LAWN: Boosting the performance of NVMM File System through Reducing Write Amplification

Chundong Wang
Singapore University of Technology and Design
cd_wang@outlook.com

Sudipta Chattopadhyay
Singapore University of Technology and Design
sudipta_chattopadhyay@sutd.edu.sg

ABSTRACT

Byte-addressable non-volatile memories can be used with DRAM to build a hybrid memory system of volatile/non-volatile main memory (NVMM). NVMM file systems demand consistency techniques such as logging and copy-on-write to guarantee data consistency in case of system crashes. However, conventional consistency techniques may incur write amplification that severely degrades the file system performance. In this paper, we propose LAWN (*logless, alternate writing for NVMM*), a novel approach that achieves data consistency and significantly improves performance via reducing write amplification. Our evaluation reveals that LAWN boosts the performance of a state-of-the-art NVMM file system by up to 12.0×.

CCS CONCEPTS

• **Software and its engineering** → **File systems management**; **Consistency**; • **Information systems** → **Storage class memory**; • **Hardware** → **Non-volatile memory**;

KEYWORDS

NVMM File System, Alternate Writing, Data Consistency, NVM

ACM Reference Format:
Chundong Wang and Sudipta Chattopadhyay. 2018. LAWN: Boosting the performance of NVMM File System through Reducing Write Amplification. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196066

1 INTRODUCTION

The byte-addressable non-volatile memory (NVM) technologies, such as PCRAM [1, 2], STT-RAM [3–6], 3D XPoint [7], and ReRAM [8], have DRAM-like access latency and disk-like non-volatility. Therefore, NVM can be placed on the memory bus alongside DRAM to build a hybrid memory system of volatile/non-volatile main memory (NVMM). NVMM blurs the conventional boundary between main memory and storage device [9–18].

To leverage the performance potential of NVMM, NVMM file systems have been proposed with a technique called *Direct Access* (DAX) [9–13]. As illustrated by Figure 1, DAX facilitates an NVMM

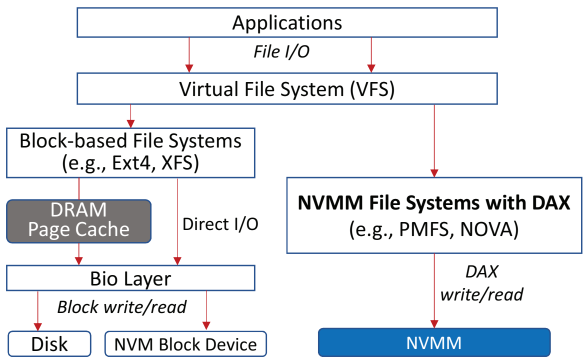

Applications
File I/O
Virtual File System (VFS)
Block-based File Systems (e.g., Ext4, XFS)
DRAM Page Cache
Direct I/O
NVMM File Systems with DAX (e.g., PMFS, NOVA)
Bio Layer
Block write/read
DAX write/read
Disk
NVM Block Device
NVMM

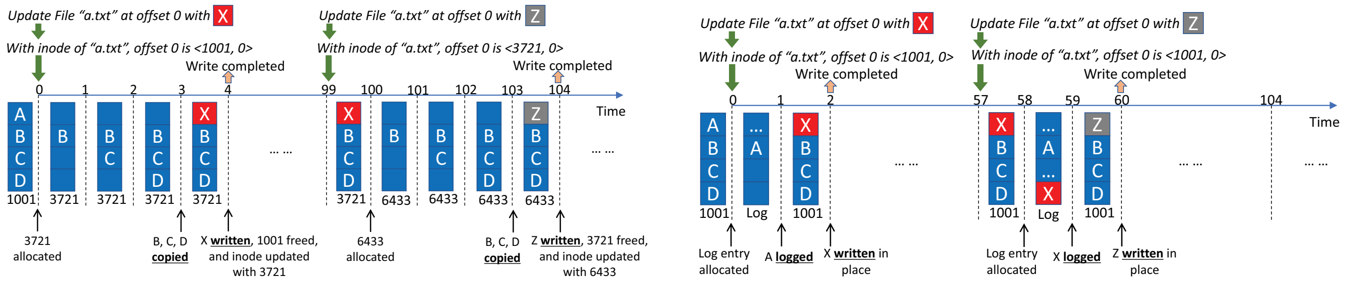
Figure 1: The Architecture of NVMM File Systems with DAX

file system to bypass the DRAM page cache by directly transferring application data between user space and NVMM space. Furthermore, DAX enables byte-level flexible data access across NVMM without involving the DRAM page cache.

A file system should attend the *crash consistency* due to the inherent unpredictability of system crashes (e.g., a sudden power failure). In particular, the consistency of file-system metadata (*metadata consistency*), such as an `inode` of each file, is of paramount importance and hence, it must be preserved in the event of system crashes. Some NVMM file systems further guarantee the consistency of user data stored in files (*data consistency*) [10–12, 14, 17]. In this paper, our focus is on the data consistency. Copy-on-write (COW) and logging are two prevalent techniques for crash consistency. COW updates the data of files out of place and substitutes the original data by changing the metadata of respective files. In contrast, logging does not change the metadata of a file, but it makes a backup copy of the file data for recovery. As an example, redo logging first commits modified data to a log and subsequently, updates file data in place. In short, logging has to write the same data twice [10, 17–20].

NVMM file systems primarily manage NVMM space in the unit of 4KB page to simplify the underlying design and implementation [9–11]. In light of DAX, page-based logging successively writes two pages to NVMM *in the critical path* [16, 18], which incurs severe performance penalty. NVMM file systems that cover data consistency hence prefer page-based COW [9, 11, 13]. However, page-based COW degrades performance with significant *write amplification* in updating small pieces of data. For example, on modifying a small fragment of a 4KB page, say, a cache line of 64B, page-based COW will copy the unmodified 4032B to a newly-allocated NVMM page alongside writing the modified 64B. Such a strategy results in 64× execution time compared to solely updating 64B data. Added to this, page-based COW compromises the access efficiency of byte-addressable NVM, as it reshapes all writes to be page-level.

A feasible solution to mitigate the write amplification of page-based COW is to replace it with fine-grained logging that only


Update File "a.txt" at offset 0 with X
With inode of "a.txt", offset 0 is <1001, 0>
Write completed
Update File "a.txt" at offset 0 with Z
With inode of "a.txt", offset 0 is <3721, 0>
Write completed
Time
3721 allocated
B, C, D copied
X written, 1001 freed, and inode updated with 3721
6433 allocated
B, C, D copied
Z written, 3721 freed, and inode updated with 6433
Update File "a.txt" at offset 0 with X
With inode of "a.txt", offset 0 is <1001, 0>
Write completed
Update File "a.txt" at offset 0 with Z
With inode of "a.txt", offset 0 is <1001, 0>
Write completed
Time
Log entry allocated
A logged
X written in place
Log entry allocated
X logged
Z written in place

(a) Execution time of Page-based COW

(b) Execution Time of Fine-grained Logging

Figure 2: An Illustration of Write Amplification in NVMM File System with Page-based COW and Fine-grained Log

records the updated data for backup. In this fashion, fine-grained logging writes much less data as compared to page-based COW. However, it still entails write amplification and yields suboptimal performance due to writing the same data twice in the critical path.

To avoid write amplification for higher performance, we propose LAWN (*logless, alternate writing for NVMM*). The main ideas behind LAWN are as follows.

- LAWN manages a fine-grained zone for files in NVMM. Given a small piece of file data to be updated, LAWN makes the copy in the zone and the copy in the file be *mutual backup copies*.
- LAWN leverages a scheme called *alternate writing* that writes only once in the critical path on updating a small piece of file data, but strictly regulates how the updated data is written to the zone or the file so as to avoid any inconsistencies.

We have prototyped LAWN in state-of-the-art NOVA [11] and performed extensive experiments. Experimental results show that LAWN is able to boost the performance of NOVA by up to 12.0×. This is due to the significant reduction in write amplification.

The remainder of the paper is organized as follows. Section 2 details the background of NVMM file systems and Section 3 outlines the cause of write amplification. Section 4 presents the building blocks of LAWN. In Section 5, we describe our LAWN prototype and the evaluation results. Section 6 concludes the paper.

2 BACKGROUND

The next-generation byte-addressable are under prosperous development and supposed to sit alongside DRAM on the memory bus for CPU to directly load and store data. PCRAM, ReRAM, 3D XPoint, and STT-RAM are good candidates in building NVMM with DRAM. The first three have longer write latencies but higher density than DRAM. STT-RAM has lower power consumption than DRAM and shorter access latencies than PCRAM.

State-of-the-art NVMM file systems, like BPFS [9], PMFS [10], and NOVA [11, 13], define and manipulate file system metadata, like `inodes`, in a byte-addressable way while managing the NVMM space for file data in the unit of page. They use DAX to access file data. DAX does not buffer data pages in DRAM page cache but directly writes and reads data with NVMM. Bypassing DRAM page cache helps to avoid copying data between DRAM and NVMM in the storage stack and enables flexible access across NVMM.

A system crash may happen at any time and result in data loss. File systems employ different techniques for different levels of crash consistency [10–12]. In regard to data consistency that guarantees the consistency of both file system metadata and file data, NVMM file systems prefer COW for updating file data, because COW is more efficient than logging in light of DAX. On updating a data page of a file, logging must write two pages to NVMM in the critical path. Yet COW just writes one data page as well as a small pointer.

There are also other challenges for data consistency in NVMM. Unlike hard disks, CPUs just support 8B atomic write to memory. Instructions like `cmpxchg16b` (with the LOCK prefix) can atomically write data of 16B. Atomically writing data greater than 16B to memory demands special architectural support [10]. Worse, writing multiple cache lines to memory may not adhere to the programmed order [10, 14, 16]. Inconsistency issues may arise upon an altered writing order. For example, recording a new file in a directory must be done after the file creation. If the directory was modified before creating the file but a crash occurred, a directory entry would show the existence of a non-existent file.

We can utilize cache line flush (e.g., `clflush`) and memory fence (e.g., `sfence`) to follow regular store instructions (e.g., `mov`) for a desired writing order. `clflush` explicitly invalidates data in a cache line and flushes it to memory. `sfence` enforces that store operations after an `sfence` cannot proceed unless those before it have been completed. Thus, a series of {`sfence`, `clflush`, `sfence`} persists multiple cache lines to memory in order. New instructions for cache line flush (`clflushopt` or `clwb`) have been proposed. In this paper we use `clflush` for illustration because of its wide availability.

3 MOTIVATION

Page-based COW incurs severe write amplification in the critical path when an application asks an NVMM file system with DAX to update only a small part of a page. Figure 2(a) shows an example on how page-based COW behaves in a time series. Assume that a page has four pieces of data indexed by 0, 1, 2, and 3. The scale unit of time axis is the execution time for writing one piece of data to NVMM. An application is to use data '*X*' to replace '*A*' in file a.txt, which stays at the offset 0 of page 1001. Page-based COW first allocates a free page, i.e., 3721 in Figure 2(a). It copies three unmodified pieces of data, i.e., '*B*', '*C*', and '*D*', to page 3721. Then '*X*' is written. When '*X*' is to be replaced by '*Z*' later, page-based COW will copy unchanged '*B*', '*C*', and '*D*' again before writing '*Z*'. Thus, page-based COW makes writing a small piece of data identical to writing an entire page, which badly impairs performance.

Write amplification caused by COW also exist for hard disk drives and flash-based solid state drives. However, due to their access unit and slow speed, writing data to them is performed with

DRAM page cache that helps to alleviate write amplification via buffering. NVMM with DAX yet bypasses DRAM page cache.

One straightforward method to mitigate write amplification of COW is to make the unit of COW fine-grained, either dynamic or fixed. Nonetheless, fine-grained COW is difficult in practice for two reasons. First, as applications can write data in any size at any offset of a file, NVMM file system must consistently maintain a complicated index structure in each file to trace every modified piece. This demands considerable efforts in design and implementation, especially with a dynamic COW unit [15]. Second, compared to managing NVMM space in the uniform pages, fine-grained COW shall dramatically increase the complexity of space management.

A doable approach to reduce write amplification is using a fine-grained log that is like a database transaction log. Take redo logging for illustration. Before updating data in place, fine-grained logging commits the write request's information along with data into an in-NVM log in the format of ⟨`inode` *number*, *offset in file*, *length*, *data*⟩. Figure 2(b) indicates how the fine-grained redo logging deals with the aforesaid example. It first commits '*X*' to log as a backup copy for recovery and then writes '*X*' in situ. When '*Z*' is used to overwrite '*X*', '*Z*' will be logged. Obviously fine-grained logging spends much less execution time than page-based COW. Whereas, it is still suboptimal as with write amplification. As shown in Figure 2(b), fine-grained logging has to write the same data twice in the critical path, one to the log and the other one to the file. The logged write request's information also has to be recorded in NVMM.

4 LAWN

In order to reduce write amplification without loss of data consistency, we propose LAWN (*l*ogless, *a*lternate *w*riting for *N*VMM). LAWN includes a fine-grained *zone* in NVMM and a scheme called *alternate writing*. In brief, with alternate writing, LAWN writes data to the zone or file only once in the critical path for each small update request but enforces a strict execution order for consistency.

4.1 LAWN's Components

Figure 3 illustrates the components of LAWN in NVMM and DRAM.

- The zone is a logical area in NVMM, separated from the space of NVMM file system. It comprises sub-zones, the number of which is equal to the number of CPU cores. Each sub-zone is designated with a CPU core for serving a quantity of files (`inode`s), the number of which depends on a specific NVMM file system, the NVMM size, and the number of sub-zones. In Figure 3, a sub-zone is dedicated to 200 `inode`s ('ino' means `inode` number). A sub-zone is cache line-aligned and contains numerous zone slots. A zone slot has a uniform size of one or multiple times of cache line size for cache efficiency. We call data stored in a zone slot a *data slice*. A file page is evenly divided into data slices. A data slice can be freely placed in any zone slot of the file's designated sub-zone. So a sub-zone can be viewed as a fully associative cache to relevant files.
- The descriptor table keeps a descriptor for each slot in NVMM and consists of sub-tables for sub-zones. A descriptor has two fields for a data slice: `inode` number (8B) and in-file offset (8B), and can be modified in a 16B atomic write, which is extremely useful for consistency. When a data slice becomes obsolete, like

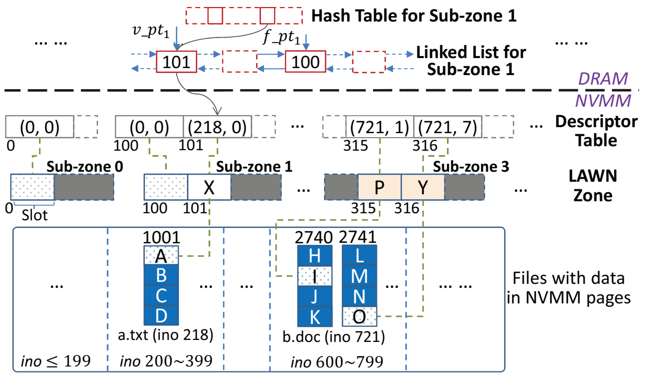


Figure 3: An Illustration of LAWN's Components

ones in slots 0 and 100 in Figure 3, its corresponding descriptor is cleared to be zeros (the zero `inode` number is usually reserved by file system and inapplicable for ordinary files).
- We maintain a hash table and a linked list for each sub-zone in DRAM for two purposes. First, the hash table can accelerate lookups to check if the up-to-date version of a data slice is stored in a zone slot or not. Second, a combination of hash table and linked list helps us to locate slots with the least-recently-used (LRU) valid data as well as free slots with obsolete data. A *free pointer* (f_pt_i) and a *valid pointer* (v_pt_i) are used to trace these two types of slots in the ith sub-zone. In Figure 3, f_pt_1 and v_pt_1 refer to slots 100 and 101 of sub-zone 1, respectively.

Initially we preallocate 3% of NVMM space for the use of LAWN. Given an NVM device of 128GB, the entire zone takes 3.8GB, which is sufficiently large as compared to the default 128MB log (journal) of Ext4 [17, 19]. If the percentage of free slots in all slots of a sub-zone drops below a threshold, say, 10%, which may entail swapping data with files to release slots, we will allocate more NVMM space to expand that sub-zone. To further avoid swapping in the critical path of updating data, we employ a background thread that writes back the LRU valid data slices at the system's idle time.

In-DRAM structures can be reconstructed by scanning the descriptor table, so we keep them volatile in DRAM and rebuild them at mounting file system. The spatial cost for them is insignificant. To index a slot we use 4B (32-bit) so that the maximum zone size can be 256GB with 64B per slot. In implementation a hash table and a linked list require 4B and 12B for a slot, respectively. Given the aforesaid 3.8GB zone, in-DRAM structures take up $\frac{3.8\text{GB}}{64\text{B}} \times 16\text{B} = 0.95\text{GB}$. This is about 0.8% in size as compared to the overall NVMM space.

4.2 Alternate Writing of LAWN

Leveraging aforementioned components, LAWN employs a scheme called *alternate writing* that updates file data with minimized write amplification and achieves data consistency. As its name suggests, alternate writing writes updated data to LAWN zone and file system in turn at runtime. Algorithm 1 illustrates how it regulates the process of writing a data slice d for an application.

(1) LAWN looks up in the corresponding ith descriptor sub-table with d's `inode` number and in-file offset to check whether the up-to-date version of d is stored in a zone slot or the file (Line 1).
(2) If the up-to-date version of d is not in the ith sub-zone, LAWN will write the newer d into a free zone slot (Lines 2 to 6).

Algorithm 1 LAWN's Alternate Writing (`alternate_write()`)

```
Input: A data slice $d$ to be written; //$d$ is with inode number and in-file offset
 1: if (is_valid_data_in_zone($d$) == false) then   //Not in a zone slot
 2:     Get a free slot $\zeta$ in the $i$th sub-zone with an exclusive use of $f\_pt_i$;
 3:     Write $d$ to $\zeta$ through memcpy;
 4:     clflush($\zeta$), sfence;
 5:     Write $d$'s information into $\zeta$'s descriptor ($\gamma$) in an atomic write;
 6:     clflush($\gamma$), sfence;
 7: else //the up-to-date version of $d$ is found with descriptor $\theta$ in the $i$th sub-zone
 8:     Write $d$ to its location ($\varphi$) in file through memcpy;
 9:     clflush($\varphi$), sfence;
10:     Clear $d$'s descriptor ($\theta$) to be zeros in an atomic write;
11:     clflush($\theta$), sfence;
12: end if
13: Update in-DRAM structures where necessary;
14: Return completion or fail of writing $d$;
```

(a) LAWN exclusively acquires f_pt_i for a free slot ζ (Line 2). Then it moves f_pt_i forward by one and releases f_pt_i.
(b) LAWN commits d to ζ with `clflush` and `sfence` followed to enforce a writing order (Lines 3 to 4).
(c) LAWN atomically writes the information of d (`inode` number, d's in-file offset, and the size of d) into ζ's descriptor with `clflush` and `sfence` followed (Lines 5 to 6).

(3) Otherwise, if the up-to-date version of d is in a zone slot, LAWN writes the newer d into file system (Lines 8 to 11).
(a) LAWN commits d to the file with `clflush` and `sfence` followed (Lines 8 to 9).
(b) LAWN clears d's corresponding descriptor in an atomic write with `clflush` and `sfence` followed (Lines 10 to 11).

(4) LAWN updates in-DRAM structures where necessary (Line 13).
(5) At last LAWN returns the completion or fail of writing d (Line 14).

The essence of alternate writing is that, LAWN makes the two versions of a data slice, which are respectively stored in the zone and file, be *mutual backup copies*. A crash that happens in writing the newly-arrived d to the file never results in inconsistency because the last up-to-date version is still valid and retrievable in a zone slot; vice versa. Hence, LAWN does not make a backup copy for updating data like logging or COW. It substantially reduces write amplification by writing a data slice only once in the critical path.

In Algorithm 1, we use `clflush`, `sfence`, and 16B atomic write in alternate writing to ensure data consistency. A combination of `clflush` and `sfence` imposes that writing updated data to a zone slot or file is always completed prior to modifying a corresponding descriptor. By doing so, a crash cannot leave a modified descriptor that indicates unreliable data. The 16B atomic write that sets or clears a descriptor determines whether the updated data slice is committed to NVMM. Assume that we should write newer d to a zone slot with the up-to-date version being in file. If a crash occurs before atomically setting the descriptor, in recovery there will be no descriptor related to d. So newer d has not been committed and the in-file verison is deemed to be valid. If the crash occurs after setting the descriptor, newer d has been successfully committed to the zone and will be used as the valid version. This is identical to the data consistency achieved by the classic Ext4 at the mode of `data=journal`, in which data committed to the log (journal) would be valid after a crash [17, 19].

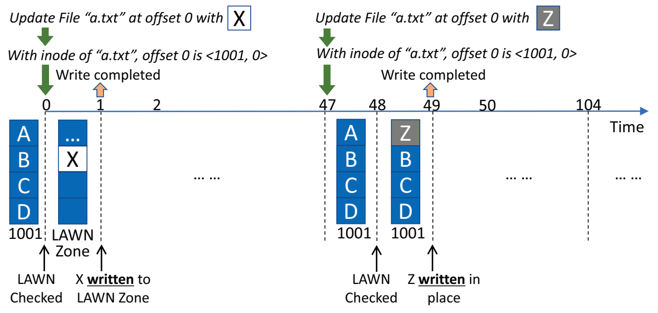


Figure 4: Execution Time of LAWN with Alternate Writing

4.3 File Write and Read with LAWN

A file write request goes through LAWN first. LAWN divides data D in a write request into n parts when D spans n pages according to the page boundaries in NVMM file system. In practice, with a write request taking data for n pages ($n \geq 2$), only the head and/or tail pages can be fit for alternate writing as these two pages may receive less data than a page size; the remaining pages will follow the routine of page-based COW. For the data to be written via alternate writing, LAWN proceeds as follows.

(1) In line with slot boundaries in a page, LAWN chunks the data into k slices, each of which can be fitted in a zone slot. Ones that are less than a slot size will be patched with unmodified data in NVMM or zeros if at either end of file.
(2) LAWN calls `alternate_write` to write k slices one by one.
(3) The file system returns the completion or fail of writing D.

Figure 4 shows the execution time of LAWN in processing the aforesaid example. First, 'X' is written to a zone slot. Then, once the information of 'X' is recorded in the descriptor table, writing 'X' is completed. Later, to substitute 'X' with 'Z', LAWN writes 'Z' to the file in place as 'X' is stored in the zone. A comparison of Figure 2 and Figure 4 evidently shows that with alternate writing, LAWN remarkably saves time by avoidance of write amplification.

A file read request also needs to ask LAWN because the up-to-date version of data may reside in a zone slot. At memory access speed, sequential read and random read yield comparable performance. So reading data from zone slots and/or file pages should achieve identical performance as compared to reading data in original NVMM file system. This will be tested in Section 5.2 and 5.4.

LAWN supports concurrent write and read requests in a multitasking environment. There are two cases for applications that concurrently write data to files via LAWN. If they fall into different sub-zones, LAWN will process their requests in parallel. If they enter the same ith sub-zone, only f_pt_i needs to be locked to avoid a contention in slot allocation. The lock period is much shorter than writing a data slice to a slot with `clflush` and `sfence` [14, 16]. In both cases LAWN well supports concurrency.

5 EVALUATION

5.1 Prototype and Evaluation Setup

Prototype We have built a prototype of LAWN within NOVA. In NOVA, writing file data calls `nova_cow_file_write` that triggers page-based COW. We modify this function to make updating small data through `alternate_write` of LAWN. For reading data, we

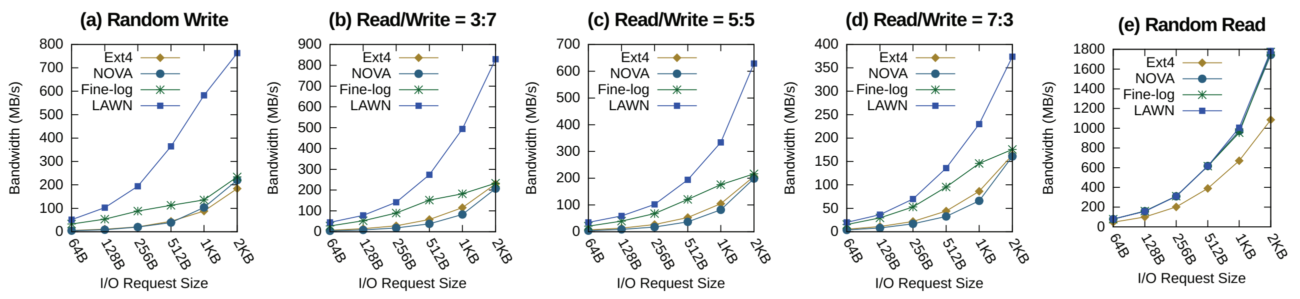


Figure 5: Bandwidth of Four Competitors with Fio

change the read routine `do_dax_mapping_read` and handle a read request by first checking whether the valid version is in a zone slot or in the file. As for the NVMM space for LAWN, we revise the page allocation and deallocation routines of NOVA. We use `kmem_cache` in DRAM to support hash tables and linked lists, which can be rebuilt in the function `nova_fill_super` on remounting NOVA.

Setup All experiments were performed on an Intel Xeon E5-2660 v3 machine with 64B per cache line and supporting `clflush` instruction. Out of 128GB DRAM space, we isolate 116GB in the GRUB boot loader via configuring the `memmap` option and emulate this space to be STT-RAM [10–12]. The remaining 12GB is used for main memory. All tests were run using Ubuntu 16.04.3 with kernel version 4.4.79 and GCC 5.4.0. We ran with 8 CPU cores for eight sub-zones by setting `maxcpus` in the GRUB boot loader. The default slot size of LAWN zone was configured as one cache line size (64B).

We compare NOVA with LAWN (referred to as `LAWN`) to the original NOVA, NOVA with fine-grained logging, and Ext4 (`data=journal` mode) on a memory-based block device. They will be referenced as `NOVA`, `Fine-log`, and `Ext4`, respectively. All four competitors guarantee data consistency. As to benchmarks, we have used Fio [21], Flexible Filesystem Benchmark (FFSB) [22], and Filebench [23]. The key metric to measure performance is the bandwidth (MB/s).

5.2 Fio

We evaluate four approaches for five different settings with Fio (cf. Figure 5), specifically with random reads and writes as well as with three different read/write ratios (i.e. 3/7, 5/5 and 7/3). We used Fio to generate a 60GB file (five times bigger than the size of main memory) and set six different sizes for write and/or read requests. We performed file operations for 30 minutes under each setting. The bandwidths of four competitors are shown in Figure 5. From the left four diagrams, we observe that `LAWN` significantly outperforms other three competitors with random write and mixed workloads. For example, with random writes at a request size of 128B, the bandwidth of `LAWN` is 9.5×, 12.0×, and 1.9× that of `Ext4`, `NOVA`, and `Fine-log`, respectively. Hence, LAWN boosts the performance of NOVA up to a factor of 12. With larger requests, such as with 1KB request, the bandwidth of `LAWN` is 6.6×, 5.6×, and 4.3× that of other three competitors, respectively. The higher performance of `LAWN` is accredited to the reduced write amplification. `LAWN` writes data only once through alternate writing. In contrast, for `Ext4` and `NOVA`, they write almost twice the size of an entire page (block) where the updated data resides. `Fine-log` also writes the update data twice. Figure 6 captures the amount of data written by the four competitors with random write at 1KB request size. `LAWN` writes

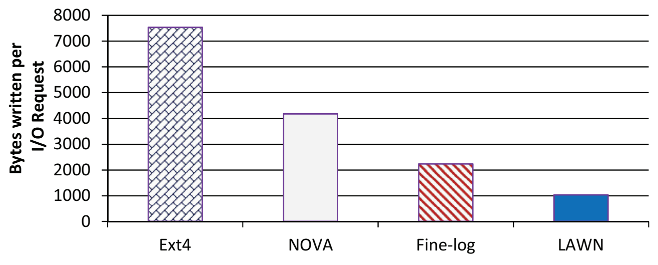


Figure 6: Bytes Written per I/O Request with Fio

86.3%, 75.3%, and 53.8% less data per write request than `Ext4`, `NOVA`, and `Fine-log`, respectively. In summary, our results quantitatively demonstrate the reduction of write amplification by LAWN.

We note that the performance gap between `LAWN` and `Fine-log` widens with an increase in the size of read/write request. Concretely, with larger size of requests, `Fine-log` spends more time in writing the updated data twice in the critical path, while `LAWN` just writes the requested data once and swiftly proceeds to the next I/O request.

With random reads, as captured in Figure 5(e), `LAWN` achieves identical performance compared to `NOVA` and `Fine-log`. This confirms that LAWN hardly impacts the performance of file reads.

5.3 FFSB

FFSB is a macro-benchmark that synthesizes real-world workloads. It supports conducting a configurable mix of different file operations over a number of files and with different file sizes. We ran the default workload provided by FFSB except 1) changing the sizes of write and read requests to be 1KB, 2) setting an initial dataset to be 60GB, and 3) varying the number of threads. All tests of FFSB were run for 30 minutes. Figure 7 captures the comparison of bandwidths obtained from the four competitors. We observe that the bandwidth of `LAWN` is 2.1×, 3.8×, and 3.2× that of `Ext4`, `NOVA`, and `Fine-log`,

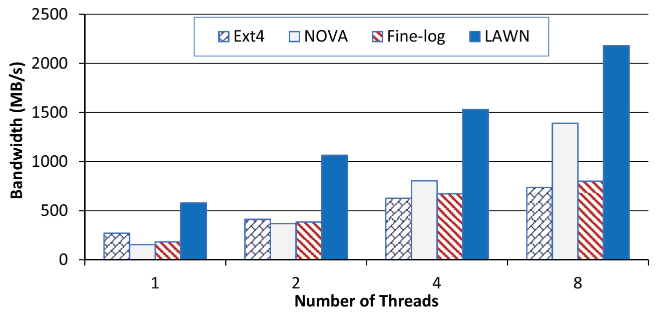


Figure 7: Bandwidths of Four Competitors with FFSB

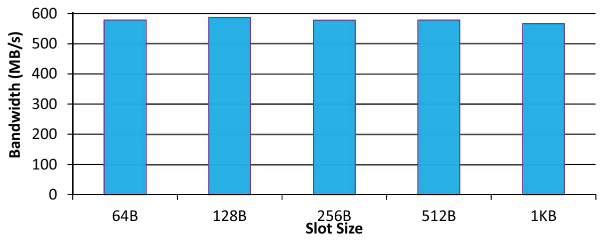


Figure 8: The impact of Slot Size on LAWN with FFSB

respectively. In a workload of macro-benchmark, different file operations, such as sequential or random write, fsync, and delete, are continuously issued. The results with FFSB reflects that, via reducing write amplification, LAWN efficiently handles such a workload and significantly outperforms the other three competitors.

We varied the number of threads to test the capability of LAWN in processing concurrent write and read requests. In Figure 7, the bandwidth of LAWN gradually increases with more threads. In particular, the bandwidth of LAWN improves almost by a factor of two (1.9×) when the number of threads is increased to two. Even from four to eight threads, the bandwidth of LAWN increases by 1.5×. These results confirm that with sub-zones as well as quick acquisition and release of pointers, LAWN effectively supports concurrency.

Using FFSB, we also performed a test with different slot sizes. We set five slot sizes, and the bandwidths of LAWN running with one thread are shown in Figure 8. The performance of LAWN does not deviate significantly with respect to slot size. Yet in-DRAM structures may take up less space given a greater slot size.

5.4 Filebench

Filebench is a widely-used macro-benchmark [10–12]. It provides numerous real-world workloads and we chose four representative ones (cf. Figure 9). When configuring these workloads, we set the size of dataset to be about 60GB, the mean size of I/O requests to be 1KB and the running time to be 30 minutes. Bandwidths of four competitors are presented in Figure 9. From Figure 9, we observe that with the write-intensive fileserver, the bandwidth of LAWN is 3.2×, 1.9×, and 2.0× that of `Ext4`, `NOVA`, and `Fine-log`, respectively. Due to the substantial reduction in write amplification, LAWN dramatically outperforms the other three competitive approaches. With metadata-intensive varmail and read-intensive webproxy and webserver, LAWN achieves comparable performance compared to `NOVA` and `Fine-log`. Although LAWN might scatter data into zone slots, assembling these scattered data in a fast memory device hardly takes more time than sequential read used by `NOVA` and `Fine-log`.

6 CONCLUSION

We propose LAWN to reduce the write amplification in NVMM file systems with DAX. LAWN leverages alternate writing that entails writing updated data only once. It also respects data consistency via imposing a strict execution order. Our evaluation reveals that LAWN substantially boosts the performance of NOVA by up to 12×.

ACKNOWLEDGEMENT

This work is partially supported by Singapore University of Technology and Design under the research grant SRIS17123.

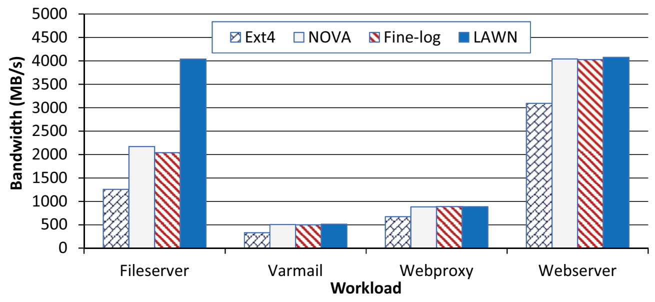


Figure 9: Bandwidths of Four Competitors with Filebench

REFERENCES

[1] P. Zhou et al. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.
[2] X. Zhang and G. Sun. Toss-up wear leveling: Protecting phase-change memories from inconsistent write patterns. In *Proceedings of the 54th Design Automation Conference*, DAC '17, pages 3:1–3:6, New York, NY, USA, 2017. ACM.
[3] J. Ahn et al. DASCA: Dead write prediction assisted STT-RAM cache architecture. In *Proceedings of 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36, 2014.
[4] X. Chen et al. AOS: Adaptive overwrite scheme for energy-efficient MLC STT-RAM cache. In *Proceedings of the 53rd Design Automation Conference*, DAC '16, pages 170:1–170:6, New York, NY, USA, 2016. ACM.
[5] H. Luo et al. Two-step state transition minimization for lifetime and performance improvement on MLC STT-RAM. In *Proceedings of the 53rd Design Automation Conference*, DAC '16, pages 171:1–171:6, New York, NY, USA, 2016. ACM.
[6] S. Yin et al. Disturbance aware memory partitioning for parallel data access in STT-RAM. In *Proceedings of the 54th Design Automation Conference 2017*, DAC '17, pages 84:1–84:6, New York, NY, USA, 2017. ACM.
[7] Micron and Intel. 3D XPoint technology. http://www.micron.com/about/innovations/3d-xpoint-technology.
[8] C. Xu et al. Understanding the trade-offs in multi-level cell ReRAM memory design. In *Proceedings of the 50th Design Automation Conference*, DAC '13, pages 108:1–108:6, New York, NY, USA, 2013. ACM.
[9] J. Condit et al. Better I/O through byte-addressable, persistent memory. In *SOSP '09*, pages 133–146, New York, NY, USA, 2009. ACM.
[10] S. R. Dulloor et al. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.
[11] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.
[12] J. Ou et al. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 12:1–12:16, New York, NY, USA, 2016. ACM.
[13] J. Xu et al. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.
[14] J. Ren et al. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685, 2015.
[15] Q. Hu et al. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Conference on Annual Technical Conference*, USENIX ATC '17, pages 703–717, Santa Clara, CA, 2017. USENIX Association.
[16] S. Pelley et al. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.
[17] T.-Y. Chen et al. Enabling write-reduction strategy for journaling file systems over byte-addressable NVRAM. In *Proceedings of the 54th Design Automation Conference 2017*, DAC '17, pages 44:1–44:6, New York, NY, USA, 2017. ACM.
[18] A. Memaripour et al. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.
[19] Q. Wei et al. Transactional NVM cache with high performance and crash consistency. In *SC '17*, pages 56:1–56:12, New York, NY, USA, 2017. ACM.
[20] C. Wang et al. Persisting RB-tree into NVM in a consistency perspective. *ACM Trans. Storage*, 14(1):6:1–6:27, February 2018.
[21] Flexible IO (Fio) Tester. Fio. https://github.com/axboe/fio, 2017.
[22] J. Santos and S. Rao. FFSB, 2013. https://sourceforge.net/projects/ffsb/.
[23] FileBench. https://github.com/filebench/filebench, 2017.