Leveraging SSD's Flexible Address Mapping to Accelerate Data Copy Operations

Fan Ni†, Xingbo Wu⊥, Weijun Li‡, Lei Wang±, Song Jiang†
†*University of Texas at Arlington, Arlington, Texas, USA* ⊥*University of Illinois at Chicago, Chicago, Illinois, USA*
‡*Shenzhen Dapu Microelectronics Co. Ltd, Shenzhen, China* ±*Beihang University, Beijing, China*
†*fan.ni@mavs.uta.edu,* ⊥*wuxb@uic.edu,* ‡*liweijun@dputech.com,* ±*wanglei@buaa.edu.cn,* †*song.jiang@uta.edu*

***Abstract*—On-disk data copy is commonly performed in various software. Its current implementation with read and write commands needs to go through the I/O stack, introducing significant performance overhead. Actually when only one disk is involved in a copy operation, the data do not have to be read out of the device and the operation can be all performed within it. Furthermore, if the device is an SSD supporting flexible mapping from logical to physical addresses, the data do not need to be physically duplicated and the data copy purpose can be fulfilled by establishing a new address mapping. In the paper, we propose a copy primitive for SSD to support copy of block(s) of data within a device with almost zero cost regardless of amount of data. We show that it is relatively easy to implement the primitive, named *copyless copy*, or CC in short. We also evaluate two example uses of CC, database journaling and block-level deduplication, where the new primitive can help for efficient data copy as the case study. Our experiments with the use cases show that CC can dramatically increase the software's performance, reduce data writes to the device for improving SSD's endurance and space efficiency.**

***Keywords*-SSD; FTL; address mapping; copy; deduplication; journaling;**

I. Introduction

Copying of on-disk data is common in various system software and applications, including support of journaling in file systems[1], disk defragmentation [2], replication of files or directories requested by users and systems such as OverlayFS [3], UnionFS [4], support of snapshots of logical volume [5], and garbage collection in a log [6]. Its efficiency can have a significant impact on the system's performance, reliability, and space efficiency, which usually conflict with each other and are hardly to achieve at the same time.

Here is an example illustrating its implication on data reliability. The journaling technique is employed to keep a file system's reliability by maintaining its metadata and possibly data consistency. However, it relies on additional copying of the metadata/data and costly synchronization operations, such as flushes, to achieve the goal. Concerned of the high cost, often users would have to tolerate risks of data corruption by not writing data into the journal to avoid their later copy to the file. While this so-called metadata journal can substantially improve performance [7], it is at the expense of data integrity. If the copy cost can be significantly reduced or even eliminated, one can have both high performance and data reliability in a file system. Another example is about its implication on space efficiency. Major costs of garbage collection and disk defragmentation are attributed to data copy operations. Because the operation is expensive, garbages in a log cannot be quickly collected and the space held by them cannot be efficiently used. For the same reason, the disk has to remain fragmented for an extended period of time, which makes file systems, such as Ext4, ineffective on using their extent-based allocation feature and under-utilized space for metadata. If the cost of data copy operation can be dramatically reduced, garbage collection and defragmentation can be initiated whenever there is such a need without concerns on the high cost.

Enabling Copy Functionality in Disks: A copy operation copies a data block from one disk location to another one. There are two fundamental misconceptions in current practices about implementing and using the operation that lead to its unnecessarily high cost and unacceptable constraints on the design of upper-level software and applications.

First, when both the source and destination of the data are on the same disk, the operation can be more efficient performed if offloaded to the device. In this way, the software only needs to issue a copy command with parameters such as source and destination addresses and number of data blocks to be copied. However, the copy operation is currently treated as a software function that entails reading the data off the disk and then writing back to it. With the disk operations and possibly multiple layers of software involved and their respective overheads added into the function's execution, there is no doubt that the cost of the copy operation is much higher than what is necessary. While the side-effect of leaving a copy of the data in the memory buffer may be desired in some scenarios, reading the data into the buffer does not have to be on the critical path of the copy operation. Even if the data have been buffered in the memory, executing the write-back operation is more expensive than in-disk copying by consuming extra I/O bandwidth.

Second, implementing copy operations outside of the disk creates a semantic gap between the upper-level software that uses the functionality and the disk where the functionality is actually performed. The association between two parties of the operation, i.e., source and destination addresses, is only known to the software. Positioning the disk as a dumb device at the bottom of the I/O stack, current system design does not make such knowledge available at the disk. However, with

such knowledge the disk can enable the functionality in place with substantial convenience and performance advantage to the software.

Enabling Copyless Copy in SSDs: For the flash-based solid-state disk (SSD), the benefit of in-disk copy can be more than saving I/O bandwidth and reducing pollution of the system buffer. This is achieved by our proposed copyless copy, or *CC* in short, that leverages SSD's flexible address mapping to enable a copy without physical data replication.

A block device, such as hard disk and SSD, usually provides an indirection from logical block addresses (LBA), which exposes to the upper-level software, to physical block addresses (PBA). To this end, metadata needs to be maintained to facilitate the indirection so that operation requested by the software can be carried out on the corresponding physical blocks. For the hard disk, the metadata can be made small as correlation of the two set of addresses is well structured and remains stable. One example is that one segment of contiguous logical addresses are mapped to a segment of contiguous physical addresses on the same disk track, and the mapping relation usually does not change unless some unusual incidents, such as sector corruption, occur. However, flash-based SSD has to provide a much more flexible address mapping, allowing almost any LBA to be mapped to any PBA. One reason is that flash memory does not support write-in-place, or overwrite. That is, every write to an LBA will be re-directed to a new PBA, rather than to the PBA currently mapped to the LBA. Accordingly the mapping from LBA to PBA is updated. A firmware in modern SSDs, named flash translation layer (FTL), is in charge of managing the address mapping table as well as other corresponding data structures.

If the copy operation is implemented in SSDs, one can entirely avoid physical data duplications by simply updating the address mapping table. For example, copying a block from one logical address (LBA_1), which has been mapped to physical address (PBA) where the data is stored, to another logical address (LBA_2), requires only adding a new mapping from LBA_2 to PBA into the metadata in the FTL and remembering that now the PBA has one more logical address mapped to it. Figure 1 shows an example of copying data from one file to another with read/write operations and CC, respectively. This COW (copy-on-write)-style implementation of copy may raise the concern on compromised spatial locality with future random writes into the LBAs that have been involved in the copying if it was proposed for the hard disk [8], which relies on the locality to maintain high performance. However, for SSD such issue does not exist. SSD always uses out-of-place write and the locality is not expected anyway after random writes. Moreover, performance of today's SSD is little affected by the locality [9], [10].

One remaining question about CC is whether it can indeed avoid flash writes, or at least remove them out of critical path

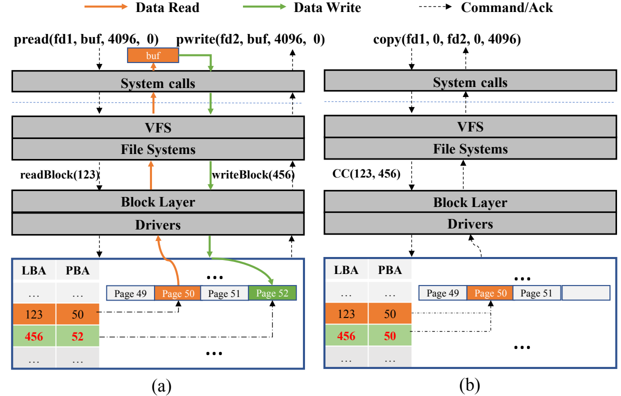


Figure 1: *Copying a 4KB block from the source file (fd1) to the destination file (fd2) with read/write system calls (a) or CC (b).*

of copy operations. As we have mentioned, CC does need to update corresponding metadata, specifically the mapping information and number of LBAs mapped to a PBA, and the metadata must not be lost even after an unexpected power loss. In SSDs at least recently used and updated metadata are cached in the DRAM. While the metadata updated by CC are not required to be written to the flash before the copy operation is considered completed, a mechanism should be in place to safeguard them against loss and corruption during a power loss. Fortunately, today's SSDs, including almost all enterprise class SSDs and many consumer class ones, have employed capacitors and techniques such as PLP (Power loss protection) [11] or PLI (Power Loss Imminent) [12] to flush the critical data in the buffer to the flash. Taking advantage of this readily available mechanism, the proposed copy operation can be truly copyless, or implemented without any access of the flash.

Our Contributions: SSD has become the mainstream storage device. With its clear performance advantage as well as increasingly larger capacity and lower price, more and more on-line processed data are stored on it. In the paper, by introducing copyless copy (CC) into the SSD we make several contributions: 1) We propose to leverage the flexible address mapping readily available in SSDs to perform copy operations within the device without physical data replication for high efficiency. 2) By introducing CC as a new primitive of SSDs, we design its implementation logic on a generic SSD architecture to show its incorporation needs minimal efforts and its execution can be of high efficiency. 3) To illustrate potential uses of CC, we showcase its applications in two important systems, database journaling and block-level deduplication. We show often dramatic performance improvements with little or moderate changes to the existing software, and the improvement of SSD's endurance and SSD's usable capacity.

II. Enabling Copyless Copy in SSDs

The CC (copyless copy) primitive has three arguments, which are source LBA, destination LBA, and number of

blocks to be copied. There are three design objectives for enabling CC in an SSD architecture. First, its implementation should require minimal changes to the SSD's FTL logic. This helps the primitive to be widely and easily incorporated into SSDs from various vendors. Though SSDs usually share a common generic architecture [9], they may have unique design and implementation specifics on aspects such as organization of mapping table(s) as well as use and maintenance of the table(s) for address translation. To minimize incompatibility with these existing implementations, we need to avoid modifications of the data structures and algorithms. Second, new data structures and operations on them for enabling CC should represent simple extension of existing ones, rather than disruptive ones that may produce less-predictable implications on logic and performance of existing operations, such as garbage collection. Third, while CC can certainly reduce service time for writes involved in the corresponding copy by only updating some metadata, it should not affect read with use of the metadata.

Updating Address Mapping Table.: NAND flash memory in SSDs does not support in-place write. Reading or writing data are performed at the unit of page, whose size can be a few kilobytes, such as 2KB, 4KB, or 8KB. However, a page cannot be re-written until it is erased, and the erase operation is performed in the unit of block, which can be as large as 128 or 256 pages, or a few megabytes. When a block is erased, idle pages in the block can be re-allocated to serve out-of-place writes. Servicing a write request entails allocation of idle page(s), writing the data, and updating LBA-to-PBA address mapping. To service a CC request, one only needs to simply remove the first two steps, and maps the destination LBA(s) to the PBA(s) mapped to by the source LBA(s).

As long as the FTL uses the page-level address mapping, i.e., the SSD's FTL allows an LBA to map to any PBA in the physical address space, the CC primitive can be seamlessly enabled in current FTL designs. With the increasingly large RAM space and existence of the access locality in the SSD's workloads, the page-level mapping has been widely used [13], [14], [15]. To save consumption of DRAM space block-level mapping [16], [17] was proposed, where the mapping scheme only determines the physical block address for an LBA and leaves its in-block offset (page) address the same as that of the lowest bits in the LBA. Being concerned with the high garbage collection cost of the mapping scheme, researchers have proposed hybrid mapping, where the mapping of an LBA to a fixed offset in a physical block is postponed [18], [19]. Because multiple pages in a block share one common entry in a block mapping table for translation from a logical block address to a physical block address, we cannot update the table according to a destination address in a CC operation. Instead, we attach a page-level-style extension to any table entry where CC's destination addresses have been mapped. Essentially the extension contains pointer(s) to the CC's source pages. While page-level mapping has advantages on addressing flexibility, space utilization, and write amplification, we will assume a page-level mapping hereafter, though the CC's implementation in FTL can also be applied to other styles of mappings with adaptation to their specific addressing schemes.

Expanding Valid Bit and Using Shadow OOB.: Besides an address mapping structure, SSD maintains two types of important metadata. One is the valid bit indicating whether a physical page in a block contains valid data. A page can be reclaimed and erased if its valid bit is cleared. The other is a 128B out-of-band (OOB) region associated with each physical page. The space is atomically written to the flash with the writing of the page. In addition to error-detection information, the LBA currently mapped to the page is stored in the region. With the mapping information about individual physical pages distributed in the flash, the address mapping table can be rebuilt even if the table or part of it gets lost accidentally for reasons such as power failure. However, rebuilding the table based on the mapping information embedded in the OOB region requires scanning every page in the SSD, which is slow and may significantly compromise availability of the device especially for a large-capacity SSD. Therefore, SSDs designed for performance are usually equipped with super-capacitors to prevent metadata in the RAM from being lost before they are persisted to the flash during a power loss.

To implement the CC primitive, we only need to enhance these two types of metadata (and introduce additions to the address mapping table if a block-level-style mapping scheme is used in the existing FTL). The first enhancement is to replace the valid bit with a n-bit reference count. We name all LBAs that are mapped to a common PBA by the CC primitive the PBA's *tag LBAs*. A PBA's reference count tracks number of tag LBAs of the PBA, and an n-bit reference count supports up to $2^n - 1$ tag LBAs for a PBA. A reference count of zero is equivalent to a valid bit indicating the physical page is invalid and ready for reclamation. Whenever a new destination LBA (specified in a CC request) is mapped to a PBA, the PBA's reference count is incremented by one. Whenever there is a write to a PBA's tag LBA, the PBA's reference count is decremented by one. This is similar to the COW (copy-on-write) operation in other memory or file management systems [20], [21]. However, as flash's use of out-of-place writes, the space required by a COW write is not more than that with a regular write. In other words, one CC operation involving k pages indeed removes writing of k pages, even when future COW writes to these pages are considered. While each physical page has limited erase/program cycles (typically 10,000 to 100,000), using CC can improve the SSD's endurance.

Most uses of address translation in an SSD are to translate an LBA to its mapped PBA. Meanwhile, there is one

scenario where a translation from PBA to LBA is required, which is to relocate a live page to a different physical page address. Two example uses of the operation are garbage collection and static wear-levelling. To collect garbage, or the invalid pages, in a block and make the block ready for erase, the FTL needs to relocate all live pages out of it. In another example, because flash memory has limited write/erase cycles, the FTL usually attempts to spread writes evenly across the SSD space. However, less frequently updated data can make the physical pages storing them less used. To address the issue, static wear leveling operation relocates the data in the page to a different location. In both examples, LBA that is mapped to the physical page being relocated (PBA) can be found in the OOB region associated with the PBA, and using the LBA the mapping for the LBA can be updated to the new PBA. With introduction of the CC primitive this logic of address updating would require little change if the OOB region could be updated, i.e., when a CC request is executed on the PBA, a new LBA was added into the OOB, and when the COW write is executed on the PBA, the corresponding LBA is removed from the OOB. The only difference is to operate possibly on multiple LBAs in an OOB region. However, usually the OOB region cannot be updated without re-writing its corresponding page (after an erase). To address this challenge, we create a shadow OOB, which can contain up to 2^n LBAs (n is number of bits for a reference count), in the SSD's DRAM for any physical page involved in a CC request to track its current tag LBA(s). The shadow OOB entry is removed when its corresponding physical page becomes invalid (or when its reference count becomes zero). For efficiency, shadow OOB entries usually are not immediately persisted after being updated. If the FTL relies on LBA information in the OOB regions to recover a corrupted LBA-to-PBA address mapping table, the entries need to be flushed to the flash with the power from a capacitor whenever there is a sudden loss of power.

The space held by the shadow OOB entries and time spent on their maintenance and use can be well justified. For each shadow OOB entry, there is a space saving of one or more physical pages on the flash and a time saving of one or more page writes. These savings further reduce garbage collection cost and improve the flash's endurance, as use of CC essentially increases the over-provisioning space, which is directly related to SSDs' performance [22]. Meanwhile, use of the OOB entries is usually not on the critical path of servicing users' read/write/CC requests. With the mapping table and its operations being minimally changed, we can dramatically accelerate the copy operation without affecting performance of regular read and write operations.

III. Use Cases of the CC Primitive

The CC primitive can be employed in various application scenarios and system software as long as block copying is involved. Generally, there are three opportunities of using CC in the software. One is existence of explicit copy operations, the second is to identify hidden copy operations, and the third is to create CC-applicable copy operations. Due to space limitation, we only discuss the the first two in the below.

For the first opportunity, data block copy operations have been in the existing software, and the software requires few modifications to take advantage of CC and immediately benefits from its performance advantage. This opportunity is substantially present in journaling operations, either in applications (such as SQLite [23]), or in file systems (such as Ext3 and Ext4 [7]), and in file-level Copy-on-write (COW) operations used in OverlayFS [24] and UnionFS [4] for creating COW containers from templates. For the second opportunity, the software may make efforts to determine whether data to be written has been resident on the same disk using techniques such as fingerprint comparison. If yes, an expensive disk write operation can be transformed into a very-low-cost CC operation (we will show how this approach can achieve a much higher I/O efficiency than a conventional deduplication system in Section III-B).

As an SSD device with the CC primitive enabled is not available yet, we implement it on the host Linux server at the generic operating system block device layer as a device mapper target [25], which maps a physical block device into a higher level virtual block device. Within the mapper we did not re-implement the entire FTL. Instead, we map literally every block address received at the virtual device to the same (LBA) address at the SSD. For the simulated CC primitive at the virtual device interface we avoid actual writes to the destination (LBA) address(es) at the SSD and maintain reference counts and shadow OOBs to help redirect affected block accesses. The virtual device periodically (once every second) persists these metadata onto a reserve space on the SSD. We believe this implementation emulating a hypothetical one within SSD carries a comparable, or even higher, request service cost as additional host-SSD communications are required, and the experiment measurements to be reported represent conservative results. In the design we assume block size at the device's interface is equal to or larger than the SSD's flash page size. Otherwise, even writing a block of data would require reading a flash page. To make the CC functionality available to user-level code, we provide a system call that supports CC copy between blocks in one or two files.

All our experiments were conducted on a Dell R630 server with two Xeon E5-2680v3 2.50GHz CPUs, each has twelve cores and 30MB last-level cache. The server equips with 128GB DDR4 memory, and a Samsung 840 EVO 1TB SSD.

A. **Replace Existing Copy Operations with CC**

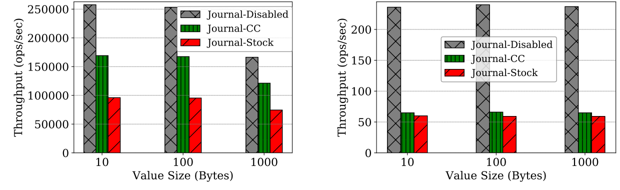
In the below we will use SQLite [23] as an example to show how CC can be directly applied to accelerate

journaling by replacing existing copy operations.

The journaling technique has been used for data consistency in face of application or system crashes and providing transaction support for databases and file systems, where an auxiliary log is maintained to record changes (write-ahead or redo log) or old data (rollback or undo log) before the changes are applied to the data. Journaling can be expensive when there is an extra write for every new or to-be-modified block. Concerned about the high cost, some file system users resort to use logical journals, in which only changes to metadata are logged. However, this may leave unlogged data and logged metadata inconsistent with each other, and causing data corruption. A database system, such as InnoDB [26], the default storage engine for MySQL, and SQLite, arguably the most widely deployed database engine [23], can use both redo log and redo log. As redo log has a number of disadvantages, such as its demand on support of shared memory primitive and extra checkpointing operation, undo log, which is SQLite's default logging method, can be a preferred choice [27]. However, using undo log can make the system slower, as copy old data blocks into the log stays on the critical path of a transaction's execution – it has to be completed before the original data can be modified. The CC technique helps remove the bottleneck.

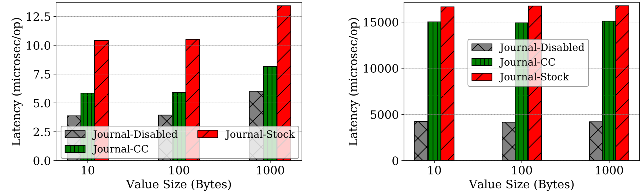
Applying CC in SQLite is straightforward, as SQLite uses page-level granularity and writes the entire original data pages along with the page numbers to the log as a new log record. One issue is that pages in the log file may not be page-aligned as they are mixed with the page number information. To address the issue, we set up a header page in a log file and collect the page numbers, total number of pages, and each page's checksum for removing one disk flush operation [28] into the page, and make the remaining data pages aligned, facilitating the use of CC to copy the data to the log. In the experiment, we use SQLite 3.16.2 with undo log, and configure the page size to 4KB. In addition to comparing the CC-enhanced SQLite, named *Journal-CC*, to the stock SQLite, named *Journal-Stock*, we also compare it to SQLite whose logging is disabled, named *Journal-Disabled*.

The benchmark is the one selected from a suite of benchmarks released by Google to compare performance of various databases, including SQLite, to that of LevelDB [29]. In the configuration, we let the selected benchmark for SQLite (*db_bench_sqlite3.cc*) issue a stream of (one million) requests, each for inserting a key and its associated value of certain size into the database. The keys can be issued in either sequential or random orders. The data records, or pairs of key and value, are indexed in a B tree. The database has a *synchronous* option, which can be configured to either *FULL* mode or *OFF* mode. In the FULL mode, *xSync* will be issued after a write, such as writing a log record or data file, and cleaning a log record. All contents are safely written to



(a) synchronous=OFF (b) synchronous=FULL

Figure 2: **Throughput of SQLite.**



(a) synchronous=OFF (b) synchronous=FULL

Figure 3: **Latency of SQLite.**

the disk before proceeding with subsequent requests, data safety and their consistency are ensured even in case of power failure. In the OFF mode, SQLite continues as soon as it has passed data to the operating system without waiting them to be persisted on the disk. Data and its consistency are protected if the SQLite applications fail as long as there is not a power loss and the operating system does not crash [30].

Figures 2 and 3 show the throughput and latency of *Journal-Disabled*, *Journal-CC*, and *Journal-Stock* with random keys and different value sizes when the *synchronous* option is FULL or OFF, respectively. All the measurements are reported by the benchmark itself. As we can see, with the synchronous OFF mode, the CC operation helps significantly improve SQLite's performance. For example, with random access, Journal-CC's improvements of throughput over Journal-Stock are 1.76X, 1.75X, and 1.63X, respectively. The improvements for sequential access are similar (not shown due to space limitation), implying that spatial access locality plays a less important role on SSD's performance. While the CC operation may compromise the locality on SSD, its impact is minimal. Journal-CC with small values is shown to have slightly higher performance advantage. The reason is that the same number of smaller values may reside in fewer pages, leading to fewer writes to the data files. However, each request causes writing of one log record, and the write amount is little affected by the value size. While CC helps to reduce the log-record write, it enjoys relatively higher performance advantage with smaller values.

The performance trend in terms of latency is similar. As we can see, Journal-CC's latency is almost always larger than that of Journal-Disabled by 2ms with different values. For example, their respective latencies for 100Byte-values

are 4ms and 6ms. As use of CC operation can remove all the cost with logging data pages and leave only one metadata page for logging, the consistent latency gap corresponds to this one-page logging cost (Journal-Disabled does not have any logging cost).

When *synchronous* option is FULL, the improvements by Journal-CC is relatively small, mostly at around 10%. In this mode, there are two or even more flushes with service of each request, flushes are expensive and their cost dominates the execution time. Removal of data writing cost in Journal-CC becomes less significant. It is noted that as SQLite is commonly used in embedded systems, such as smart phones, a power failure is less likely, and a *synchronous* OFF mode is sufficient to protect the data.

B. Identifying Hidden Copy Opportunity

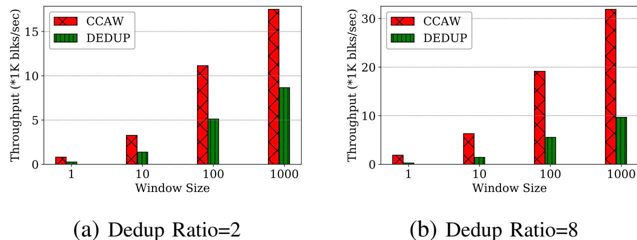
While introduction of the CC primitive makes it possible for copying blocks to be much cheaper than writing blocks of data into the SSD, we consider opportunity of converting writing operations into copy operations to take advantage of CC's performance advantage. To this end, for each block of data to be written we try to identify a block currently on the SSD with the exactly same content. If such a block is found, we replace the write with a CC from the block. When there are potentially a large number of redundant blocks on the SSD, this optimization assisted by CC can substantially improve write performance, make the disk space less occupied, and reduce wearing on the flash. These are benefits also claimed by a block-level deduplication system. However, we will explain and experimentally show why our CC-Assisted Writing design, named *CCAW*, has a clear advantage.

Design of the CC-Assisted Writing Scheme.: Like a deduplication system, CCAW uses a cryptographic hash function, such as MD5, to compute a signature for any block of data written to a logical block address (LBA) on the disk. When there have been multiple blocks of the same content on the disk, one signature can be associated to multiple LBAs (of the blocks). We maintain a signature table of mappings, each for a signature to LBA(s). By looking into the table with signature of the data to be written one knows whether a write can be transformed into a CC copying. Specifically, suppose that the original request is writing a block of data to disk address (LBA_{dest}), and the signature of the block is found in the signature table. From possibly multiple LBAs associated with the signature CCAW can picks any one, say LBA_{src}. Then a CC copying from LBA_{src} to LBA_{dest} replaces the original write.

However, the signature table needs to be carefully maintained. First, before a primitive of CC copying from LBA_{src} to LBA_{dest} is issued, we have to ensure that LBA_{dest} is removed from the signature table if it had been associated with a signature in the table to prevent it from mistakenly being identified as a copy source using its out-of-date signature. Furthermore, only after the CC primitive is complete can we associate LBA_{dest} to the signature of the currently written data for a similar reason. Second, for high efficiency we have to keep the table in the memory, whose size is between 0.1% and 0.5% of the amount of data on the SSD, depending on its duplication ratio. As an example, for a 1TB SSD filled with data, the table size is between 1GB and 5GB. Compared to the proposal of keeping a similar table in the SSD's RAM to enable in-SSD deduplication [31], using host memory to cache the table is much more affordable. If indeed the memory space is limited, CCAW shrinks the in-memory table by either discarding some less used entries or swapping them to the disk without compromising its correctness. Third, to maximize the opportunity for using CC to improve performance after a system restarts or recovers from a failure, it is necessary to keep a copy of the signature table on the disk, so that it can be loaded into memory to immediately help identify duplicate data on the disk, rather than rebuilding the table incrementally from scratch. To this end, it seems that we need to keep the on-disk table consistent to the in-memory table at any time so that an unexpected system failure does not invalidate the entire on-disk table. However, it is too costly to maintain a real-time consistency by writing to the disk with every update on the in-memory table. To address the issue, we partition the table into a number of small segments (4MB by default), and invalidate an on-disk segment once there is an update in it for the first time. The segment will be re-validated after all updates in it are later batch committed to the disk. The invalidation is infrequent compared to the table updates. In this way, a tradeoff is well made between usability of the on-disk table after a failure and additional I/O cost for maintaining the consistency.

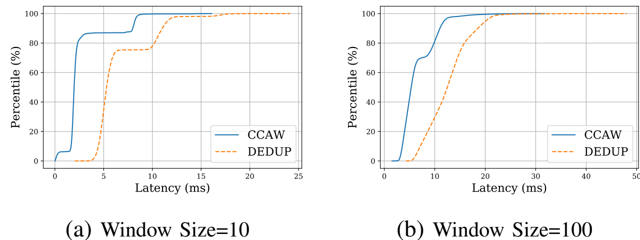
Evaluation of the CCAW Scheme.: To evaluate the CCAW scheme, we implement a prototype in the device mapper target where the CC primitive is enabled [25]. We use 4KB block size and 128-bit MD5 signature for a block. We generate synthetic block write traces and issue them continuously to the SSD. Each write is to a random LBA address in a 120GB volume on the SSD. In generation of the traces, we control the percentage of blocks whose writes can be converted into the CC operations among all written blocks, or conversion ratio. In the experiments we find that CCAW can improve the write throughput by almost $1/(1 - conversion_ratio)$, which is expected as CC removes actual disk writes in the percentage of the conversion ratio. To make a stronger case for CCAW's performance advantage, we focus on a comparison with a block-level deduplication system on the host server. For every write converted to a CC copying, it is removed from the stream of write requests to the disk, or deduplicated, in the deduplication system.

If CCAW is considered as a deduplication scheme, it represents a software-hardware co-design. However, existing



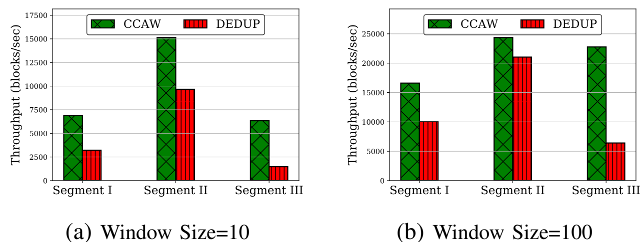
(a) Dedup Ratio=2 (b) Dedup Ratio=8

Figure 4: **Throughput of CCAW and the deduplication system (DEDUP) with various deduplication ratios and window sizes.**



(a) Window Size=10 (b) Window Size=100

Figure 5: **CDF curves of write latency for CCAW and DEDUP with various window sizes when deduplication ratio is 2.**



(a) Window Size=10 (b) Window Size=100

Figure 6: **Throughput of CCAW and DEDUP with a real-world trace for various window sizes. Three segments (Segment I, II, and III) of traces are used with deduplication ratios being 1.42, 1.55, and 4.22, respectively.**

systems have to maintain critical metadata required by deduplication all by themselves. First, they have to introduce a new layer of indirection from an address space exposed to upper-level software using the deduplication service to the LBA address space exposed by the disk, and maintain corresponding mappings. They also track reference count for each block in addition to the management of fingerprints. Second, they need to provide necessary persistency and consistency to the metadata. To this end, they have to use additional writes to persist metadata and flushes to enforce write order between metadata and data. For example, the address mappings, which can be (much) smaller than a block size, need to be frequently persisted to the disk in the unit of block. Regarding consistency, incrementing reference count for an LBA has to be performed before this LBA is mapped to by a new deduplicated block. The address mapping must be persisted after writing of the corresponding non-deduplicated block is complete. Expensive flushes are required to enforce the orders.

For comparison, we implement a recently proposed deduplication scheme, OrderMergeDedup [32], which makes substantial efforts to reduce metadata writes and flushes, and use it as a representative of block-level deduplication system, named *DEDUP* in short. In the implementation, write requests are grouped into windows according to their arrival time. Metadata related to requests in a window are persisted after all data writes are issued and a flush is complete. After the batched metadata write, another flush is issued to ensure all requests in the window are successfully serviced. For a fair comparison we also add a flush at the end of each window for CCAW. In the experiments, we vary window size (in terms of number of blocks for writing) and deduplication ratio, which is $1/(1 - conversion_ratio)$.

Figure 4 shows the write throughput of CCAW and DEDUP with various deduplication ratios and window sizes. As shown, at a certain deduplication ratio increasing window size can substantially improve both systems' throughput, as it reduces frequency of issuing flushes and writing metadata. Because DEDUP carries higher overheads with the two operations, its throughput is more significantly improved due to the reduced frequency than CCAW's. However, for each window size CCAW's throughput is much higher than DEDUP's by eliminating most metadata writes and extra flushes. For example, with a deduplication ratio of 2, CCAW's improvements are 3.1×, 2.3×, 2.2×, and 2.0× at window sizes of 1, 10, 100, and 1000, respectively, over DEDUP. With a higher deduplication ratio, metadata operations, which can be mostly removed in CCAW, account for a higher portion of request service time in DEDUP. As expected, with a higher ratio, higher throughput improvements are made by CCAW. For example, at a window size of 10, the improvements are 2.37× and 4.39× for deduplication ratios of 2 and 8, respectively, over DEDUP.

Figure 5 shows CDF curves of write latencies for CCAW and DEDUP with window size being 10 and 100 and deduplication ratio being 2. The latency measures the time period from a request entering a window to its service completion. A higher throughput can be achieved by increasing the window size at the expense of higher latency. However, CCAW can retain low latency even if the window size increases. For example, at a window size of 100, 80% of requests have latency lower than 10ms in CCAW, while only 30% of requests have latency lower than 10ms in DEDUP. If one wants to reduce DEDUP's latency, at a window size of 10 its 80 percentile latency can be reduced to 11ms with its throughput reduced to only 24% of that at the window size of 100.

Figure 6 shows the throughput for a real-world trace collected on a mail server at FIU [33]. It covers requests in 21 continuous days. We selects three segments, each for a day, with distinct deduplication ratios. In the experiment, the

system is warmed up with requests preceding the selected segments. In general CCAW achieves a higher throughput than DEDUP at a rate in a range from 7% to 331%. The lower improvements occur when read requests dominate the workload (e.g., Segment II), the window size is large, and/or, the deduplication rate is low.

IV. Related Work

There are many efforts related to the CC work, including reduction of SSD writes, and offloading data transfer.

Removal of SSD Writes: SSD's write is especially expensive as it is slow and incurs subsequent erase and garbage collections. To address this issue, Delta-FTL replaces a full-block re-write with a postponed partial block write if the change made in the new write is small [34]. However, for synchronous writes persistency of the partial block data can be an issue. CAFTL implements a block-level deduplication in the SSD modified FTL [31]. We have demonstrated that CC can help to enable an efficienct deduplication system at the host. In contrast, the all-in-SSD deduplication has a few drawbacks. First, it may complicate FTL logic and a large amount of metadata including finerprints, are requried to be held in SSD's DRAM, which can otherwise be used to cache hot data. Second, processor in the SSD, which is much less powerful than the one on the host, has to calculate fingerprints. Using a hardware-software co-dedesign, CC-assiated deduplication well addresses the problems.

Offloading Data Transfer Operation: Data transfer operations are often offloaded to reduce involvement main CPU and data movements. For example, data transfer between memory and the disk is offloaded to direct memory access (DMA) engine [35]. Intel proposes I/O Acceleration Technology (I/OAT) to enable DMA between memory and NIC [36]. The zero-copy technology eliminates data transfer between user and kernel spaces during data transfer between files on devices such as disks and network sockets [37], [38]. Examples include TransmitFile function [39] in Windows and *sendfile* system calls in Linux.

The EXTENDED COPY (XCOPY) command in SCSI standard is another example of offloading data copying operation to *copy manager*, which copies data from source to destination devices [40]. It is commonly used for accelerate backup tasks. Though main CPU or even the host server is not involved in the execution, I/O accesses are not reduced: data are still read from and then written to device(s).

In contrast, the CC primitive offloads the copy operation to the FTL engine in SSD. A unique feature is that CC completely eliminates data writes in its service of copy requests, and achieves an almost-zero-cost copying on storage devices, rather than just a zero-copy for in-memory data transfer, by leveraging SSD's flexible address mapping capability.

V. Conclusion

In this paper, we propose Copyless Copy, an SSD primitive, to enable highly efficient data block copying. With the primitive, the copy operation can be performed by only updating metadata without physical data read and write. We showcase its applications in a wide range of important systems and user software and experimentally demonstrate its significant performance benefits with little or moderate changes of the existing software. It also helps with SSD's endurance and effective capacity.

Acknowledgment

We are grateful to the reviewers for their valuable comments and feedback. This work was supported in part by Song Jiang's UTA startup fund. Weijun Li was supported by Shenzhen Peacock Plan (KQTD2015091716453118). Lei Wang was supported by National Natural Science Foundation of China (No. 61672073).

References

[1] S. C. Tweedie, "Journaling the Linux ext2fs Filesystem," in *The Fourth Annual Linux Expo*, 1998.

[2] J. K. Edwards and J. Heller, "File system defragmentation technique via write allocation," 2005, uS Patent 6,978,283.

[3] N. Stahl and A. Khan, "Copy-on-write mapping file system," 2004, uS Patent App. 10/841,808.

[4] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok, "Unionfs: User-and community-oriented development of a unification filesystem," in *Proceedings of the 2006 Linux Symposium*, vol. 2, 2006, pp. 349–362.

[5] M. Hasenstein, "The logical volume manager (LVM)," *White paper*, 2001.

[6] D. Woodhouse, "JFFS: The journalling flash file system," in *Ottawa linux symposium*, vol. 2001, 2001.

[7] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and Evolution of Journaling File Systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 8–8.

[8] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, Inline Data Deduplication for Primary Storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 24–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208485

[9] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08, Berkeley, CA, USA, 2008, pp. 57–70.

[10] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37. ACM, 2009, pp. 181–192.

[11] Samsung, "Samsung: Power loss protection (PLP) – Protect your data against sudden power loss," https://goo.gl/0CMk0a.

[12] Intel Corporation, "Intel: Power Loss Imminent (PLI) Technology," https://goo.gl/zfHXzD, 2014.

[13] A. Ban, "Flash file system," April 1995, uS Patent 5,404,485.

[14] 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 XIV. New York, NY, USA: ACM, 2009, pp. 229–240.

[15] D. Ma, J. Feng, and G. Li, "LazyFTL: A Page-level Flash Translation Layer Optimized for NAND Flash Memory," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1–12.

[16] D. Liu, T. Wang, Y. Wang, Z. Qin, and Z. Shao, "A block-level flash memory management scheme for reducing write activities in PCM-based embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 1447–1450.

[17] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Demand-based block-level address mapping in large-scale NAND flash storage systems," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*. IEEE, 2010, pp. 173–182.

[18] D. Jung, J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "Superblock FTL: a superblock-based flash translation layer with a hybrid address translation scheme," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 4, p. 40, 2010.

[19] D. Park, B. Debnath, and D. Du, "CFTL: A convertible flash translation layer adaptive to data access patterns," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1, pp. 365–366, 2010.

[20] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[21] D. Hitz, M. Malcolm, J. Lau, and B. Rakitzis, "Copy on write file system consistency and block usage," May 2005, uS Patent 6,892,211.

[22] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: ACM, 2009, pp. 10:1–10:9.

[23] P. domain, "SQLite Home Page," https://sqlite.org/, 2000.

[24] N. Brown, "Overlay Filesystem," http://goo.gl/VlBg98, 2014.

[25] Wikimedia Foundation, Inc, "Linux Device Mapper," https://en.wikipedia.org/wiki/Device_mapper.

[26] P. Frühwirt, M. Huber, M. Mulazzani, and E. R. Weippl, "Innodb database forensics," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 2010, pp. 1028–1036.

[27] P. domain, "SQLite Home Page," https://www.sqlite.org/wal.html, 2000.

[28] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *IRON File Systems*, ser. SOSP '05. New York, NY, USA: ACM, 2005.

[29] Symas Corporation, "Database microbenchmarks," http://www.lmdb.tech/bench/microbench/.

[30] P. domain, "SQLite: PRAGMA Statements," https://www.sqlite.org/pragma.html#pragma_synchronous, 2000.

[31] F. Chen, T. Luo, and 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*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 6–6.

[32] Z. Chen and K. Shen, "OrderMergeDedup: Efficient, Failure-consistent Deduplication on Flash," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 291–299.

[33] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," *Trans. Storage*, vol. 6, no. 3, pp. 13:1–13:26, Sep. 2010.

[34] G. Wu and X. He, "Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 253–266.

[35] G. V. Kabenjian, "Method and apparatus for performing efficient direct memory access data transfers," March 1997, uS Patent 5,613,162.

[36] K. Lauritzen, T. Sawicki, T. Stachura, and C. E. Wilson, "Intel (R) I/O acceleration technology improves network performance, reliability and efficiently," *Technology@ Intel Magazine*, pp. 3–11, 2005.

[37] D. Stancevic, "Zero copy I: user-mode perspective," *Linux Journal*, vol. 2003, no. 105, p. 3, 2003.

[38] M. N. Thadani and Y. A. Khalidi, *An efficient zero-copy I/O framework for UNIX*. Citeseer, 1995.

[39] Microsoft, "Transmitfile function," https://msdn.microsoft.com/en-us/library/windows/desktop/ms740565(v=vs.85).aspx.

[40] T10.org, "EXTENDED COPY command," www.t10.org/ftp/t10/document.99/99-143r1.pdf, 1999.