WOJ: Enabling *W*rite-*O*nce Full-data *J*ournaling in SSDs by Using Weak-Hashing-based Deduplication

Fan Ni[a,*], Xingbo Wu[b], Weijun Li[c], Lei Wang[d], Song Jiang[a]

[a] *University of Texas at Arlington, Arlington, TX, US, 76010*
[b] *University of Illinois at Chicago, Chicago, IL, US, 60607*
[c] *Shenzhen Dapu Microelectronics Co. Ltd, Shenzhen, China, 518100*
[d] *Beihang University, Beijing, China, 100191*

Abstract

Journaling is a commonly used technique in file systems to provide data reliability for applications. Full-data journaling, which stores all file system (data and metadata) updates in a journal before they are applied to their home locations, provides the strongest data reliability, reduces application developers' efforts on application-level crash consistency, and helps to remove most crash-consistency vulnerabilities. However, file system users usually hesitate to use it as it doubles the write volume to the disk, leading to compromised performance. While fast SSDs have the potential to make full-data journaling affordable, its doubled writes threaten the devices' durability, which is their Achilles heel.

While data deduplication technique can be used to remove the second writes to the home locations, it can be too expensive to be a practical solution due to its high computation and space overheads as it has to compute and cache collision-resistant hashing values (fingerprints). The issue is especially serious for SSDs, which are becoming increasingly large and fast, but less tolerant of additional overhead in the I/O stack. Leveraging the fact that with data journal mode all writes to the home locations in a file system are preceded by corresponding writes to the journal, we propose Write-Once data Journaling (WOJ), which uses a weak-hashing-based deduplication dedicated for removing the second writes in data journaling. WOJ can reduce regular deduplication's computation and space overheads significantly without compromising the correctness. To further reduce metadata persistency cost, WOJ is integrated with SSD's FTL within the device. Experiment results show that the ext3 file system with data journaling on WOJ-enabled SSDs can deliver up to 2.7X higher throughput than that with regular deduplication, while both remove about half of the writes to the SSD.

Keywords:
Journaling, File systems, Deduplication, SSD

1. Introduction

Journaling is a commonly used technique in today's file systems (e.g., ext3 [1] and ext4 [2]) to ensure data consistency in the face of a system failure. In a journaling file system, updates

*Corresponding author
Email address: fan.ni@mavs.uta.edu (Fan Ni)

are first recorded in the journal (in the *commit* phase) and later applied to their home locations in the file system (in the *checkpoint* phase). In case of a system crash, all completed records (transactions) in the journal are replayed and applied, while uncompleted ones are discarded, to ensure the file system's consistency.

File system journaling can be in one of the two modes, *data* or *metadata* journaling, based on the contents written to the journal. In the data journaling mode, all file system (data and metadata) updates are written to the journal before being checkpointed to the files later on. In contrast, in the metadata journaling mode, only the updated metadata of the file system, such as inode and free block bitmap, are first written to and protected by the journal, while data are written directly to their home locations in the files. Due to the double write of both data and metadata in the data journaling mode, file system users are usually reluctant to use it and resort to metadata journaling for its fast speed. However, they have to tolerate compromised data reliability.

1.1. Data Journaling is Necessary

Metadata journaling has several limitations. First, as data are written directly to a file, without explicit synchronization control, updated data blocks can be mixed with un-updated ones in any order even with sequential write pattern if the system crashes during the file's overwriting [3]. Second, metadata journaling cannot guarantee even the consistency of metadata. For example, with the metadata journaling in the *ordered* mode in ext3 and ext4, the modify time ("mtime") of a file may remain unchanged after the file is updated. The anomaly is due to a critical rule, which is to write data before committing metadata to the journal in the ordered mode. This metadata inconsistency can raise an issue with applications relying on the *mtime* attribute to decide their next actions, including GNU make [4, 5] and file integrity checks using signatures [6, 7]. Last but not least, recent research has revealed that metadata journaling is prone to introduce vulnerabilities to user-level applications as it reorders applications' write operations for performance [8, 9, 10]. To avoid the vulnerabilities, application developers have to be aware of write re-ordering in file systems and eliminate their side effects with extra efforts in their programming, such as inserting extra flushes between file system operations to enforce the right order. However, it is not easy to avoid the vulnerabilities at the application level even for expert programmers. These vulnerabilities are found in widely-used applications [9], including Google's LevelDB [11] and Linus Torvalds's Git [12].

As Linus Torvalds stated "*Filesystem people should aim to make 'badly written' code 'just work*"' [13], use of data journaling adheres to the belief. With data journaling, the file system maintains a total write order and preserves application order for both metadata and data updates, which provides the strongest data consistency support for applications. As shown in existing studies, most crash-consistency vulnerabilities in commonly-used applications can be avoided with the use of data journaling, and the remaining ones have minor consequences and are ready to be masked or fixed [8, 9]. With the clear advantages in providing data reliability and stronger file system consistency support for upper-level applications, application developers will prefer to use data journaling if the performance is not a concern. With the extensive use of SSDs and their ever-improving I/O performance, data journaling's higher demand on write bandwidth is likely to be accommodated, and the once-thought expensive journaling approach may become affordable.

1.2. SSD's Endurance is now a Barrier

While SSD can provide much higher write throughput to potentially support data journaling well, its endurance becomes a new barrier to the practical use of data journaling, which doubles

Table 1: *Throughput of different hash functions under different CPU frequencies*

Hash functions	**MiB/s (3.0GHz CPU)**	**MiB/s (1.2GHz CPU)**
MD5	526.1	215.3
SHA-1	470.1	172.5
SHA-256	204.1	79.2
SHA-512	327.5	130.5
CRC32C	7631.3	3050.7
xxHash	5715.4	2285.2

write traffic to the disk. For most flash-based SSD devices, each flash memory cell can only be written several thousand times in its lifespan [14, 15, 16, 17]. While high-end SSDs can deliver GB/s-level throughput and TB-level capacity, their lifetime is not improved accordingly [18]. Actually, due to the adoption of MLC (multi-level cells) and TLC (triple-level cells) for larger capacity, SSDs' lifetime is worsening [14, 17]. For example, Intel 750 Series NVMe 400GB SSDs can provide up to 2.2GB/s and 900MB/s throughput for sequential read and write accesses, respectively, while the endurance is rated for up to a maximum of 127TB written (70GB per day) over the course of its 5-year limited warranty [19]. That is, even if the cells of the device are written evenly (an ideal scenario), it cannot surely admit new writes successfully after each flash cell is written about 317 times (127TB/400GB) on average. In other words, if the device keeps admitting writes at its full write speed, the total write time of the device can be only about 37 hours (127TB/900MB seconds). While these numbers may be derived from highly conservative estimates on SSD's lifetime, endurance of SSDs is indisputably a major concern. If the write-twice issue could be addressed by efficiently removing the second write (for checkpointing) that follows the corresponding first one to the journal, data journaling would not be a concern to the SSD's endurance. As contents of the two writes are the same, block-level deduplication technique [20] is a potentially viable solution.

1.3. Regular Deduplication is too Expensive

Though deduplication can be a promising solution, it is too expensive to be effective for fast SSDs in terms of its computation, space, and synchronization overheads. Existing deduplication techniques rely on fingerprints, which are hash values computed on individual blocks' contents with a collision-resistant hash function, to identify duplicate data. Example collision-resistant hash functions include SHA-1 [21] and MD5 [22]. As write of every block requires computation of its fingerprint, the impact of the computation can be substantial. Table 1 shows throughput of computing hash values of 4KB blocks on a Dell server with Xeon E5-2680v3 CPUs (1.2GHz~3.0GHz with DVFS technique) and 30MB LLC when different hash functions are used. As we can see, the collision-resistant hash functions, such as MD5, SHA-1, SHA-256, and SHA-512, have throughput higher than or comparable to that of hard disks or slow SSDs. These functions are affordable when the slow devices are used. However, the time of fingerprint computation can be larger than the access time of fast SSDs. For example, Intel 750 NVMe SSD has 900MB/s sequential write throughput [19], much higher than that for computing SHA-1 fingerprints (470MB/s). When the fingerprint computation becomes the performance bottleneck on the I/O path, deduplication is unlikely to be applied. Though non-collision-resistant hash functions, such as CRC32C and xxHash[23], have throughput higher by more than ten times (even on less powerful embedded CPUs [24], as illustrated in Table 1 showing throughput under lower CPU frequency), their use can compromise correctness. As we will show, deduplication is preferred to be implemented within a disk [25]. High-performance multi-core processors are less

likely to be used inside an SSD. Furthermore, today's high-performance SSDs, such as Intel Optane SSDs [26], have throughput as high as more than 2GB/s and demand any computation on their I/O paths to be very light.

To make matters even worse, regular deduplication techniques consume a large amount of memory for caching its metadata, including fingerprints, block address mappings, and reference counts, for its efficient operations. The space demand can be substantial. For example, assuming a disk of 4TB (1G 4KB blocks) and a fingerprint of 20B (a SHA-1 value) and each physical block is mapped to two logical blocks on average, the space demand for block address mapping and fingerprints can be 28GB (4B*1G*2+20B*1G). While fingerprints usually have weak locality, almost all of them have to be in the memory to achieve an optimal deduplication ratio.

Finally, deduplication requires periodical synchronization of its metadata, in particular, the block address mappings, onto the disk on a timely manner for data reliability and short response time. Additionally, different types of metadata should be persisted onto the disk in a particular order. All these require frequent use of expensive *flush* operations [27]. For deduplication not implemented in the disk, metadata synchronization operations can significantly degrade I/O throughput and offset deduplication's potential performance advantage.

1.4. A Lightweight Built-in Solution

In this work we propose a solution that is built in the SSD to transparently support **W**rite-**O**nce data **J**ournaling, named WOJ. While it still uses fingerprints to detect duplicate blocks and leverages deduplication technique to remove the second writes, it addresses all three issues with the regular deduplication technique. First, WOJ can use ultra-lightweight non-collision-resistant hashing to identify second writes of duplicated blocks without compromising correctness. Second, WOJ only needs to maintain a small amount of metadata, which is thousand times smaller than that for regular deduplication and can be fit in the SSD's internal memory. Third, WOJ integrates its block mapping with SSD's FTL and does not require frequent flushes from the host to the device.

Our contributions in the paper are threefold: (1) We investigate potential benefits and challenges of enabling data journaling in SSDs. We address the challenges with a design with little compromise on I/O performance even for high-end SSDs that are sensitive to even small overheads added into their I/O stack. Also, it protects SSDs' durability as well as regular deduplication does; (2) We prototype WOJ as a device mapper target supporting real file systems (ext3 and ext4) and perform I/O on real SSDs, instead of SSD simulators, to demonstrate its practicality and efficacy; (3) We extensively evaluate WOJ with micro-benchmarks, widely-used file system benchmarks, database workload, and workloads with real-world data. The results show that WOJ removes about half of the writes in data journaling and provides significant performance improvement over full deduplication schemes.

2. The Design of WOJ

The design goal of WOJ is to address challenges on minimizing time and space costs that are required by the deduplication technique and are usually too high to fit in the I/O stack of high-performance SSDs. As we have mentioned, deduplication on the host needs to periodically and often frequently flush its metadata to the disk for their persistency and consistency [27]. This excessive overhead is unlikely to be removed on most of today's general-purpose computers as long as the deduplication is performed at the host side. In contrast, WOJ is situated within the

SSD to leverage the non-volatile memory that is often found in today's SSDs in the form of DRAM protected by battery or super capacitor. An additional benefit of performing deduplication in the SSD is that its address mapping table overlaps with the SSD's existing mapping table (as part of the SSD's FTL), and additional space and maintenance costs for the table required by the deduplication can be avoided.

However, the resources available for deduplication, including computing power and memory, are highly constrained in the SSD. Accordingly, objectives of WOJ's design include (1) use of ultra-lightweight fingerprints without compromising correctness, and (2) very small demand on metadata space with its size decoupled from the SSD's capacity. WOJ achieves the design goals by taking advantage of a priori knowledge on the source of data duplication (checkpointing data that have been in the journal) and of very limited amount and lifetime of the data in the journal.

2.1. SSD with File-system-level Knowledge

To be simple and effective, WOJ performs deduplication at the block level. That is, it identifies and removes writes of duplicate data at the unit of blocks defined by the SSD's interface. WOJ is designed to remove the second writes in a journaling file system in the data journaling mode. It requires that the whole blocks (not only the updated portions) where (data or metadata) modifications take place are recorded in the journal and later written to the file system. This requirement is met by any file systems using physical journal [28, 29] (including ext3 and ext4), in which file system updates are logged in their original blocks of data and written to the journal. We are aware that file systems with logical journal (like XFS [30]) only record the deltas (the changes) made by the writes in the journal to reduce the amount of logged data. This is at the expense of increased complexity. While WOJ can remove all the cost of second writes and enable a physical journal that is more efficient than logical journal, we believe that the simpler physical journal will become the preferred journaling approach and take the place of logical journal.

A unique feature of WOJ, as a block-level deduplication design, is to leverage knowledge that is only available at the upper level, i.e., file system level, to distinguish whether a block is written to the journal or not. With this knowledge WOJ can use non-collision-resistant fingerprints and maintain a very small set of metadata for lightweight deduplication without compromising correctness (details in the next subsection). As WOJ is implemented in the SSD, a journaling file system must inform WOJ that data journaling and WOJ functionality be enabled when the file system is mounted. In a journaling file system, such as ext3, ext4, and ReiserFS, a special file with contiguous space allocation at a fixed disk address and of fixed size is designated as the file system's journal to store records (transactions), which are then periodically checkpointed into the file system. The journal is used as a circular buffer, where space holding records that have been checkpointed can be reused. As long as the journal's address space, possibly in the form of its starting disk address and length, is disclosed by the file system to the disk, WOJ knows which of the writes to the disk are commit ones (*the first writes*) to the journal and which are checkpoint ones (*the second writes*) to the home locations. When there are more than one partition on the SSD, a file system needs to specify the address space the partition occupies, and WOJ distinguishes the two types of writes and performs its deduplication operations separately for individual WOJ-enabled partitions. WOJ can be enabled at the time of mounting a file system (with a *mount* system call). The journal's address space can be passed to the SSD via an SSD primitive similar to the *ptrim* and *exists* commands proposed by FusionIO [31]. When the WOJ functionality is not enabled or is turned off after being enabled, the SSD functions as a normal SSD device without deduplication.

2.2. Deduplication with Non-collision-resistant Fingerprints

WOJ does not pursue full deduplication, where all duplicate blocks are detected and only one copy of the duplicate blocks is physically stored in the disk. Instead, WOJ uses non-collision-resistant fingerprints to deduplicate the second writes (writes in the checkpoint phase) in a data-journaling file system with very low overheads. In the process it is required that (1) the correctness is not compromised; and (2) the collision rate is low so that deduplication ratio is negligibly affected. The key difference in its use of fingerprints from regular deduplication is that WOJ does not rely on fingerprints to determine the existence of duplication between a given block in the checkpoint phase and those that are recently committed to the journal and have not been checkpointed yet, as in the data journaling mode *"all new data is written to the journal first, and then to its final location"* [32] and the duplication is guaranteed. Instead, a block's fingerprint is used only to identify which block in the journal has identical contents as the block under consideration in the checkpoint phase. As long as a fingerprint is not shared by more than one block in the journal, it can be used to identify the corresponding block in the checkpoint phase (even if non-collision-resistant fingerprints are used) and avoid writing it to the disk.

To facilitate the identification, WOJ maintains a fingerprint pool. A fingerprint will be inserted in the pool and used for detecting duplicate blocks in the checkpoint phase only when two conditions are satisfied. First, the fingerprint is computed over the contents of a block in the commit phase. Second, the fingerprint is unique in the pool. For each fingerprint in the pool, it is associated with a unique physical page address (PPA) indicating where the corresponding block of data is stored. We assume the block size exposed by the SSD's interface is identical to the flash memory page size inside SSD. If not, an adaptation is straightforward [33]. To this end, when a block is written into the journal (in the commit phase), its fingerprint is computed (Step 1.1 in Figure 1) and used as the key to search in the pool. If it is not found, the fingerprint is added into the pool (Step 1.3 in Figure 1). Otherwise, a collision occurs (Step 1.2 in Figure 1). A straightforward solution is to mark the fingerprint as invalid and abort the deduplication attempt on the blocks of this fingerprint. Note that when a fingerprint is designated as invalid, it is not immediately removed from the pool. Otherwise it can cause correctness issue as there can be further collisions on the fingerprint. An alternative is to introduce secondary fingerprints for the blocks in collision and enable a second chance after a collision. We do not take this option in the design as the probability of the collision is small. As disclosed in a recent study [34], the collision rate of CRC32 (and CRC32C) for typical storage workloads is lower than $8X10^{-5}$, and for *xxh64* (64-bit version of xxhash) 4 billion hashes have a 50% chance of getting one collision [35]. Additionally, the fingerprint pool is small, whose size is capped by number of blocks in a journal, whose size is usually configured as a few hundreds of megabytes to several gigabytes. Further, WOJ removes a fingerprint from the pool right after it is used for a successful removal of a second write. Otherwise, it will be removed when the space for its corresponding block is reclaimed in the journal. In either case the lifetime of a fingerprint in the pool is short and the collision is expected to be rare. It is noted that, no blocks in the commit phase will be deduplicated regardless of the uniqueness of their fingerprints.

For a block in the second write (in the checkpoint phase), its fingerprint is computed and searched in the fingerprint pool (Step 2.1 in Figure 1) for a match. Note that there always exists such a match. If it matches a valid fingerprint (one that has not experienced any collision), the block is deduplicated (Step 2.2 in Figure 1). Instead of actually writing the data again to the disk, only an entry in the FTL's address mapping table (from a block's logical page address (LPA) to its logical page address (PPA)) is updated to reflect that the block's LPA is mapped to the PPA of the block with the matching fingerprint. Note that WOJ does not need to maintain a separate

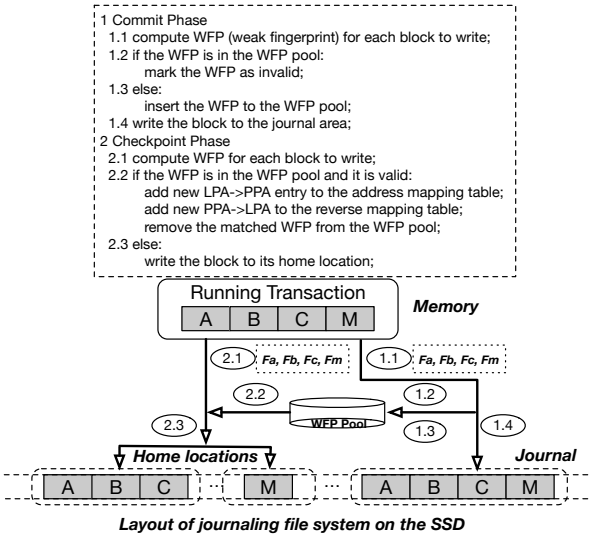


Figure 1: *Sequence of operations in WOJ for committing and checkpointing blocks. In the figure, F_a, F_b, F_c, and F_m refer to weak fingerprints of blocks A, B, C, and M, respectively.*

address mapping table. If the block matches an invalid fingerprint (Step 2.3 in Figure 1), it is written to the disk as usual, instead of being deduplicated.

2.3. Metadata Supporting Movements of Physical Blocks

As mentioned above, to serve writes and reads from users WOJ only needs to use FTL's address mapping table and maintains a very small fingerprint pool (tens of KB for a journal of a few hundreds of MB) due to the small number of much shorter fingerprints (8B for xxhash vs. 20B for SHA-1). The pool can be easily held in the SSD's memory.

However, WOJ needs to maintain a reverse address mapping table (from PPA to LPA) to support SSD's internal operations such as garbage collection and static wear leveling. When a physical block at a PPA is migrated during the operations, one has to know the LPA(s) that are mapped to the PPA, so that the address mapping table ($LPA \rightarrow PPA$) can be properly updated. When a block in the checkpoint phase is deduplicated, and its logical page address (LPA_2) is mapped to the PPA of a matching block in the journal, whose logical page address is (LPA_1), the PPA is mapped to two LPAs ($PPA \rightarrow LPA_1$ and $PPA \rightarrow LPA_2$). The first one is automatically recorded in the physical page's OOB (out-of-band) area when the page is written. The second one is recorded into a reverse mapping table. Note that a PPA is mapped to no more than two LPAs in WOJ, and a reference count maintained for each PPA in the regular deduplication is not necessary. To reclaim a physical block, both block's OOB area and the reverse table are checked to ensure no LPAs are still mapped to its PPA .

Most of the reverse mapping table will be stored on the flash. The organization of the table can be similar to the directory-based one for the address mapping table in DFTL [36]. As updates of the table have strong spatial locality due to usually sequential writes to PPAs in the journal, they can be done efficiently in batches. Though lookups into the table can be expensive by involving flash reads, they may not be on the critical path of servicing user requests. More

Table 2: *SSDs used in the experiments*

Type	Fast SSD	Moderate SSD	Slow SSD
Disk Size	400GB	240GB	80GB
Model Family	Intel 750 Series [19]	Intel 520 Series [38]	Intel X18-M/X25-M/X25-V G2 [39]
Device Model	INTEL SSDPEDMW400G4	INTEL SSDSC2CW240A3	INTEL SSDSA2M080G2GC
Sequential Read	2200MB/s	550MB/s	250MB/s
Sequential Write	900MB/s	520MB/s	70MB/s
Interface	PCI-Express 3.0 X4	SATA 3.0, 6.0GB/s	SATA 2.6, 3.0Gb/s
Endurance Rating	70GB/day for 5 years	20GB/day for 5 years	100GB/day for 5 years

importantly, they are often followed by erase operations, which can be hundreds of times more expensive than the lookups and make their impact negligible.

3. Evaluation

We extensively evaluate WOJ with a variety of workloads on a WOJ prototype to reveal its performance insights. In particular, we will answer the following questions: 1) can WOJ retain data journaling's performance on fast SSDs, and to what extent can WOJ reduce the performance overheads introduced by deduplication? 2) in terms of reducing writes to the disks, can WOJ address the write-twice issue in data journaling by removing the duplicate writes to the disk?

3.1. Experiment Methodology

As an SSD with built-in WOJ is not available yet, we prototyped a virtual SSD with WOJ functionality enabled. In the virtual SSD, the WOJ functionality is implemented in Dmdedup [37], an open-source deduplication framework, as a device mapper target at the generic operating system block device layer in Linux kernel 4.12.4. All the data written to the virtual disk are first processed in the target and those to be persisted are directed to a real SSD. Block size of the device is set to be the file system's default page size, which is 4KB. Since WOJ is to be implemented inside SSDs, where the metadata can be cached in the NVM space (e.g., battery-backed RAM) to avoid frequent data persistence, we choose the *INRAM* backend of Dmdedup to manage the deduplication metadata. With the INRAM backend, the deduplication metadata is only stored in DRAM, rather than writing to the disk. In the prototype, WOJ can be enabled/disabled by using the "*dmsetup message*" command. An the ext3 file system is installed on the virtual SSD, and data journaling mode (*data = journal*) is enabled with a 256MB journal. In the prototype we implemented the deduplication operations as well as their supporting data structures including an address mapping table, a reverse mapping table, and a fingerprint pool. It is noted that, using Dmdedup framework adds additional overheads (compared to an in-SSD implementations) to WOJ and its counterparts, which are described in the below. Because WOJ itself is more lightweight than its counterparts, the overhead represents a larger percentage of its service time, and accordingly WOJ's reported relative performance advantages are conservative.

The experiments are conducted on a Dell R630 server with two Xeon E5-2680v3 2.50GHz CPUs, each with twelve cores and 30MB last-level cache. The server is equipped with 128GB DDR4 memory. Three SSDs with different performance (as listed in Table 2) are used in the experiments to create different evaluation scenarios.

In the evaluation, the file system is configured with one of following three configurations:

- *WOJ_X*: data journaling mode with WOJ enabled. "*X*" indicates the hash function for fingerprinting. In the experiments, we use *xxHash* [23], a fast non-cryptographic hash algorithm, to generate weak fingerprints as a representative of non-collision-resistant hash functions[1]. Specifically, we applied the xxHash function over the first 64 bytes of a 4KB block to generate a 64-bit fingerprint for the block. Accordingly this WOJ is named *WOJ_xxh64*. In addition, to show the impact of hash functions, we also use collision-resistant hash function, namely, SHA-1 and MD5, in WOJ (*WOJ_sha1* and *WOJ_md5*, respectively.)

- *No_Dedup_DJ*: data journaling mode without using any deduplication. For a fair comparison, the I/O requests are also directed to the Dmdedup framework, and then sent to the device without deduplication.

- *Dmdedup_sha1*: data journaling mode with *Dmdedup* enabled for full deduplication using SHA-1 values as fingerprints, where any blocks with identical fingerprints are detected and deduplicated.

We use four types of workloads to evaluate WOJ:

- Running write-only micro benchmarks. We continuously write 4KB data to a file. Both sequential and random writes are tested.

- Running Filebench benchmarks. We conduct experiments on write-intensive benchmarks in Filebench [40], a widely-used file system and storage benchmark suite, which includes common file operations, such as create, open, close, delete, read, and write. We fill the data blocks in the write operations with randomly-generated contents.

- Hosting database workloads. In the above two workloads, each user file access had been aligned to 4KB blocks before being sent to the disk. In the database workloads, we choose a popular KV store (LevelDB [11]) and run the off-the-shelf benchmark (*db_bench*) on it. In the tests, users' data are usually re-organized before they are written to the disk, which may have implication on WOJ's deduplication ratio.

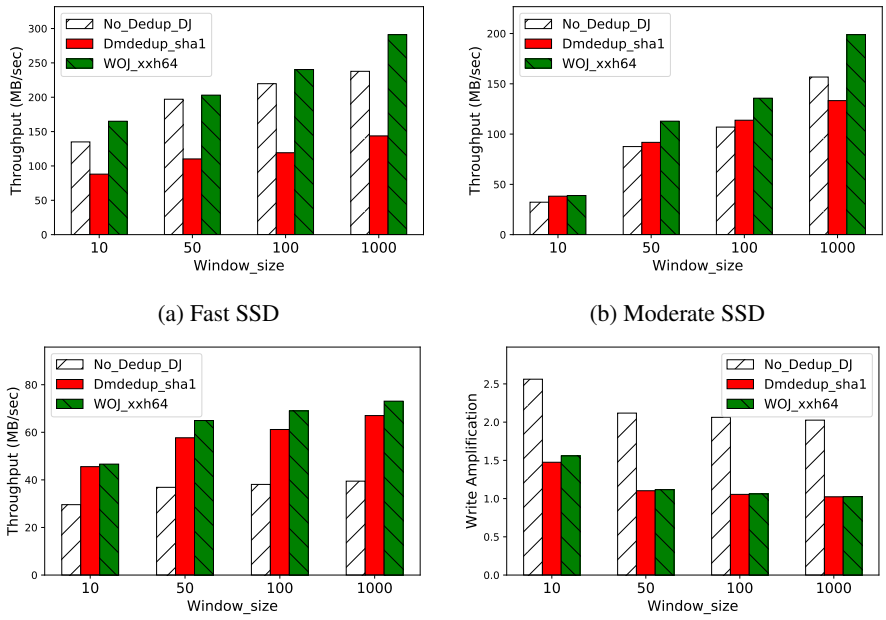
- Serving workloads with real-world data. In this experiment we conduct some common file system operations (file creations, reads, and writes) on real-world data.

3.2. Results with Write-only Micro Benchmarks

In these workloads write requests of 4KB data are issued continuously to a file, and a flush operation (issued with the *fsync* system call) is issued after a given number of writes for data persistency. We name this number the *flush window size*. Using a smaller flush window reduces chance of losing data during a system crash but may suffer a higher performance penalty.

We first perform sequential writes to a new file until it grows to 8GB. Because of use of journaling and file system operations, the actual amount of data written to the disk is larger than the amount of data requested by user programs for writing. The ratio between these two amounts is named *write amplification*, or *WA*. Figure 2 shows the throughput and WA of the sequential

[1]Using crc32c for fingerprinting can provide even better performance (see Table 1) and reduce space overhead, but its high speed depends on special Intel instructions that are not available in most embedded processors.



(a) Fast SSD

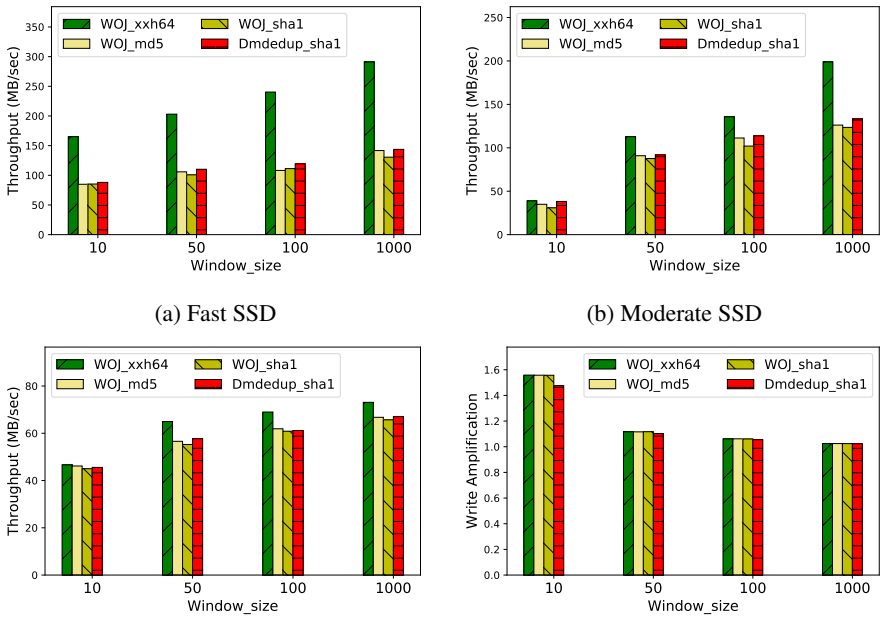
(b) Moderate SSD

(c) Slow SSD

(d) Write amplification

Figure 2: *Throughput and write amplification for sequential write accesses with different window sizes on the three SSDs.*

writes with different flush window sizes on different SSDs. The results reveal a number of insights. First, WOJ_xxh64 achieves the highest throughput among the three configurations as shown in Figures 2a, 2b and 2c. Second, throughput difference among the three configurations varies on different SSDs. For the fast SSD, Dmdedup_sha1 shows much lower throughput than No_Dedup_DJ. For example, its throughput is only about 60% of No_Dedup_DJ's throughput when the window size is 1000. Although applying deduplication removes almost half of the writes to the disk in data journaling as shown in Figure 2d, it introduces fingerprinting overheads, which is too expensive for the fast SSD as indicated in Tables 1 and 2. These results confirm our belief that regular deduplication schemes are too expensive for high-performance SSDs. For moderate and slow SSDs, since their I/O performance is much lower (as shown in Table 2), their overhead of fingerprinting becomes less significant, making Dmdedup_sha1 provide comparable (as shown in Figure 2b) or higher (as shown in Figure 2c) throughput than that with No_Dedup_DJ. Third, there is an anomaly in the performance difference between No_Dedup_DJ and Dmdedup_sha1 in Figure 2b when the window size is 1000. Although the throughput of fingerprinting with SHA-1 (470MB/s) is a little lower than the write throughput of the moderate SSD (550MB/s) as shown in Tables 1 and 2, Dmdedup_sha1 achieves higher throughput when the window size is small as the frequently issued flush operations add extra cost to the I/O operations. However, when the window size becomes larger, No_Dedup_DJ has higher performance than Dmdedup_sha1. This is because the overhead of the expensive flush operations is amortized by more write operations, which improves the I/O performance and makes the overhead of fingerprinting more significant. In general, WOJ improves the throughput by about 1.85X to 2.03X compared to Dmdedup_sha1 for the the fast SSD.



(a) Fast SSD

(b) Moderate SSD

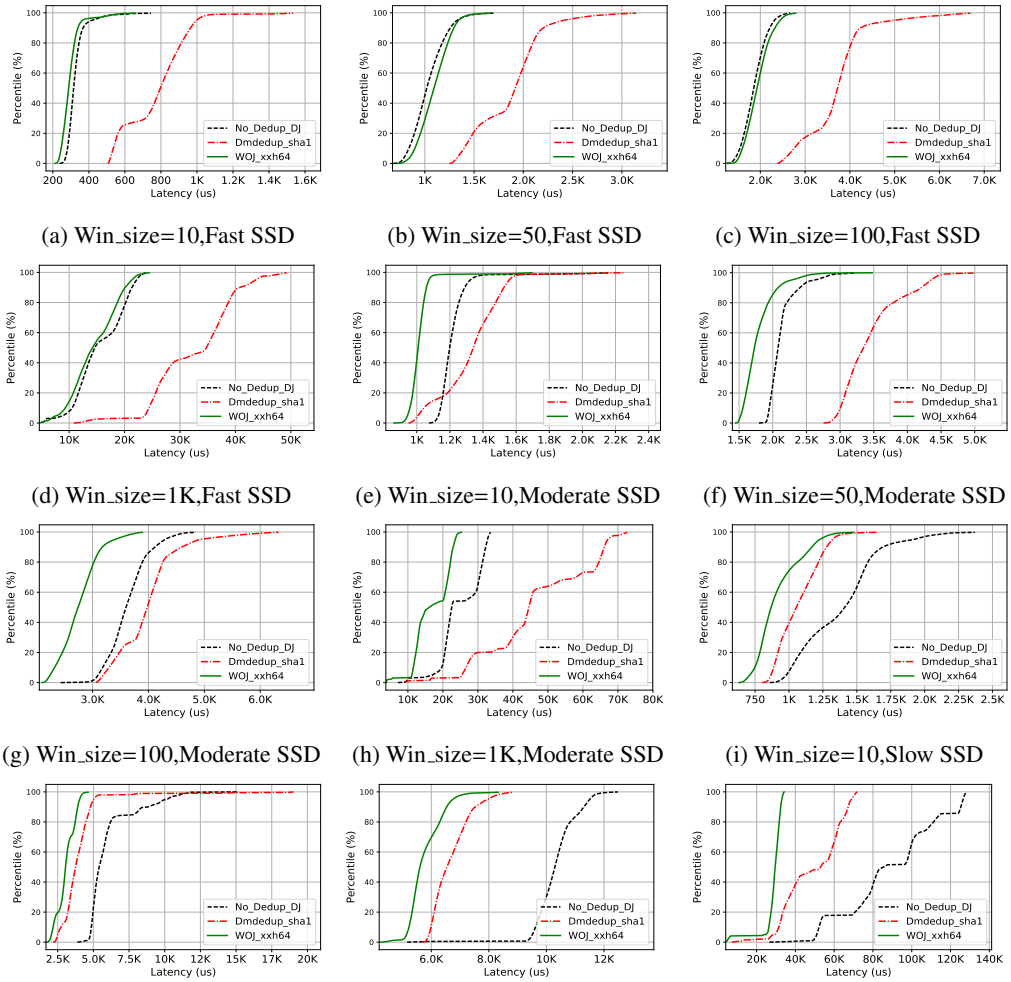
(c) Slow SSD

(d) Write amplification

Figure 3: *Throughput and write amplification of WOJ for sequential write accesses with different hash functions for different window sizes.*

As shown in Figure 2d, WOJ_xxh64 removes about half of the data written to the disk as Dmdedup_sha1 does. The results show that hash collisions due to the use of weak fingerprint are rare in WOJ. We also see that WA is lower when the window size is larger. For example, WA for No_Dedup_DJ is about 2 when the window size is 1000, while it increases to 2.6 when the window becomes 10. There are two reasons for this. First, with a larger window size file system metadata updates to the same metadata blocks in the same window can be merged and thus fewer metadata blocks are written to the disk. Second, with larger window size flush operations are less frequently issued, the file system can fit more file system updates into a single transaction, which will improve file system efficiency and reduce the journal metadata blocks (e.g., descriptor blocks and commit blocks).

To understand the contribution of the use of a weak hash function in WOJ to its performance improvement, we replace the hash function with collision-resistant ones for generating fingerprints in WOJ. The results for WOJ_xxh64, WOJ_md5, and WOJ_sha1 are shown in Figure 3. There are several observations. First, WOJ_xxh64 provides much better performance than the other two, as *xxhash* is much more lightweight than the other two as shown in Table 1. Second, the throughput difference between the three is more significant on the fast SSD, where a larger proportion of run time is spent on computing fingerprints. Third, WOJ_sha1 has a little lower performance than Dmdedup_sha1. This is because WOJ_sha1 detects and removes duplicates only in the checkpoint phase, while Dmdedup_sha1 achieves full deduplication, which may reduce a little more writes to the disk as shown in Figure 3d. Last, comparing Figures 3a and 3b, we can find that with strong hash functions, the throughput of WOJ on the fast SSD and moderate SSD



(a) Win_size=10,Fast SSD (b) Win_size=50,Fast SSD (c) Win_size=100,Fast SSD

(d) Win_size=1K,Fast SSD (e) Win_size=10,Moderate SSD (f) Win_size=50,Moderate SSD

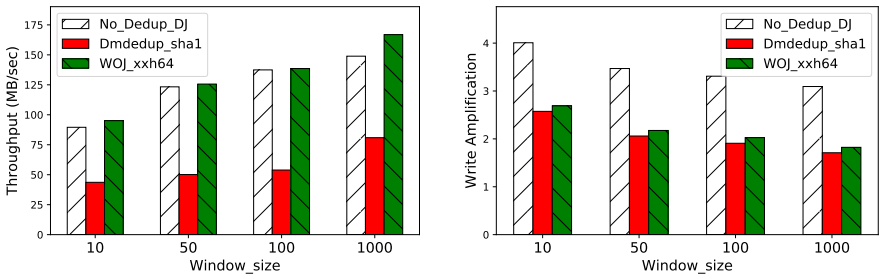
(g) Win_size=100,Moderate SSD (h) Win_size=1K,Moderate SSD (i) Win_size=10,Slow SSD

(j) Win_size=50,Slow SSD (k) Win_size=100,Slow SSD (l) Win_size=1K,Slow SSD

Figure 4: *Sequential write latency with different window sizes on the three SSDs.*

does not show significant difference for large window size even though the raw performance of the SSDs is quite different. The reason is that the computation of strong fingerprints becomes performance bottleneck for fast SSDs and offsets the performance advantage of the fast SSD. In general, with faster hash functions, such as xxHash, WOJ provides up to 2.24X throughput improvement over that using slow hash functions such as SHA-1.

For file system users, I/O request latency is also a critical performance metric. Figure 4 shows the CDF curves of write latencies of the three schemes on the three SSDs with different window sizes. The latency increases when the window grows for all the three schemes as a request is acknowledged only when the flush operation at the end of the window completes, which ensures all data blocks in the window are persisted on the flash. On the fast SSD, the latency with WOJ_xxh64 is similar to that with No_Dedup_DJ, and much lower than that with Dmdeup_sha1



(a) Throughput (b) Write amplification

Figure 5: *Throughput and write amplification for random write accesses with different window sizes on the fast SSD.*

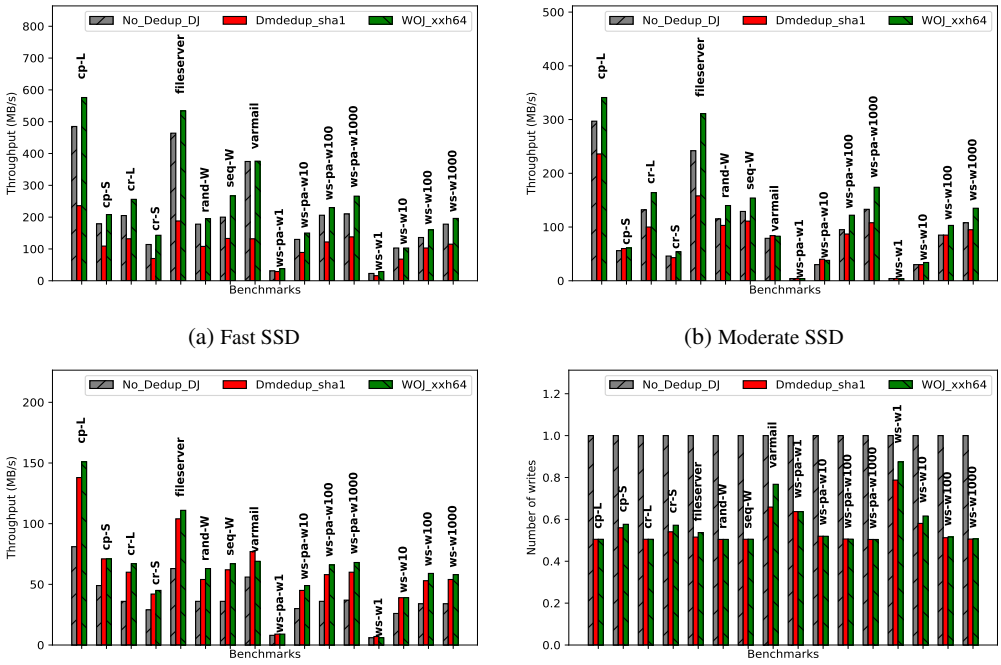
for all window sizes. For the fast SSD with high bandwidth, the doubled amount of writes poses a modest impact on the write latency for No_Dedup_DJ. For WOJ_xxh64, although half of the writes are removed, it has write latency similar to that of No_Dedup_DJ. On the moderate SSD, the benefit of reducing half of the write traffic outweighs the overhead of computing weak fingerprints. Accordingly, WOJ_xxh64 provides lower latency than No_Dedup_DJ. In contrast, computing strong fingerprint is still too expensive. Thus, the latency of Dmdedup_sha1 is the highest. On the slow SSD with low bandwidth, the doubled write traffic in No_Dedup_DJ poses a significant overhead, causing No_Dedup_DJ to have the highest latency. For the fast SSD, the 90th percentile latency of WOJ_xxh64 can be reduced to around 50% of that with Dmdedup_sha1.

To understand impact of different access patterns, we also conduct experiments with 4KB writes at random locations in a 8GB file. The experiments demonstrate similar performance trends, though random access has lower performance than sequential access. Figure 5 shows experiment results on the fast SSD. Compared to Figure 2, we see two significant differences. First, the throughput is much lower than that with sequential access. For example, for a window size of 1000 writes, the throughput of WOJ_xxh64 is about 165MB/s for random writes, while it is about 290MB/s for sequential writes. Second, the WA is higher for random access, which contributes to its throughput degradation. For random writes, the metadata updated in a flush window are less likely to be merged as they usually scatter in different metadata blocks, which causes more metadata blocks to be written to the disk.

Results with sequential and random writes show that WOJ reduces almost the same amount of duplicated data as that in full deduplication schemes with strong fingerprints. Meanwhile, WOJ reduces the performance overhead to significantly improve both throughput and latency.

3.3. Results with Filebench Benchmarks

The Filebench benchmarks include commonly-used file system operations, such as create, open, read, append, overwrite, close, and delete. Figure 6 shows the throughput and amount of data written to the disk for selected benchmarks on the three SSDs. For almost all the benchmarks, WOJ_xxh64 provides the highest performance. On the fast SSD, WOJ_xxh64 achieves much higher throughput than Dmdedup_sha1 for benchmarks with large flush window sizes, such as *cp_L*, *cr_L*, *ws_w1000* and *ws_pa_w1000*. The improvement can be up to 2.44X. For the benchmarks *varmail* and *fileserver*, WOJ_xxh64 achieves high throughput improvement for a different reason. These two benchmarks generate about the same amount of read and write operations and the I/O time dominates their execution time. In such workloads, read and write requests compete



(a) Fast SSD

(b) Moderate SSD

(c) Slow SSD

(d) Number of Writes

Figure 6: *Throughput and normalized number of writes on the three SSDs with filebench benchmarks. Meaning of the abbreviations: cp=copy; cr=create; L=Large file (1GB); S=Small file (64KB); W=Write; ws=Write with sync; pa=Preallocate file.*

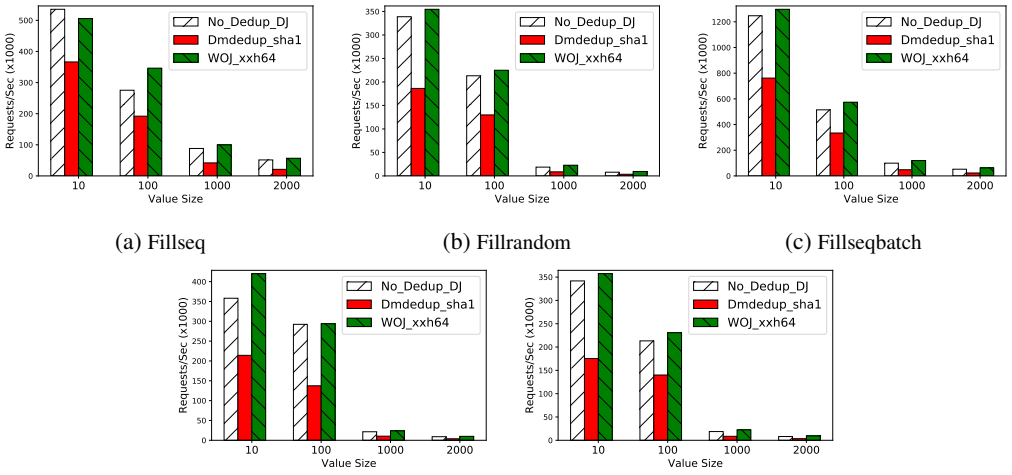
for the disk's bandwidth. While WOJ removes about half of the writes and makes the corresponding disk bandwidth available, read throughput also increases. Although Dmdedup_sha1 can remove about the same amount of writes, its high fingerprinting overhead largely offsets this benefit.

In a case with *varmail* shown in Figure 6c, the throughput of Dmdedup_sha1 is a little higher than WOJ_xxh64 on the slow SSD. In *varmail* about 10% more data written to the disk are removed by Dmdedup_sha1 than WOJ_xxh64 (shown in Figure 6d). WOJ only removes duplicate data in the checkpoint phase, while Dmdedup implements full deduplication. WOJ misses some deduplication opportunities hidden within the journal area and takes more I/O time.

The results with Filebench workloads also show that in most cases WOJ can perform as well as full deduplication schemes in reducing redundant data, as shown in Figure 6d. Except for *varmail* and *ws-w1*, where many blocks are journal metadata and less likely to be deduplicated even by full deduplication schemes, WOJ can remove about 38% to 50% blocks written to the disk, which will substantially help to improve the disk's lifetime.

3.4. Results with Database Workload

User applications usually rely on the database for data storage and retrieval, and expect it to be highly reliable as a crash in a database can affect all upper-layer applications that use its service. As databases, like LevelDB [11], are usually built on top of file systems and store their data



(a) Fillseq (b) Fillrandom (c) Fillseqbatch

(d) Fillrandbatch (e) Overwrite

Figure 7: *Throughput for PUT operations with different value sizes in LevelDB on fast SSD. The key size is 16 bytes.*

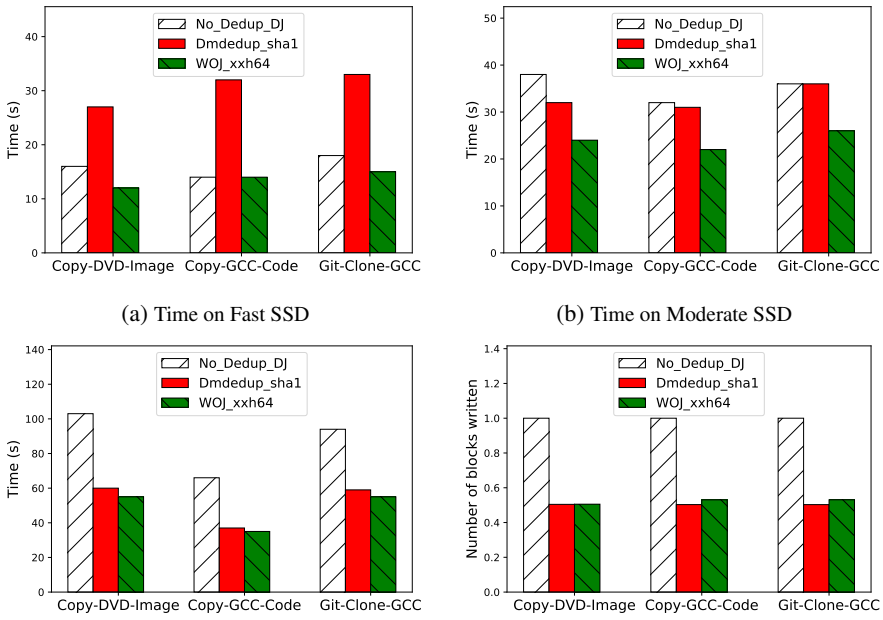
in files, the development of database programs can be much easier if the underlying file systems provide stronger data consistency support and preserve write orders by using data journaling [8]. In this experiment we mount the ext3 file system with the data journaling mode enabled. We then extensively conduct experiments on PUT workloads with different access patterns on LevelDB. We use all the default configurations of LevelDB during the experiments. In particular, the SSTable's size is 2MB. We run *db_bench* released with the LevelDB code [41]. Due to space constraint, we only show experiment results on the fast SSD, which poses significant performance challenges to existing deduplication schemes.

Figure 7 shows the throughput for LevelDB benchmarks on the fast SSD. For all the benchmarks, WOJ_xxh64 provides better performance than Dmdedup_sha1. The improvement is about 1.38X to 2.7X, and higher improvements are achieved with larger value sizes. Given a fixed number of requests, requests of larger value sizes will generate more blocks for writing and more I/O operations. On the contrary, with smaller value sizes more requests' data (key-value items) can fit in a single block and written to the disk together, leading to reduced I/O operations, and the advantage of using WOJ to reduce write traffic is weakened.

3.5. Results with Workloads Using Real-world data

In this section, we evaluate WOJ with workloads with real-world data, rather than randomly generated data, in the I/O operations. In the first experiment (*Copy-DVD-Image*), we copy a Debian DVD image [7] (about 3.7GB) from the tmpfs (/tmp) to the tested file system. In the second experiment (*Copy-GCC-Code*), we copy a compressed gcc source code [42], which includes more than 4000 files and has a size of about 3.1GB, from the tmpfs (/tmp) to the tested file system. In the third experiment (*Git-Clone-GCC*), we use *git clone* to clone a gcc repository to the tested file system. The results are shown in Figure 8.

Consistent with observations on other workloads, WOJ shows substantial performance advantages in the three experiments on the three SSDs. Compared to Dmdedup_sha1, WOJ_xxh64 reduces the execution time by about 55.6%, 56.3%, and 54.5% in the three experiments on the



(a) Time on Fast SSD

(b) Time on Moderate SSD

(c) Time on Slow SSD

(d) Number of Writes

Figure 8: *Run time and number of writes of workloads with real-world data on the three SSDs. The number of writes is normalized based on the number of blocks written with No_Dedup_DJ.*

fast SSD, respectively. As expected, the improvements are less significant on the slow SSD, which are about 8.3%, 5.4%, and 9.3%, respectively. As shown in Figure 8d, WOJ_xxh64 performs almost as well as Dmdedup_sha1 in terms of write reduction due to deduplication. In all the experiments, WOJ removes about 50% of data blocks being written to the disk, demonstrating that use of non-collision-resistant hash function has negligible impact on the deduplication effectiveness.

3.6. Memory Space Overheads

Reducing memory space overhead is critical to effectively enable WOJ inside SSDs. In WOJ, three main data structures are maintained in memory: an address mapping table, a fingerprint pool, and a reverse mapping table. For the address mapping table WOJ does not require any extra space beyond that currently available in the SSD's FTL.

The size of the fingerprint pool is proportional to the journal size. In our evaluation, we have configured the ext3 file system with journals of different sizes (from 32MB to 1GB). The performance results show little difference. So we choose a moderate journal size, which is 256MB. With this journal size, the signature pool size is capped at 768KB[2]. For the reverse mapping table, only the segment where the PPAs mapped by current journal blocks needs to stay in memory, and

[2]Each item in the pool is 12B, including an 8B fingerprint (an *xxh64* hash value) and a 4B PPA. There are at most $256MB/4KB$ items.

has a size of 512KB ($256MB/4KB * 8$). Therefore, WOJ requires only about 1.28MB memory in the SSD for its operations.

In contrast, if a full deduplication scheme with collision-resistant hashing is deployed in an SSD, a few gigabytes of metadata have to be maintained assuming a 400GB SSD [25], like the 400GB fast disk listed in Table 2. In the scheme even a fingerprint pool can be larger than 1GB. Specifically, in the pool each item can be of 24 bytes (a 20-byte SHA-1 value and a 4-byte PPA) and total size can be 2.4GB ($400GB/4KB * 24B$). For efficient and effective deduplication, it is often expected that its metadata are mostly cached in the memory. Otherwise, additional flash reads for cache miss and writes for metadata persistency are required, which compromises its performance. In the aforementioned experiment, we conservatively assume a sufficiently large non-volatile memory for it to hold all the metadata.

4. Related Work

The importance of removing duplicate writes to SSDs for better performance and extended device lifetime is well recognized and many efforts have been made. Existing studies include efforts on reusing existing data on SSD devices and on the improvement of performance of journaling file system without compromising data reliability.

4.1. Efforts on Reusing Data on SSDs.

WOJ reuses data in a file-system journal to avoid re-writing duplicate ones to their home locations. Data deduplication has been widely used to remove redundant writes to flash-based SSDs for longer device lifetime and better performance [43, 25, 44]. All existing deduplication approaches rely on collision-resistant (strong) hash functions to detect duplicate data, which makes them too expensive to be used on fast SSDs. CAFTL is a deduplication scheme designed to be in the SSDs [25]. To find a duplicate block to remove a write, it has to apply expensive strong hash function on the block. It does attempt to use weak hash function in its detection of duplicate blocks. However, a strong hash still has to be used to ascertain its duplication. Furthermore, it adds a second pool of weak fingerprints in addition to the pool of strong fingerprints. It also needs to maintain all the other metadata required by regular deduplication schemes. Though CAFTL may use DRAM in the SSD as a cache for the metadata, the concern on the metadata access speed and persistency may not be well addressed. First, some metadata, such as fingerprints, usually do not exhibit strong locality. Second, the DRAM, especially battery-backed one, in the SSD can be too small to hold the working set of the large metadata set. In contrast, WOJ only needs to compute a weak fingerprint to recognize duplicate blocks, which minimizes fingerprinting overhead, and keeps metadata small without compromising deduplication ratio.

JFTL[45] was proposed to remove duplicate writes introduced by journaling. It is different from WOJ on its approach of detecting the second writes in the checkpoint phase to remove them. A physical journal consists of two types of blocks, among which data blocks are copies of file system blocks, and journal metadata blocks contain home addresses of the data blocks in the file system. Format of journal metadata blocks is defined by individual file systems. JFTL needs to understand the format for its analysis of the journal metadata blocks and know home addresses of the data blocks. It compares addresses of non-journal writes and the home addresses to detect the second writes. As there is no such well-accepted protocol describing the format, it is unlikely to pass the information from the file system into the SSD as WOJ does to know the journal's address space. Consequently, JFTL customizes its implementation to a particular

journal format, making this approach less likely to be accepted into SSD. Any approaches based on XCopy interface [46, 47] suffer from the same issue as JFTL.In contrast, WOJ only needs to know the journal's location in its disk partition, which can be easily obtained from the file system.

4.2. Efforts on Improving Journaling File System Performance.

Being a commonly used approach for data consistency in a file system, journaling introduces write amplification and degrades the I/O performance. CCFS [8] conducts a detailed analysis on how file system behaviours affect the user-level crash consistency, and identifies a few performance issues in journaling file systems. To address the performance issues, its major effort is to use a stream-based abstraction to reduce the flush cost on the critical I/O path in a file system with data journaling mode enabled. In contrast, WOJ complementarily removes the second writes in the checkpoint phase of journaling operations for further performance improvement. More importantly, WOJ has its unique advantage with SSDs – the reduction of writes can extend SSDs' lifetime. This is especially critical to high-performance and high-price SSDs. Rocha et al. analyzed performance of the ext3 file system using an external journal stored on a separate hard drive, and reported that the I/O throughput can be improved by about 40% [48]. However, the approach requires an extra disk, which is preferred to be a high-performance SSD if the main disk is an SSD. In addition to extra cost, still doubled writes would reduce SSD's lifetime. Furthermore, to recover a crash failure both disks have to be available to form a complete and consistent image, making it impossible to just move one disk from a failed server to a replacement one. WOJ does not use extra hardware. Instead, it removes duplicate data within SSD in a manner almost transparent to file system and its users.

5. Conclusion

In this paper we describe WOJ, a weak-hashing-based deduplication scheme for deployment inside SSDs to address the write-twice issue of data journaling with negligible performance and space overheads. With a prototype implemented as a device mapper target, we extensively conducted experiments with a variety of workloads, and the results show that the ext3 file system with data journaling on WOJ-enabled SSDs can achieve up to 2.7X higher throughput than that with regular deduplication, and it also removes about half of the writes to the SSD.

6. Acknowledgements

We are grateful to reviewers of the paper for their constructive comments, which helps to improve the papers quality. This work was mainly supported by US National Science Foundation under CNS 1527076. In addition, Weijun Li was supported by Shenzhen Peacock Plan (KQTD20150917164 53118), and Lei Wang was supported by National Natural Science Foundation of China (No. 61672073).

References

[1] S. Tweedie, Ext3, journaling filesystem, in: Ottawa Linux Symposium, 2000, pp. 24–29.
[2] M. Cao, S. Bhattacharya, T. Ts'o, Ext4: The next generation of Ext2/3 filesystem. (2007).
[3] D. Robbins, Advanced filesystem implementor's guide, part 8, `https://www.ibm.com/developerworks/library/l-fs8/index.html` (2001).

[4] GNU Operating System, Timestamp resolution and make, https://www.gnu.org/software/autoconf/manual/autoconf-2.61/html_node/Timestamps-and-Make.html (2008).
[5] J. Graham-Cumming, The GNU make book, No Starch Press, 2015.
[6] J. Graham-Cumming, Rebuilding when a file's checksum changes, https://www.cmcrossroads.com/article/rebuilding-when-files-checksum-changes (2006).
[7] Debian Project, Debian CD Image Download page, https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/ (2017).
[8] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Application crash consistency and performance with ccfs, in: Proceedings of the 15th Usenix Conference on File and Storage Technologies, FAST'17, USENIX Association, Berkeley, CA, USA, 2017, pp. 181–196.
URL http://dl.acm.org/citation.cfm?id=3129633.3129650
[9] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, All file systems are not created equal: On the complexity of crafting crash-consistent applications, in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 433–448.
URL http://dl.acm.org/citation.cfm?id=2685048.2685082
[10] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Crash consistency, Commun. ACM 58 (10) (2015) 46–51. doi:10.1145/2788401.
URL http://doi.acm.org/10.1145/2788401
[11] S. Ghemawat, J. Dean, LevelDB source code, https://github.com/google/leveldb,http://leveldb.org (2011).
[12] L. Torvalds, J. Hamano, Git: Fast version control system, http://git-scm.com (2010).
[13] L. Torvalds, Linux 2.6.29., https://lwn.net/Articles/326505/ (2009).
[14] HP Workstations, Technical white paper: SSD endurance, http://h20195.www2.hp.com/v2/getpdf.aspx/4AA5-7601ENW.pdf?ver=1.0 (2015).
[15] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, T. Wobber, Extending SSD lifetimes with disk-based write caches, in: Proceedings of the 8th USENIX Conference on File and Storage Technologies, FAST'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 8–8.
URL http://dl.acm.org/citation.cfm?id=1855511.1855519
[16] R.-S. Liu, C.-L. Yang, C.-H. Li, G.-Y. Chen, Duracache: A durable SSD cache using mlc nand flash, in: Proceedings of the 50th Annual Design Automation Conference, DAC'13, ACM, New York, NY, USA, 2013, pp. 166:1–166:6. doi:10.1145/2463209.2488939.
URL http://doi.acm.org/10.1145/2463209.2488939
[17] L. E. Inc., How to properly calculate write endurance, http://h20195.www2.hp.com/v2/getpdf.aspx/4AA5-7601ENW.pdf?ver=1.0 (2015).
[18] B. Schroeder, R. Lagisetty, A. Merchant, Flash reliability in production: The expected and the unexpected, in: Proceedings of the 14th Usenix Conference on File and Storage Technologies, FAST'16, USENIX Association, Berkeley, CA, USA, 2016, pp. 67–80.
URL http://dl.acm.org/citation.cfm?id=2930583.2930589
[19] INTEL, Intel Solid State Drive 750 Series Product Specification, Revision 004, https://www.intel.com/content/www/us/en/products/\\memory-storage/solid-state-drives/gaming-enthusiast-ssds/750-series/750-400gb-2-5-inch-20nm.html (2015).
[20] B. Zhu, K. Li, R. H. Patterson, Avoiding the disk bottleneck in the data domain deduplication file system., in: Proceedings of the 6th USENIX Conference on File and Storage Technologies, Vol. 8 of FAST'08, 2008, pp. 1–14.
[21] D. Eastlake 3rd, P. Jones, US secure hash algorithm 1 (SHA1) (2001).
[22] R. Rivest, The md5 message-digest algorithm, https://tools.ietf.org/html/rfc1321 (1992).
[23] Y. Collet, xxHash–Extremeley fast hash algorithm, http://cyan4973.github.io/xxHash/ (2016).
[24] A. Pranckeviius, Performance tests of various non-cryptographic hash functions, on various cpus., http://aras-p.info/blog/2016/08/09/More-Hash-Function-Tests/ (2016).
[25] F. Chen, T. Luo, X. Zhang, Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives, in: Proceedings of the 9th USENIX Conference on File and Stroage Technologies, FAST'11, USENIX Association, Berkeley, CA, USA, 2011, pp. 6–6.
URL http://dl.acm.org/citation.cfm?id=1960475.1960481
[26] Intel, Intel Optane SSD DC P4800X Series, https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series/p4800x-750gb-2-5-inch.html (2017).
[27] Z. Chen, K. Shen, Ordermergededup: Efficient, failure-consistent deduplication on flash, in: Proceedings of the 14th Usenix Conference on File and Storage Technologies, FAST'16, USENIX Association, Berkeley, CA, USA, 2016, pp. 291–299.

URL http://dl.acm.org/citation.cfm?id=2930583.2930605

[28] V. Prabhakaran, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Analysis and evolution of journaling file systems, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATC '05, USENIX Association, Berkeley, CA, USA, 2005, pp. 8–8.
URL http://dl.acm.org/citation.cfm?id=1247360.1247368

[29] Wikipedia, Wiki Page: Journaling file system, https://en.wikipedia.org/wiki/Journaling_file_system (2017).

[30] Wikipedia, Wiki Page: XFS-Journaling, https://en.wikipedia.org/wiki/XFS\#Journaling (2017).

[31] D. Nellans, M. Zappe, J. Axboe, D. Flynn, ptrim ()+ exists (): Exposing new ftl primitives to applications (2011).

[32] L. Kernel, Ext4 Filesystem, https://www.kernel.org/doc/Documentation/filesystems/ext4.txt (2017).

[33] M. Bjørling, J. González, P. Bonnet, LightNVM: The linux open-channel SSD subsystem, in: Proceedings of the 15th Usenix Conference on File and Storage Technologies, FAST'17, USENIX Association, Berkeley, CA, USA, 2017, pp. 359–373.
URL http://dl.acm.org/citation.cfm?id=3129633.3129666

[34] J. An, D. Shin, Offline deduplication-aware block separation for solid state disk (2013).

[35] Y. Collet, xxHash wider: assessing quality of a 64-bits hash function, http://fastcompression.blogspot.com/2014/07/xxhash-wider-64-bits.html (2014).

[36] A. Gupta, Y. Kim, 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, ASPLOS XIV, ACM, New York, NY, USA, 2009, pp. 229–240.

[37] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, E. Zadok, Dmdedup: Device mapper target for data deduplication (2014).

[38] Intel, Intel solid-state drive 520 series product specification, https://www.intel.com/content/dam/support/us/en/documents/ssdc/hpssd/sb/Intel_SSD_520_Series_Product_specification.pdf (2012).

[39] Intel, Intel X25-M and X18-M Mainstream SATA Solid-State Drives, https://www.intel.com/content/dam/support/us/en/documents/ssdc/hpssd/sb/Intel_SSD_50nm_X25-M_X18M_Series_Product-brief.pdf (2008).

[40] R. McDougall, J. Mauro, Filebench, http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf(Citedonpage56.).

[41] H. Chu, Database benchmarks, https://github.com/hyc/leveldb/tree/benches (2014).

[42] G. operating system, Mirror of all gcc svn branches and tags, https://gcc.gnu.org/git/?p=gcc.git;a=summary (2017).

[43] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H. u. Lee, S. Kang, Y. Won, J. Cha, Deduplication in SSDs: Model and quantitative analysis, in: 012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012, pp. 1–12. doi:10.1109/MSST.2012.6232379.

[44] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, G. Wallace, Nitro: A capacity-optimized SSD cache for primary storage, in: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 501–512.
URL http://dl.acm.org/citation.cfm?id=2643634.2643686

[45] H. J. Choi, S.-H. Lim, K. H. Park, Jftl: A flash translation layer based on a journal remapping for flash memory, Trans. Storage 4 (4) (2009) 14:1–14:22. doi:10.1145/1480439.1480443.
URL http://doi.acm.org/10.1145/1480439.1480443

[46] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck, Scalability in the xfs file system, in: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC '96, USENIX Association, Berkeley, CA, USA, 1996, pp. 1–1.
URL http://dl.acm.org/citation.cfm?id=1268299.1268300

[47] SNIA, Hypervisor storage interfaces for storage optimization white paper, https://www.snia.org/sites/default/files/HSI_Copy_Offload_WP-r12.pdf (2010).

[48] P. E. Rocha, L. C. Bona, Analyzing the performance of an externally journaled filesystem, in: Proceedings of the 2012 Brazilian Symposium on Computing System Engineering, SBESC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 93–98. doi:10.1109/SBESC.2012.26.
URL http://dx.doi.org/10.1109/SBESC.2012.26