between the required RAM size for the mapping tables and garbage collection overhead. Low-end flash storage devices such as microSD cards and eMMCs adopt hybrid FTL schemes due to their smaller RAM requirements. In contrast, high-end flash storage devices such as SSDs adopt page-level FTL schemes due to their higher performance with lower garbage collection overhead.

2.2.2 Flash-Friendly Write Patterns

Garbage collection (GC) is universal to all FTL schemes that perform out-of-place writes. Since GC involves moving valid pages in victim blocks to new locations, it generates hidden writes and fundamentally limits write performance in NAND flash storage devices [23, 33, 34, 44, 45]. The efficiency of GC is generally measured in terms of the write amplification factor (WAF) – the ratio of total internal writes including copying induced by GC to externally requested writes. Since higher WAF means that more hidden writes are generated during GC, it negatively impacts performance and lifetime.

Modeling flash write performance is non-trivial because various factors including write pattern, I/O range, GC policy, and the amount of free space interact with each other non-linearly. To characterize flash write performance, previous studies use various techniques: black-box testing using heuristically generated patterns [33, 46, 47], mathematical modeling [34, 44, 45], and statistical machine learning [48]. Unfortunately, they mostly focus on relatively simple patterns such as completely sequential writes, uniform random writes and stripe writes. Thus, to optimize flash write performance, system designers rely on heuristics such that (1) sequential writes are

much faster than random writes, (2) random write performance converges with sequential write performance when requests are aligned in block size and their sizes are also in block size [23, 47], and (3) the performance of random writes in a narrow I/O range is higher than that in a wide I/O range [49].

Page cache replacement algorithms are responsible for shaping write requests from applications into desirable write patterns for NAND flash storage devices. Therefore, we need to understand *"what flash-friendly write patterns are"* and evict dirty pages for them to form such flash-friendly write patterns. To do this, we formulate a hypothesis that, *when writing dirty pages in a block by the sector number, the WAF becomes smaller as the block utilization (i.e., the ratio of dirty pages in a block) becomes higher*. Since writing more pages to a block is likely to invalidate more pages in that block, GC can easily select a victim block with few valid pages and thus can minimize the overhead of copying the valid pages. To confirm the hypothesis, we performed trace-driven simulations for synthetic traces with different block utilizations. The synthetic trace with $x\%$ block utilization is where we randomly select $x\%$ of pages in a randomly selected block and then issue write operations for the selected pages by the sector number. In addition, to investigate how I/O ranges affect the WAF in our traces, we generated synthetic traces with different I/O ranges. The traces were run on an FTL simulator with two representative FTL schemes (FAST FTL [30] as a representative hybrid FTL scheme and page-level FTL [42]).

Our experimental results in Figure 2-1 clearly show two things regardless of the FTL scheme used:

- As the block utilization of a trace increases, the WAF decreases. In all cases, traces with 100% block utilization have an ideal WAF, 1, with no hidden write overhead. In contrast, for traces with 25% block utilization, the WAFs significantly increase: 4.9 for FAST FTL and 3.5 for page-level FTL.

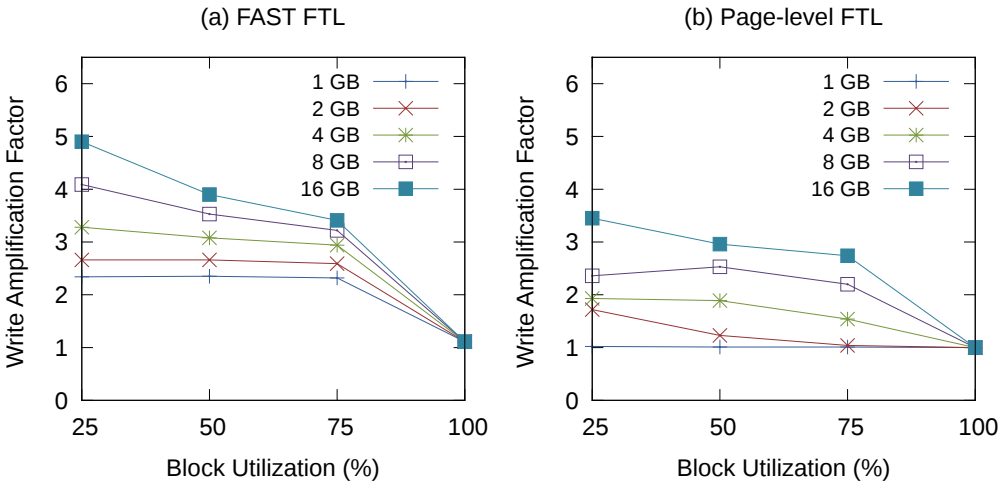


Fig. 2-1: Write amplification for synthetic traces with different block utilizations and different I/O ranges. In our FTL simulation, the capacity of the devices is 16 GB and the over-provisioning space is 2.4 GB (15%).

- As the I/O range of a trace decreases, the WAF decreases. For traces with 25% block utilization and 1 GB range, the WAFs of FAST FTL and page-level FTL are 2.3 and 1.02, respectively.

Since the I/O range is the characteristics of workloads, which replacement algorithms cannot control, we use the block utilization as our main design rationale behind the TS-CLOCK algorithm. For brevity, we use the term *pseudo sequential write patterns* or *flash-friendly write patterns*, which are write patterns with high block utilization. While we omitted experimental results on real NAND flash storage devices due to space limitations, throughputs on the real devices were inversely proportional to the WAFs in Figure 2-1.

2.3 The TS-CLOCK Algorithm

In this section, we present a novel replacement algorithm called TS-CLOCK for NAND flash storage devices. Guided by the unique performance characteristics of NAND flash storage devices described in Section 2.2, the design objectives of the TS-CLOCK are stated as follows:

- Maximize cache hit ratio by exploiting temporal locality. To achieve high performance, it is important to reduce the number of I/O requests issued to NAND flash storage devices.
- Prefer to evict clean pages over dirty pages. Since write operations are slow and hurt the lifetime of a device, it is important to evict less dirty pages for higher performance and longer lifetime.
- Shape evicted dirty pages to flash-friendly write patterns which have high block utilization. The flash-friendly pseudo sequential patterns can minimize hidden writes inside NAND flash storage devices and thus improve performance and lifetime.

Now, let us explain the details of the TS-CLOCK algorithm based on the pseudo-code depicted in Figure 2-2. To exploit temporal locality and thus keep hit ratio high, TS-CLOCK extends a well-known CLOCK replacement algorithm [35]. CLOCK is an LRU approximation algorithm which maintains a reference bit. While at every cache hit LRU needs to move a page to the most recently used (MRU) position in the list, CLOCK only monitors whether a page has been recently referenced or not. When a page is accessed, the page unit hardware sets the reference bit of the page to 1. Whenever free page frames are needed, the *clock-hand*, which points to the last evicted page, scans pages in a circular list until a page with a reference bit of zero is found, and that page

is then replaced. In the course of the scan, CLOCK clears a reference bit in every page to zero. After all, CLOCK replaces such pages that have not been referenced upon one rotation of the *clock-hand*. To grant one full life of a new page, CLOCK inserts the new page into the position of the evicted page.

```
page *t−hand = null, *s−hand = null;

void TS−CLOCK(page *p) {
  if (p is not in the buffer cache) {
    if (the buffer cache is full) {
      page *v = choose_victim();
      evict v;
    }
    if (t−hand is null)
      t−hand = p;
    else
      insert p into the position of t−hand;
    if (p.status is dirty)
      insert p into the sorted list of p.block;
  }
  if (p.status is clean)
    p.reference_count = 1;
  else {
    if (p.reference_count is 0)
      p.block.non_zero++;
    p.reference_count = ceil(4 * p.block.non_zero / number of pages per block);
  }
}

page *choose_victim() {
  while (true) {
    if (t−hand.reference_count is 0) {
      if (t−hand.status is clean)
        return t−hand;
      if (s−hand is null)
        s−hand = the first page
          in the sorted list of t−hand.block;
      while (true) {
        page *s = s−hand;
        if (s−hand is the last of the block)
```

```
36              s−hand = the first page
37                in the sorted list of t−hand.block;
38            else
39              s−hand = the next page in the sorted list of the block;
40            if (s.reference_count is 0)
41              return s;
42          }
43        }
44        t−hand.reference_count−−;
45        if (t−hand.reference_count is 0)
46          p.block.non_zero−−;
47        t−hand = the next page of t−hand in the circular list;
48      }
49  }
```

Fig. 2-2: The pseudo-code of the TS-CLOCK algorithm

TS-CLOCK maintains a *reference count* for each page rather than a reference bit to give each page a different level of opportunity to stay in the page cache. Similar to CLOCK, a reference count for each page is set to R when that page is accessed (Line 17, 21), and it is decremented by one when the *t-hand* (i.e., the *clock-hand* in CLOCK) sweeps the page (Line 44). A page with a reference count of R is granted to stay in the page cache for R times the rotation of the *t-hand*. Though a reference count of every clean page is always set to one (Line 16 - 17), that of every dirty page is determined by its update likelihood (Line 21). We approximately calculate the update likelihood of a page from the update likelihood of the block that belongs to it. The update likelihood of a block is calculated as the ratio of dirty pages, such that they belong to that block and their reference count is not zero, to the number of pages per block. If a page is likely to be updated, its reference count is set to large number for that page to stay longer in the page cache. For four equally divided ranges of update likelihood, 0 - 25%, 25 - 50%, 50 - 75 %, and 75 - 100%, we set a reference count to 1, 2, 3, and 4, respectively (Line 20 - 21, 45 - 46). Since a reference count of a dirty page can be set to larger than one, TS-CLOCK prefers to evict clean pages over dirty pages

and, among dirty pages, it prefers to evict dirty pages that are unlikely to be updated. If a selected page with a reference count of zero is clean, it is immediately evicted (Line 28 - 29). Otherwise, TS-CLOCK re-selects a victim page using the *s-hand*, as we will explain in next paragraph. Note that our reference counting scheme can be efficiently implemented using paging unit hardware: when the *t-hand* scans pages in the circular list, TS-CLOCK resets a reference count of a page whose reference bit is one.

In order to shape evicted dirty pages to flash-friendly pseudo sequential patterns with high block utilization, TS-CLOCK manages the *s-hand* in addition to the *t-hand*. The *s-hand* scans dirty pages, which belong to the same block, by their sector numbers to maintain spatial locality of evicted dirty pages (Line 13 - 14, 30 - 42). TS-CLOCK replaces a dirty page via a two-step process: the *t-hand* selects a block which is unlikely to be updated (Line 31 - 32, 35 - 37) and then the *s-hand* finally selects a dirty victim page in the block (Line 34, 40 - 41). Let's consider that the *t-hand* selects a page with a reference counter of zero. If the selected page is clean, it is immediately evicted as we explained (Line 28 - 29). But, if it is dirty, the *s-hand* scans dirty pages by their sector numbers until a page with a reference count of zero is found (Line 13 - 14, 30 - 42). When the *s-hand* points to *null* (its initial value) or the last dirty page in a block, TS-CLOCK sets the *s-hand* to the dirty page that belongs to a block pointed by the *t-hand* and that is the first page in that block (Line 30 - 32, 35 - 37). In contrast to the *t-hand*, while the *s-hand* scans dirty pages, it does not decrement a reference count of each page. That is because a reference count is designed to reflect temporal locality. After evicting a page, TS-CLOCK inserts a new page into the position of the *t-hand* to grant at least one full life to the page (Line 12). In this way, TS-CLOCK can generate flash-friendly pseudo sequential write patterns with high block utilization and thus minimize hidden writes inside NAND flash storage devices.

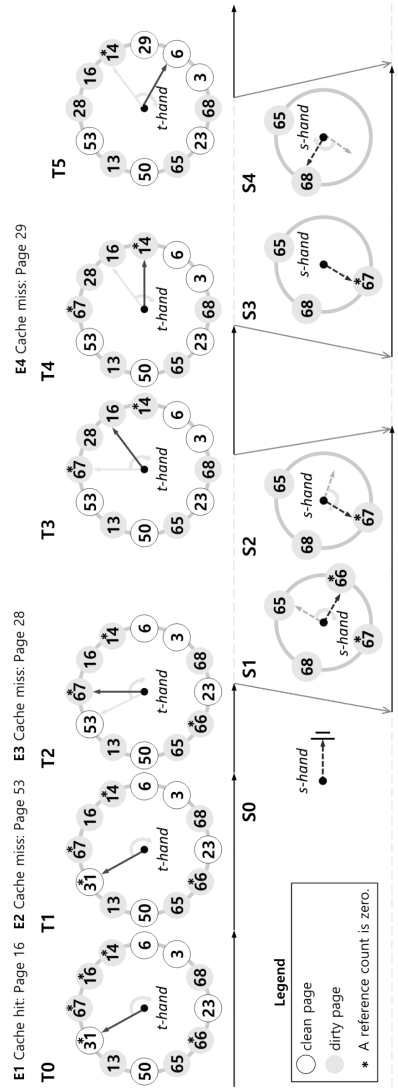


Fig. 2-3: An example of TS-CLOCK operations showing how TS-CLOCK handles events E1 - E4 from the initial status of T0 and S0. In this example, the size of a block is four pages.

Figure 2-3 illustrates an example of TS-CLOCK operations which show how TS-CLOCK handles events E1 - E4 from the initial status of T0 and S0. In this example, the size of a block is four pages. While accessing Page 16 (E1) at T0, a cache hit occurs and the reference counter of Page 16 is set to 2 (T1). Then, a cache miss occurs while accessing Page 53 (E2). Page 31, which is clean and is being pointed to by the *t-hand*, is replaced by a new page, Page 31 and now the *t-hand* moves to the next page in the circular list (T2). Next, a cache miss occurs while accessing Page 28 (E3). The *t-hand* selects Page 67, which is dirty, and then TS-CLOCK starts to scan the *s-hand* of Block 16, which Page 67 belongs to. Since the *s-hand* is initially *null* (S0), the *s-hand* is set to the smallest page in Block 16, which is Page 65 (S1). TS-CLOCKS sweeps the *s-hand* until a page with a reference count of zero is found. As a result, Page 66 is evicted (S2). Then, the new page, Page 28, is inserted at the position of the *t-hand* and TS-CLOCK moves the *t-hand* one step forward (T3). Finally, a cache miss occurs while accessing Page 29 (E4), and TS-CLOCK sweeps the *t-hand* to select a victim page and then selects Page 14 (T4). Since the selected Page 14 is dirty, TS-CLOCK uses the *s-hand* to select a dirty victim page. Since the *s-hand* already points to pages in Block 16 (S3), TS-CLOCK finds a victim page, Page 67, by sweeping the *s-hand*. The victim page pointed by the *s-hand* is evicted (S4), the new page, Page 29, is inserted (T5).

	Storage-A	Storage-B	Storage-C
Vendor	Patriot	Adata	Samsung
Type	microSD card	microSD card	SSD
Capacity	16 GB	16 GB	120 GB
Erase Block Size	4 MB	4 MB	24 MB
Flash Memory	MLC	MLC	TLC

Table 2-1: Specification of the NAND flash storage devices

2.4 Evaluation

In this section, we present the performance evaluation and analysis to assess the effectiveness of TS-CLOCK. We first compare the performance results of TS-CLOCK with those of well-known page cache replacement algorithms: LRU, CLOCK, Linux2Q, CFLRU, FAB, LB-CLOCK, and Sp.Clock.

2.4.1 Experimental Setup

For evaluation, we follow the evaluation methodology used by Kim et al. [17]. Our evaluation proceeds in three steps: first, we collect traces of read/write operations to the page cache from mobile and server workloads. We call this the *before-cache traces*. Second, we use the collected traces as the input of a simulator that implements the seven page cache replacement algorithms. The simulator produces a *cache hit ratio* and a storage access trace evicted by a page cache replacement algorithm, which we call the *after-cache trace*. Finally, the after-cache traces are replayed on top of real devices or on a FTL simulator. To measure the performance of each replacement algorithm, we replayed the after-cache traces on real devices in Table 2-1 with the `O_DIRECT` option. We also enable the Native Command Queuing (NCQ) to exploit the internal parallelism of the devices.

Workload	Read (MB)	Write (MB)	Write Range (MB)
Video Streaming	0.2	1514.4	266.6
Mixed Applications	526.9	413.2	133.0
File Server	7252.8	5880.1	14568.2
TPC-C	1129.2	2357.6	15124.4

Table 2-2: Summary statistics of our test traces

To broadly exercise the replacement algorithms, we choose two microSD cards and a SSD such that their manufactures are different and their flash memory types include MLC and TLC. We measure the size of the erase block in each NAND flash storage by following the measurement methodology in Kim et al. [47]. Also, to evaluate how each replacement algorithm affects lifetime, we replayed the after-cache traces on a FTL simulator. Our FTL simulator is trace-driven and supports FAST FTL and page-level FTL.

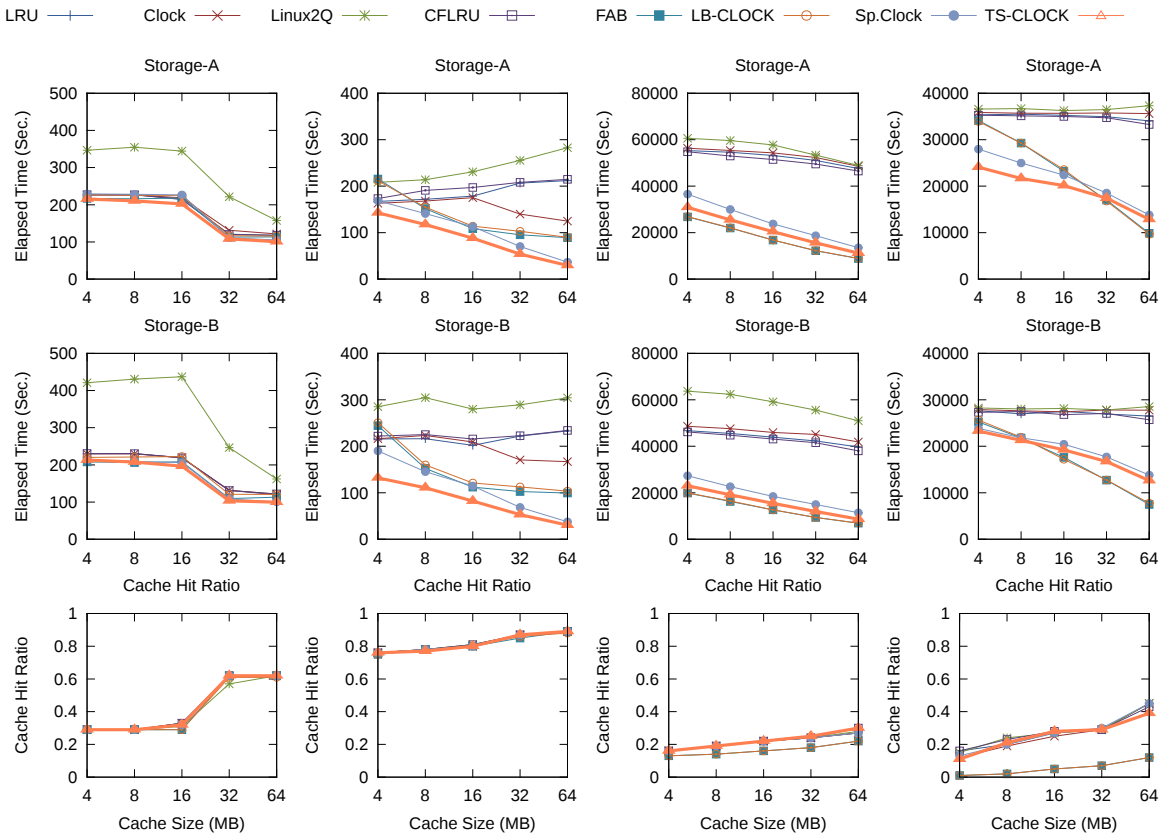
2.4.2 Evaluation Workloads

We used four before-cache traces: the two are workloads on mobile devices and the others are workloads on servers. Table 2-2 summarizes their statistics. The mobile workloads – video streaming and the mixed applications – are from Kim et al. [17]. These two workloads are collected from a real Android smart phone: the video streaming is collected while watching YouTube video and the mixed applications workload is collected while running multiple mobile applications, such as Facebook, Maps, Camera, Internet Browser, YouTube, Gallery, etc., for several hours. We modified the Linux kernel (version 3.2.0) to collect server traces. The file server workload is collected by running the Dbench benchmark [50], which simulates file server I/O. The TPC-C workload is collected while running the TPC-C benchmark [51] on a MySQL database system with 50 warehouses.

2.4.3 Experiments on the MicroSD Cards

We first evaluate the performance of the eight replacement algorithms on the two microSD cards. Figure 2-4 presents the elapsed times and hit ratios of each replacement algorithm varying in page cache size from 4 MB to 64 MB. According to our measurements in Table 2-1, we configured the block sizes of TS-CLOCK, FAB, and LB-CLOCK to 4 MB. Also, we set the window size of CFLRU as 25%, which is advised by the authors of the paper, for their static algorithm. As we described, we measured elapsed time by replaying a generated *after-cache trace* of each replacement algorithm on the microSD cards. Figure 2-4 show that TS-CLOCK maintains high cache hit ratios, which are comparable to those of traditional page cache replacement algorithms, in all cases. In addition, TS-CLOCK re-arranges the eviction order of dirty pages to form *pseudo sequential write patterns*, and thus it significantly outperforms the other algorithms. In the rest of this section, we will analyze results in more detail.

Mobile Workloads: For the video streaming workload, which has mostly sequential accesses, TS-CLOCK outperforms the other algorithms: on average 12.8% for Storage-A and 15.3% for Storage-B (Figure 2-4a). For the mixed applications workload, whose significant portions are random accesses, TS-CLOCK significantly outperforms the other algorithms: by up to 89.5% for Storage-A and 90.0% for Storage-B (Figure 2-4b). To understand why TS-CLOCK performs better, we analyze the distribution of block utilizations from the after-cache traces. We count the number of sorted write operations in the same block. Specifically, Figure 2-5 shows the distribution of the mixed applications workload. It clearly shows that TS-CLOCK issues significantly less number of write operations and it clusters more write operations for a block to generate flash-friendly write patterns.



(a) Video Streaming (b) Mixed Applications (c) File Server (d) TPC-C

Fig. 2-4: Elapsed time and hit ratio on the microSD cards

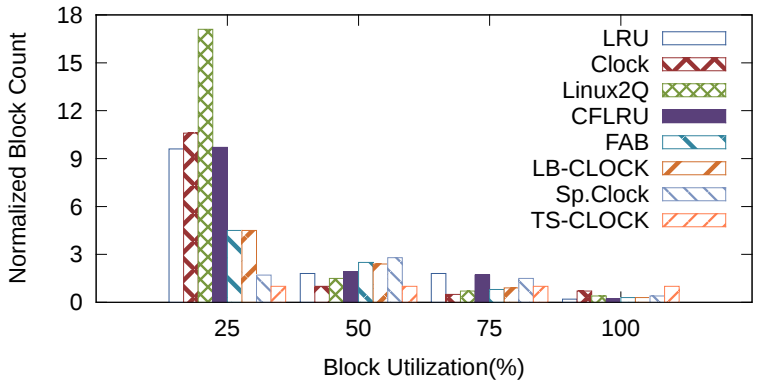


Fig. 2-5: Distribution of block utilizations for write operations in the mixed applications workload. Cache size and Block size are set to 4 MB.

Server Workloads: For the file server and TPC-C workloads, TS-CLOCK significantly outperforms the others, excluding FAB and LB-CLOCK, by up to 78.1% for the file server workload and 65.6% for the TPC-C workload (Figure 2-4c and 2-4d). That is because the server workloads are mostly random accesses and their I/O ranges are wider than those of the mobile workloads. While FAB and LB-CLOCK are slightly faster than TS-CLOCK, they issue significant write operations than TS-CLOCK does. That is because their cache hit ratios are significantly lower, the lowest among the eight algorithms, due to the *early eviction problem* caused by their coarse-grained cache management. Since they issue significantly more write operations, they shorten the lifetime of the devices.

Our experimental results reveal that write pattern strongly affects performance on low-end NAND flash devices which mostly use hybrid FTLs due to smaller memory requirement. In terms of performance, TS-CLOCK, FAB, and LB-CLOCK perform well since they all consider write patterns. However, in terms of lifetime, the coarse-grained cache management of FAB and LB-CLOCK generate significantly more number of write operations than TS-CLOCK does. As a re-

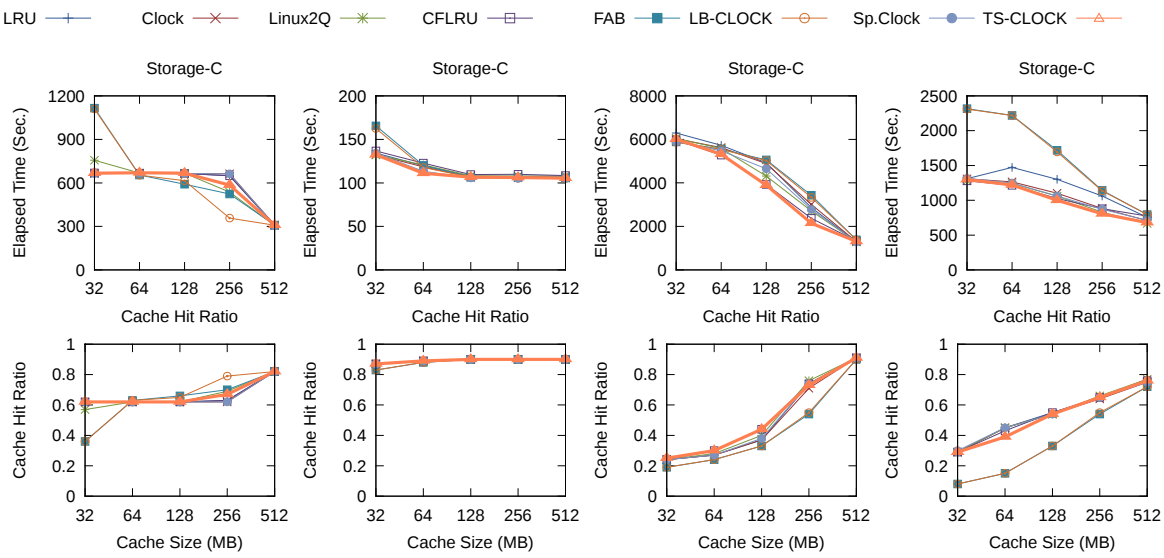
sult, TS-CLOCK improves performance and lifetime at the same time by maintaining high hit ratios and generating flash-friendly write patterns.

2.4.4 Experiments on the SSD

Now, we investigate our experimental results on our TLC-based SSDs. Performance on SSDs could be different from that on microSD cards, since SSDs typically adopt page-level FTL to achieve high performance and their erase block size is much larger than that of microSD cards to exploit higher degrees of internal parallelism (24 MB in our case). Also, we set cache sizes to more realistic ranges for computing devices with high performance SSDs: from 32 MB to 512 MB.

Mobile Workloads: For the video streaming workload, which consists mostly of sequential accesses with some meta-data updates, TS-CLOCK performs well in all cache sizes (Figure 2-6a). However, under large enough cache size, e.g., 256 MB, to keep all metadata in memory, FAB and LB-CLOCK also performs well with high cache hit ratios. In particular, TS-CLOCK outperforms Sp.Clock, which is the best performing replacement algorithm for mobile workloads in the recent literature, by up to 11.8%. This result confirms that our approach to generate *flash-friendly write patterns* is more effective than the sorted eviction of Sp.Clock.

For the mixed applications workload of which significant portions are random accesses, results on the SSDs in Figure 2-6b show quite different trends than on the microSD devices in Figure 2-4b; all replacement algorithms show similar cache hit ratios and similar performance. However, these results are not surprising. As we discussed in Figure 2-1b, since I/O ranges of random writes are small compared to the capacity of a device, the WAF approaches to a value of one. Since the write range of the mixed applications workloads is quite small (133 MB over 120 GB, see Table 2-2 and



(a) Video Streaming (b) Mixed Applications (c) File Server (d) TPC-C

Fig. 2-6: Elapsed time and hit ratio on the SSD

2-1), performance will be mostly limited by the amount of write operations. Since all replacement algorithms show similar cache hit ratios, they all show similar performance.

Server Workloads: For the server workloads, TS-CLOCK performs best in all cases: on average 9.3% for the file server workload and 14.2% for the TPC-C workload (Figure 2-6). Interestingly, comparing to performance results on microSD cards, CFLRU performs much better while FAB and LB-CLOCK performs much worse. We found the reason by analyzing the after-cache traces. Figure 2-7 compares the amount of generated I/O requests from each of the replacement algorithms. It clearly reveals the limitations of FAB and LB-CLOCK. The amount of generated I/O requests from FAB and LB-CLOCK are the largest in all cases due to the coarse-grained cache management. On the other hands, TS-CLOCK and CFLRU show lower eviction counts, especially in write operations, than the others due to their fine-grained cache management and an eviction

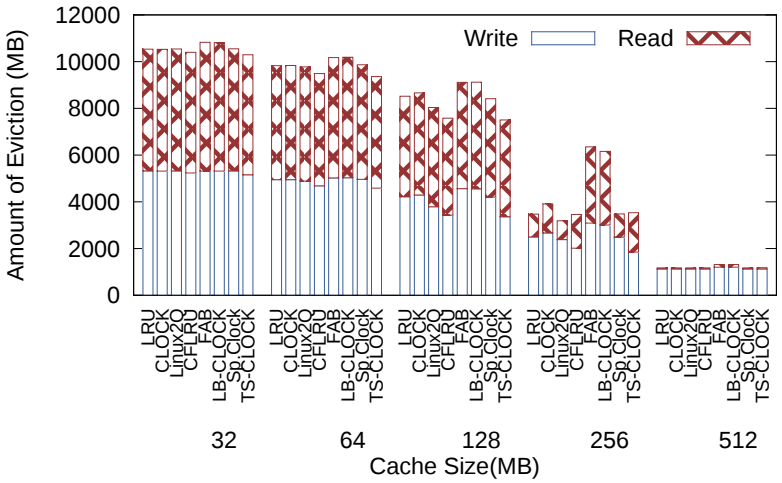


Fig. 2-7: The amount of generated I/O requests for the file server workload

policy that prefers to evict clean pages over dirty pages. In addition, since TS-CLOCK generates pseudo sequential write patterns which minimize GC costs, it performs best in all cases.

In summary, the number of write operations strongly affects the performance of SSDs, which typically adopt page-level FTL with large overprovisioning space. It clearly shows that a page cache replacement algorithm should consider both of the number of generated write operations and their patterns to work well on a wide range of NAND flash storage devices.

2.4.5 Effect on Lifetime

In order to investigate how each replacement algorithm affect the lifetime of NAND flash storage devices, we ran the after-traces on our FTL simulator. The FTL simulator is trace-driven and supports FAST FTL and page-level FTL. For FAST FTL, it is configured as typical microSD cards:16 GB capacity, 15% over-provisioned space, 4 KB page, and 4 MB block. For page-level FTL, its block size is configured to 24 MB and the other parameters are the same as FAST FTL. Before run-

ning each workload, we run aging traces, which are 16 GB and are mixed with sequential accesses and random accesses.

Figure 2-8 and Figure 2-9 compare erase counts. Our experimental results clearly show that TS-CLOCK significantly extends the lifetime of NAND flash storage devices more than the others. More specifically, it can extends the lifetime by up to 29% compared to CFLRU on FAST FTL (Figure 2-8b) and by up to 53% compared to FAB on page-level FTL (Figure 2-9d). The reason behind these results is that TS-CLOCK prefers to evict clean pages to minimize write operations as well as it shapes evicted dirty pages in flash-friendly write patterns to minimize hidden write operations.

2.4.6 Write Patterns

To intuitively understand how each replacement algorithm behaves, we plot the before-cache traces of the mixed applications workload and the after-cache traces of the five selected algorithms, LRU, CFLRU, FAB, Sp.Clock, and TS-CLOCK. As we expected, LRU and CFLRU generates random I/O patterns since they do not consider write patterns (Figure 2-10b and Figure 2-10c). On the other hand, TS-CLOCK seems to evict multiple pages in a unit of a block like FAB although it actually evicts at the granularity of a page (Figure 2-10f). This is because that TS-CLOCK shapes evicted dirty pages to flash-friendly write patterns which have high block utilization. Especially, we also found that TS-CLOCK performs more sequential write operations compared to FAB since TS-CLOCK maximizes block utilization by clustering page based on reference count. Moreover, as shown in Figure 2-4 and Figure 2-6, TS-CLOCK outperforms Sp.Clock in all combinations of workloads and page cache sizes, even though Sp.Clock writes almost all pages sequentially (Figure 2-10e). This reveals that flash-friendly pseudo sequential writes are more effective than the sorted writes of Sp.Clock.

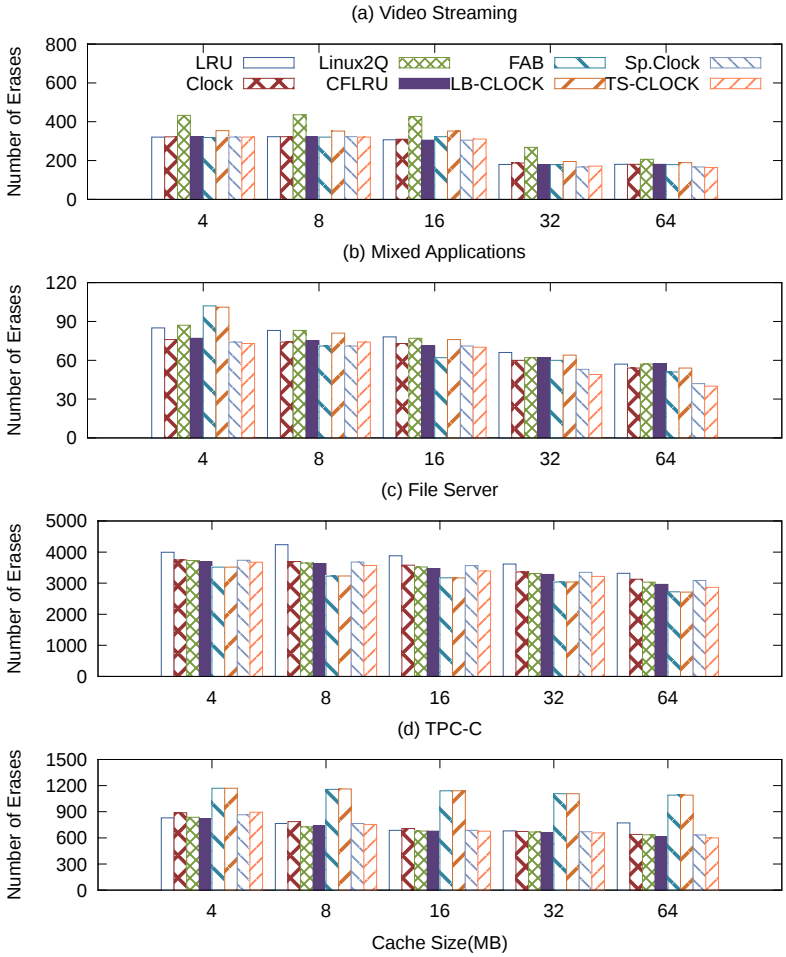


Fig. 2-8: Comparison of erase count on FAST FTL

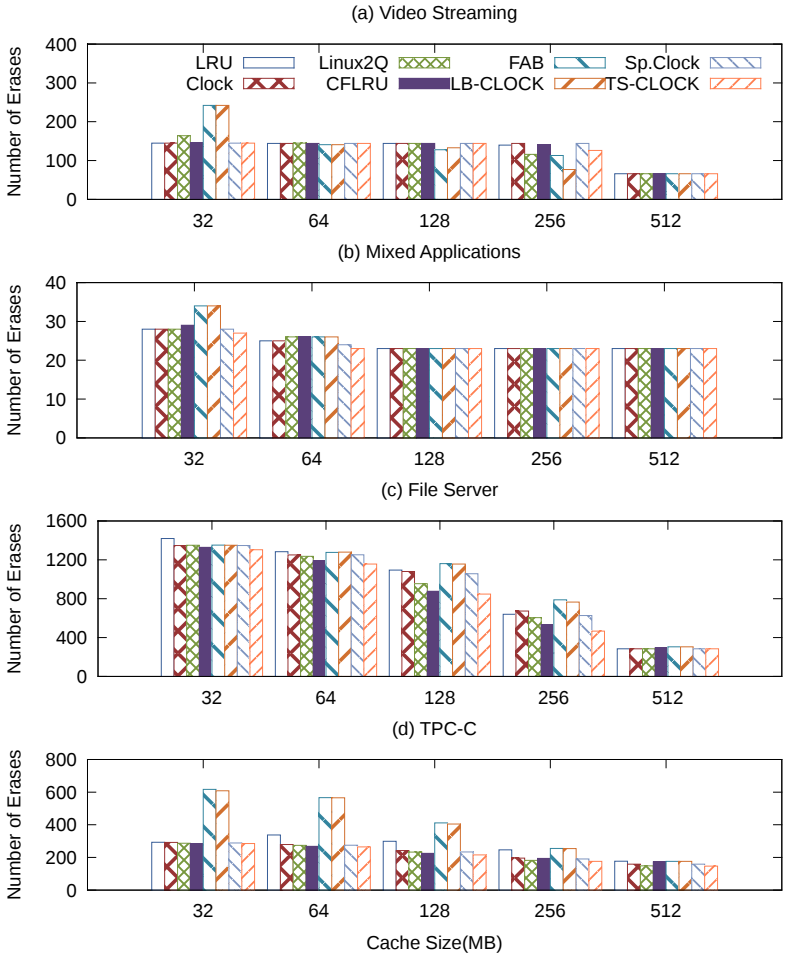


Fig. 2-9: Comparison of erase count on page-level FTL

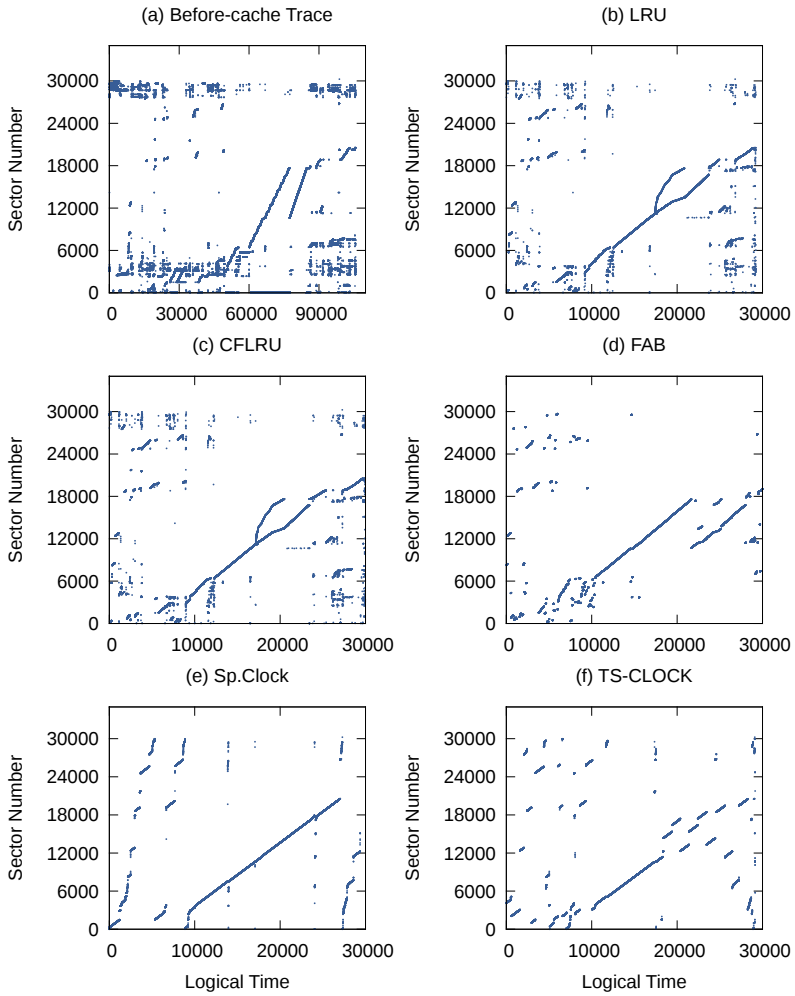


Fig. 2-10: Comparison among the before-cache trace and the after-cache traces. Only write operations are presented.

2.5 Related work

Over the years, the HDD has been considered to be a performance bottleneck because it has poor access latency due to the characteristic of having mechanical movement. To relieve this problem, various replacement and I/O scheduling algorithms have been proposed. Most replacement algorithms [35, 36, 52–54] have been designed to reduce the number of I/O operations by exploiting temporal locality. However, this involves seeking and rotating overheads because they do not consider write patterns of evicted pages. In order to mitigate these overheads, I/O scheduling algorithms, such as SSTF, SCAN, VSCAN, and FSCAN, have focused on spatial locality, which can minimize seek and rotate times. DULO [37], WOW [38], and STOW [55] combine the above two approaches to improve I/O performance. However, these algorithms only consider characteristics of the HDD.

Recently, many approaches have been extensively proposed to consider the characteristics of NAND flash storage devices in the page cache replacement layer. Previous flash-aware page cache replacement algorithms can be classified into two: First, CFLRU [13], LRU-WSR [15], and FOR [16] consider the asymmetric read and write cost and they prefer to evict clean pages over dirty pages to reduce the number of more expensive write operations. However, these algorithms cannot guarantee I/O performance, because they generate random write patterns. Second, the other algorithms [14, 17–19, 39] focus on write patterns. FAB, BPLRU, and LB-CLOCK select a block including the largest number of pages and evicts all pages in the block for generating sequential write patterns. Sp.Clock manages pages by its logical sector number and uses the sorted eviction scheme. The major drawback of these algorithms is that they slightly ignore the recency by evict-

ing whole pages in the same block and sorting pages in the block by the sector number. BPAC exploits the temporal and spatial locality by using the dual-list. Unfortunately, BPAC does not consider read operations because it has been designed for the write buffer in SSD.

2.6 Summary

In this chapter, we proposed a novel page cache replacement algorithm, called TS-CLOCK, for NAND flash storage devices. TS-CLOCK extends the CLOCK algorithm for temporal locality and exploits spatial locality for generating the *flash-friendly write pattern*. Moreover, it prefers to evict clean pages over dirty pages to minimize expensive write operations. Our experimental results clearly confirm that TS-CLOCK outperforms existing replacement algorithms in terms of performance and lifetime of the devices.

3. AFS: Making Application-level Crash Consistency Practical on Flash Storage Devices

3.1 Introduction

File system problems: To guarantee the consistency of file metadata, data blocks, and versions, modern file systems have heavily resorted to various techniques such as journaling and copy-on-write [56]. Unfortunately, they suffer from heavy read/write amplification incurred by the mechanisms inherent in each scheme, including redundant write [57, 58], segment cleaning [25, 59], and tree wandering [60]. Furthermore, because they do not provide the higher *application-level crash consistency* (hereafter, for short, crash consistency) [61–65], many consistency-critical applications (*e.g.*, MySQL [66], SQLite [67], git, and VMware) should implement their own idiosyncratic mechanisms for ensuring the secure recovery of their data from unexpected crashes, which are, in some cases, still crash-vulnerable [64, 65, 68].

***SHARE* and its opportunities:** Considering that flash become the main storage, especially for performance critical applications, it is an urgent and practical problem for file system communities to develop flash-tailored solutions for higher consistency level (*e.g.*, crash consistency) and for higher performance (*e.g.*, no redundant write), by leveraging the new interface such as *SHARE*. To this end, we address two problems in file systems, IO amplification for data consistency and

lack of crash consistency, especially focusing on ext4 journaling mechanism, with the *SHARE* flash storage interface [69]. The *SHARE* interface allows host programs to explicitly remap one or more pairs of LBAs atomically at the flash storage FTL layer. Though simple, it is very effective in eliminating the overhead of *redundant writes for guaranteeing atomic write, excessive read-/writes in compaction, and the tree-wandering problem* in database applications such as MySQL DWB [66] and Couchbase storage engine [69]. One coincident and intriguing observation is that these database overheads have essentially same characteristics with those consistency overheads in modern file systems.

Our contribution, *AFS*: Based on this observation, in this chapter, we propose *AFS* (*Application-crash-consistent File System*), which extends the existing ext4 journaling file system naturally and minimally so as to utilize the *SHARE* interface, thus achieving both higher performance (*i.e.*, no redundant writes) and higher consistency (*i.e.*, crash consistency). *AFS* makes two main contributions: single-write journaling and application-level crash consistency. For single-write journaling, *AFS* provides *SHARE*-aware data journaling (SDJ) mode, which can achieve the highest data consistency (*i.e.*, version consistency [56]) at the same performance of ordered journal (OJ) mode. We slightly modified the data journal (DJ) mode in ext4 file system so that, after (metadata and data) blocks have been successfully written in journal area, an *SHARE* call for the multiple blocks is made, instead of making `pdflush daemon` to write them redundantly in their original locations. Thus, *AFS* can achieve high data consistency with single write.

Next and more importantly, *AFS* guarantees the atomic write of multiple scattered pages in either single or multiple files opened with `O_ATOMIC` flag. We adopted the semantics of `fsync()` and `syncv()` calls slightly, added two new system calls, `abort()` and `abortv()`, and modified the commit/checkpoint/recovery operations in DJ mode (minimal changes and very compat-

ible to DJ mode). Without *SHARE*, as will be detailed later, this light-weight implementation of solid application-level crash consistency would not be possible even with double-write journaling. While designing *AFS*, we identified an interesting recovery property, called *A-property*.

The focus of *AFS* is on crash consistency and thus it leaves concurrency control management (*i.e.*, isolation) to the applications, as other file systems [61–65].

Prototype and Evaluation: We prototyped *AFS* by modifying ext4 journal (kernel version: 4.6.7) on top of a commercial SSD available in the market, inside which we implemented the *SHARE* interface as a firmware. Our preliminary evaluations confirm that the effect of *AFS* is very promising. As an example, when we ran an OLTP benchmark using MySQL/InnoDB DBMS under *AFS*, we observed 6x TPS improvement over the default configuration where the benchmark was run with InnoDB engine's own journaling mode (*i.e.*, DWB: double-write buffer) enabled. This surprising performance improvement can be explained as follows: by offloading the responsibility of guaranteeing the atomic full page write data from InnoDB engine itself to *AFS*, it can halve the amount of data being written to the flash storage, and can, more importantly, reduce the number of `fsync()` system calls by 16.4 times. Also in SQLite database, we observed that, compared to the case when it was ran in either RBJ (roll-back journal) or WAL (write-ahead logging) mode under ext4 ordered journaling mode, SQLite on *AFS* can achieve better performance as well as the same crash consistency even when its journaling mode is turned off. From these results, we confirm that *AFS* can make many consistency-critical applications high-performance and also free from the burden of devising their own idiosyncratic mechanisms for crash consistency.

Benefit of *AFS*: 1. TRIM-like: As will be shown in this chapter, like the well known TRIM command, which has successfully been incorporated into major OS/file system kernel, 1) the

SHARE interface could be easily supported by commercial SSDs, 2) *AFS* can be built with minimal extension of the existing file systems (*e.g.*, 400 lines added to DJ mode), and 3) it will have high performance impact on a variety of applications. **2. Portability [64]:** The existing legacy applications can run under *AFS* without any modification, and they are provided with higher data consistency. And, *AFS* allows applications to be made crash consistent simply by adding `O_ATOMIC` flag to `fopen()` system call. Two types of applications can run concurrently under *AFS*.

In the rest of this chapter, we will discuss background (section 3.2) and related work (section 3.3). Next, we will present the details of *AFS* design (section 3.4) and implementation (section 3.5). Then, we will show our evaluation results (section 3.6). Finally, we discuss how *SHARE* interface can be used to minimize segment cleaning overhead in a log-structured file system (section 3.7), and conclude (section 3.8).

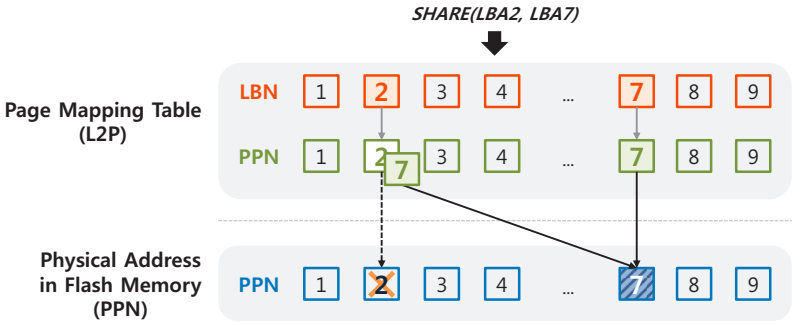


Fig. 3-1: *SHARE* Interface

3.2 Flash memory, FTL, and *SHARE*

Because flash memory does not allow to update pages in place, an out-of-place update strategy is commonly taken by every flash storage devices. Thus, to maintain the ever-changing mapping between logical addresses and physical flash memory addresses, every flash storage device are equipped with a firmware module called *FTL* (flash translation layer), and the fine-grained page-mapping approach is popular mainly for performance reasons.

In order to leverage this indirection of page-level address mapping in flash storage, recently in database community, Oh *et al.* [69] proposed the *SHARE* interface. It exposes an abstraction that allows host applications to explicitly ask FTL to change the *internal* address mapping maintained by FTL. To be concrete, as illustrated in Figure 3-1, upon receiving a `share` command from the host with a pair of two logical block addresses, *LBA2* and *LBA7*, as its parameter, FTL changes the PPN (physical page number) of *LBA2* in its page-mapping table to that of *LBA7*, thus the latter physical page being shared by the former logical address. A `share` command can have an optional third argument, `length`, when the length of data to be shared is longer than the FTL

mapping granularity (*i.e.*, 4KB). Though the description so far assumes that a `share` command is associated with a single pair of LBAs, it can have multiple LBA pairs in a batch. In this case, FTL should be able to support the *atomic address remapping* for the given set of LBA pairs upon a system crash or power-off failure.

The *SHARE* interface, though simple, has proven to be very effective in reducing the read/write amplification in various database applications [69]: with the help of *SHARE*, 1) the atomic full page write, which is critical to MySQL database, can be achieved without the double-write mechanism in InnoDB engine, 2) the read/write amplification of the compaction operation in NoSQL databases can be replaced by zero-copy compaction, and 3) the write amplification problem in CoW B-tree, so called tree-wandering, can be avoided in Couchase NoSQL storage engine. One strong motivation of our work on *AFS* is that journaling-based file systems can also benefit from *SHARE* by applying the single-write journaling of *SHARE*-aware InnoDB engine to the ext4 data journal mode. The idea of applying *SHARE* to file systems is not limited to journaling file system, and would be also very helpful in optimizing the segment cleansing overhead in log-structured file systems [25, 59] and the tree-wandering problem in CoW B-tree file systems [60].

3.3 Related Work

Three types of existing works are closely related to our work: 1) file system consistency 2) application-level crash consistent file systems, and 3) address block remapping, and here we briefly review and compare each work with *AFS*.

File System Consistency By using a variety of techniques such as journaling and copy-on-write, modern file systems provide various consistency levels including metadata, data, and version consistency [56]. However, the system-wide consistency currently provided by file systems is a broken abstraction for application-level crash consistency [64]. Therefore, many applications such as SQLite [67] and Vim [70] should craft their own complex update protocols that ensure its data can be correctly recovered upon unexpected system crash or power failure, mainly by calling `fsync()` system calls for ordering and durability. Unfortunately, they suffer from poor performance due to frequent `fsync()` calls [56, 71], and some of them are still even vulnerable to crashes [65]. We argue that *AFS* is a principled and practical way to change this landscape, which supports crash consistency as first-class citizen inside file system [61–64]. In addition, *AFS* can, with the help of *SHARE*, provide all consistency levels at no cost of redundant writes.

Application-Level Crash Consistent File System As far as we know, two crash consistent file system have been proposed: Failure-atomic Msync [61, 62] and CFS [63]. CFS is similar in spirit to our *AFS* in that it achieves application-level crash consistency by utilizing a transactional storage, called X-FTL [72], which can atomically update multiple (scattered) pages in place, and by extending an existing file system. However, our *AFS* is more practical than CFS in that *SHARE*

is simple enough to be easily added to the existing SSDs while X-FTL requires SSD to support complex concepts including transaction identifier, commit, and abort. Moreover, while CFS introduced new APIs such as `cfs_begin` and `cfs_end` to define the transactional scope, *AFS* utilizes the existing APIs such as `fsync()` and `syncv()` for that purpose. For this reason, the existing applications can be made more portably crash consistent with *AFS* than CFS.

Next, Park *et al.* [61] proposed Failure-atomic msync(), which can atomically update the changes of an mmap-ed file using REDO journaling, and Verma *et al.* [62] extended the work mainly in two directions; firstly, to avoid the redundant write, data blocks are managed in a CoW style, and secondly, to support the failure-atomicity for multiple files, they suggested the `syncv()` call. This work is unique in two folds: 1) they showed that the crash consistency can be achieved without help of special hardware, and 2) they proposed a small but elegant set of APIs for developing crash consistency applications. However, its CoW style data management will newly introduce the space fragmentation and thus require the costly garbage collection overhead. In contrast, our *AFS* is built with minimal changes in the existing ext4 file system with the help of *SHARE* interface, and thus we believe this would be more practical approach for crash consistent file system.

Address Remapping Our *SHARE*-aware *AFS* is not the first work on exploiting the address remapping for file system optimization, and here we will compare ANViL [24] and JFTL [73] with *AFS*. Weiss *et al.* [24] proposed a small set of storage APIs, based on address remapping at the *block device layer*, and showed that those primitives are useful in a variety of case studies, such as single-write journaling, snapshot, file copy, and de-duplication. In this respect, our *AFS* could not be regarded as unique at all, and is no less than implementing just one scenario of single-write journaling with a similar *SHARE* interface. But, we argue that our work of *AFS* is in stark contrast with ANViL from two perspectives. First, *AFS* shows for the first time that the application-level

crash consistent file system can be made practical with the help of atomic address remapping from flash storages. Second, from the perspective of performance, we think that the right place to embody the functionality of atomic address remapping is not the host-side block device layer as in ANViL, but the FTL layer as in *SHARE*.

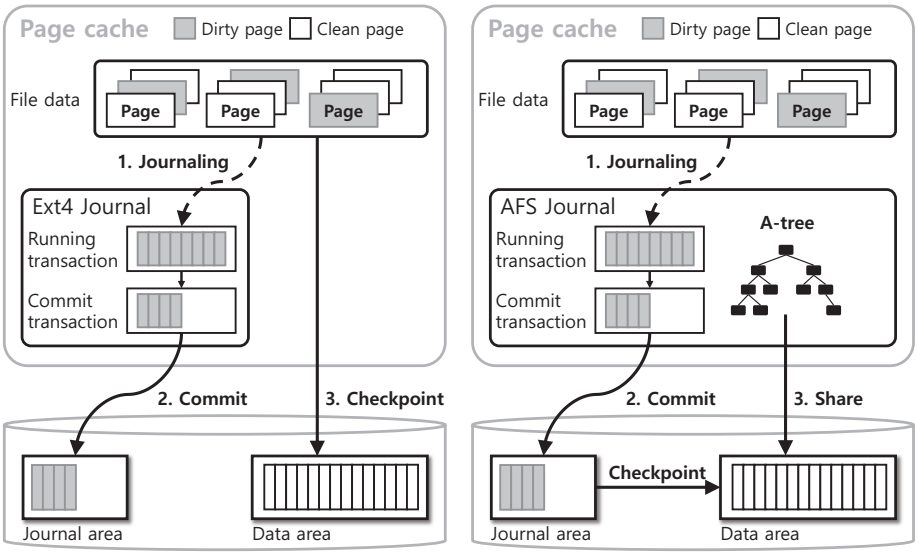
To our knowledge, JFTL [73] is the first approach to suggest the atomic address remapping functionality in FTL so as to avoid the redundant write overhead in journaling file system. In this sense, it is the closest approach to *AFS*. But, unlike *AFS*, the authors of JFTL did not consider the application-level crash consistency at all, and because it use a proprietary interface between host and flash storage for remapping only the journaled data, its capability of atomic address remapping, unlike ANViL and *SHARE*, is prevented from being fully and flexibly exploited in host systems [24].

3.4 Design of *AFS*

3.4.1 Overview

Guaranteeing crash consistency is one of the most important factors in designing a file system. But, there is a trade-off between consistency level and performance. For this reason, the ext4 file system takes a relaxed consistency, i.e., the ordered journaling mode (OJ), as its default mode. This mode of OJ provides *data consistency* [56], which only guarantees that metadata are entirely consistent to the data and that the same data read by a file legitimately belongs to that file. Therefore, under the OJ mode, a file can point to the older version of its data, which is the source of the well-known *torn page* problem in database systems. In contrast, the full data journal mode (DJ) supports *version consistency* [56], where the metadata version is guaranteed to match to the version of the referred data. But, this higher consistency in the DJ mode comes at the expense of considerable performance degradation due to double-write journaling of data as well as metadata.

Basically, *AFS* is based on ext4 journaling file system. But, unlike ext4 file system, one of its main design goal is to provide higher consistency at no compromise of performance. To satisfy this goal, *AFS* takes advantage of the atomic address remapping provided by *SHARE* interface at the flash storage layer, thus offloading the burden of guaranteeing system-wide version consistency from file system to flash storage and, at the same time, achieving higher performance almost for free without resorting to costly journaling scheme. In addition, by slightly modifying the existing data journaling mode of ext4 and also causing no extra run-time overhead, *AFS* can support higher application-level crash consistency as its first-class citizen functionality, thus freeing the applica-



(a) ext4 (b) *AFS*

Fig. 3-2: The overview of journaling process in ext4 and *AFS* file systems

tion developers from the burden of devising a complex and costly update protocol for application crash consistency.

Figure 3-2 illustrates the overview of journaling process in ext4 and *AFS*. As depicted in Figure 3-2, while the journaling and commit processes in *AFS* are almost same to those in ext4, the checkpoint process in *AFS* is in stark contrast with that in ext4. While the second write of each journaled block to its home location in ext4 happens, the second write of the same block is replaced by *SHARE* command in *AFS*. *AFS* uses an auxiliary red-black tree, called A-tree (atomic-tree), which helps to batch multiple journal writes into a single `share` command by keeping a set of LBA pairs of home location and journaled location on DRAM. When commit operation is triggered in *AFS*, each write operation for journaling is first recorded in the journal area and then the relevant LBA pair is inserted into A-tree for *SHARE* interface. At each checkpoint, *AFS* generates `share`

command by searching LBA pairs on A-tree belonging to the checkpoint transaction, and issues a `share` command to the underlying storage. At this time, the home locations in the storage are atomically shared with journaled locations at hardware level. Finally, for the next checkpoint, the previous LBA pairs in A-tree are discarded. In this way, *AFS* can avoid the unnecessary overhead caused by redundant journaling writes, which we call single-write journaling.

3.4.2 *SHARE*-aware Data Journaling (SDJ)

For efficient single-write journaling, *AFS* provides *SHARE*-aware data journaling (SDJ) mode that can achieve system-wide version consistency at the same performance of ordered journal (OJ) mode. In this section, we show in detail how *SHARE* interface can be leveraged to guarantee the highest data consistency (*i.e.*, version consistency). Figure 3-3 illustrates the SDJ procedure in comparison with ordered journaling (OJ) mode and data journaling (DJ) mode in ext4. When a commit operation is triggered by time (*e.g.*, 5 second) or a synchronous operation (*e.g.*, `fsync()`, `fdatasync()`, and `msync()`), OJ mode first asynchronously writes dirty pages (A' and B') to their home locations and then synchronously writes a journal descriptor (JD) block and metadata pages ($M_{A'}$ and $M_{B'}$) to the journal area. The JD block has the home locations of journaled metadata blocks [57] for recovery. After that, OJ mode writes a journal commit (JC) block together with the force unit access (FUA) command to the journal area, to mark the end of a journal commit transaction. Note that dirty metadata pages have not yet been written to the home location. These metadata pages asynchronously flush to their home location ($M_{A'}$ and $M_{B'}$) by either checkpoint or flush daemon. On the other hand, upon a commit operation, DJ mode in ext4 synchronously writes data (A' and B') and metadata blocks ($M_{A'}$ and $M_{B'}$) with a JD block to the journal area and then synchronously writes the JC block together with FUA command to the journal area. Since the same version of data and metadata blocks are in the journal area, DJ mode completely

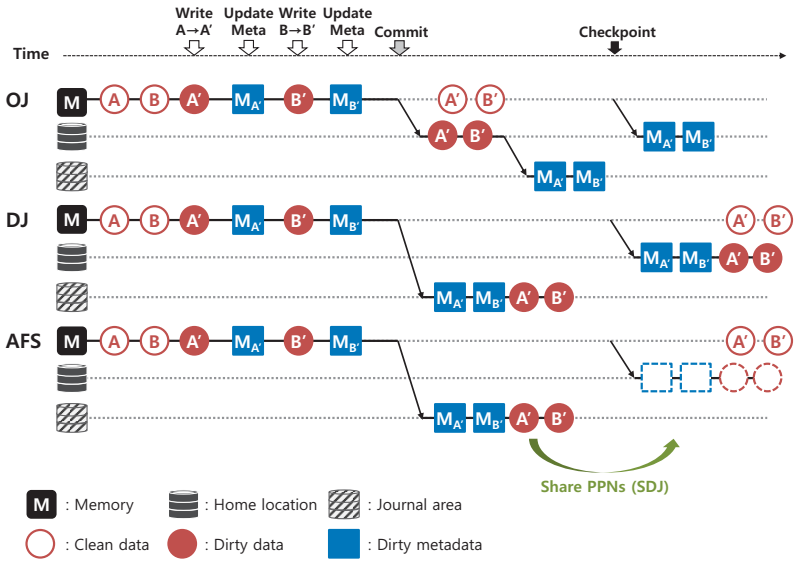


Fig. 3-3: Comparison of journaling modes in ext4 and *AFS*. Ext4 file system provides two journal modes: ordered journaling mode (OJ) and full data journaling mode (DJ). Checkerboard rectangles denote journaled blocks into the journal area. OJ mode in ext4 guarantees a minimal crash consistency: data and metadata of the file system will be preserved in order manner and flushed to the home location at checkpoint time (total 6 block writes). DJ mode in ext4 provides *version consistency*: data and metadata of the file system are synchronously logged in the journal area and then flushed to the home location by periodic checkpoint operation (total 8 block writes). SDJ in *AFS* follows the default rules of DJ mode except for checkpoint operation: data and metadata of the file system are synchronously logged and then reflected to the home location through *SHARE* interface (total 4 block writes with 1 `share` command).

guarantees the version consistency. Later upon a checkpoint operation, DJ mode asynchronously writes dirty pages (A′, B′, $M_{A'}$, and $M_{B'}$) to their home location of journaled blocks.

SDJ in *AFS* follows the similar steps to the DJ mode in ext4 until the commit operation is completed. At commit time, unlike DJ mode, SDJ fetches the LBA address for the home location (in which is preserved the persistent original data), and the LBA address for the journaled location (in which is kept the up-to-date data) from the `journal_head` structure, after each journal write operation is persistently placed in the underlying storage. A given LBA pair is inserted into A-tree

with integer key that is used to find the pair at checkpoint time. Once a checkpoint operation is triggered, SDJ first searches for a set of LBA pairs, which belong to the checkpoint transaction, on A-tree and then builds `share` command to send it to the storage. After finishing the checkpoint operation, SDJ discards not only a set of LBA pairs, which were reflected to the storage via the last `share` command, but also a set of invalid LBA pairs that were invalidated during the last commit operation. In this way, *AFS* builds more robust and reliable system-wide version consistency while improving the overall performance. Therefore, *AFS* uses this as its default consistency mechanism.

3.4.3 *SHARE*-aware Application-level Data Journaling (SADJ)

For some applications such as databases and key-value stores, even the system-wide version consistency by ext4 DJ mode, despite its double-write journaling, fails to meet their stringent requirements for transactional atomicity. For this reason, each application should devise its own application-level crash consistency mechanism. For instance, it is well-known that SQLite relies on costly application-level journaling mechanisms for transactional atomicity. However, such application-level crash consistency mechanisms bring about two problems. First, they usually suffer from poor performance and the reduced lifespan of the underlying flash storage mainly because of write amplification and frequent `fsync()` calls from the application layer. Second, the application-level complex update protocols are complex and error-prone so that, as is shown in recent studies [64, 65, 68], there still exist some subtle bugs even in widely-deployed applications. Therefore, it is imperative for file systems to support application-level crash consistency. In this section, we describe *SHARE*-aware application-level data journaling (SADJ), which can provide application-level crash consistency by slightly extending SDJ. It was designed with the following three goals in mind:

- The interface for using SADJ should be simple and intuitive so that application developers can easily adopt it.
- Legacy applications should be able to run together with SADJ-based one without any modification.
- The changes made in the existing file system should be minimal.

For an application to run in SADJ mode in *AFS*, it can use the failure-atomic update APIs (*i.e.*, `O_ATOMIC`, `syncv()`, and `msync()`) [61, 62]. Some applications (*e.g.*, SQLite [67]) require `abort()` protocols to roll back the changes to the most recent successful committed state. To this end, *AFS* newly introduced `abort()` and `abortv()` systems calls. Multiple files can be committed or roll-backed at once using `syncv()` or `abortv()`, respectively. These APIs can be easily incorporated to conventional applications with only a few lines of code changes. Once a file is opened with `O_ATOMIC` option and thus runs in SADJ mode, dirty pages of the file should be handled according to the following two rules.

- *Rule 1: When a checkpoint is triggered, each page should not be reflected to its home location even though it belongs to the checkpoint transaction.*
- *Rule 2: When an application executes synchronous operations (e.g., `fsync()`, `fdatasync()`, `msync()`, or `syncv()`), all dirty pages should be handled in the atomic manner of "all or nothing."*

In order to enforce these rules on-the-fly, there should exist a mechanism which allows us to easily determine whether each journaled block belongs to a file opened with `O_ATOMIC`. For this purpose, `JBD2_A_FLAG` was introduced. For every each block, `JBD2_A_FLAG` is stored both

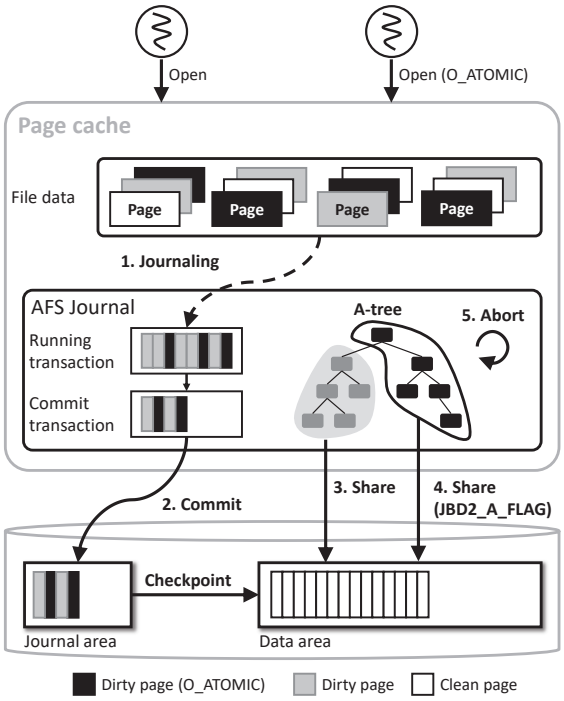


Fig. 3-4: Application-level data journaling in *AFS*.

in `journal_head` structure of the storage and in A-tree together with the relevant LBA pair at commit time. Now let us explain in detail how *AFS* can guarantee both the application-level crash consistency and system-wide version consistency with *SHARE* interface as shown Figure 3-4.

1. Journaling: Once a page on the page cache is written by an application, its status is changed from clean to dirty. Like ext4, SADJ in *AFS* inserts information for each dirty page, which keeps file data, to the running transaction.

2. Commit: At commit time, the running transaction is updated to the commit transaction. SADJ synchronously write all dirty pages of data and metadata belonging to the committing transaction to the journal area in the storage. Also, `journal_heads` of each dirty page are written

together simultaneously. While writing each dirty page to the journal area, SADJ checks whether the page belongs to the file that was opened with `O_ATOMIC`, and in that case SADJ sets its `JBD2_A_FLAG` inside `journal_head` structure and inserts a LBA pair with `JBD2_A_FLAG` into A-tree.

3. Share: When checkpoint is triggered, SADJ first searches for LBA pairs belonging to the checkpoint transaction in A-tree and then checks their `JBD2_A_FLAG`. If a `JBD2_A_FLAG` is set, SADJ skips the LBA pair and move on to the next for building `share` command based on the *Rule 1*. Finally, like SDJ, SADJ also discards a set of LBA pairs and a set of invalid LBA pairs in A-tree.

4. Share (`JBD2_A_FLAG`): Meanwhile, a set of LBA pairs whose `JBD2_A_FLAG` was set to 1 at commit time is sent to the storage via `share` command when an application calls `fsync()` to make the updates data of the file (which was opened with `O_ATOMIC`) persistent. After issuing the `share` command, SADJ discards those LBA pairs in A-tree for the next synchronous operation. Fortunately, *Rule 2* is satisfied by using the `share` command that supports atomicity at hardware level.

5. Abort: When an application calls `abort()` against a specific file, SADJ first searches for all dirty pages (which belong to the file) on the page cache and then fetches each journaled LBA address from the `journal_head` of each page. Finally, SADJ disrupts a set of LBA pairs by scanning A-tree for them using the searched LBA addresses. Note that SADJ does nothing for storage operations for rollback because journal blocks that contain data to be ignored were only placed in the journal area. Those journal blocks will be never reflected to the home locations because SADJ reuses them based on a round-robin order.

3.4.4 Technical issues of *AFS*

In comparison to ext4, *AFS* raises two technical issues about performance and correctness: optimal size of journal area and identification of valid journal block. Let us discuss each issue in turn.

Size of Journal Area: In *AFS*, the size of journal area has a huge effect on the overall performance because *AFS* triggers a checkpoint operation to reclaim journal blocks in the storage when it runs out of journal space. And, since *AFS* keeps data blocks as well as metadata blocks in the journal area, this performance issue is exacerbated when it uses the small-sized journal area (*e.g.*, 128MB). In addition, in terms of application-level crash consistency, perhaps some applications would likely need large journal area to guarantee their application-level consistency. To resolve this issue caused by small-size journal area, we decided to allocate rather large-sized journal area (*e.g.*, 1GB) which can be expected to preserve all data requiring application-level crash consistency. Of course, one possible solution to completely address the issue is to extend the fixed journal area in a dynamic way. We will leave it for our future work.

Valid Journal Block Identification: SADJ in *AFS* should maintain some journal blocks (which we call valid journal block) in the journal area for guaranteeing the application-level crash consistency. However, those journal blocks can be unintentionally over-written because a new journal block is assigned in a round-robin manner. To prevent such data corruption, SADJ allows skipping those journal blocks and allocates a new journal block in the journal area; this allocation is similar to the slack space recycle (SSR) of F2FS [74]. To achieve this, SADJ looks for a new LBA address in A-tree, which holds LBA addresses for all journaled blocks, before assigning a new journal block. If the new LBA address exist in A-tree, SADJ increments the LBA address until it finds an LBA address that does not exist in A-tree (*i.e.*, invalid journal block). Fortunately,

this situation hardly happens because journal area of *AFS* (*e.g.*, 1GB) is large enough to guarantee application-level crash consistency without any data corruption.

3.4.5 Recovery

In the event of system crash or application failure, *AFS* can completely guarantees both system-wide version consistency and application-level crash consistency. Basically, it takes the roll-back recovery in the ext4 journaling scheme. But the recovery process in *AFS* is unique in two aspects: (1) *SHARE*-aware zero-copy for all committed blocks with `JBD2_A_FLAG` disabled and (2) *A-property*, which states that all `JBD2_A_FLAG`-ed blocks can be safely ignored during the recovery process.

***SHARE*-aware zero-copy recovery:** During the recovery, For each journal commit block (JC) encountered while scanning the journal area, *AFS* finds its corresponding journal descriptor (JD) block [57]. Then, for all non-`JBD2_A_FLAG`-ed blocks between JD and JC, it generates and issues a `share` command, which contains pairs of LBAs, so as to keep data and metadata up-to-date. This step is repeated until the last commit block is encountered in the journal area. As in ext4, all the remaining blocks after the last commit block can be safely ignored from the recovery perspective. Considering that the redundant write of every journaled block to its home location in ext4 is replaced by a zero-copy `share` command in *AFS*, it is quite obvious that *AFS* can recover much faster than ext4. Note that this *SHARE*-based recovery process is idempotent and thus the recovery process can be simply repeated when another crash is encounter during the recovery.

A-property: Now let us explain the *A-property* in *AFS* recovery. It states that every `JBD2_A_FLAG`-ed block in the journal area can be safely ignored during the recovery. As stated

above, at the moment when an application calls `fsync()` against a specific file, every data or metadata block with `JBD2_A_FLAG` enabled belonging to the file will be propagated to its original location in an atomic manner by a `share` command, which will be executed only after the JC block for the `fsync()` is synchronously saved in the journal area. Therefore, it is guaranteed that all the blocks of a successfully `fsync()`-ed file is propagated to their original location. In case when the system crashed before the application is acknowledged for the `fsync()` call, each block belonging to the file should not be propagated to its original location. Consequently, all the `JBD2_A_FLAG`-ed blocks could be simply skipped during the recovery. Also, *A-property* makes it easy to implement `abort()` and `abortv()` APIs because these APIs do not need to remove the journaled blocks in the journal area at the time of `abort()`. In summary, this property is essential in making our *AFS* guarantee application-level crash consistency.

One interesting question with regard to *A-property* is whether it can also be embodied in the existing data journaling mode only if the concepts of A-tree and `JBD2_A_FLAG` are introduced, and the answer is no. In fact, *A-property* is a combined effect of taking all three techniques, A-tree, `JBD2_A_FLAG`, and *SHARE* interface. Let us assume that a file opened with `O_ATOMIC` is being updated by an application and for any reason one or more `JBD2_A_FLAG`-ed blocks from the file are already journaled before the application invokes the `fsync()` call to make its recent update durable. In this case, upon recovery, the existing data journal mode can not decide whether those blocks should be copy-backed to their original locations because it has no information regarding whether the `fsync()` call for those blocks succeeded.

3.5 Implementation

We implemented *AFS* in Linux kernel 4.6.7 by modifying 400 lines of code (LoC) of ext4 and JBD2. We note that other file systems supporting application-level crash consistency [62, 63] need significant changes (*e.g.*, 5,800 LoC for CFS [63]), which inhibit wide and rapid adoption of the new storage interface in practice. A-tree is implemented using a red-black tree maintaining mappings between a destination LBA and source LBA, associated file and page. We added two new `ioctl` flags for `syncv()` and `abortv()`. We ran all experiments on a system with a quad-core processor (Intel i7-6700) and 8GB memory.

We implemented a *SHARE*-enabled SSD by modifying an FTL firmware of a commercial high-end PCIe M.2 SSD supporting 360K and 280K IOPS for random read and write operations, respectively. And, since no matching command exists in current storage interface such as SATA and NVMe, the `share` command has been added as a *vendor unique command* (VUC) in the NVMe SSD. If you are interested in how to implement the `share` command atomically, please refer to [69].

3.6 Evaluation

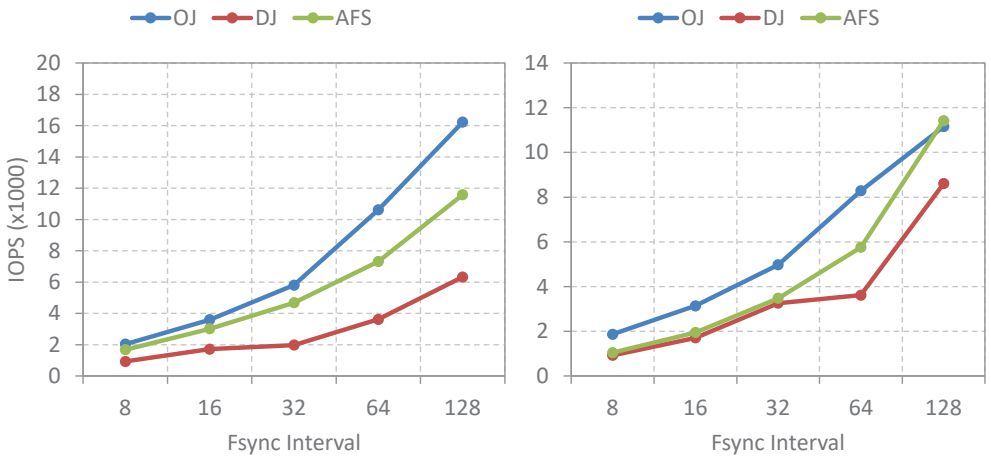
In this section, we present experiments that answers following questions:

- Does SDJ in *AFS* provide the high performance while guaranteeing the *version consistency*? (subsection 3.6.1)
- How much can *version consistency* of SDJ help to improve real application performance? (subsection 3.6.2)
- How well does SADJ in *AFS* guarantee application-level consistency? (subsection 3.6.3)

3.6.1 Effect of *AFS* on Microbenchmarks

Normally, microbenchmark is widely used to evaluate the performance impact of file systems. So, we used two microbenchmarks, Flexible I/O (FIO) [75] and Filebench [76] benchmark, to compare *AFS* with the conventional ext4 file system. We compare three journal mode: ordered mode (OJ) journal, data mode (DJ) journal, and SDJ in *AFS*.

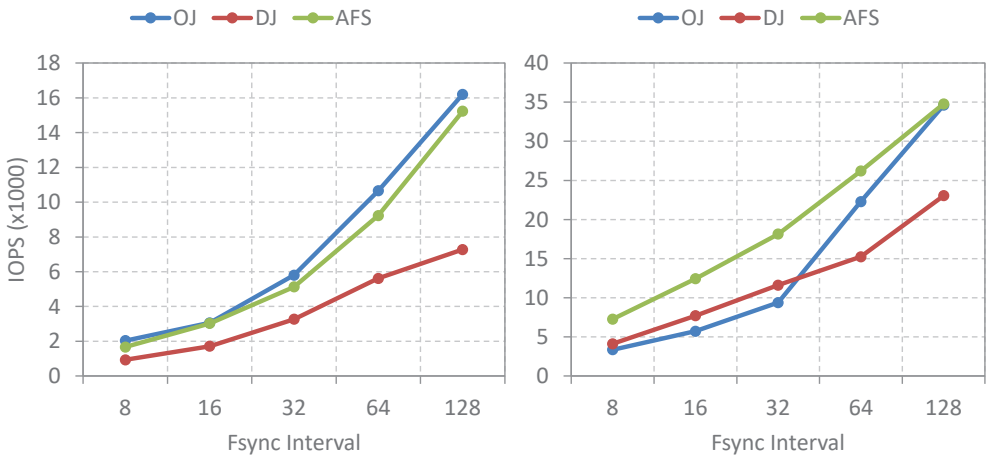
FIO microbenchmark: We first evaluated *AFS* using the FIO microbenchmark, which was configured to simulate data-heavy workload. We performed random writes 10GB of data with 8KB write granularity, and varied the number of threads and files to better investigate the performance of *AFS*. Since `fsync()` directly affects the amount of journal data and the overall performance, we ran the same pair of FIO with varying `fsync()` interval, which indicates the number of write operations between two consecutive `fsync()` calls.



(a) 1 job / 1000 files (b) 10 jobs / 100 files

Fig. 3-5: Performance comparison between *AFS* and ext4 for FIO microbenchmark (128MB journal size)

Figure 3-5 presents the throughput in IOPS for all the experiments when the journal size is 128MB, which is default for the ext4 file system. As we expected, this figure shows the notable performance gap between OJ and *AFS*. The major reason of the performance gap is that *AFS* frequently triggers checkpoint operations to reclaim journal blocks in the storage as mentioned in subsection 3.4.4, because it preserves both data and metadata in the journal area unlike OJ mode. To confirm this consideration at runtime, we monitored the block traces by using Blktrace while running the benchmark and figured out that *AFS* significantly increases the number of `fsync()` operations for a short time. For fair comparison, we extended the journal size to 1GB, as in previous work [71] and again evaluated the same pair of FIO (Figure 3-6). As expected, 3-6a clearly shows that *AFS* provides similar performance to OJ mode in all cases while it guarantees the system-wide *version consistency*. In addition, 3-6b presents that the performance of *AFS* outperforms that of OJ mode by up to 2.16x. Meanwhile, one interesting finding from the results is that



(a) 1 job / 1000 files

(b) 10 jobs / 100 files

Fig. 3-6: Performance comparison between *AFS* and ext4 for FIO microbenchmark (1GB journal size)

OJ mode reveals performance drop compared with DJ when it runs 10 FIO threads with short `fsync()` intervals, such as 8, 16, and 32 interval. This is because OJ mode suffers from the scalability issue of file systems [77] and random pattern writes. Another interesting finding is that 1GB journal size for *AFS* is large enough to hide the overhead (*e.g.*, frequent checkpoints) caused by the journal size while guaranteeing the system-wide version consistency.

Filebench microbenchmark: To emulate real-world I/O workload, we used the write-intensive Varmail and the read-intensive Webserver workload. The Varmail consists of 16 concurrent threads to simulate a mail server and each thread performs a set of create-append-sync and read-append-sync operations. The Webserver workload is also composed of 100 threads, each of which sequentially reads a whole file and then writes a small chuck of data. Figure 3-7 demonstrates the throughput of Filebench. From this figure, we can confirm that *AFS* works well in real-world workload. In addition, Even when the journal size is 128MB, the performance of *AFS*

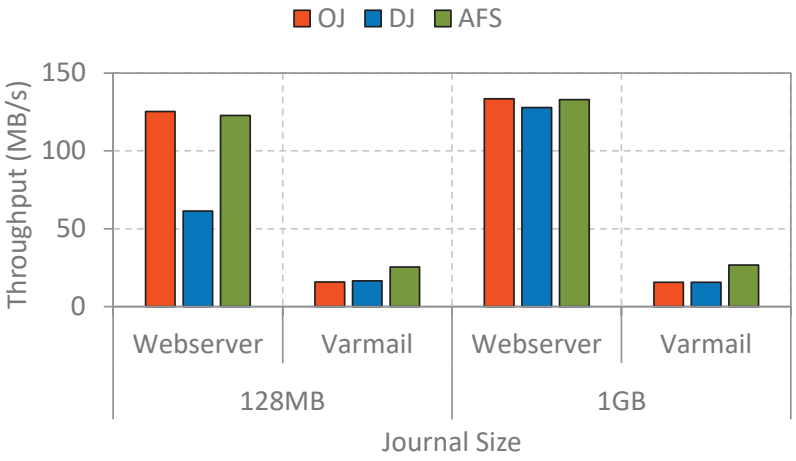


Fig. 3-7: Performance comparison between *AFS* and ext4

outperforms other modes in most cases. These results do not match to those of Figure 3-5. Therefore, we analyzed read and write performance, respectively. We found two reasons behind such improvements. First, *AFS* can improve the read performance by sequentially writing data in the journal area. In other words, *AFS* can maximize the effect of read-ahead within the flash storage (*i.e.*, internal parallelism). Second, *AFS* reduces the number of write operations without any redundant write as shown in Figure 3-3. This observation is indeed interesting because we had not expected any improvement of read operations with *SHARE* interface.

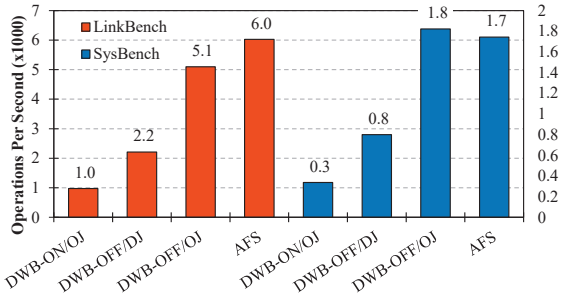
3.6.2 Effect of AFS on MySQL/InnoDB

The atomicity of each page write is a uncompromisable assumption in database storage engines because a *torn* page can not be restored even with the Aries-style recovery scheme. However, since modern file systems and storage devices do not generally guarantee page write atomicity, every database engine has its own update protocol to prevent the *torn* page problem. For example, the MySQL/InnoDB storage engine takes a variant of journaling, called *double-write-buffer* (for short,

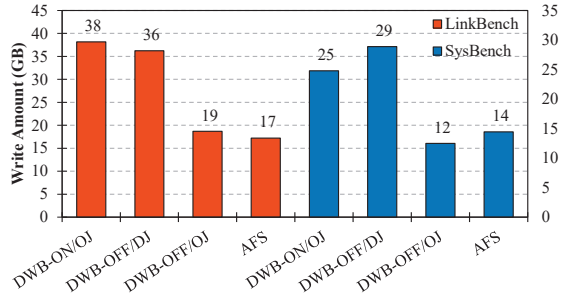
DWB) [66]: when a dirty page is replaced from the buffer cache, its new copy is first appended to a separate journal area, *double-write-buffer*, and then the old copy in its original location is overwritten. In each step, an `fsync()` call is made to enforce ordering and durability.

Because SDJ mode in *AFS* can guarantee the *version consistency*, MySQL/InnoDB on *AFS* is safe from the torn page problem even when the DWB mode is turned off. And due to the system-wide *version consistency* of *AFS*, the amount of writes to the storage is halved, and hence the performance could be doubled. Hence, to evaluate the effect of *AFS* on MySQL/InnoDB database, we ran two popular OLTP benchmarks, SysBench [78] and LinkBench [79] under four different modes: (1) `DWB-ON/OJ`(default), (2) `DWB-OFF/DJ`, (3) `DWB-OFF/OJ`, and (4) `DWB-OFF/SDJ`(*AFS*-based version). And, the results are presented in Figure 3-8. Note that the third mode `DWB-OFF/OJ` does not prevent the torn page problem while the other three modes do. We deliberately added the *crash-inconsistent* `DWB-OFF/OJ` mode to Figure 3-8 so as to stress that the *AFS*-based version can outperform the crash-inconsistent mode even in terms of performance.

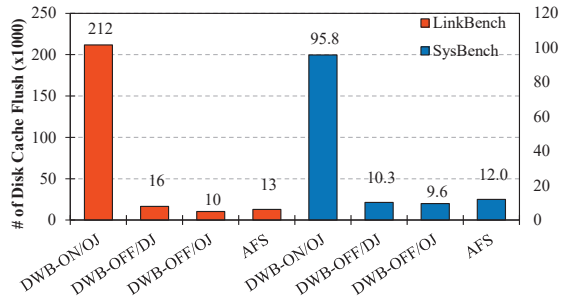
As 3-8a shows, the *AFS*-based MySQL outperforms the default mode `DWB-ON/OJ` by 6.16 times and the second option `DWB-OFF/DJ` by 2.73 times. This performance gain is, as is clearly shown in 3-8b, in part due to the write reduction by replacing the redundant write at either `DWB` or `DJ` mode with *AFS*'s single-write journaling. However, the wider performance gap between `DWB-ON/OJ` and `DWB-OFF/SDJ` modes can not be explained solely with the write reduction. The other main reason for the gap is the difference in the numbers of disk flush operations invoked in two modes. While the default `DWB-ON/OJ` mode, as explained before, calls `fsync()` in every step of database file writes and double-write-buffer write, the *AFS*-based version calls one disk flush after writing all database files together. Thus, as 3-8c shows, the *AFS*-based version invokes 16.4x less disk flush operations than the original version. In summary, MySQL/InnoDB can, by



(a) Performance



(b) Write amount



(c) Disk cache flush count

Fig. 3-8: OLTP benchmark results of Sysbench and LinkBench using MySQL. The original MySQL versions were tested in three different configurations: (1) `DWB-ON/OJ`(default), (2) `DWB-OFF/DJ`, and (3) `DWB-OFF/OJ`, while MySQL on *AFS* were in `DWB-OFF/SDJ`. SysBench in an OLTP mode (10 GB database (20 files) with 40 million rows for 1,000,000 operations. LinkBench: we ran 4,800,000 operations for a 50 GB database (24 files) after a two minute warm-up. In both experiments, MySQL/InnoDB engine was configured to use 5 GB as a buffer pool with sixteen concurrent threads, and all under buffered I/O mode.

offloading the responsibility for preventing the torn page problem to *AFS*, benefit significantly in terms of performance at no compromise of data consistency.

3.6.3 Effect of AFS on SQLite

SQLite is a light-weight library based DBMS widely used in mobile devices. Hence, unlike enterprise-class DBMS engines such as MySQL, it takes a less complicated page-oriented scheme for its transactional atomicity support: the force policy for commit and the steal policy for buffer replacement [72]. For this reason, when a transaction commits, all the updated pages (from single or multiple files) by the transaction should be atomically propagated to the storage. Please note that this requirement is more stringent than the data consistency provided in either ext4 DJ mode or MySQL/InnoDB's DWB scheme. Therefore, in order to meet this application-level crash consistency, SQLite takes costlier journaling modes of rollback journaling (RBJ) [80] and write-ahead-logging (WAL) [81].

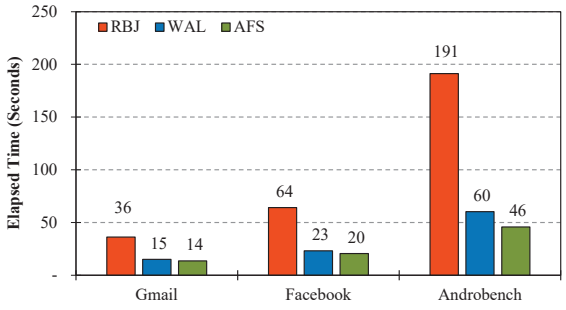
RBJ mode takes a *undo-based journaling* in that the original content of a page is copied to the rollback journal before updating the page. In contrast, the WAL mode takes a *redo-based journaling* in that the original content is preserved in the database and the modified page is appended to a write-ahead-log file. The change is then later propagated to the database by periodic checkpoint operation. Unfortunately, the WAL mode can not, although faster than the RBJ mode, guarantee the transactional atomicity when updates made by a transaction are spanning over multiple database files [81], and in this respect, it is an *incomplete* solution to the crash consistency.

In order to evaluate the effect of *AFS* on SQLite database, we ran a set of representative mobile traces in three different SQLite modes, `RBJ/OJ`, `WAL/OJ`, and `WRITEBACK/SADJ`. As noted earlier, our SDJ mode in *AFS* can not meet the crash consistency requirement in SQLite and thus the SAJD mode should be used instead. To guarantee the crash consistency in `WRITEBACK/SADJ`

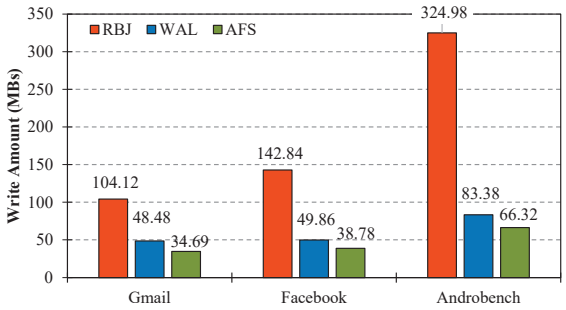
mode, where any journaling mode of SQLite is turned off, the only change made in SQLite source code is to add `O_ATOMIC` flag to an `fopen()` call which opens SQLite database files. We used three SQLite traces, and one trace is a synthetic AndroBench [82], and the other two real traces are collected from running Facebook and Gmail applications on an Andriod 4.1.2 Jelly Bean SDK. And the experimental results are presented in Figure 3-9.

As shown in Figure 3-9, the *AFS*-based version outperforms the two SQLite journaling modes consistently over all three workloads by approximately 3.3 times and 1.2 times, respectively. It is well known that the RBJ mode suffers from its double-write journaling and excessive `fsync()` calls [8, 72]: the frequent `fsync()` calls are in part due to journal file creation/deletion per every transaction, and are in part necessary to guarantee the durability of database and journal files, and also to ensure the strict write ordering between those two files. As a result, it is not surprising to see from 3-9b and 3-9c that the original RBJ mode generates about 3.8x more writes and 3.2x more disk cache flush operations than the *AFS*-based version.

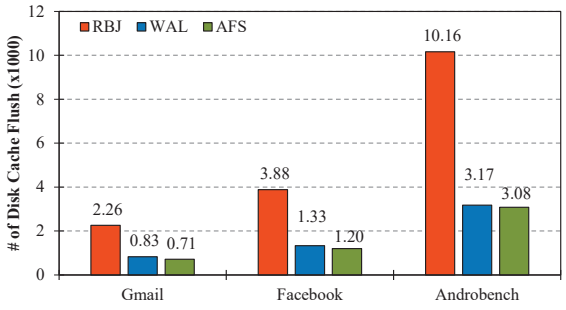
In the WAL mode, the updated pages are appended to a WAL file and then they are later checkpointed to the database file. For this reason, when the database size is relatively small and the workload has clear locality in write patterns, the write amplification in the WAL mode could be significantly smaller than that in the RBJ mode. And because the WAL file is reused once created, the `fsync()` calls are not made so frequently as in the RBJ mode. As a result, as shown in 3-9b and 3-9c, the WAL mode generates less writes and disk cache flush operations than the RBJ mode consistently over all three traces. This is why the performance gap between *AFS*-based version and WAL mode is rather marginal. However, it should be recalled that the original WAL mode in SQLite does not guarantees the crash consistency against multiple files, while *AFS*-based SQLite and the RBJ mode do guarantee.



(a) Performance



(b) Write amount



(c) Disk cache flush count

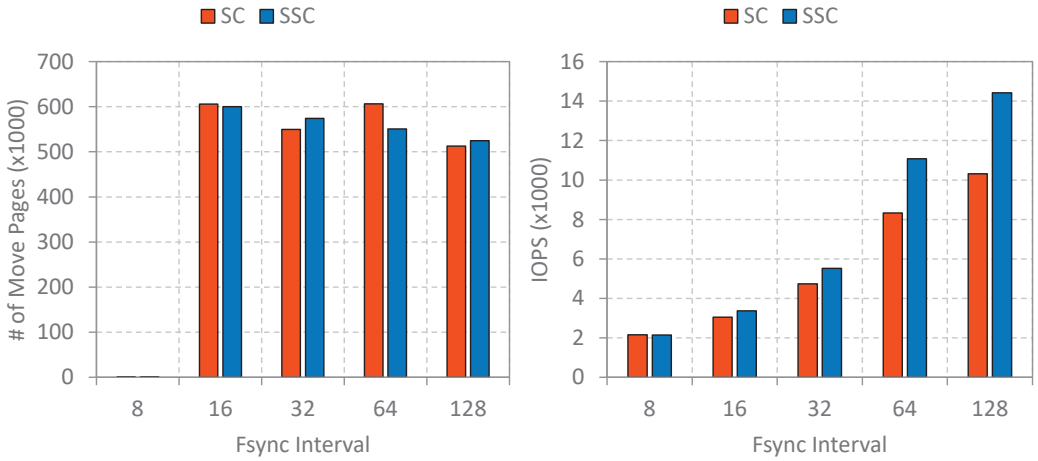
Fig. 3-9: SQLite Performance: RBJ vs. WAL vs. *AFS*. The original SQLite database (version 3.8.13) were tested under three different modes : 1) `RBJ/OJ`(default), 2) `WAL/OJ`, and 3) `WRITEBACK/SADJ`. A set of three mobile workloads was used in the experiment: Facebook, Gmail, and AndroBench.

Before closing this subsection, we would like to stress that with *AFS*, the existing application can be easily made crash-consistent. Please recall that the SQLite `WRITEBACK` mode becomes crash-consistent by adding one flag to the `fopen()` call in SQLite. In contrast, in version 3.8.13 of SQLite, the RBJ and WAL mode consist of about 14,500 lines of code, and CFS also requires to add 38 lines of its system calls in SQLite source code [63].

3.7 Discussions

As we briefly discussed in section 3.1, *SHARE* can flexibly support various uses cases, including journaling and log-structured writing. In this section, we explore another use case of *SHARE* in log-structured file system (LFS) [59]. A log-structured writing scheme is widely adopted for flash storage devices but it still suffers from enviable segment cleaning overhead to get large chunk of free space. Various techniques, such as data grouping [23], slack space recycling [83], and in-place-update (IPU) mode in F2FS [74], have been proposed but none of them remove copy-back overhead of valid blocks. With incorporating *SHARE* with the segment cleaning procedure, we can fundamentally remove the copy-back overhead of valid blocks. Instead of copying valid blocks in a victim segment to a new segment, we simply call *SHARE* from the victim segment to the new segment for segment cleaning.

We implemented this *SHARE*-aware segment cleaning (SSC) scheme by modifying F2FS 100 LoC. For evaluation, we first filled up the file system utilization to 50% of total space. Then we performed FIO benchmark, which was configured to perform random writes to 40% of total storage capacity, by varying `fsync()` interval from 8 to 128. Figure 3-10 shows the total number of move pages during the segment cleaning and the performance results of each segment cleaning. 3-10a shows how many valid pages are moved during the segment cleaning. Interestingly, when `fsync()` interval is 8, SSC and SC do nothing. This is because current F2FS was modified to allow an in-place update when the `fsync()` interval is smaller than 16. Meanwhile, 3-10b demonstrates that SSC outperforms the original segment cleaning by 10%–39%. In SSC, copy-back overhead of data blocks are completely removed and only meta data blocks, such as segment information



(a) # of move pages

(b) Performance

Fig. 3-10: Performance comparison between segment clean (SC) and *SHARE*-aware segment cleaning (SSC). The performance results were measured using FIO benchmark.

table, node address table, and segment summary area, are updated. Figure 3-10 clearly confirms the possibility that *SHARE* interface is easily adopted by other file systems.

3.8 Summary

We have presented *AFS*, which natively supports both *system-wide version consistency* and the *application-level crash consistency* on flash storage with an atomic address remapping interface, called *SHARE*. Our *AFS* can relieve the applications of the burden of guaranteeing their crash consistency as well as data consistency, and boost the application performance because of its single-write journaling and less frequent `fsync()` calls from applications. Therefore, with *AFS*, consistency-critical applications do not need to devise complex, tardy, error-prone update protocols by themselves. The existing applications can be ran without any changes under *AFS*, while enjoying the higher data consistency, and they can be easily made application-level crash-consistent simply by opening their data files in `O_ATOMIC` mode. In addition, the single-write journaling in *AFS* will double the life span of flash storage. We have prototyped *AFS* by modifying *ext4* file system with only minimal changes, and also have implemented the *SHARE* interface inside a commercial SSD as firmware. Using the *AFS* prototype and the *SHARE*-enabled M.2 SSD, we have carried out a set of synthetic and realistic benchmarks. Our experimental results show that *AFS*-based applications are 2–6x faster than their original versions.

4. LDJ: Version Consistency Is Almost Free on Flash Storage Devices

4.1 Introduction

The *metadata consistency* guarantee has traditionally been opted for in designing or configuring file systems [56], since it provides the file system consistency at high performance (*e.g.*, ordered journaling mode in ext4 and in XFS [58]). Under *metadata consistency*, however, a file can point to an older version of data after a system crash or power failure. That is, the consistency in metadata does not necessarily guarantee the consistency of data itself pointed to by the metadata. This kind of weak consistency has forced many applications, such as SQLite, MySQL, and CePH, to devise their own ways to guarantee the application-level crash consistency. To mitigate the situation, some researchers have begun shifting towards a *version consistency* [56], which ensures that the metadata correctly points to its data (*e.g.*, data journaling mode in ext4, logging in LFS [59], and copy-on-write scheme in btrfs [60]). But, the existing schemes can achieve the *version consistency* only at the cost of significant performance degradation such as redundant journaling and cleansing [24, 61–63, 71, 84]. For instance, the double writes of data pages in journaling-based file system will, particularly in flash-based storage devices, have adverse effect on the performance and lifespan of storage devices.

Meanwhile, during the last decade, we have witnessed a few intriguing trends in the storage market. First, the capacity of each individual storage device has exponentially grown. As a consequence, it is not uncommon nowadays to see flash SSDs of 16 TB [5]. Accordingly, the size of the file system has also been increased significantly. Second, the ever-evolving storage interfaces and related techniques have continuously made the read and write operations on the storage more lightweight. For example, SATA 3.0 interface provides the throughput of 6 Gbps, two orders of magnitude faster compared to SATA 2.0 [3]. Third, flash SSDs have been rapidly becoming another main-stream storage device. One intrinsic characteristic of flash SSDs is that sequential write pattern is more preferable to random one in those storage devices. These trends, interestingly, provide us an opportunity to revisit the data journaling mode of file system for higher performance; the data journaling mode can be optimized so as to perform very comparably to, and in some cases, to outperform the ordered mode, while preserving its higher consistency level.

In this chapter, we propose a simple but effective optimization scheme for data journaling mode in ext4, called *lightweight data journaling* mode (*LDJ*). The main ideas of *LDJ* are as follows: (1) Since the sequential write bandwidth in SSD is much larger than the random one, the performance overhead of sequentially writing data itself, as well as metadata, to the journal area would be marginal. (2) By enlarging the size of journal area (*e.g.*, from 128 MB to 5 GB) and thus making the checkpoints triggered more lazily, we can alleviate the overhead of the so-called *forced checkpoint* [84] and thus the foreground transactions will commit quickly without being blocked any more by the sluggish checkpoint [84]. (3) By compressing journal blocks on-the-fly prior to writing them to the journal area, we may reduce the amount of data to be journaled, which in turn will shorten the write completion, prolong the lifespan of flash SSDs, and, most importantly, make checkpoints further delayed with the same size journal area. In addition, the compression enables the atomicity of each journal write without issuing an intervening `FLUSH` command be-

tween journal data blocks and commit block, thus halving the number of costly `FLUSH` calls in *LDJ*.

While limited when each idea is individually applied, the performance improvement is quite significant when all the three ideas are applied in combination. In particular, we made an observation that the problem of *forced checkpoint* could not be perfectly solved simply by enlarging journal area [84]. Instead, when combined with the compression, the enlarged journal area allows the *foreground* committing transactions to proceed their journal write operations, while perfectly overlapping them with the *background* checkpoint operations. From the technical perspective, this is in stark contrast with the recent *ext4-lazy* scheme which would still suffer from the *forced checkpoint* while running applications with frequent `fsync()` calls on top of high-end SSDs. The main contributions of this chapter can be summarized as follows:

- We made an observation that the technological characteristics and trends in storage devices provide an opportunity to revisit the data journal mode for further performance optimization.
- We have designed and implemented the *lightweight data journaling* mode (*LDJ*) by slightly modifying the existing ext4 with jbd2 for journaling and also *e2fsck* for recovery (*i.e.*, less than 300 lines of source code were added or updated).
- We carried out comprehensive performance evaluations by running four standard benchmarks and two real application workloads in three different modes of OJ mode, DJ mode, and *LDJ* mode, on top of the state-of-the-art storage devices including Samsung SSD, and intel NVMe SSDs. Our evaluation results confirm that the overall performance of *LDJ* is comparable or even better than that of ordered mode, while preserving version consistency.

The rest of the chapter is organized as follows. section 4.2 describes the journaling mechanism and investigates the IO behaviors of each journal mode. section 4.3 gives our motivation based on

the experimental performance results of the traditional journal modes on the latest SSD. section 4.4 and section 4.5 describes the details of *LDJ* design and implementation, respectively. section 4.6 shows our evaluation results and section 4.7 compares *LDJ* with prior work. Finally, section 4.8 provides the conclusion.

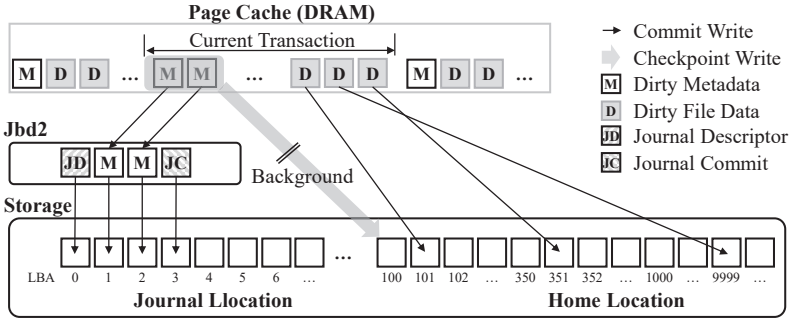
4.2 Background

Guaranteeing crash consistency is one of the most important factors in file systems. In this section, we first review two journaling modes of ext4 file system, focusing on the commit procedure of each mode taken to guarantee its crash consistency. We then characterize and compare the two modes from the perspective of several technical issues of each commit procedure.

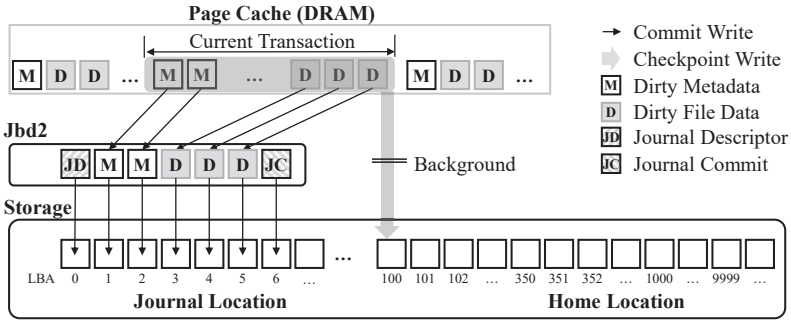
4.2.1 Journaling Mechanism of Ext4

Ext4 file system is commonly used in modern computing systems, such as enterprise, desktop, and mobile systems, and its journaling layer in Linux (*i.e.*, jbd2) is the key component to ensure crash consistency upon a system crash or power failure. As a compromise between the performance and consistency, ordered journaling (OJ) is used as the default journaling mode in Linux; it journals only metadata and guarantees only the *metadata consistency*, therefore writing less data onto the storage and performing faster than the data journaling (DJ) mode. In contrast, DJ mode guarantees higher level data consistency, so called *version consistency*, as well as metadata consistency, by journaling data blocks too.

Figure 4-1 shows how each mode works to guarantee its consistency level upon *commit* which is triggered either by pre-defined time (*e.g.*, 5 seconds) or one of the synchronous operations (*e.g.*, `fsync()`, `fdatasync()`, and `msync()`). On a commit in OJ mode, as illustrated in 4-1a, jbd2 first reflects each modified data block to its home location in a synchronous way, and then issues a series of journal blocks, including a journal descriptor (JD) block, journal data blocks (metadata only), and a journal commit (JC) block, to the journal area. We would like to note that, during



(a) Ordered journaling mode on ext4



(b) Data journaling mode on ext4

Fig. 4-1: The traditional journaling procedure of jbd2.

commit operations in OJ mode, the write pattern onto the storage device tends to be *random* since the logical block addresses (LBAs) of the home location of modified data blocks are likely to be scattered. This random write pattern in OJ mode may, as will be explained in subsection 4.2.2, have an adverse effect on the block layer in Linux and the storage media.

In contrast, in DJ mode, when a commit operation is triggered, as illustrated in 4-1b, jbd2 writes a series of journal blocks, including a JD block, journal data blocks (metadata and data), and a JC block, to the journal area. Therefore, the dominant write pattern onto the storage device under the DJ mode will be *sequential*, and any random write is not in the critical path of committing transactions. All the metadata and data blocks once journaled will be asynchronously copy-backed to their home locations later when a checkpointing is triggered. In DJ mode, every metadata and data blocks are written twice in theory for higher consistency, and this write amplification is the main cause of performance degradation and will also shorten the lifespan of the flash memory. But in practice, if metadata or data block with strong update locality is committed two or more times prior to the next checkpoint, the write amplification ratio could be lowered because only the latest copy of the block need to be copy-backed to its corresponding home location. Unfortunately, both modes suffer from two consecutive `FLUSH` commands during committing a transaction; the first `FLUSH` is issued to the storage for ensuring the order between journal data blocks and journal commit block before writing the commit block, and then the second `FLUSH` command is issued to make journal blocks *durable* right after writing the commit block.

In either mode, once a metadata or data block is successfully written to the journal area by jbd2, its durability is ensured even though the block has not yet been propagated to its home location; the block is recoverable upon crash. For this reason, the propagation of blocks to their home locations can be safely delayed to the next checkpoint which is triggered either in pre-defined interval (*e.g.*, 5 minutes) or by a kernel thread (*e.g.*, pdflush in Linux). The checkpoint will be

carried out by a dedicated kernel daemon, which writes all the dirty blocks to their home location asynchronously, and this *background checkpoint* will not pose severe overhead on the committing foreground processes.

4.2.2 Performance-related Issues in Ext4 Journal Modes

With the background in Section 2.1 in mind, we now compare the characteristics of the two journaling modes in ext4 and discuss a few critical technical issues outstanding in each mode mainly from the performance perspective, as summarized in Table 4-1. As discussed before, although DJ mode writes more than OJ mode, the write pattern in DJ mode tends to be sequential while that in OJ mode is inherently random.

Let us first discuss the technical issues resulting from the random write pattern in OJ mode. It is well known that the random pattern incurs noticeable overhead on the underlying IO stack layers [9, 10, 23, 25, 84]. Therefore, the random write pattern which is dominating in OJ mode will make the storage performance optimization hard. For example, the random writes give unintended overhead to the block layer in Linux, mainly because the block layer has to spend a lot of time in making an object of block IO (*i.e.*, *bio* structure), and also in finding an opportunity for merging the incoming block IOs even when the IOs eventually cannot be merged due to their random LBAs. In addition, the random write pattern is harmful in SSDs; it will cause costly garbage collection overhead in SSDs [9, 10, 85]. In contrast, DJ mode does not suffer from these technical issues because the dominating write pattern in the mode tends to be sequential.

Meanwhile, the amplified journal write in DJ mode raises two technical issues different from the ones above. The first and most crucial one is the *forced checkpoint* problem [84]. In DJ mode, a small-sized journal area (*e.g.*, 128 MB) will be quickly filled up with data as well as metadata blocks before the background checkpoint is triggered, especially when the applications seldom

	OJ mode	DJ mode
Consistency level	Metadata	Version
Write pattern	Random	Sequential
Write amount	1x	2x
IO stacks	Block layer	–
Jbd2	–	FC [84]
SSDs	GC	Endurance

Table 4-1: Ordered vs. data journal mode in ext4 jdb2. "GC" and "FC" indicates the garbage collection and forced checkpoint, respectively.

issue the `fsync()` calls. In this case, jbd2 will force the checkpoint operation to reclaim free space in the journal area. And, unfortunately, during such a *foreground checkpoint*, a committing transaction has to wait until some amount of free space in the journal area is secured. For this reason, the checkpoints in DJ mode does not proceed in background mode; instead they are blocking the committing transactions and thus become the critical path on the performance of committing transactions. This phenomenon is called as *forced checkpoint* [84]. The second technical issue in DJ mode is that the double amount of writes can halve the endurance of flash memory storage.

4.3 Motivation

With the tremendous growth in storage industry, new storage technologies and devices and technologies have been relentlessly developed during the last decade: SATA 3.0 [3] and PCIe/NVMe interface [4], internal parallelism in flash storage, data recording [86], large and/or battery-backed write buffer [87, 88], and so on. Today, they are commonly used for optimization of the storage performance [9,27,84,89,90]. In order to understand the performance trends of both journal modes in ext4 on the latest storage devices and also to get an insight about the limitations and opportunities of DJ mode in comparison to OJ mode, we measured the performance of both modes while running three standard file system benchmarks, Varmail, Fileserver, and Dbench, on top of one representative storage device of Samsung 256 GB 850 PRO SSD (see Figure 4-2).

The experimental results in Figure 4-2 reveal both the promises and challenges of DJ mode together. For SSD, while DJ mode slightly underperforms OJ mode in case of Dbench workload, the former even slightly outperforms the latter in case of Varmail workload. This promising result of DJ mode is consistent with the observations made in other recent research [91]. Both Varmail and Dbench workloads are common in that `fsync()` calls are made frequently. Under the heavy `fsync()` calls, the random write pattern in OJ mode will have adverse effect on performance while the sequential write pattern in DJ mode, despite of its doubled amount of writes of data blocks, makes its performance comparable to that of OJ mode.

Meanwhile, from the performance result of Fileserver workload in Figure 4-2, we know that DJ mode is still quite inferior to OJ mode consistently for both storages. Fileserver benchmark is a write-intensive workload with almost no `fsync()` call so that DJ mode suffers from both re-

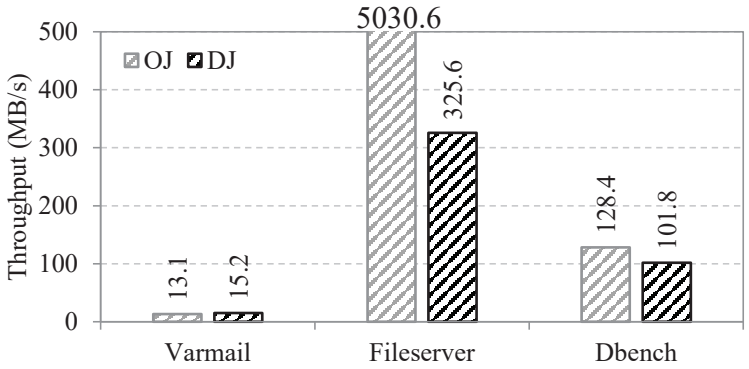


Fig. 4-2: Throughput of two journal modes in ext4, ordered journal (OJ) and data journal (DJ), while running three standard benchmarks under the latest SSD (Samsung 850 PRO 256 GB).

dundant writes and excessive *forced checkpoints* [84]. It is clear from this discussion that reducing both the amount of data to be written to the journal area and the frequency of *forced checkpoint* is critical in further optimizing DJ mode. It is also evident that the sequential write pattern in DJ mode is helpful in making DJ mode perform comparably to OJ mode. These are the main motivations behind the *lightweight data journaling* (*LDJ*) mode.

4.4 Design of *LDJ*

One of the key objectives in designing *LDJ* is to trade expensive IO costs in DJ mode with less-expensive CPU costs by compressing journal blocks, while preserving the *version consistency*. To achieve our design goal, we extends the DJ mode of ext4 file system and utilizes *compression-properties* of the lossless compression algorithms (*e.g.*, Lempel-Ziv [92, 93]). In this section, we first describe *compression-properties* (subsection 4.4.1) and present the journaling procedure of *LDJ* in detail (subsection 4.4.2). Finally, we introduce the *false recovery* and explain *self-trimming* scheme that completely prevents the false recovery (subsection 4.4.3).

4.4.1 Compression-properties

Compression techniques are widely used in various fields, including storage systems and computer networks, to reduce the amount of space needed to store or to transfer data [94–96]. In storage systems, the compressed data can increase the efficiency of data transfers between the host and storage, reduce space requirement on the storage and, in case of flash memory, can shorten the write completion time and prolong the lifespan of the storage. Now, let us elaborate on two important properties that the journal compression in *LDJ* provides in optimizing the data journaling.

Compression-property 1. The first *compression-property* is that the compression reduces the amount of space needed to store data. Obviously, this property allows *LDJ* to reduce the amount of journal writes by compressing the whole data that need to be journaled. In turn, this reduction is obviously beneficial to the commit latency and the lifespan of flash storage devices; (1) it can shorten the commit latency in *LDJ* by reducing the number of IO operations (*e.g.*, insert/merge/is-

sue operations on the *bio* structure) and by mitigating the interference of IO traffic, and (2) it can prolong the endurance of flash storage devices by reducing the amount of data itself to be written to the storage and the resulting reduction of garbage collection overhead inside flash storage.

Compression-property 2. The second property that the compression technique provides is about the *write atomicity* of the compressed journal data (*i.e.*, all-or-nothing). This *compression-property* is inherited from the *lossless compression* algorithm that is based on Lempel-Ziv [92, 93] and fixed Huffman coding [97]:

- The compression algorithm first finds the repeated literals while scanning raw data sequentially, and the matches are encoded into the dictionary that is composed of a set of pairs of literal length and matched length
- The decompression algorithm, while reading the dictionary from the compressed data, restores the original raw data by referencing the literal length and matched length; if one of the matches on the dictionary is omitted (or removed), decompression fails to rebuild the raw data.

An important implication of this property is that *LDJ* can achieve the write atomicity and durability of a compressed journal spanning multiple blocks only by issuing one FLUSH command after write all the blocks. That is, if the decompression procedure succeeds in restoring the original data from a compressed journal, it means that all the blocks belonging to the compressed journal were successfully written. Otherwise, it means that one or more blocks belonging to the compressed journal were not properly written to the storage. In DJ mode, two FLUSH commands are required to ensure the ordering between journal data blocks and journal commit block and their durability [56].

4.4.2 Journaling Procedure

LDJ is simple and straightforward but it completely guarantees *version consistency* and efficiently improves the overall performance by utilizing the *compression-properties* in committing the transactions (*i.e.*, the running transactions in Linux). Now, we describe *LDJ* under ext4 file system in detail. Figure 4-3 shows the journaling procedure of *LDJ* with an example.

Compressed Commit. When a commit operation is triggered either by pre-defined time (*e.g.*, 5 seconds) or one of the synchronous operations (*e.g.*, `fsync()`, `fdatasync()`, and `msync()`), *LDJ* first journals the whole dirty data belonging to the current transaction, including modified data and its metadata (see ❶). Then, *LDJ* transforms the raw data to the *compressed journal* by compressing the data blocks along with the corresponding journal metadata blocks, journal descriptor and journal commit block (see ❷); one *compressed journal* consists of compressed data (variable length) and its size (4 Byte), and it is stored over one or more multiple blocks aligned at a page granularity (*e.g.*, 4 KB). Such a compression is advantageous in committing the transaction compared with the traditional OJ mode, in that *LDJ* can reduce the amount of storage writes when a large amount of raw data is transformed to a smaller amount of *compressed journal*; if the compression ratio is more than half a percent (*e.g.*, 50%), *LDJ* can halve the amount of writes compared with OJ mode. Finally, *LDJ* starts writing a series of blocks belonging to one *compressed journal* into the journal area on the underlying storage device (see ❸).

Note that, because the write request for each block belonging to the *compressed journal* is made to the storage device individually at page granularity, the atomic propagation of all blocks of the *compressed journal* to the storage is not guaranteed at all either by the storage device itself or by the kernel. But, recall that *LDJ* can ensure the atomic write of each *compressed journal* to the journal area according to the *compression-property* 2. Therefore, *LDJ* can achieve the atomicity

as well as the durability for the *compressed journal* by calling only one FLUSH command after issuing the write call for its final block. In contrast, under the DJ mode, two FLUSH commands should be called to guarantee the atomic write of one journal data [56]. The first FLUSH command should be called, prior to writing the journal commit block, to ensure the write ordering between the journal data blocks and the journal commit block. And, after writing the journal commit block, the second FLUSH command should be called to ensure the durability of the commit block.

Of course, the benefits of the compression in *LDJ*, such as reduction in the amount of writes and halved `FLUSH` calls, do not come for free - the compression of the journal data requires CPU power and time, both of which are not negligible. That is, there exists a trade-off between the IO benefit and the CPU overhead. In many realistic applications, as will be demonstrated in section 4.6, the benefits of write efficiency by compression and higher-version consistency in *LDJ* far outweigh the additional CPU overhead taken for compression. In addition, the write efficiency that comes from compression in *LDJ* will be more outstanding when combined with the lazy checkpoint, which is described below.

Lazy Checkpoint. During the last decade, we have witnessed the capacity of single storage device has been growing exponentially [98]. However, we believe that existing journaling mechanisms (*i.e.*, jbd2) fail to take full advantage of an opportunity out of the ever-growing storage capacity. That is, although it is now affordable to invest much more space as journal area for better performance, the size of journal area in jbd2 is still set to 128MB by default, which we believe is too small and thus is the root cause for performance detriment in DJ mode (*i.e.*, *the forced checkpoint*). This observation led us to extend the size of journal area in designing *LDJ*, and one obvious benefit of this enlarged journal area is that the *forced checkpoint* operations will be naturally deferred in proportion to the size of the journal area.

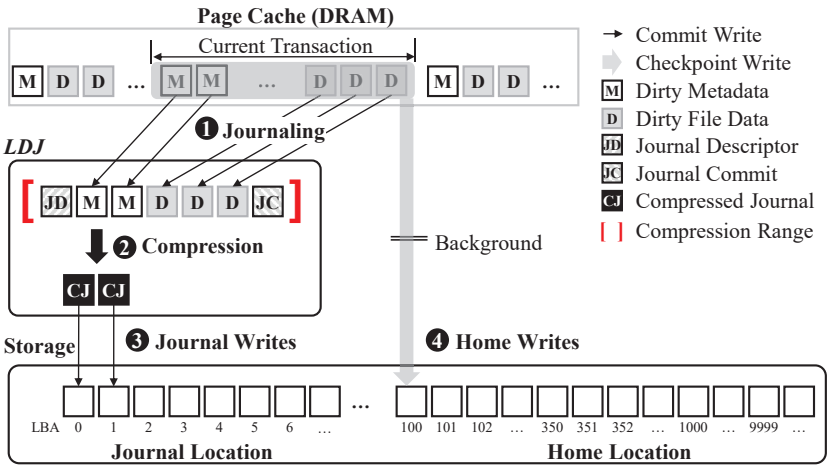


Fig. 4-3: The journaling procedure under ext4 with *LDJ*. *LDJ* compresses modified metadata and data on-the-fly (❶ – ❷) and issues the compressed journal instead of raw data (❸). Finally, modified metadata and data are reflected on their home location by background checkpoints (❹).

Basically, the checkpoint procedure in *LDJ* proceeds equally as in DJ mode of jbd2. During checkpoint, a background job flushes every journaled block of either data or metadata from the buffer cache to its home location (see ❹). The enlarged journal area in *LDJ* will trigger the checkpoints in a much longer interval than the DJ mode and thus the *forced checkpoint* can be transformed to a background activity. In contrast to the baseline checkpoint in DJ mode, this lazy checkpoint in *LDJ*, though much more IOs have to be carried out upon every checkpoint, fortunately will not impose a significant burden on the file system because the checkpoint operation is a background activity. More importantly, the compression in *LDJ* allows to further delay the checkpoint operations because more transactions can be journaled in a compressed format against the same size of journal area. This in turn reduces the costlier *forced checkpoints* and thus the total amount of writes and the number of FLUSH commands are significantly decreased.

Crash Recovery. In the event of system crash or power failure, *LDJ* can still guarantee its *version consistency*. Basically, the recovery process in *LDJ* is exactly same to the UNDO-based recovery in the jbd2, only except for the additional step of *journal decompression*.

As explained before, *LDJ* keeps a set of *compressed journals* in the journal area, and each *compressed journal* consists of compressed data (variable length) and its size (4 Byte). Therefore, upon recovery, for each *compressed journal* in sequence starting from the start of the journal (*i.e.*, `s_start` in Linux), *LDJ* first reads its compressed size to find out the offset of its compressed data and then restores the original data by decompressing the compressed data. This step of decompressing each *compressed journal* repeats until the decompression of the current *compressed journal* fails due to the dictionary mismatch according to compression-property 2. The decompression-failed compressed journal will be simply discarded from the recovery because its one or more blocks were lost out of the crash and thus its corresponding transaction can be safely regarded as invalid. After finishing decompressing all valid compressed journals, *LDJ* then, likewise the DJ mode, scans each decompressed journal until reaching its journal commit block and replays every valid transactions having its own commit block but not checkpointed yet in sequence.

4.4.3 False Recovery and Self-trimming

Unfortunately, since the journal data is appended to the journal area just in a round-robin fashion, *LDJ* may cause data corruption and inconsistency by recovering incorrect data. We call such a recovery as *false recovery*. For example, let us assume that a head pointer of the journal is reset to the first offset in the jbd2 because the journal area was already full of the previously written *compressed journals*. Then, let us suppose that, at the next commit time, *LDJ* issues three blocks from a *compressed journal* to the journal area: **CJ1**, **CJ2**, and **CJ3**. And further assume that a system crash was encountered after only two of the three blocks had been stored in the journal

area (*e.g.*, CJ1 – **CJ2** – **CJ3**, **CJ1** – CJ2 – **CJ3**, or **CJ1** – **CJ2** – CJ3). If the dictionary from the up-to-date durable blocks (*i.e.*, two black boxes) coincidently matches with the dictionary from the outdated version of a *compressed journal* (*i.e.*, one red box), the recovery procedure in *LDJ* might recover incorrect data. This false recovery might result in data corruption and inconsistency. Unfortunately, we cannot assume that the false recovery, though very rare in practice, will not happen in *LDJ*.

In order to completely prevent the *false recovery*, we decided to take a lightweight cleaning mechanism, called *self-trimming*, that erases the older versions on the journal area. By default, immediately after each checkpoint ends, *self-trimming* is asynchronously carried out as a background activity which issues a set of TRIM commands (or 0x00) against the checkpointed region in the journal area. With *self-trimming* enabled, *LDJ* can now detect a mismatch of the dictionary under the above scenario because literal and matched length stored on the black boxes will now point to the trimmed blank block, filled with 0x00, instead of outdated block. In this way, the *false recovery* problem can be avoided in *LDJ*. Note that the overhead of *self-trimming* is negligible because it run as a background task and thus will not block other foreground operations such as committing transactions and checkpoint operations.

4.5 Implementation

We have implemented *LDJ* by extending jbd2 journaling layer in ext4 file system slightly. In addition, 13 lines of code (LoC) were added to ext4 to make three alternative schemes in *LDJ* (*i.e.*, compressed commit with self-trimming (CMP), lazy checkpoint (lazy), and LDJ) available as mount options in ext4. Now, we will elaborate how *LDJ* is implemented on jbd2 layer.

To speed up the (de)compression, numerous hardware-assisted (de)compression techniques have been developed [94, 99], and it is well known that the hardware-based approach can outperform the software-based one by 20x or more [99]. But, in this chapter, we decided to use the software-based compression technique for two reasons. The first one is to implement *LDJ* easily and the second is to show that the performance of *LDJ* is comparable to that of OJ mode even on commodity servers with no extra support for (de)compression. We used the optimized compression algorithm, lz4, that presents the best compression speed and ratio among modern compression algorithms.

To compress all the dirty blocks that are not yet committed, *LDJ* allocates a large amount of contiguous DRAM space (*e.g.*, 1 GB) at boot time. The memory space is divided into two buffers: `I_BUF` and `O_BUF`: The former is used to hold raw data to be compressed and the latter to store the results of the compression (*i.e.*, *compressed journal*). To compress the whole data of a committing transaction, *LDJ* first copies both the raw data and journal metadata (*i.e.*, journal descriptor and commit block) to the `I_BUF` and then calls `lz4_compress()` along with `O_BUF`. Note that, in the resulting *compressed journal*, the first 4 bytes at offset 0 keeps the size of the *compressed journal* and the variable-sized real content of the *compressed journal* follows the size information. Next,

likewise the existing ext4 journal modes, *LDJ* crafts log blocks to store the *compressed journal* by calling `jbd2_journal_next_log_block()` and copies the contents of the `O_BUF` to the log blocks. Finally, *LDJ* writes the *compressed journal* to the journal area. In total, we modified 171 LoC of jbd2 layer.

Meanwhile, we can embody *LDJ*'s *lazy checkpoint* without any source code change simply by setting the size of journal area to 5 GB at mount time[1)]. In order to implement the *self-trimming* functionality, we need to know the target range to be trimmed in the journal area. For this purpose, 11 LoC was added to the existing checkpoint code to find the range of journal LBAs covered by the current checkpoint transaction: the `Head` and `Tail` pointers on the journal area[2)]. Against this range, a set of `TRIM` commands is issued in sequence by *self-trimming* which is implemented as a *kernel worker* thread; it only required 14 LoC modification.

In general, *e2fsck* is used to verify the ext4 file system because it can detect file system corruption from various system crashes or power failures. In addition, *e2fsck* is responsible for restoring a damaged file system to a durable and consistent state [100]. We have implemented the recovery procedure of *LDJ* by adding 85 LoC to e2fsck [101]. As mentioned in subsection 4.4.2, the recovery procedure of *LDJ* involves one additional step of *journal decompression*. The *journal decompression* process is carried out in the reverse order of the compression process along with `lz4_decompress_safe()`. It first reads 4 Bytes to find the offset of the compressed data and then restores the raw data and journal metadata by scanning and decompressing the compressed data. In order to store the restored journal data, the *journal decompression* process requires an extra memory space. The *journal decompression* process repeats until the decompression fails or

1) the default journal size is changed by performing "*tune2fs* -J size=5120"
2) the `Head` and `Tail` was captured by `s_start` of the journal super block at the start and end time of the checkpoint operation, respectively

reaches the last *compressed journal*. Finally, *LDJ* replays data belonging to the transaction, which was committed but not yet checkpointed, to guarantee crash consistency.

	NVMe-A	NVMe-B	SSD
Vendor	Intel	Intel	Samsung
Model	750	DC 3600	850 PRO
Interface	PCIe 3.0	PCIe 3.0	SATA 3.0
`fsync()`	Disabled	Disabled	Enabled
Bottleneck	CPU	CPU	`fsync()`
Performance	Very high	High	Medium
Price / GB	High	High	Medium
Capacity	400 GB	400 GB	256 GB

Table 4-2: The specification of storage devices used in our experiments.

4.6 Evaluation

We evaluate *LDJ* by driving the following questions:

- How does *LDJ* affect the overall performance with standard benchmark and real-world workloads? And is it efficient enough in guaranteeing *version consistency*? (subsection 4.6.2)
- How effective, in compression time and ratio, is the compression in accelerating the performance of *LDJ*, and what are the limitations in the current version of *LDJ*? (subsection 4.6.3)
- How much recovery time does *LDJ* require, including the decompression time of the *compressed journal*? (subsection 4.6.4)

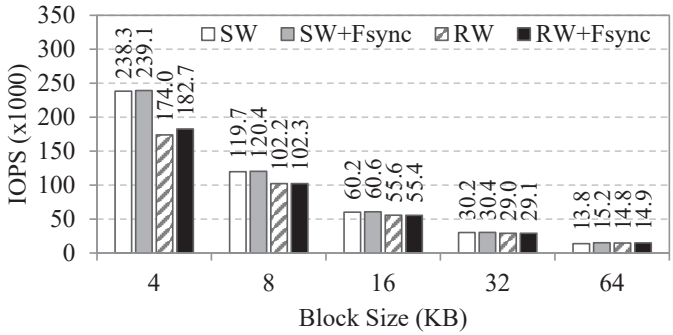
4.6.1 Experimental Setup

To perform a comprehensive evaluation, we validated *LDJ* with four standard benchmarks and two real-world applications on top of three up-to-date commercial storage devices. Now, we present our experimental setup and workload in detail.

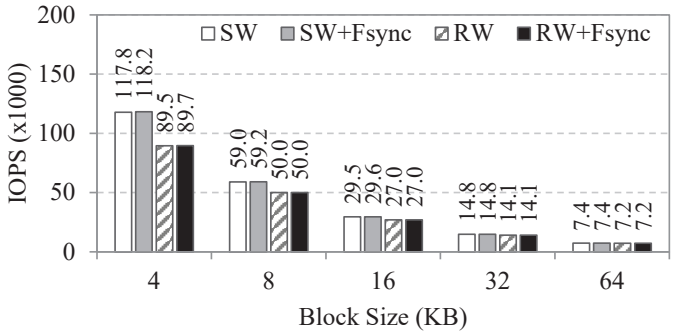
Hardware. We conducted all experiments on a machine with Intel Xeon E5-2620 processors of 2.1 GHz with 8 cores and 128 GB DRAM running the Linux kernel 4.9 on the Ubuntu 14.04.5 LTS.

For accuracy, we used three different types of storage devices as listed in Table 4-2. Figure 4-4 shows the baseline performance of each storage device. The performance is measured by running FIO benchmark while varying the block size[3)]. The storage devices involve different characteristics and they can be classified into two storage types: *Fsync-disabled* and *Fsync-enabled*. Some costly NVMe SSDs, NVMe-A and NVMe-B in Figure 4-4, use a small battery-backed memory as their write buffer inside the device, and thus, no device-level `FLUSH` is needed. For this reason, the device driver for such storage devices omits the `FLUSH` flag in flight when writes are followed by `fsync()`. In addition, such Fsync-disabled devices provide high IO performance (*e.g.*, multi-GB/s IO rates) using PCIe interface and multiple queues for I/O commands. Therefore, with these devices, the performance of CPU may be the bottleneck of the storage devices. On the other hand, the performance of SSD in Figure 4-4, is dominantly affected by `fsync()` and their block layer (*i.e.*, Blk-mq in Linux) handles SATA devices differently from PCIe ones. Consequently, their performance is worse than NVMe SSDs. Especially, the storage performance might be notably degraded under `fsync()` intensive workloads.

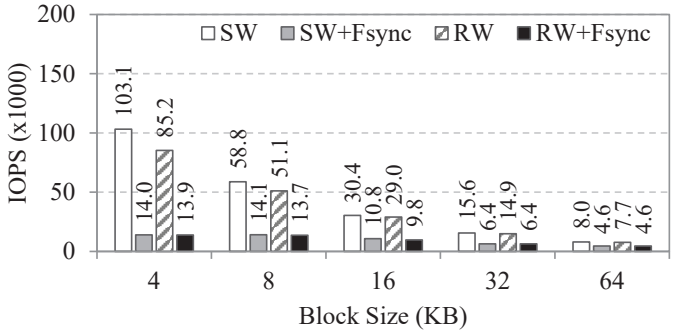
3) –ioengine=libaio –iodepth=64 –size=50GB –direct=1 –fsync=0 or 100



(a) NVMe-A



(b) NVMe-B



(c) SSD

Fig. 4-4: The write performance comparison of FIO benchmark. "SW" and "RW" indicates sequential write and random write, respectively.

	R/W IO size	# of files	R/W ratio	# of thread
Varmail	1 MB / 16 KB	1000	1/1	16
Webserver	1 MB / 16 KB	1000	10/1	100
Fileserver	1 MB / 16 KB	10000	1/2	50
Fileserver-MF	1 MB / 16 KB	50000	1/2	50

Table 4-3: The characteristics of Filebench workloads.

Workloads. To evaluate the effectiveness of *LDJ*, we first used a well-known standard benchmark tool, Filebench 1.5 version [76], that supports to evaluate the performance of file systems along with data reduction techniques (*e.g.*, data duplication and compression); this version generates different types of data unlike its earlier version that fills data to be written with all zeros or some arbitrary values [102]. This new functionality is especially important in that *LDJ* uses the compression technique as its main feature. To demonstrate *LDJ* under various conditions, we carefully selected four workloads in the Filebench: Varmail, Webserver, Fileserver, and Fileserver-MF. Table 4-3 summarizes the characteristics of each workload. Varmail is one of the `fsync()`-intensive workloads because it generates many small writes and then issues `fsync()` system calls frequently. On the other hand, Webserver emulates web server workloads that mostly reads and rarely appends data (*i.e.*read-intensive), while performing the operations in random way. Meanwhile, Fileserver is a write-intensive workload that involves file operations such as create, append, delete, etc. Finally, Fileserver-MF is processed exactly the same way as the Fileserver except that it handles more files.

To confirm how *LDJ* can improve the performance of real workloads, we evaluated *LDJ* using three real-world applications: Git, OLTP, and File-copy (*i.e.*, cp). In general, Git is primarily used for source code management and generates a lot of IO activities when pulling source code from the server. To reduce the variance of network bandwidth, we installed Git on top of our local server that is connected to the local network and measured the time to clone data on the remote repository,

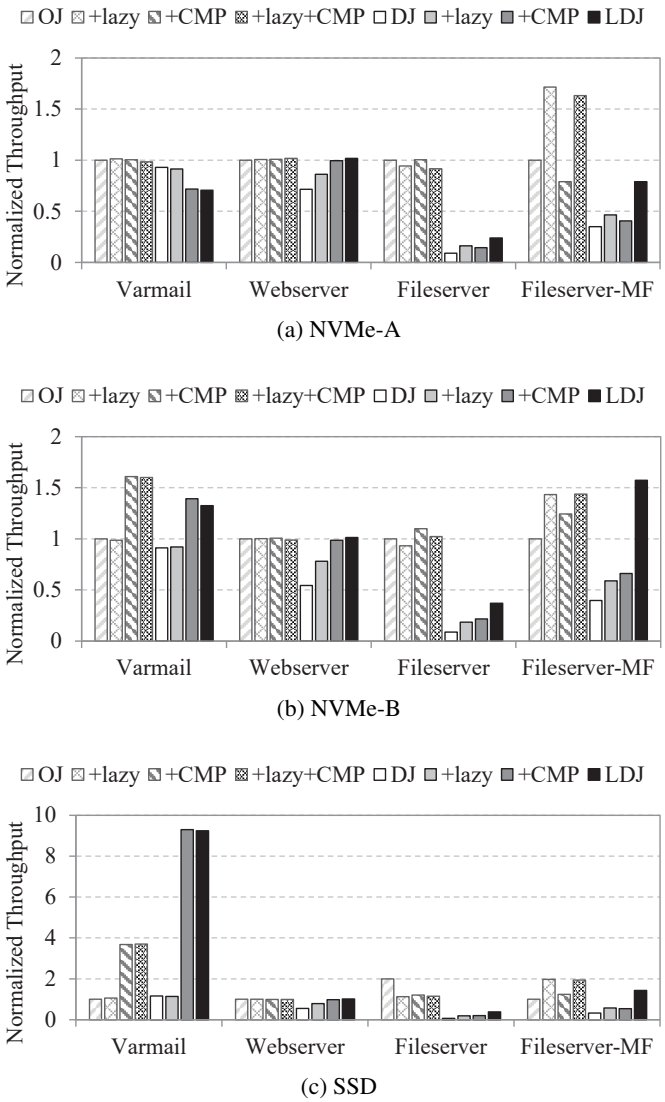


Fig. 4-5: Throughput of each workload on four different storage devices; the results are normalized to OJ mode. Note that "+" sign means that an additional feature was added to the base journal mode (*i.e.*, +lazy and +CMP indicate that the base journal mode adopted lazy checkpoint and compressed commit with self-trimming, respectively.)

where 19 GB of the Linux kernel source is stored. As OLTP worklod, we ran Sysbench [78] benchmark on top of MySQL [66]. This is because it generates IO intensive online transactions and supports sophisticated concurrency control protocol to ensure isolation among the threads. For Sysbench in an OLTP mode, we configured it with InnoDB engine, having 128 concurrent threads, and 40 GB database that is composed of 128 files with 1 million records. Both applications generate write-intensive workloads but they are different in that OLTP frequently issues `fsync()` calls. To demonstrate a worst-case scenario, we performed the copy (*i.e.*cp) operation along with 5000 JPG files (total 20 GB); the JPG file is one of the uncompressed data types.

Methodology. For a fair comparison, we initialized the target storage by performing a series of operations, such as `fdisk()`, `format()`, and `mount()`, right before each evaluation. Especially, at format time, we enabled discard option and disabled lazy initialization options[4)] and delayed allocation option[5)] because they would give disturbance to our experiments. Each experiment was run for 10 minutes and it was repeated 3 times to stabilize the results. Basically, we compared *LDJ* with the traditional journaling modes, ordered journaling (OJ) mode and data journaling (DJ) mode. Furthermore, to accurately analyze the evaluation results, we evaluated and compared *LDJ* with the journaling modes optimized by adding additional features; "+" sign is used as the prefix for the additional feature in the rest of this section. For example, +lazy indicates that base journal mode adopted lazy checkpoint feature. Finally, in case of the OLTP workload, we differently configured the option for *double-write-buffer* according to the journaling mode because it is primarily used on top of the OJ mode to complement the limitation of *metadata consistency*; the option is enabled on the OJ mode, but it is disabled on the DJ mode.

4) the options are configured by passing "-E discard,`lazy_itable_init` =0,`lazy_journal_init` =0" to mkfs.ext4
5) the option is disabled by passing "nodelalloc" to mount

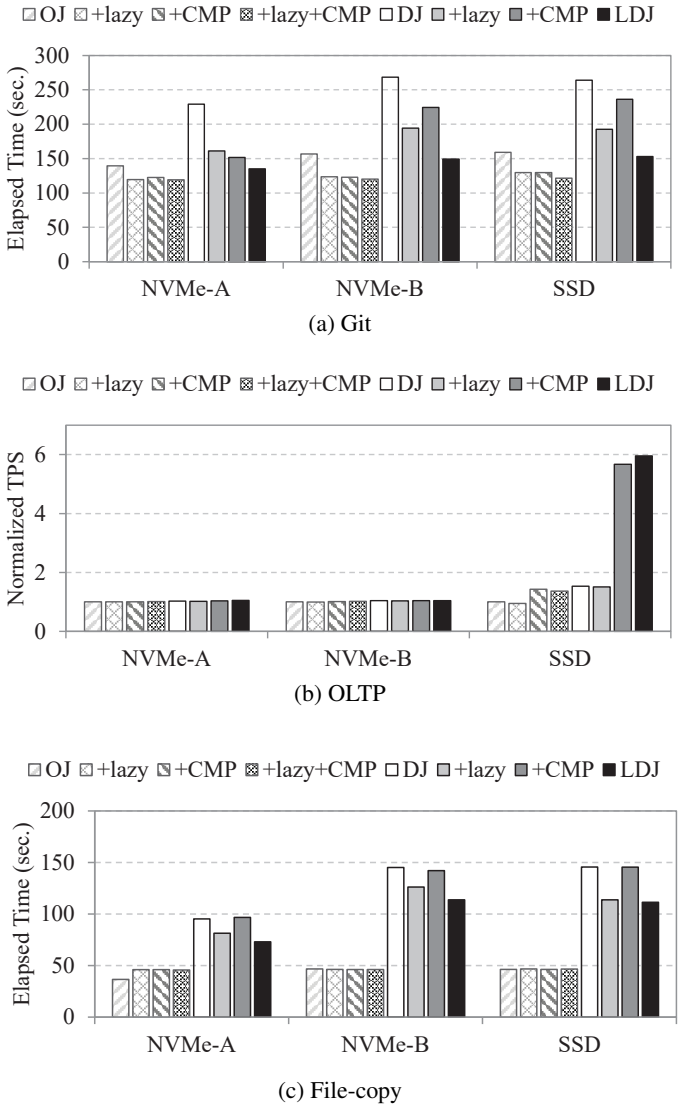


Fig. 4-6: The performance results of each journaling mode with three real-world applications.

4.6.2 Performance Comparison

We first start from the performance for understanding the effectiveness of *LDJ*. To accurately account the benefits of our scheme, we ran the existing journal modes along with additional features: +lazy, +CMP, and +lazy+CMP.

Standard Workloads. In order to emulate the IO activities in server environments, we first evaluated *LDJ* by running a set of server workloads in Filebench. Figure 4-5 presents the throughput of each workload on four storage media; the results are normalized to OJ mode. The performance results in Figure 4-5 shows that *LDJ* achieves performance improvement over OJ mode on most storage devices although *LDJ* completely guarantees *version consistency*. However, *LDJ* unfortunately performs comparable to or even worse than the OJ mode in case of the NVMe-A. These results are reasonable in that the compression cost of CPU is more expensive than the IO cost of NVMe-A storage. We will study this trade-off in more detail in the following sections.

Meanwhile, under other storage devices, *LDJ* improves the throughput of each workload compared with the existing journaling modes except Fileserver workload. These surprising results are explained by the fact that modern storage devices prefer to write data in a sequential way and the *compression-property* definitely gives *LDJ* the opportunities to accelerate storage performance. Especially, in Varmail, *LDJ* shows by up to 9.2x higher throughput compared to OJ mode, and by up to 8x compared to DJ mode. The reason behind this improvement is that *LDJ* can halve the number of `FLUSH` commands in every commit procedure due to *compression-property* 2. In case of the NVMe-B device, the results show little performance gap according to journaling modes and configurations despite of frequent `fsync()` calls. This is possible because the device does not require the device-level `FLUSH` command as mentioned before. Figure 4-5 also verifies that *LDJ* completely ensure the *version consistency* with high throughput that is comparable to OJ

mode in all storage devices. Surprisingly, NVMe-B and SSD reveal a performance collapse for Fileserver workload, but we believe this is acceptable because OJ mode only journals the metadata that includes the file information of data and thus 128 MB journal area is large enough. In this experiment, our assumption can be directly confirmed via the evaluation of Fileserver-MF that handles 50,000 files; *LDJ* improves the performance of SSD by up to 40% compared with OJ mode.

Real Application. Now, we focus on three real-world applications, Git, OLTP, and File-copy, to further explore the opportunities of *LDJ*. Figure 4-6 presents the performance results of *LDJ* on the real-world applications; 4-6a shows the elapsed time that was measured while cloning data from the server to the local storage device, 4-6b reports the normalized results for OLTP workload, and 4-6c shows the elapsed time of copy operation.

Unlike the results in Figure 4-5, Figure 4-6 shows that *LDJ* outperforms OJ mode in all storage devices and workloads. Such consistent results are especially important in that the compression cost of *LDJ* is negligible and acceptable in the real-world environments even though it journals both data and its metadata for *version consistency*. Also, 4-6a and 4-6b clearly confirm that *LDJ* takes 3%–47% speedup on Git and 0.04%–5.9x performance improvement on OLTP, compared with OJ mode. We believe that this performance benefit comes from the reduced amount of writes by compressing journal data on-the-fly as well as sequentializing write pattern, which is more important in low-end storage devices, such as SSD. On the other hand, as we expected, *LDJ* reveals worse performance than OJ mode in 4-6c. This is because *LDJ* should consume a lot of time for compression of JPG files even though the files are the uncompressed format. However, as shown in 4-6c, *LDJ* outperforms DJ mode by up to 23%. Meanwhile, the trends of Figure 4-6 is very similar to those of Figure 4-5; Git and File-copy have similar trends to the Fileserver or

	Benchmark	OJ	+lazy	+CMP	+lazy+CMP	DJ	+lazy	+CMP	LDJ
NVMe-A	**Varmail**	518.0	501.6	278.1	268.8	520.6	481.7	101.0	98.1
	Webserver	95.9	94.2	95.8	95.7	132.3	159.3	91.5	94.1
	Fileserver	141.5	141.5	140.1	140.5	374.2	216.6	36.7	12.7
	Fileserver-MF	467.2	504.7	566.7	503.5	497.2	417.9	159.3	57.3
	Git	19.6	19.2	18.5	18.5	38.6	38.3	25.0	24.6
	OLTP	85.8	84.2	63.6	63.6	100.3	103.1	59.5	59.1
	File-copy	20.34	20.34	20.34	20.34	40.77	40.75	40.73	40.72
NVMe-B	**Varmail**	268.7	268.3	245.1	262.8	269.7	269.5	100.2	98.3
	Webserver	94.4	95.1	95.2	95.1	99.4	144.6	91.1	94.0
	Fileserver	135.1	134.7	133.9	133.8	299.9	162.8	38.0	12.7
	Fileserver-MF	272.9	297.9	294.3	295.5	328.1	307.0	166.3	57.2
	Git	19.6	19.2	18.5	18.5	38.5	38.3	25.0	24.6
	OLTP	83.8	7.0	82.7	7.0	62.6	8.1	62.8	7.8
	File-copy	20.34	20.34	20.34	20.34	40.77	40.75	40.73	40.72
SSD	**Varmail**	8.2	8.3	15.2	15.1	10.6	10.3	22.5	22.7
	Webserver	94.1	93.8	94.0	93.9	96.9	147.6	91.4	93.5
	Fileserver	141.5	141.5	140.8	140.4	204.3	172.3	35.4	12.6
	Fileserver-MF	268.4	316.1	304.3	315.5	302.7	313.1	132.2	57.1
	Git	19.6	19.2	18.5	18.5	38.5	38.3	25.0	24.6
	OLTP	101.8	9.6	98.4	9.4	58.9	18.8	57.7	18.8
	File-copy	20.34	20.34	20.34	20.34	40.77	40.75	40.73	40.72

Table 4-4: The amount of storage writes that were captured with the *blktrace* while running each benchmark. The unit is GB.

Fileserver-MF and OLTP shows similar trends to Varmail. The reason behind these trends is that they have similar IO behaviors and `fsync()` intervals, while running each workload.

4.6.3 Performance Analysis

As described in subsection 4.6.2, *LDJ* is highly effective on all kinds of commercial storage devices. In this section, we present a variety of analyses on our experiments to deeply understand the feasibility and effectiveness of our *LDJ*.

Amount of Writes. We first start our performance analysis on the amount of writes because it is significantly correlated with the performance of *LDJ*. To achieve this, we captured whole writes of each workload, issued to the underlying storage device, by using *blktrace* tool.

Table 4-4 lists the amount of writes captured while running each benchmark on different storage devices. As shown in Table 4-4, *LDJ* notably reduces the amount of writes compared with OJ mode while guaranteeing the *version consistency* (except Git workload). These results of *LDJ* are quite different from those of DJ mode; DJ mode always performs on average 46%+ more writes than OJ mode. However, we expected these results before performing the measurement because *LDJ* can transform a large amount of raw data (*i.e.*, data and metadata) to a small size compressed journal. In addition, such a transformation further reduces the amount of writes issued when `fsync()` is called to force data consistency and durability. The reason is that *LDJ* compresses both data and its metadata needed to be durable and then writes *compressed journal* whose size is definitely smaller than original data; unfortunately, OJ mode should write the whole data to its home location to make a small metadata durable. For this reason, it can describe why *LDJ* generates the small amount of writes even though it guarantees *version consistency*. Meanwhile, Now, let's see the results of Git workload; *LDJ* issues 2x more writes than OJ mode in this case.

However, the huge amount of writes is reasonable since the workload does not carry out in-place update for data blocks. As a result, *LDJ* writes 30% more journal data in case of the Git workload.

Effect of Compression. To further understand the performance variation in Figure 4-5, we measured two metrics of the compression while running each workload: compression time and its ratio. The compression time is measured by placing `ktime_get()` before and after the compression function in jbd2 and the compression ratio is computed as the ratio of the size of data after compression to that before compression.

Table 4-5 and Table 4-6 summarizes the compression time and ratio of *LDJ* for each workload. As we expected, each workload shows good compression time and ratios, but they are different according to the types of storage devices and workloads. For example, in NVMe-A, *LDJ* can reduce the size of data to be journaled to 0.09% of the size of raw data before compression on average. Fortunately, such a dramatical reduction is important in that it can give the flash storage devices, such as NVMe-A, NVMe-B, and SSD, an opportunity to prolong their limited endurance. However, *LDJ* should pay the cost for compression overhead (*i.e.*, the compression time) to support the compression on-the-fly. Note that since the compression time gives some effects on the commit latency, the performance of *LDJ* can be affected by the trade-off between the IO benefit and the CPU overhead; if the CPU overhead is higher than the IO benefit, its performance will be decrease. For example, in NVMe-A with Varmail workload, *LDJ* shows 30% worse performance than the OJ mode (see Figure 4-5) even though it reduces the amount of writes by up to 6x (see Table 4-4). Meanwhile, Table 4-5 gives us an intuitive proof of the trade-off; the 4 KB write performance of NVMe-A is 4.3 us that is calculated based on the Intel 750 series specification [6], but the compression time of *LDJ* is 109 us. For this reason, *LDJ* tends to exhibit high CPU overheads, but

	Standard benchmarks							
	Varmail		**Webserver**		**Fileserver**		**Fileserver-MF**	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
NVMe-A	109 us	0.09%	128 ms	0.004%	103 ms	0.007%	164 ms	0.01%
NVMe-B	108 us	0.09%	134 ms	0.004%	108 ms	0.008%	166 ms	0.009%
SSD	88 us	0.1%	132 ms	0.004%	103 ms	0.008%	170 ms	0.009%

Table 4-5: The average compression time per transaction (Time) and compression ratio per transaction (Ratio) of *LDJ* for standard benchmarks; a small compression ratio indicates a good compressibility.

	Real-world workloads					
	Git		OLTP		File-copy	
	Time	Ratio	Time	Ratio	Time	Ratio
NVMe-A	1.4 s	0.39%	77 us	0.20%	360 ms	0.95%
NVMe-B	1.2 s	0.38%	76 us	0.20%	291 ms	0.77%
SSD	1.3 s	0.38%	123 us	0.27%	267 ms	0.76%

Table 4-6: The average compression time per transaction (Time) and compression ratio per transaction (Ratio) of *LDJ* for real-world workloads; a small compression ratio indicates a good compressibility.

we believe that CPU technologies will compromise it with their steady technological development in the near future.

Impact of Journal Size. Now, we carefully revisit the size of journal area because, if the journal size is set too small, jbd2 frequently triggers the *forced checkpoint* that considerably exacerbate both the performance and the amount of writes. To evaluate how the size affects the performance, we repeated the same experiments while varying the journal size. In this chapter, we only show the performance results of Git workload (see Figure 4-7). In this figure, the red dot line indicates the performance of OJ mode. As shown in Figure 4-7, the performance results are slightly different according to the storage type, but they show a similar trend. As we expected, as the journal size is increased, *LDJ* shows better performance. The reason behind this result is that *LDJ* uses the journal area more efficiently with less checkpoint operations. Especially, in Figure 4-7, when the size of journal area is extended to 5 GB, *LDJ* shows similar or slightly better performance compared to OJ mode. That is why we extends the journal area as 5 GB in this chapter. On the other hand, DJ mode always shows worse performance compared with *LDJ* although the journal size is extended with *Lazy checkpoint*. This result clearly confirms that *Lazy checkpoint* cannot avoid triggering checkpoint operations frequently.

4.6.4 Recovery Overhead

Now, we verify the recovery process of *LDJ* and compare its performance with those of existing journaling modes, OJ mode and DJ mode. Since the verification of crash recovery is possible only when a system crash or power failure occurs, we forced the system reboot by sending 'b' character to the system request key[6)] while running each benchmark. As mentioned in section 4.4, *LDJ*

6) the character is set by passing "echo b > /proc/sysrq-trigger"

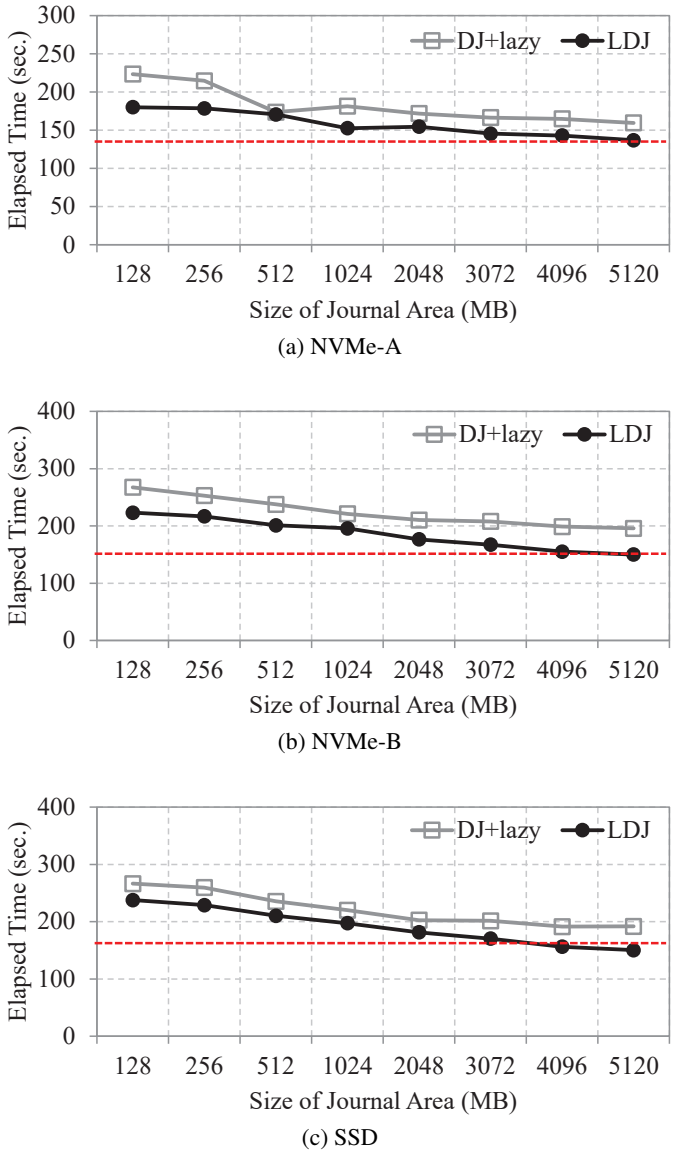


Fig. 4-7: Performance results of git according to the size of journal area. Red dot line means the performance of OJ mode.

introduced one more step of decompressing the compressed journal data in its recovery process; this step first reads data stored in compressed form and then detects a mismatch of the dictionary to prevent *false recovery*. Therefore, it is not surprising at all that *LDJ* will take longer than default journaling modes in terms of recovery time. For comparison, we measured the time taken for crash recovery in *e2fsck*, which is modified to implement the recovery procedure of *LDJ*.

Figure 4-8 shows the recovery time measured under different types of storage devices; it only includes the results from three representative workloads: Varmail, Fileserver-MF, and Git. In contrast to the performance results in subsection 4.6.2, *LDJ* is quite inferior to the existing two journaling modes in terms of crash recovery time. Unfortunately, In the worst case, the recovery time in *LDJ* takes longer than that in OJ mode and DJ mode, respectively, by up to 126x and 47x. Taking into account the high performance gain from *LDJ* during the normal execution and the higher *version consistency* and also considering that the crashes tend to be rare, we believe this rather long recovery time in *LDJ* is compromisable in practice. In addition, the recovery time in *LDJ* could be, though not realized in this chapter, significantly shortened by the parallelism and hardware-assisted decompression.

Next, to validate the possibility of the *false recovery*, we conducted an experiment that intentionally omits some journal blocks belonging to a commit transaction. For evaluation, we performed the system reboot with the same methodology described above. As a result, we confirmed the fact that *LDJ* completely prevents the *false recovery*. The reason behind this is that *LDJ* fails to decompress the compressed journal blocks in all cases and experiments.

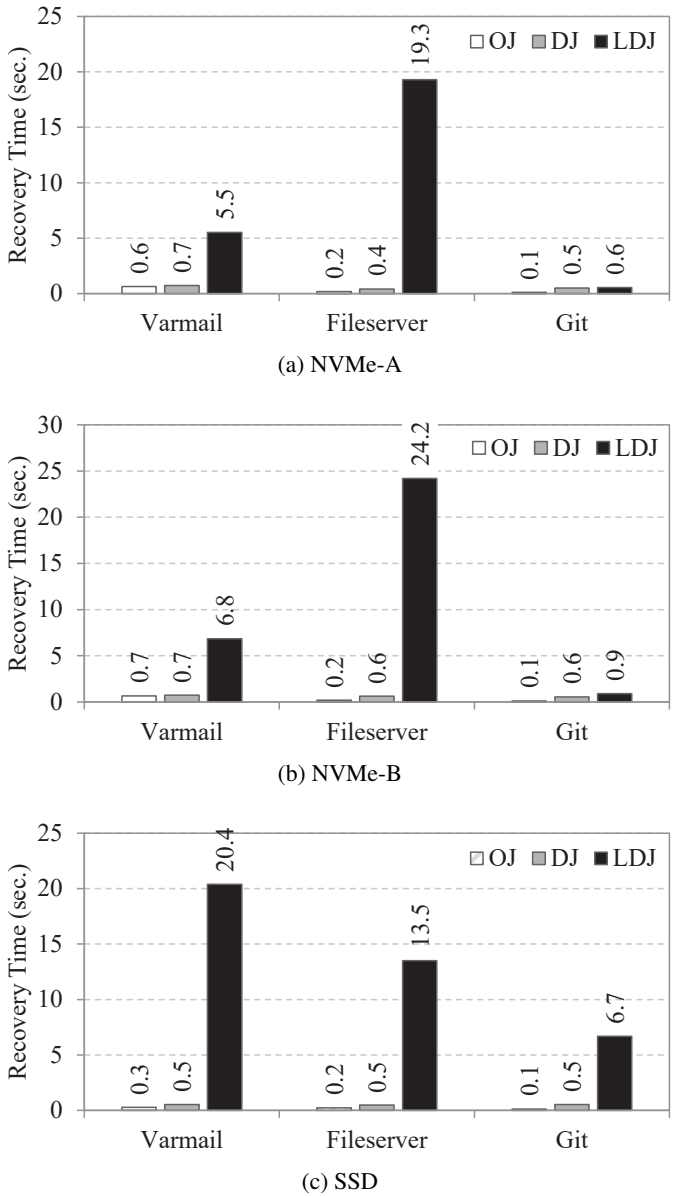


Fig. 4-8: Comparison of recovery time for different storage devices and workloads.

4.7 Related Work

There have been numerous efforts to improve the performance of crash consistency mechanism in both industry and academia. Most of prior efforts has focused on optimizing the traditional consistency mechanisms in the file system layer [24, 27, 71, 84, 103, 104], on reducing the consistency overhead in the application layer [11, 61, 62, 91, 95, 105–110], or on developing hardware-assisted crash consistency techniques [63, 69, 72, 90, 111, 112].

File system layer. Closest to our *LDJ* is the ext4-lazy mechanism [84] that can reduce the journaling overhead by remapping the location of metadata blocks to journal area. In particular, its checkpoint approach was borrowed by *LDJ* as one of the main journaling functionalities. In recent years, many researchers have been actively working to develop low-overhead but higher-level file system consistency. For instance, the checksum-based journaling [71] and padding-based journaling [27, 103] and they could relieve `fsync()` overhead by those approaches. Park *et al.* [104] improved the traditional journaling mechanism so as to reduce the overhead caused by `fsync()`. Weiss *et al.* [24] proposed the ext4 based journaling approach that improves the overall performance by reducing the redundant write overhead in journaling.

Application layer. Recently, in order to achieve application-level crash consistency as well as high performance, Pillai *et al.* [91] proposed stream APIs in their CCFS work, which are used to preserve the write order of each application. In the application layer, most work has focused on improving SQLite, which is an embedded lightweight database engine but with high runtime overhead. To reduce the overhead, Hu *et al.* [105], Lee *et al.* [11], Kim *et al.* [106], Shen *et*

al. [107], and Chen *et al.* [108] revisited both the consistency mechanism of SQLite and the file system underneath, and then redesigned them for their collaboration. Other researchers studied the overhead of application consistency in the file system layer and they suggested new interfaces that can be used for better performance [61, 62, 109]. Meanwhile, some researches utilized the compression technology to reduce the size of log data to be stored or to be transferred [95, 110].

Hardware approaches. There has been a large number of researches to exploit special hardware features to achieve better crash consistency at low overhead. For example, CFS [63] and XFTL [72] are designed on the transactional flash storage device that extends SCSI interface to send or receive some hints on a transaction. In addition, some researchers have focused on the non-volatile memory (NVM) to leverage its byte-addressable and non-volatile features in improving the performance of file system consistency. UBJ [111] efficiently removes the journaling overhead by using in-memory commit and checkpoint and WORT [112] enhances the radix tree structure along with 8-byte failure-atomic feature that is a granularity for data consistency. Also, Kang *et al.* [90] introduced a special flash storage device that can significantly reduces the `fsync()` overhead with the help of the battery-backed write cache technology which ensures durability of data on the write buffer inside the flash storage device. Oh *et al.* [69] enhanced the no in-place update of flash translator layer (FTL) and eliminates the overhead of double write logging for improving the performance of database engine.

4.8 Summary

In this chapter, we first discussed existing journaling modes in terms of consistency-level, performance, and their own benefits. Then we introduced a novel journaling mode, called *LDJ*, that combines the benefits of DJ mode with the compression technology for efficient journaling. In addition, we implemented *LDJ* on top of Linux kernel to perform a comprehensive evaluation and analysis; less than 300 lines of source code were changed. In evaluation, *LDJ* clearly shows performance improvement compared with the existing journaling modes of ext4 file system. In the best case, *LDJ* outperforms the OJ mode by up to 9.2x and 5.9x on the standard benchmark and real application, respectively.

5. Conclusion

With the tremendous growth in the flash storage industry, the redesign of existing storage stacks becomes an important issue for the flash storage devices. Therefore, in this dissertation, we explored some characteristics of the flash storage devices that are the focus of much interest in terms of both system performance and the device's endurance. In Chapter 2, we first investigated the *flash-friendly write pattern* and introduced a new page cache replacement algorithm that enhances the traditional CLOCK algorithm by exploiting both temporal and spacial locality. In Chapter 3, we proposed a novel file system that efficiently guarantees application-level crash consistency via *SHARE* command. In addition, we prototyped *AFS* with only minimal changes and carried out various experiments by running *AFS* on top of the M.2 SSD. In Chapter 4, we focused on the crash consistency mechanism once again. We presented a novel journaling mode for ext4 file system by carefully analyzing the write patterns generated by the ext4 and its consistency mechanism.

In our future work, we will continue to optimize the key idea of TS-CLOCK and explore the effectiveness of TS-CLOCK against a variety of flash storage devices. Also, we plan to further apply *SHARE* interface to other file systems because we believe that the idea of leveraging the *SHARE* interface in file systems is not limited to journaling file system. We expect the interface would be also helpful in mitigating the tree-wandering problem in CoW B-tree file systems. Finally, we have a plan to enhance the recovery procedure of *LDJ* because it is currently implemented to decompress data in a serialized way. This would be an interesting topic for future work.

References

[1] V. Kristian, "Samsung releases tlc nand based 840 ssd," http://www.anandtech.com/show/6329/samsung-releases-tlc-nand-based-840-ssd.

[2] A. L. Shimpi, "Plextor's dabbles in tlc nand," http://www.anandtech.com/show/6577/plextors-dabbles-in-tlc-nand.

[3] "Serial ATA," https://en.wikipedia.org/wiki/Serial_ATA.

[4] "NVM Express," https://en.wikipedia.org/wiki/NVM_Express.

[5] "Enterprise SSD MZILS15THMLS (PM1633a)," http://www.samsung.com/semiconductor/products/flash-storage/enterprise-ssd/MZILS15THMLS?ia=832.

[6] "Intel SSD 750 Series," https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/750-series/750-400gb-aic-20nm.html.

[7] "Solid-state Drive," https://en.wikipedia.org/wiki/Solid-state_drive.

[8] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O Stack Optimization for Smartphones," in *Proceedings of the 2013 USENIX Annual Technical Conference*, ser. ATC '13, 2013, pp. 309–320.

[9] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 271–283.

[10] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim, "Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter," in *Proceedings of the 2017 USENIX Annual Technical Conference*, ser. ATC'17, 2017, pp. 759–771.

[11] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly," in *Proceedings of the 2015 USENIX Annual Technical Conferences*, ser. ATC'15, 2015, pp. 235–247.

[12] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting Storage for Smartphones," in *Proceedings of USENIX conference on File and Storage Technologies*, ser. FAST '12, 2012, pp. 1–14.

[13] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06, 2006, pp. 234–241.

[14] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-aware Buffer Management Policy for Portable Media Players," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 2, pp. 485–493, May 2006.

[15] H. Jung, H. Sim, P. Sungmin, S. Kang, and J. Cha, "LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 3, pp. 1215–1223, Aug. 2008.

[16] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-aware Buffer Management in Flash-based Systems," in *Proceedings of the 2011 International Conference on Management of data*, ser. SIGMOD '11, 2011, pp. 13–24.

[17] H. Kim, M. Ryu, and U. Ramachandran, "What is a Good Buffer Cache Replacement Scheme for Mobile Flash Storage?" in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12, 2012, pp. 235–246.

[18] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST '08, 2008, pp. 239–252.

[19] B. Debnath, S. Subramanya, D. Du, and D. J. Lilja, "Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks," in *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '09, 2009, pp. 1–9.

[20] D. Woodhouse, "JFFS: The Journalling Flash File System," in *Proceedings of the Ottawa linux symposium*, 2001.

[21] YAFFS, "Yet another flash file system," http://www.yaffs.net/.

[22] UBIFS, "Unsorted block image file system," http://www.linux-mtd.infradead.org/doc/ubifs.html.

[23] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random Write Considered Harmful in Solid State Drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST '12, 2012, pp. 1–12.

[24] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ANViL: Advanced Virtualization for Modern Non-volatile Memory Devices," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15, 2015, pp. 111–118.

[25] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST '15, 2015, pp. 273–286.

[26] J. Park, D. H. Kang, and Y. I. Eom, "File Defragmentation Scheme for a Log-Structured File System," in *Proceedings of the 7th ACM Asia-Pacific Workshop on Systems*, ser. APSys'16, 2016, pp. 1–7.

[27] D. H. Kang and Y. I. Eom, "TO FLUSH or NOT: Zero Padding in the File System with SSD Devices," in *Proceedings of the 8th ACM Asia-Pacific Workshop on Systems*, ser. APSys'17, 2017, pp. 1–9.

[28] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST '12, 2012, pp. 1–13.

[29] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-based SSDs," in *Proceedings of the 2013 USENIX Annual Technical Conference*, ser. ATC '13, 2013, pp. 67–78.

[30] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, pp. 1–18, Jul. 2007.

[31] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36–42, Oct. 2008.

[32] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for CompactFlash Systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, May 2002.

[33] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives," in *Proceedings of the 11th international joint conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '09, 2009, pp. 182–192.

[34] X.-Y. Hu and R. Haas, "The Fundamental Limit of Flash Random Write Performance: Understanding, Analysis and Performance Modelling," Research Report RZ 3771, IBM Research, 2010.

[35] A. Bensoussan, C. Clingen, and R. C. Daley, "The Multics Virtual Memory: Concepts and Design," *Communications of the ACM*, vol. 15, no. 5, pp. 308–318, May 1972.

[36] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[37] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality," in *Proceedings of*

the 5th USENIX Conference on File and Storage Technologies, ser. FAST '05, 2005, pp. 101–114.

[38] B. S. Gill and D. S. Modha, "WOW: Wise Ordering for Writes–Combining Spatial and Temporal Locality in Non-Volatile Caches," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '05, 2005, pp. 129–142.

[39] G. Wu, X. He, and B. Eckart, "An Adaptive Write Buffer Management Scheme for Flash-Based SSDs," *ACM Transactions on Storage*, vol. 8, no. 1, pp. 1–24, Feb. 2012.

[40] L. M. Grupp, J. D. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST '12, 2012, pp. 1–8.

[41] V. Kristian, "Samsung ssd 840: Testing the endurance of tlc nand," http://www.anandtech.com/show/6459/samsung-ssd-840-testing-the-endurance-of-tlc-nand.

[42] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-memory Based File System," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON '95, 1995, pp. 155–164.

[43] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09, 2009, pp. 229–240.

[44] S. Boboila and P. Desnoyers, “Performance Models of Flash-based Solid-state Drives for Real Workloads,” in *Proceedings of the IEEE Mass Storage Systems and Technologies*, ser. MSST ’11, 2011, pp. 1–6.

[45] P. Desnoyers, “Analytic Modeling of SSD Write Performance,” in *Proceedings of the 5th Annual International Systems and Storage Conference*, ser. SYSTOR ’12, 2012, pp. 1–12.

[46] L. Bouganim, B. Jónsson, and P. Bonnet, “uFLIP: Understanding Flash IO Patterns,” in *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*, ser. CIDR ’09, 2009, pp. 1–12.

[47] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, “Parameter-Aware I/O Management for Solid State Disks (SSDs),” *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 636–649, May 2012.

[48] H. H. Huang, S. Li, A. Szalay, and A. Terzis, “Performance Modeling and Analysis of Flash-based Storage Devices,” in *Proceedings of the IEEE Mass Storage Systems and Technologies*, ser. MSST ’11, 2011, pp. 1–6.

[49] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “De-indirection for Flash-based SSDs with Nameless Writes,” in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST ’12, 2012, pp. 1–16.

[50] “The DBENCH web pages,” http://dbench.samba.org/.

[51] Transaction Processing Performance Council, “TPC Benchmark C,” http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[52] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, ser. FAST '03, 2003, pp. 115–130.

[53] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '02, 2002, pp. 31–42.

[54] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," in *Proceedings of the 2005 USENIX Annual Technical Conference*, ser. ATC '05, 2005, pp. 323–336.

[55] B. S. Gill, M. Ko, B. Debnath, and W. Belluomini, "STOW: A Spatially and Temporally Optimized Write Caching Algorithm," in *Proceedings of the 2009 USENIX Annual Technical Conference*, ser. ATC '09, 2009, pp. 1–14.

[56] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency Without Ordering," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST '12, 2012, pp. 1–16.

[57] Ext4 Wiki, "Ext4 Disk Layout/Journal (jbd2)," https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Journal_.28jbd2.29.

[58] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," in *Proceedings of the 1996 USENIX Annual Technical Conference*, ser. ATC '96, 1996, pp. 1–15.

[59] 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, Oct. 1992.

[60] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Transactions on Storage*, vol. 9, no. 3, pp. 1–32, Aug. 2013.

[61] S. Park, T. Kelly, and K. Shen, "Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013, pp. 225–238.

[62] R. Verma, A. A. Mendez, S. Park, S. Mannarswamy, T. Kelly, and C. B. Morrey, "Failure-atomic Updates of Application Data in a Linux File System," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST '15, 2015, pp. 203–211.

[63] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight Application-Level Crash Consistency on Transactional Flash Storage," in *Proceedings of the 2015 USENIX Annual Technical Conference*, ser. ATC '15, 2015, pp. 221–234.

[64] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Crash Consistency," *Communications of the ACM*, vol. 58, no. 10, pp. 46–51, Sep. 2015.

[65] ——, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, ser. SOSP '14, 2014, pp. 433–448.

[66] "MySQL 5.7 Reference Manual," http://dev.mysql.com/doc/refman/5.7/en/.

[67] "SQLite," http://www.sqlite.org/.

[68] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing Databases for Fun and Profit," in *Proceddings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '14, 2014, pp. 449–464.

[69] G. Oh, C. Seo, R. Mayuram, Y.-S. Kee, and S.-W. Lee, "SHARE Interface in Flash Storage for Relational and NoSQL Databases," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, 2016, pp. 343–354.

[70] "Vim the editor," http://www.vim.org/index.php.

[71] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic Crash Consistency," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 228–243.

[72] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite Databases," in *Proceedings of the 2013 International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 97–108.

[73] H. J. Choi, S.-H. Lim, and K. H. Park, "JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory," *ACM Transactions on Storage*, vol. 4, no. 4, pp. 1–22, Jan. 2009.

[74] J. Kim, "f2fs: introduce flash-friendly file system," https://lwn.net/Articles/518718/.

[75] J. Axboe, "FIO (Flexible IO Tester)," http://git.kernel.dk/?p=fio.git;a=summary.

[76] "Filebench," http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench.

[77] C. Min, S. Kashyap, S. Mass, W. Kang, and T. Kim, "Understanding Manycore Scalability of File Systems," in *Proceedings of the 2016 USENIX Annual Technical Conference*, ser. ATC '16, 2016, pp. 71–85.

[78] "SysBench (Branch 1.0)," https://github.com/akopytov/sysbench, 2016.

[79] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: a Database Benchmark based on the Facebook Social Graph," in *Proceedings of the 39th International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 1185–1196.

[80] "SQLite: Atomic Commit In SQLite," http://www.sqlite.org/wal.html.

[81] "SQLite: Write-Ahead Logging," http://www.sqlite.org/wal.html.

[82] "AndroBench (SQLite Benchmark)," http://www.androbench.org/wiki/AndroBench.

[83] Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, ser. SYSTOR '10, 2010, pp. 1–9.

[84] A. Aghayev, T. Ts'o, G. Gibson, and P. Desnoyers, "Evolving Ext4 for Shingled Disks," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 105–119.

[85] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-streamed Solid-State Drive," in *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'14, 2014, pp. 1–5.

[86] "Shingled magnetic recording," https://en.wikipedia.org/wiki/Shingled_magnetic_recording.

[87] W. He and D. H. Du, "SMaRT: An Approach to Shingled Magnetic Recording Translation," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 121–133.

[88] "Seagate Archive HDD," http://www.seagate.com/enterprise-storage/hard-disk-drives/archive-hdd/.

[89] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs," in *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'16, 2016, pp. 1–5.

[90] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, "Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases," in *Proceedings of the 2014 International Conference on Management of Data*, ser. SIGMOD'14, 2014, pp. 529–540.

[91] T. S. Pillai, R. A. L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Application Crash Consistency and Performance with CCFS," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 181–196.

[92] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[93] ——, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, Sep. 1978.

[94] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik, "Compression and SSD: Where and How?" in *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, ser. INFLOW'14, 2014, pp. 1–10.

[95] F. Lautenschlager, M. Philippsen, A. Kumlehn, and J. Adersberger, "Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 229–242.

[96] F. Douglis, A. Duggal, P. Shilane, T. Wong, S. Yan, and F. Botelho, "The Logic of Physical Garbage Collection in Deduplicating Storage," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 29–43.

[97] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," in *Proceedings of the Institute of Radio Engineers*, ser. IRE'52, 1952, pp. 1098–1101.

[98] "History of hard disk drives," https://en.wikipedia.org/wiki/History_of_hard_disk_drives.

[99] H. Rahmani, C. Topal, and C. Akinlar, "A Parallel Huffman Coder on the CUDA Architecture," in *Proceedings of IEEE Visual Communications and Image Processing Conference*, ser. VCIP'14, 2014, pp. 311–314.

[100] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ffsck: The Fast File System Checker," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, ser. FAST'13, 2013, pp. 1–15.

[101] "e2fsck - Linux man page," https://linux.die.net/man/8/e2fsck.

[102] V. T. Asov, E. Zadok, and S. Shepler, "Filebench: A Flexible Framework for File System Benchmarking," *;login:Magazine*, vol. 41, no. 1, pp. 6–12, 2016.

[103] A. Rajimwale, V. Prabhakaran, D. Ramamurthi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Coerced Cache Eviction and Discreet Mode Journaling: Dealing with Misbehav-

ing Disks," in *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, ser. DSN '11, 2011, pp. 518–529.

[104] D. Park and D. Shin, "iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call," in *Proceedings of the 2017 USENIX Annual Technical Conference*, ser. ATC'17, 2017, pp. 787–798.

[105] Y. Hu, Y. Kwon, V. Chidambaram, and E. Witchel, "From Crash Consistency to Transactions," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS'17, 2017, pp. 1–8.

[106] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split," in *Proceedings of USENIX conference on File and Storage Technologies*, ser. FAST '14, 2014, pp. 273–285.

[107] K. Shen, S. Park, and M. Zhu, "Journaling of Journal Is (Almost) Free," in *Proceedings of USENIX conference on File and Storage Technologies*, ser. FAST '14, 2014, pp. 287–293.

[108] Q. Chen, L. Liang, Y. Xia, and H. Chen, "Mitigating Sync Amplification for Copy-on-write Virtual Disk," in *Proceedings of the USENIX conference on File and Storage Technologies*, ser. FAST '16, 2016, pp. 241–247.

[109] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating System Transactions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '09, 2009, pp. 1–16.

[110] X. Dou, P. M. Chen, and J. Flinn, "Knockoff: Cheap Versions in the Cloud," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 73–87.

[111] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, ser. FAST'13, 2013, pp. 73–80.

[112] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 257–270.

논문요약

플래시 스토리지 기반 일관성 유지 기법과 효율적인 페이지 캐시 관리 기법 제안

성균관대학교

전자전기컴퓨터공학과

강동현

NAND 플래시 저장 장치의 오랜 목표는 컴퓨팅 환경(예: 모바일, 데스크톱, 엔터프라이즈 서버) 에서 하드 디스크 드라이브를 완벽하게 대체하는 것이며 이를 위해, 플래시 스토리지 제조업체들은 자체 기술력과 제품을보다 높은 용량과 성능으로 발전시키고 있다. 이러한 스토리지 업계의 엄청난 성장은 스토리지 연구원들이 하드 디스크 드라이브 기반으로 설계된 기존 스토리지 스택들을 재 설계 할 수있는 기회를 제공한다.

이에, 본 학위 논문에서 우리는 운영 체제의 기존 스토리지 스택이 플래시 저장 장치의 이점을 활용할 수있는 방법을 탐색하고 스택 내부의 소프트웨어 기술을 최적화하는데 유용한 세 가지 방법을 제시합니다. (1) 우리는 플래시 친화적 쓰기 패턴(*flash-friendly write pattern*) 을 소개하고 새로운 페이지 캐시 교체 알고리즘인 TS-CLOCK을 제안한다. TS-CLOCK은 높은 캐시 적중률을 위해 시간적 지역성을 활용하며, 페이지 캐시에서 추출되는 데이터의 쓰기 패턴을 플래시 친화적으로 유지하기 위해 공간적 지역성을 이용한다. 우리의 TS-CLOCK의 효율성을

확인하기 위해 TS-CLOCK을 구현하고 이를 7 가지의 페이지 교체 알고리즘들과 비교하였으며, 그 결과 다른 알고리즘에 비해 향상된 성능을 보여주었다. (2) 우리는 새로운 파일 시스템인 *AFS* 보여준다. 우리는 ext4 파일 시스템의 데이터 저널 모드를 약간 수정함으로써, *AFS* 의 프로토타입을 구현하였으며 다양한 실험을 진행하였다. 우리의 실험 결과에서 *AFS* 는 기존 저널 모드들보다 우수한 성능을 보여준다. 또한, 우리는 *AFS* 가 MySQL/InnoDB 엔진 기반의 OLTP 벤치마크의 성능을 2–6배 향상시킬 수 있다는 것을 확인하였다. (3) 마지막으로 우리는 쓰기 패턴 관점에서 ext4의 일관성 메커니즘을 재검토하고 *LDJ* 라고하는 압축/압축 해제 기법을 활용하는 유용한 저널링 모드를 제안한다. 우리는 *LDJ* 의 저널링을 위해 ext4와 jbd2의 코드를 약간 수정하였으며 *LDJ* 의 복구을 위해 e2fsck을 수정하였다. 우리의 실험 결과에서 *LDJ* 는 기본 저널링 모드보다 표준 벤치마크에서는 9.2배, 실제 어플리케이션에서는 5.9배의 성능 향상을 보였다.

주제어: 낸드 플래시 저장 장치, 페이지 캐시 교체 알고리즘, 파일 시스템, 일관성, 저널링 메카니즘, 고성능

Providing Crash Consistency along with Efficient Page Cache Management on Flash Storage Devices 2017 Dong Hyun Kang