A High Performance File System for Non-Volatile Main Memory

Fumiya Shigemitsu[1,2] Mitsugu Suzuki[1,3]

Abstract: Emerging Non-Volatile Main Memories (NVMMs) are expected to be next-generation storage. These memories promise to enable persistent memory, which can store data persistently at the main memory level with low latency. Therefore, the traditional primary storage hierarchy is extended to the non-volatile part by them. Integrating NVMM into computer systems includes some interesting challenges though they are expected to realize a fast and reliable computer system when using them. We explore NVMMs feature and how to handle them efficiently as main storage through developing a new file system in the Linux kernel which exploits memory hierarchy including NVMMs.

Keywords: NVMM, NVDIMM, File System

1. Introduction

Non-Volatile Main Memories (NVMMs) (e.g., phase change, spin-torque transfer, resistive memories or Intel and Micron ' s 3DXPoint) are expected to be next-generation storage. These memories promise to enable persistent memory, which can store data persistently at the main memory level with low latency. Therefore, the traditional primary storage hierarchy is extended to the non-volatile part by them.

Integrating NVMM into computer systems includes some interesting challenges though they are expected to realize a fast and reliable computer system when using them. Firstly, traditional software design is not adequate for NVMMs. They are so fast compared to old one, and conventional software is designed on the assumption that the storage is too slow compared to the above memory hierarchy component. Therefore they are redundant when using NVMMs as main storage. We need to design software again for using NVMMs more efficiently[1]. Secondly, CPU cache memories still remain as the volatile while the main memory becomes non-volatile. This requires some considerations when using NVMMs. To make sure data persistent, a computer system should flush data to storage using NVMMs. CPU cache ability is not fully exploited by the system which flushes data in cache eagerly because the cache flush instruction causes significant performance overhead compared to writing data into cache due to waiting for a processing completion[2].

We explore NVMMs feature and how to handle them efficiently as main storage through developing a new file system in the Linux kernel which exploits memory hierarchy including NVMMs. Experimental results show that our developed file system provides up to 224% improvement in a micro-benchmark, and up to 471% improvement in a macro-benchmark compared to other state-of-the-art file systems.

[1] Shimane University, Matsue, Shimane 690-8504, Japan
[2] s179507@matsu.shimane-u.ac.jp
[3] suzuki@cis.shimane-u.ac.jp

2. Preliminaries

2.1 Non-volatile memory technologies

There are various types of Non-volatile memories and they have different strengths and weaknesses that make them useful in different parts of the memory hierarchy. PCM and ReRAM are denser than DRAM and may enable very large non-volatile main memories. The 3DXPoint memory technology from Intel and Micron is the technology to be expected to offer performance up to 1,000 times faster than NAND flash[3]. NVDIMM, a non-volatile dual in-line memory module, is a type of memory which uses non-volatile technologies on DIMM package. NVDIMM-N is DIMM with flash storage and DRAM on the same module, which provides as high performance as DRAM DIMM. It can offer non-volatile storage at high speed while its density is as low as DRAM. As a result, we expect to see more flexible memory hierarchies become common in a large system.

2.2 Challenges for NVMM software

NVMM technologies present several challenges to file system designers. Firstly, re-designing software storage stack is essential to exploit NVMMs performance. The latency of slow storage devices dominates access latency in conventional storage systems. In the new storage with NVMMs, however, it is much low latency than them, so software efficiency is critical for improving entire computer system performance. Secondly, considering processors cache hierarchies is important. Modern processors use caches within CPU to improve performance, so implementing software with it in mind is important to exploit performance. Caches hierarchy is a great effort to improve performance while there is a disadvantage to lost data when system crash. NVMMs ability to hold data permanently is great, but storing data does not make

the data persistent soon in conventional computer systems due to caches. Therefore, it is essential to consider it when designing software, especially in file systems.

2.3 File system technologies for NVMM

The page cache is used to buffer reads and writes to files, and provide the pages which are mapped into user-space by a call to mmap(2). When storage becomes memory-like, the page cache would be unnecessary copies of the original storage.

Considering the above case, The DAX appears[4]. The technologies of DAX provides direct access to files. It avoids pages cache and also maps an NVMMs region directly into user-space when calling mmap(2) system call. NVMM-aware file systems usually use this technology for performance.

2.4 NVMM-aware File System

We will review some proceeding NVMM-aware file systems. **EXT2-DAX** extents EXT2 with DAX, which is direct access mode to access NVMMs storage directly. **EXT4-DAX** and **XFS-DAX** also extends EXT4 and XFS with DAX. **PMFS** is an experimental DAX file system designed only for NVMMs which use undo journal for meta-data updates[1]. PMFS introduces some techniques to address NVMMs storage efficiently and safely. For example, PMFS maps entiere NVMMs region to a kernel virtual address space to access NVMMs region efficiently. PMFS uses the technique to protect the mapped region by using a write protect control feature along with mapping the region. They are incorporated succeeding NVMM-aware file systems. **NOVA** is a log-structured file system designed only for NVMMs[5][6]. NOVA uses logs to update each inode, journal to update multiple meta-data and copy-on-write to write data to NVMMs. NOVA aims to realize high-performance with providing strong consistency guarantees. **SoupFS** is an NVMM-aware filesystem with soft-updates which is re-designed for NVMMs. SoupFS provides correctness and consistency without synchronous cache flushes in the critical path[7]. It is deeply considered the timing of cache flushes and achieves high performance compared to NOVA while providing less consistency guarantee. **HiNFS** is an NVMM-aware file system considering file writes on NVMMs [8]. HiNFS provides better performance compared to PMFS and EXT4 to buffer the lazy-persistent file writes in DRAM. This idea comes from almost NVMMs write latency is slower than DRAM.

2.5 Insights for NVMM-aware FS from Experiments

Priya Sehgal et.al show the feature of a file system to design a better NVMM-aware file system[9]. They present four keys from their performance study which evaluate some Linux file systems. The following is the points.

(1) Parallel allocation strategy
(2) An in-place update
(3) An Execute-in-place I/O
(4) Fixed sized data blocks

Firstly, enabling parallel allocation is more preferable for NVMM-aware file systems. PMFS cannot scale beyond a few numbers of files and allocation requests because it uses a single list to allocate blocks. An NVMM-aware file system should have some allocation groups to scale file system performance.

Secondly, an in-place update is suitable for NVMM-aware file systems as it helps utilize the CPU and memory resource efficiently. The technologies of updating file system such as copy-on-write, a log-structured file system, use much CPU resource and memory resource and may cause much cache contention.

Thirdly, An execute-in-place, which equals DAX, leverages NVMMs to its fullest extent because it helps bypass extra layers in the software stack.

Finally, using fixed-size blocks is better than an extent based because it uses simple index based inodes which are cache friendly. Simple software design helps to improve the performance of software for NVMMs, including a file system.

3. AEON Design and Implementation

We describe the design and implementation of our file systems ”AEON”. AEON is designed to maximize NVMM-aware file system performance with light-weight consistency guarantees, especially in **NVDIMM**. We have implemented AEON in Linux Kernel 4.19.4. AEON has some features to exploit a system with NVMMs. They are optimal NVMM space management including the NVDIMM NUMA architecture support and meta-data management with scalability in mind, careful locking consideration, and in-place updates which are supported by consistency without ordering.

3.1 NVMM space management

When managing the memory subsystem, it is important to consider whether the system is UMA or NUMA. AEON can manage both UMA and NUMA architecture properly by adopting a little different management policy.

3.1.1 In case of UMA

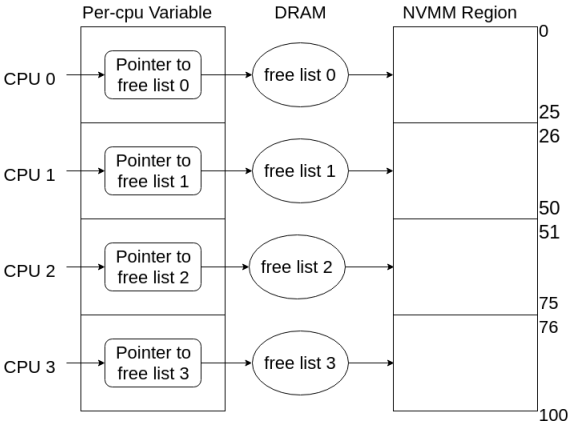

Fig. 1 AEON’s block allocation layout in UMA version

Figure 1 shows the layout of free lists for block allocation in AEON. The block allocation of AEON is designed with scalability in mind. AEON has free block lists at each CPU cores to avoid locking and scalability bottlenecks. AEON also stores each pointer to each free list in unique region per CPU[10] to access free lists quickly. When requesting a new block allocation, AEON uses the free list that belongs the CPU running at the time.

3.1.2 In case of NUMA

Figure 14 shows the layout of free lists for block allocation in NUMA version of AEON. When system architecture is NUMA, the access speed to memory depends on the distance from the CPU, so it is better to use local memory as much as possible. For example, we want to make CPU 0 use NVDIMM0 region in figure 14. AEON uses a local NVDIMM node from running CPU when requesting a new block allocation.

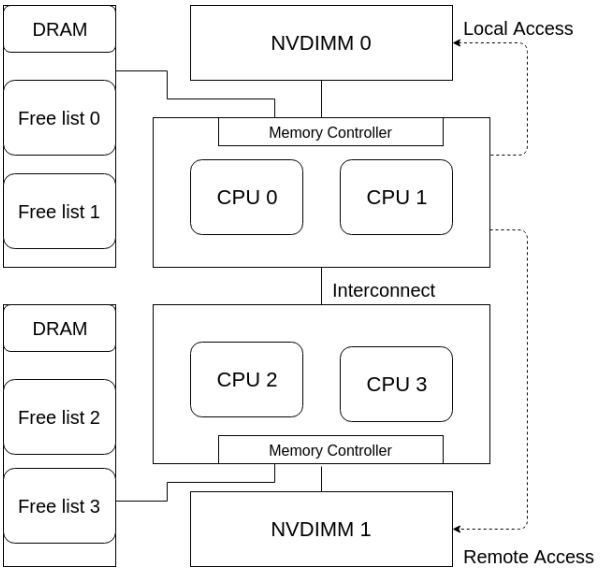

Fig. 2 AEON's block allocation layout in NUMA version

Traditional file systems with NVMM support including EXT2, EXT4, and XFS can handle some NVMM devices to combine in one device by the device mapper technology. The file system dedicated to NVMM including PMFS and NOVA etc. maps entire NVMM region to a kernel virtual address space for efficient file system operations. It is a better way to handle NVMM devices efficiently though they are unable to identify an entire linear-mapped device. They can only use the first device. Although AEON also maps the entire NVDIMM device region to a kernel virtual address space, AEON further identifies each NVDIMM node separately. The following is the example of mapping two different NUMA nodes to the kernel virtual address space.

Example

NVMM0 virtual address 0xffff966b90200000 size 16GB
NVMM1 virtual address 0xffff967fd0200000 size 16GB

AEON swithches the head address per node. To manage the allocated blocks, AEON holds the allocated block number with NUMA id.

3.2 Metadata management

The figure 3 shows how to manage inode allocation. The inode cache consists of two parts which are allocated blocks for new inodes and a linked list holding released inodes.

AEON gets some blocks for inodes allocation (one block in default). The inode size is 256 bytes. The number of inodes is determined "the number of CPUs * range + CPU id". The number of range increases every allocation. After releasing the allocated inode, connecting it to a linked list, which prepared in per CPU cores, and reuse inodes from a linked list preferentially.

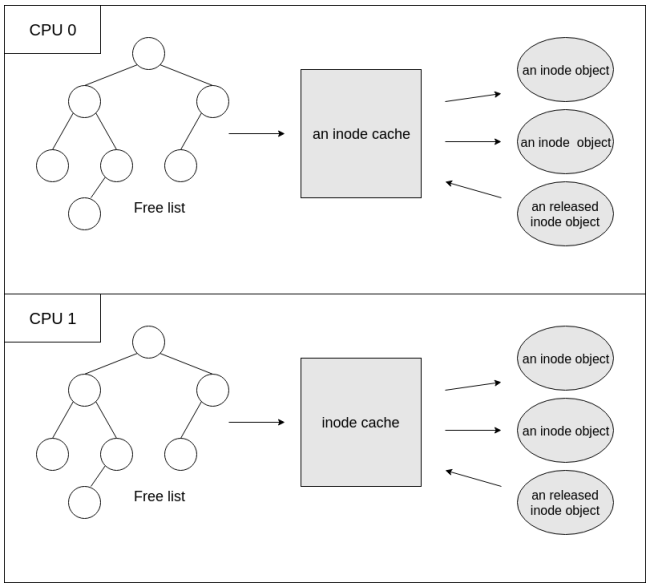

Fig. 3 Inode allocation

The how to allocate directory entry is also simple. The directory entry structure size is fixed, it is 256 bytes. One block can hold sixteen entries. When releasing directory entry spaces due to deleting files, they are connected to a linked list. The released spaces are reused preferentially. Releasing entire directory entry due to deleting the directory, allocated blocks insert the free lists.

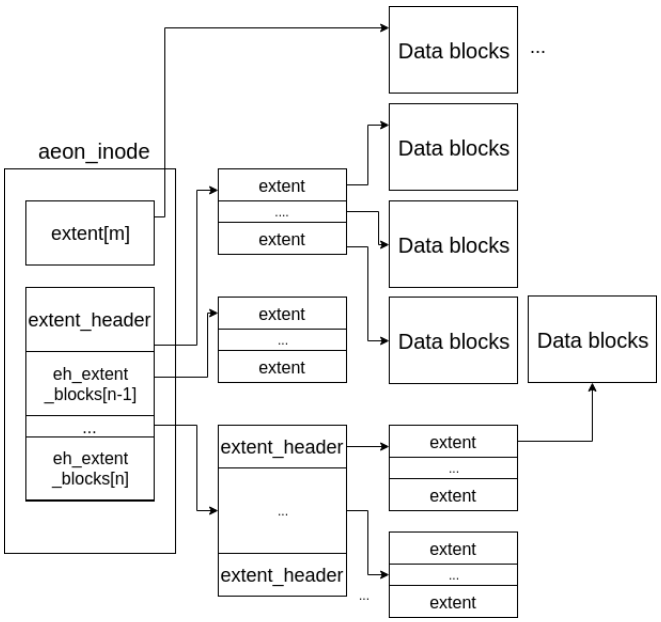

Fig. 4 The layout of extents

AEON uses extents to express file data. The figure 4 shows the layout of extent allocation in AEON. A few extents are embedded in an inode and expand a red-black tree when increasing them. The region for extents is managed by an extent header, which is

also embedded in an inode. When increasing extents and one extent header cannot manage all extent, a new block would allocate for extent headers.

3.3 Mutual exclusion

It is essential to use mutual exclusion for some critical file system operations. In the Linux kernel, many forms of mutual exclusion can be used. and they each have advantages and disadvantage[11]. It is important to choose which type of locks for improving the file system performance, especially in increasing scalability in mind[12]. We choose the type of mutual exclusion carefully in file system operations in AEON.

- Atomic operations
- Spinlocks
- Reader-writer spinlocks
- Mutex locks

We use atomic operations to modify pointers in a inode or a dentry (directory entry structure) on NVMM. Atomic operations can perform atomic read-modify-write operations on a memory location. We can modify a variable on NVMM safely by these operations in the multi-threaded scenario.

Atomic operations are useful when operating only on CPU word and double word size data, but often file systems want to bigger shared data with a bunch of instructions. Spinlocks, one of the simplest and lightweight mutual exclusion mechanisms are suitable for these cases. We use spinlocks in two cases. First is to modify free lists. Although AEON has as many free lists as CPU cores and chooses free lists by running CPU id, mutual exclusion should be used because users may run multi-thread applications or AEON chooses a free list other than running CPU id due to free blocks depletion. While there are such problems, we predict that lock contentions do not occur frequently thanks to multiple free lists. Therefore we use the most lightweight mutual exclusions, spinlocks, in this case. Second is to connect released inodes or dentries to the related linked list as a reusable cache. There is a possibility of corrupting a linked list due to competing the same resource. However, it is simple and lightweight operations so we use spinlocks.

Reader-writer spinlocks are more fine-grained mutual exclusion than the simple spinlocks. They enforce exclusion between reader and writer paths; this allows concurrent readers to share lock and a reader task will need to wait for the lock while a writer owns the lock. It is suitable to manipulate extents, which are the structure for file data. AEON can share the file data resource when reading, and update the file data safely thanks to them.

The mutual exclusion of spinlocks type is lightweight, however, it enforces busy-waiting to the CPU. Therefore, they would have a bad impact on the performance of the system in the case that a lock is held for longer, non-deterministic durations. Sleeping locks such as mutex locks are precisely designed to be engaged in such situations. These types of mutual exclusion are put into sleep and moved out into a wait for a queue and forcing a context switch allowing the CPU to run other productive tasks when a caller task attempts to acquire a mutex that is unavailable. When the mutex comes available, the task in the wait queue is woken up and moved by the unlock path of the mutex, which can then attempt to lock the mutex. We use mutex locks in two cases. First is to do truncate operations. File truncate operations hold non-deterministic durations because the length of operation time depends on the truncate size. Second is to rebuild the structure of AEON when the mount process. If the last system shutdown was invalid, AEON checks the structure on NVMM. It holds non-deterministic durations, so we mutex locks.

3.4 In-place updates

AEON updates all metadata in-place. Also, AEON reuse NVMM regions that is released recently with temporal locality in mind. All of them are designed to implement a fast and scalable file system.

3.4.1 Consistency Without Ordering for NVMM

In-place updates are a better way to update metadata from the aspect of performance. On the other hand, it cannot protect file system consistency as it is. One way of protecting file system consistency with in-place updates is journaling. Journaling is mature technology and the framework of implementation in the Linux kernel source code. The problem of journaling is lack of scarablity[12], so there is a less meaning to adopt in-place updates for metadata if using journaling for protection of file system consistency. Therefore, we should adopt the more scalable technology than journaling not to kill in-place updates advantages.

We adopt the idea of Consistency Without Ordering (CWO)[13] in AEON. This is a lightweight technique based on backpointer-based consistency to provide crash consistency without ordered writes. The original CWO is designed for block-based storages, but we believe that it is suitable for NVMM based storages due to the feature of byte addressability. We design CWO for NVMMs (CWON). The core of CWON

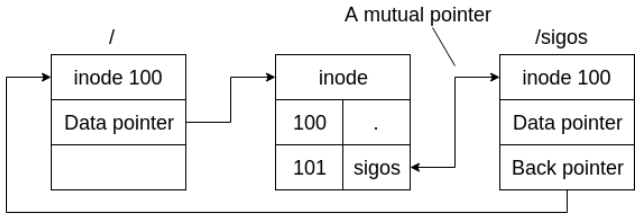

Fig. 5 The layout of meta-data structure

is circulated and a mutual pointer made of back pointers. The figure shows that the layout of metadata structure in AEON. AEON uses them to recover file system consistency from some inconsistent states which happen when system failures. We will see cases of system failures and how to detect and recover from them below.

Create: changes two data structures of AEON on NVMM. The one is a dentry, which expresses a directory entry and The other is an inode, which corresponds a file uniquely.

```
1 add_dentry() --- (1)
2 init_inode() --- (2)
```

AEON adds a directory entry (dentry) to a parent directory in the operation (1), and initializes a new inode which corresponds to the entry of (1) in the operation (2). The cases to be considered

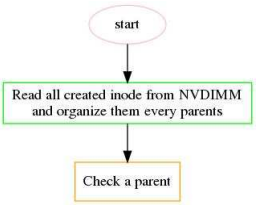

Fig. 6 Preparing for recovering file system's consistency

are four; both of them are persisted, the second is the dentry is persisted but the inode is not persisted, the third is the dentry is not persisted but the inode is persisted, the last is both of them are not persisted. The first is no problem, it is all OK. The second and third can be detected and recovered by checking whether back pointers are valid or not. If there are valid, manipulating the incompleted operation again from the information in the persisted metadata. The last is that AEON simply does not notice the create operation happened.

Delete also changes a dentry and a corresponded inode. The following is the flow of detecting a file.

```
1 remove_dentry() --- (1)
2 free_inode() --- (2)
```

The operations which should be done after a system failure are almost the same as the create operations. Checking and recovering them if needed.

Rename is the most complex operation in all file system's operations because it contains remove and create operations. The following is the flow of renaming a file.

```
1 add_old_dentry_to_newdir() --- (1)
2 fix_old_inode_info --- (2)
3 remove_old_dentry() --- (3)
```

The biggest issue is that a new directory entry is not persisted and an old directory entry is fully removed. It means that the data disappear. There is no way to recover the case. To prevent the case, AEON does cache flushes related (1) and (2) operations before doing the (3) operation. AEON does not lose the file by its cache flushes and can recover the consistent state. If a system failure happens, the considered case is that both new and old directory entry exists. It seems to be the worse case especially in the aspect of security, but AEON can remove the file that should be erased by checking the pointer to the corresponded inode and the back pointer from it because the back pointer from the inode is one.

3.5 Recovering procedure

We will describe the recovering procedure from a system failure when the mount processing. The recovering procedure consists of four parts; the preprocessing per parent directory, checking and recovering each file, inserting each valid file, checking and recovering the parent directory.

Figure 6 shows that preparing for the recovering. Before entering the recovering process, AEON reads all created inodes and arrange them per parent directories. All inodes which its valid flag does not stand are connected to inode caches in this stage.

Figure 7 shows the checking and recovering files per parent directory. In "Check a child", AEON checks the detnry pulled from

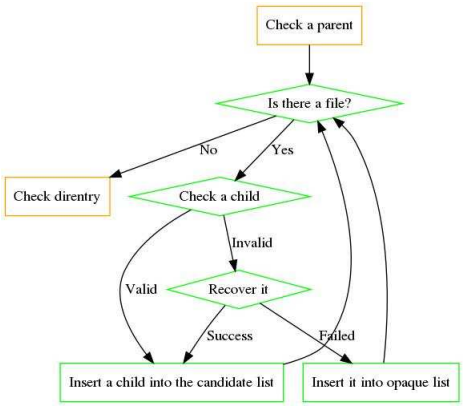

Fig. 7 Recovering each file's metadata

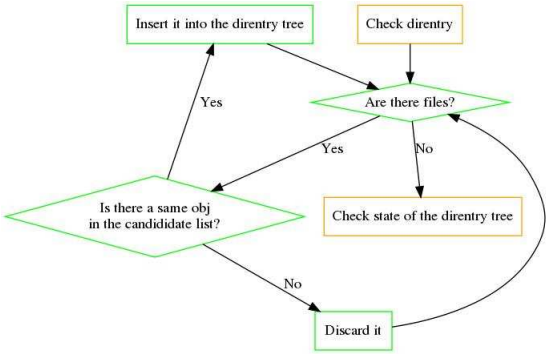

Fig. 8 Inserting valid files to a directory entry tree

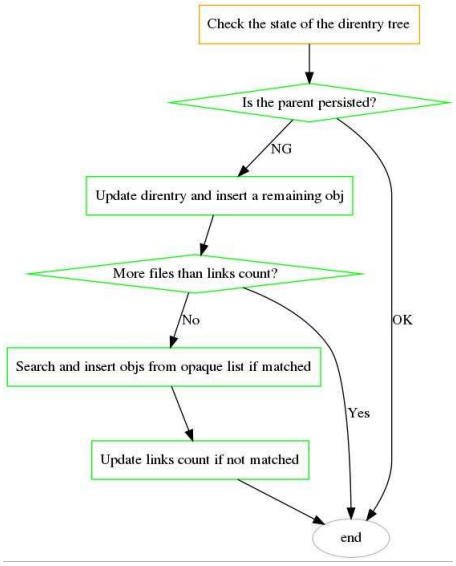

Fig. 9 Additional checking

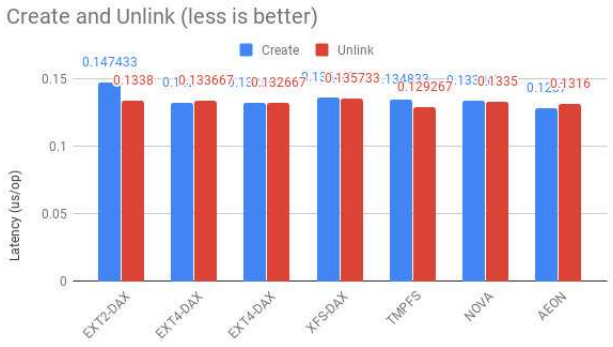

Fig. 10 Latency of create and unlink

the inode, and recover it if needed as mentioned the previous section.

Figure 8 shows the process of checking directory entries from the head of directory entry and inserting valid child files. If there is a valid directory entry but AEON cannot find the corresponded inode, it is valid flags sets invalid which means it is discarded by AEON.

Figure 9 shows the last check of the recovering procedure. If there is a possibility that the parent directory has more child files, AEON finds whether the child file exists in the opaque list created when checking inodes in the second stage.

4. Evaluation

4.1 Experimental setup

To evaluate the performance of AEON, we run benchmarks with an HPE ProLiant DL360 Gen10 Server. It equipped Intel Xeon Gold 5115, 128GB DRAM, and two NVDIMM. Each NVDIMM is a different NUMA node with 16GB.

We compare AEON against four Linux file systems: EXT2 with DAX, EXT4 with DAX, EXT4 with DAX and writeback mode, XFS with DAX, TMPFS, NOVA. We use NOVA with in-place mode because it is the standard option at the experiment day from upstream NOVA, and it seems to be the fastest mode in NOVA[14].

We use linear-mapped pmem device combined with two pmem devices for evaluation of EXT2, EXT4, XFS. NOVA cannot identify the entire region of the combined device, so we use one pmem device. We use both of pmem devices for AEON, which identifies each pmem device as different NUMA nodes.

In evaluation, we use simple Linux commands for simple evaluations and use Filebench [15] for further evaluations.

4.2 Micro benchmark

Table 1 Micro-benchmark characteristics (dir operations)

Workload	files	dirs	threads
touch command	1×10^4	1	1
rm command	1×10^4	1	1
createfiles (filebench)	5×10^5	100	1 - 30
filemicro_ delete (filebench)	5×10^5	100	1 - 30

We use four micro-benchmarks to evaluate the throuphput and latency of meta-data operations as show in Table 1.

Figure 10 shows the latency of creating and unlink tested using the shell script. AEON provides the lowest latency for the create operations, outperforms EXT2 which provides the highest latency by 115%. For delete operations TMPFS provides the lowest latency, outperforms XFS which provides the highest latency by 1.05 %

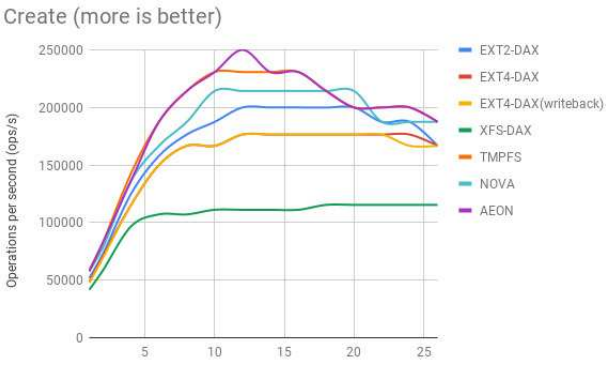

Fig. 11 Throughput of create with the different number of threads

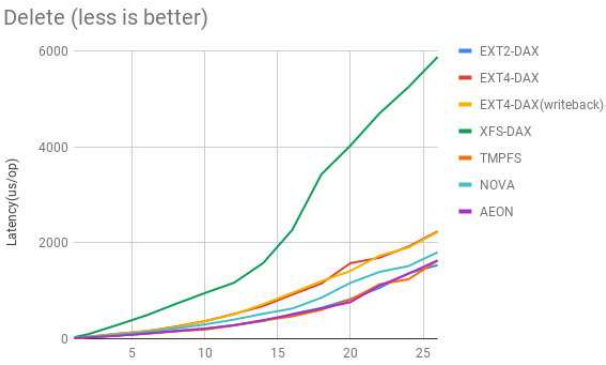

Fig. 12 Latency of delete with the different number of threads

Figure 11 and 12 shows the throuput and latency of create and unlink tested using two filebench workloads through chaging the number of threads. AEON provides the highest throughput and scarability for the create operations, outperforms XFS by up to 520%. AEON provides one of the lowest latency for the delete operations.

Single-thread workload tests do not show the remarkable difference but In multi-thread workloads, AEON has good scalability compared to the other file systems including TMPFS. AEON optimizes for the concurrent workloads and can avoid heavy additional writes cache flush instructions through operations while NOVA needs log and cache flush instructions. In addition, It is interesting that XFS is not scalable in these tests. Although XFS is designed with scalability in mind, it cannot make a contribution on NVDIMM.

Table 2 Micro-benchmark characteristics (file operations)

Workload	files	appends	sync	threads
dd command	1	1000	-	1
filemicro_ writefsync (filebench)	1	1024k	per 12500	10

We use two micro-benchmarks to ealuate the throuput and latency of writing as shown in Table 2 Figure 13 shows that the throughput of write tested using the dd command. NOVA provides the highest throughput, outperforms even TMPFS and XFS which is the lowest throughput by 142%. NOVA uses the original write method, while the other file systems without TMPFS use the method provided by the Linux kernel. Generally, there is a possibility that the optimized original method can outperform a general method, so we believe that there is still room for improvement of AEON's write method.

On the other hand, NOVA does not make a significant contribution in a multi-thread write workload. TMPFS provides the highest throughput, outperforms XFD which is the lowest throughput. EXT2 provides the second highest throughput. EXT2 has good scalability in micro benchmark from the results. It seems not to match the result between a single-thread benchmark and a multi-thread benchmark necessarily.

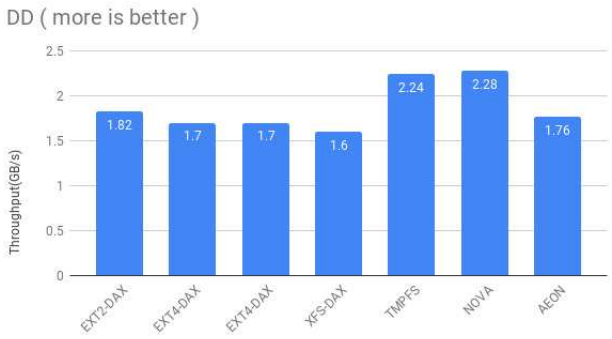

Fig. 13 Throuput of write with dd command

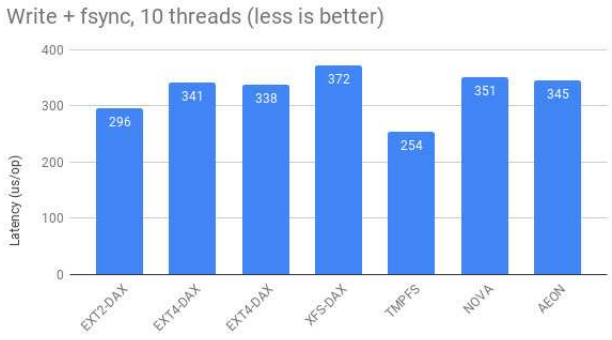

Fig. 14 Latency of multi-thread write workload

4.3 Macro benchmark

We evaluate the performance of AEON for real-world applications by running a set o macro-benchmark workloads by Filebench. Table 3 shows the characteristics of Filebench workloads.

Figure 15 shows the throughput of webserver workload. The webserver workload consists of opens, reads, and appends. AEON provides the second highest throughput next to TMPFS, outperforms EXT4-DAX(writeback) which is the lowest throughput by 1.09%

Table 3 Macro-benchmark characteristics (file operations)

Workload	files	dirs	threads
Webserver	20,000	20	100
Fileserver	60,000	100	50
Varmail	20,000	100,000	12
Mongo	200,000	40	2

Figure 16 shows the throughput of fileserver workload. The fileserver workload consists of well-balanced getting file statistics, searches, deletes, creates, writes, and appends. AEON provides the highest throughput, outperforms EXT4-DAX(ordered) which is the lowest throughput by 1.12%. AEON perform well in the comprehensive workload compared to the other file systems. We believe that it depends on the design with scalability and NUMA architecture in mind.

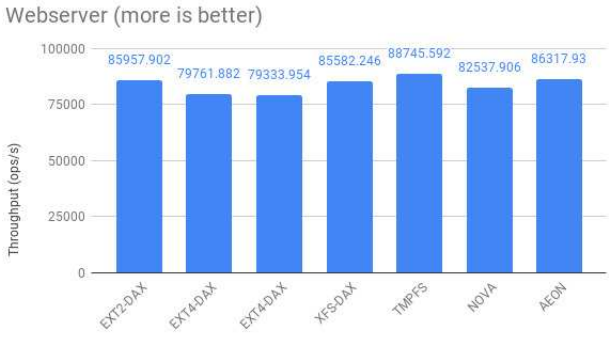

Fig. 15 The throughput of webserver workload with different file systems

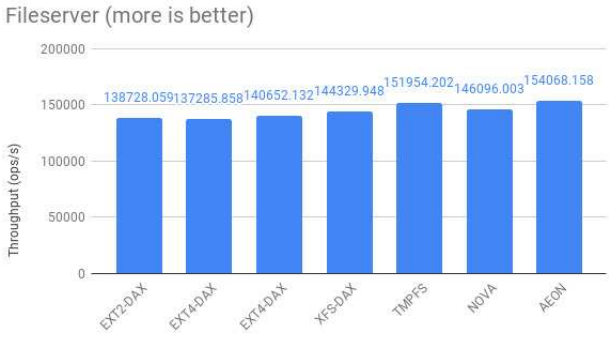

Fig. 16 The throughput of fileserver workload with different file systems

Figure 17 shows the throughput of varmail workload. The varmail workload consists of finds, deletes, writes, and syncs. AEON is the second highest throughput next to TMPFS, though almost the same. AEON outperforms EXT2-DAX by 472%. EXT2-DAX shows the good performance before this benchmark but its performance is degraded in this workload. We think that the cause is that the searching directory entries of EXT2 is a linear search. It seems to increase overhead as the directory entries increase.

Figure 18 shows the throughput of mongo workload. The mongo workload consists of finds, appends, and deletes like a

database workload. AEON outperforms the highest throughput, outperforms XFS which is the lowest throughput by 181%.

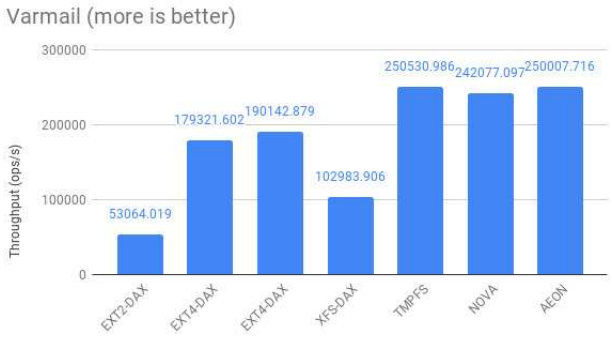

Fig. 17 The throughput of varmail workload with different file systems

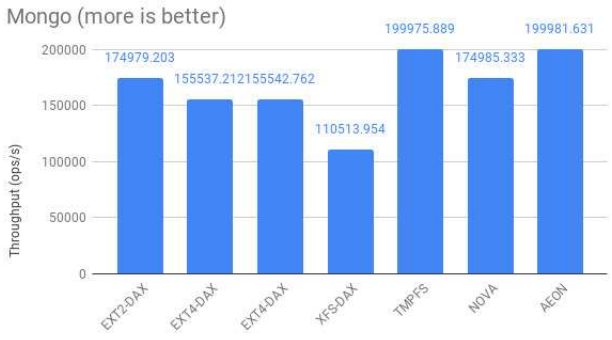

Fig. 18 The throughput of mongo workload with different file systems

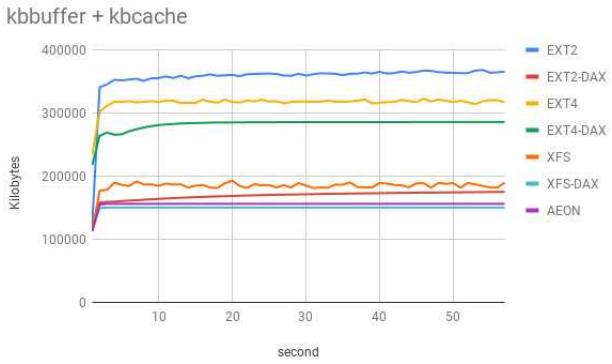

Fig. 19 The difference of each file system's memory usage

4.4 Resource Comsumption

We also evaluate the memory and disk usage with different file systems.

4.4.1 The Memory Usage

We compared to the memory usage through the varmail workload (Figure 19) among EXT2, EXT2-DAX, EXT4, EXT4-DAX, XFS, XFS-DAX, and AEON. EXT2 has the largest memory usage and XFS-DAX has the lowest memory usage. Although it is natural, the memory usage of the file system with the DAX option becomes less than when the option is not attached.

4.4.2 The Disk Usage

We compared to the disk usage by creating 10,000 of 16KB files (Figure 20) among EXT-DAX, EXT4-DAX, XFS-DAX, NOVA, and AEON. NOVA uses a very large space. It may use it for each inode log, which is 4KB.

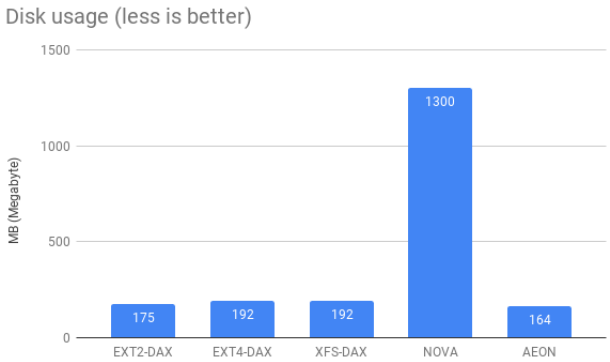

Fig. 20 The difference of each file system's disk usage

4.4.3 Evaluation for CWON

We evaluate the CWON by destroying meta-data via ioctl(2). We made 32 error cases referencing the previous CWO research. Test cases is made to reproduce the inconsistent state after a system failure (e.g. a system failure while creating a file or deleteting a file or renaming a file, and so on). Test cases contain destroying multiple meta-data. AEON can recover the consistent state from the inconsistent state which each test case make. AEON can provide atomicity operations at least related the test cases which we made.

5. Concluding Remarks

We have implemented and described AEON, a file system with scarability in mind designed for NVMM. AEON is made fast and scalable by doing in-place update with CWON. CWON makes AEON avoid most cache flushes, it affects better performance improvement. Also, AEON can handle different NVDIMM NUMA nodes. In NUMA architecture, especially in NVDIMM, it is pretty fast compared to traditional storage so it is important to consider NUMA architecture when designing new software. AEON handle it and achieves better performance.

There are some future works. One is to study AEON whether it is better performance in the other NVMMs (e.g. like HiNFS in a write operation). In addition, AEON is not always good performance in single-thread benchmarks. If it was improved, AEON could get better performance in the other file systems every cases. Also, it is a problem that CWON cannot be generally adopted to other file systems easily. We will work around above works to study better file systems from the aspect of performance and reliability.

Acknowledgments We thank Toshihiro Kamiya for his feedback. We are also thankful Hiroyuki Onizawa and Hewlett-Packart Japan, Ltd for hardware access. This research is supported by Mitou program, sponsored by IPA.

References

[1] D. S. Rao, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014* (D. C. A. Bulterman, H. Bos, A. I. T. Rowstron, and P. Druschel, eds.), pp. 15:1–15:15, ACM, 2014.

[2] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018* (M. Annavaram, T. M. Pinkston, and B. Falsafi, eds.), pp. 439–451, IEEE Computer Society, 2018.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[4] M. W. et.al, *direct access for files - Linux Kernel Documentation*. https://www.kernel.org/doc/Documentation/filesystems/dax.txt, January 31, 2019.

[5] 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)*, (Santa Clara, CA), pp. 323–338, USENIX Association, 2016.

[6] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. D. Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pp. 478–496, ACM, 2017.

[7] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 719–731, USENIX Association, 2017.

[8] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *EuroSys*, pp. 12:1–12:16, ACM, 2016.

[9] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on NVM," in *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pp. 1–14, IEEE Computer Society, 2015.

[10] J. Corbet, *Better per-CPU variables*. https://lwn.net/Articles/258238/, November 12, 2007.

[11] R. Bharadwaj, *Mastering Linux Kernel Development*. Packt Publishing, 2017.

[12] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *USENIX Annual Technical Conference*, pp. 71–85, USENIX Association, 2016.

[13] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *FAST*, p. 9, USENIX Association, 2012.

[14] "Nova: Non-volatile memory accelerated log-structured file system." https://github.com/NVSL/linux-nova.

[15] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *;login:*, vol. 41, no. 1, 2016.