Efficient Memory Mapped File I/O for In-Memory File Systems

Jungsik Choi
Sungkyunkwan University
chjs@skku.edu

Jiwon Kim
ATTO Research
jiwon.kim@atto-research.com

Hwansoo Han
Sungkyunkwan University
hhan@skku.edu

Abstract

Recently, with the emergence of low-latency NVM storage, software overhead has become a greater bottleneck than storage latency, and memory mapped file I/O has gained attention as a means to avoid software overhead. However, according to our analysis, memory mapped file I/O incurs a significant amount of additional overhead. To utilize memory mapped file I/O to its true potential, such overhead should be alleviated. We propose *map-ahead*, *mapping cache*, and *extended madvise* techniques to maximize the performance of memory mapped file I/O on low-latency NVM storage systems. This solution can avoid both page fault overhead and page table entry construction overhead. Our experimental results show throughput improvements of 38–70% in microbenchmarks and performance improvements of 6–18% in real applications compared to existing memory mapped I/O mechanisms.

1 Introduction

Although CPU performance has increased dramatically, the performance of storage systems has not improved as rapidly as their capacity. Even though CPUs can process data quickly, systems have bottlenecks due to storage devices latency, and this wasted time is the most significant component of overhead for data-intensive workloads. These days, the ever-increasing data-intensive nature of recent applications requires significant improvements in performance, and in-memory data processing has drawn attention because it can process workloads much faster by eliminating I/O overhead. In addition, techniques to ensure the durability of data in the main memory have been actively studied [19, 21]. Recently, an increase in the use of *non-volatile DIMMs* (NVDIMMs) has begun to provide high-performance memory storage by integrating DRAM with battery-backed NAND flash memory [20]. Emerging *non-volatile memories* (NVMs), including high-density STT-MRAM [9], have an almost equivalent ability to DRAM, and these are often considered to be suitable for the main memory in the future, even replacing DRAM [3]. With the use of modern memory technology, it is possible to develop high-performance storage devices, such as Intel and Micron's 3D XPoint [14]. The use of next-generation storage technologies will help remove almost all storage latency that has plagued conventional storage devices.

However, existing operating systems (OSs) have been designed to manage fast CPUs and very slow block devices. For this reason, their resource management mechanisms inefficiently manage low-latency NVM storage devices and fail to take advantage of the potential benefits of NVM devices [1]. In fact, software overhead has been identified as a new dominating overhead to be solved in the systems equipped with low-latency NVM storage devices [6, 31, 22]. Software overhead includes complicated I/O stacks, redundant memory copies, and frequent user/kernel mode switches. Such overhead is currently small enough to be ignored, but will be the largest overhead soon when storage latencies are significantly reduced by using the NVDIMMs and NVMs.

To fully exploit high-performance next-generation NVM storage, researchers have proposed various NVM-aware file systems [11, 13, 26, 28, 29, 30, 15] and system software [27, 10, 8]. In most of these studies, memory mapped file I/O has been commonly proposed to provide fast file accesses. Mapping a file onto user memory space with an `mmap` system call enables users to access the file directly in the same way as data on the memory. Therefore, it is possible to avoid the use of a complicated I/O stack for existing OSs, as well as user/kernel mode switches from `read`/`write` system calls. Consequently, this minimizes the occurrence of user/kernel mode switching. In addition, no data copies are needed between kernel space and user space, which consequently minimizes overhead. For these reasons, the `mmap` system call has a high likelihood to become a critical interface in the future [30, 25].

However, our analysis indicates that memory mapped file I/O can have a large software overhead, and this overhead comes from three sources. First, the overhead to map files to memory is much higher than expected. A detailed description is given in §2. Second, memory mapped file I/O requires a special `msync` to guarantee consistency in the files. However, the sophisticated techniques that have been studied so far are only available in certain file systems with a high overhead [30] or can only be used in secondary storage systems [23]. Third, memory mapped file I/O can not append new data at the end of the memory mapped files. To append data to a memory mapped file, you first need to resize the file with `fallocate` and then recreate the file mapping. This is an inefficient method that results in further overhead.

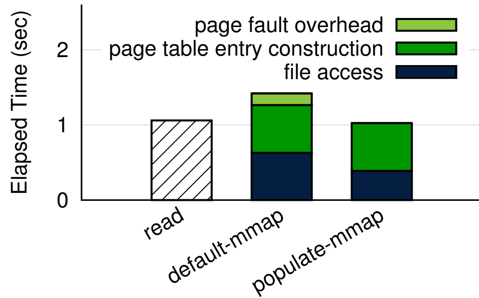


Figure 1: Sequential access to a 4GB file on Ext4-DAX

These problems must be addressed to use memory mapped file I/O on NVMs or NVDIMMs. In this paper, we address the first problem, reducing the page mapping overhead. The second and third problems will be addressed in future work. To reduce the memory mapping overhead, we present *map-ahead*, *mapping cache*, and *extended madvise* techniques. With these techniques, the performance of the memory mapped file I/O improves by up to 38%, 18%, and 70%, respectively.

2 Motivation

To analyze the overhead in memory mapped file I/O, we evaluate existing `mmap` mechanisms along with `read` system calls. Our evaluation is conducted on a Linux machine with 32GB DRAM and 32GB NVDIMM-N [20]. We installed an Ext4-DAX [28] file system through a PMEM driver on the NVDIMM-N. The Ext4-DAX simplifies the Ext4 file system to directly access memory storage, bypassing the page cache. The microbenchmark in our analysis sequentially reads a 4GB file in units of 4KB. *Read*, shown in Figure 1, is the case where the `read` system call is used to access the file. *Default-mmap* in Figure 1 is the case where `mmap` is called without any special flags and *populate-mmap* in Figure 1 is the case where `mmap` is called with the `MAP_POPULATE` flag. When the `MAP_POPULATE` flag is used, `mmap` constructs all page table entries for the entire file, whereas *default-mmap* relies on a page fault handler to construct a page table entry for the referred page (minor page fault). Thus, *populate-mmap* may take longer, but subsequent accesses to the memory mapped file can proceed without page faults. This is called a *prefault* mechanism.

Figure 1 shows the result of our microbenchmark test. It is interesting to note is that the memory mapped file I/O is not so fast as expected, when compared to the `read` system call. Reading a 4GB file sequentially takes 1.06 seconds, 1.42 seconds, and 1.02 seconds for *read*, *default-mmap*, and *populate-mmap*, respectively. As for the memory mapped file I/O, the time take for *file access* (copy from mapped area to user buffer) accounts for only about 40% of the total time. The rest of the time is spent to prepare the file accesses. For the *default-mmap*, each access to a page causes a page fault, and 11% of the total time is spent to call the page fault handler (*page fault overhead*). In addition, the time spent in the *file access* are different for the *default-mmap* and *populate-mmap*. A huge difference in TLB misses is the main reason. In sequential accesses, the TLB page walker implemented in the MMU sophisticatedly prefetches TLB entries from the page table. For the *populate-mmap*, the page table entries are already there to prefetch, but they are not ready for *default-mmap*. Since both *default-mmap* and *populate-mmap* require page mapping of a 4GB file, they take the same time for *page table entry construction*, which accounts for 45% and 62% of the whole execution time, respectively. This is absolutely necessary work to access the memory mapped files, but it is more expensive than the cost of the *file access*.

As shown in Figure 1, memory mapped file I/O still possesses a high overhead to construct page table entries. In traditional storage systems, this overhead was negligible, but not now. Relying on the page fault mechanism or populating entries for the whole file is not adequate, either. In this paper, we propose a couple of extensions to existing memory mapped file I/O to reduce the overhead involved in page table entry construction.

3 Related Works

There have been studies to reduce the page fault overhead. A premapping scheme [7] uses the hint files of the page fault history which is obtained from profiling runs. According to the experimental results, memory access patterns were quite random, and the result of the profile was required to predict the page faults.

PMFS [13] proposed file mapping with the use of hugepage or `MAP_POPULATE` flag. Compared to the use of the general page size, the use of hugepage requires a far smaller amount of page mapping, which greatly reduces the TLB misses and page faults. However, hugepage may lead to internal fragmentation and expensive copy-on-write (CoW). CoW is a popular technique to protect data consistency. If several MBs or GBs need to be copied simply to update several bytes to perform CoW on huge pages, it causes too much overhead. In the case of the `MAP_POPULATE` flag, constructing the page table entries for the entire file may be too much overhead, if only a partial contents of the file is accessed. According to the study on file system workload, large files tend to be accessed only partially [24]. In addition, analytic research on the file access pattern of the network file system reveals that partially accessed files amount to 67% of all files [16]. Therefore, the `MAP_POPULATE` flag should be used only when a sufficient portion of the file will be accessed.

In a study on memory mapped file I/O [25], researchers

proposed a modified page reclamation method that tries to reclaim pages from its own process. They also used vectored I/O for write operations of reclaimed pages. This method reduces context-switch overheads and results in a 92% performance improvement on DRAM-based SSDs. When data-intensive workloads are running, this technique will improve the performance of memory mapped file I/O.

4 Design

To take full advantage of memory mapped file I/O on low-latency NVM storage devices, the following issues need to be addressed. First, the page fault overhead should be reduced while minimizing the prefault overhead. Existing method (`MAP_POPULATE`) is not enough, since it still contains a prefault overhead. Second, the overhead in the page table entry construction needs to be reduced. In low-latency NVM systems, both are a serious burden on the whole system.

4.1 *Map-ahead*

We have designed our map-ahead mechanism to reduce the page fault overhead and prefault overhead in memory mapped file I/O. Our map-ahead technique follows the same principle as the read-ahead mechanism, but we apply it to page table entry construction. All files are maintained in the main memory for in-memory file systems on NVM storage devices. Since the purpose of read-ahead is to bring data from the secondary storage to the main memory, this stage is no longer needed for in-memory file systems. Our map-ahead technique maps the pages that are expected to be accessed before page faults occur.

The virtual memory system in our OS is managed using a demand-paging policy that was designed in the late 1960s when the amount of the main memory was scarce. Despite radical developments in computer systems, it is still used until now [5]. Modern computer systems have a much larger amount of memory and can even operate with in-memory file systems, and this new environment necessitates a new approach. When a page fault occurs, a conventional page fault handler processes only the corresponding page. If page faults occur often to access memory mapped files, the switches between the user mode and kernel mode also frequently occur to invoke the page fault handler. In low-latency NVM systems, frequent mode switches require overhead. Thus, the page fault handler would rather process additional pages that are expected to be accessed soon.

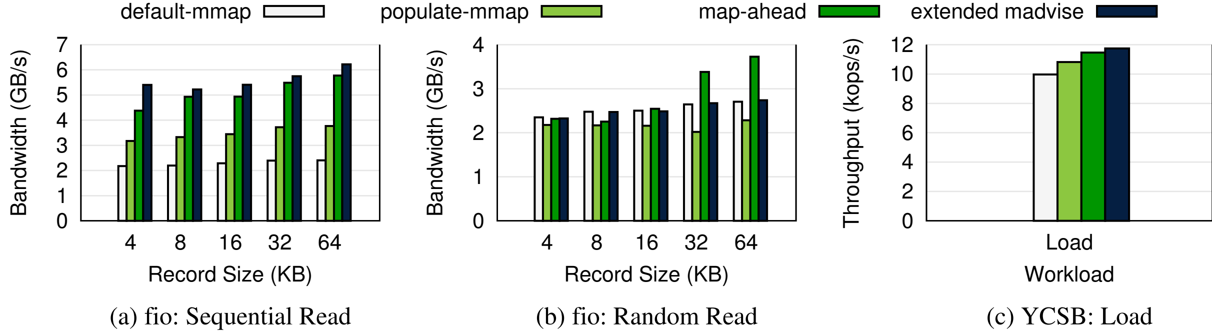
With existing memory mapped file I/O, it is possible to access files with memory operations. The kernel simply manages the memory mapped files as a part of the process address space without considering the file access patterns. However, our map-ahead mechanism dynamically analyzes the page fault patterns to predict the pages that are to be accessed. If page faults occur on sequentially placed pages, it is reasonable to predict that the next page will be accessed. In this way, the page fault handler additionally processes the predicted pages to reduce the number of page faults. Moreover, if sequential page faults occur continuously, the *map-ahead window size* increases. The map-ahead window size is the number of pages that are processed together within a page fault. A bigger window size can be used to reduce the number of page faults. On the other hand, if page faults occur in random locations, the map-ahead window size decreases. If a random page fault repeats, the map-ahead window size will be reduced to one page. In our experiments in §5, the map-ahead window size is doubled when a sequential page fault occurs and is reduced by half when a random page fault occurs.

If a sequential page fault occurs continuously, the map-ahead window size becomes very large. The response time of the page fault handler will also become too long since the pages to process at one page fault pile up. Even though many pages should be dealt with at once, the whole performance can improve by returning the control early to the application. We designed an *asynchronous map-ahead* method to ensure a short response time of the page fault handler, and this also hides the overhead of the page table entry construction. With the asynchronous map-ahead mechanism, the page fault handler synchronously processes only pages that are required immediately and returns to the application context right away. The rest of the pages are processed asynchronously by kernel threads.

In the Linux kernel x86-64, a multi-level page table is used for address translation. An entry in the page middle directory (PMD) points to a page table. This page table is 4KB in size and can contain 512 page table entries (PTEs). A PTE finally points to a page frame. When updating a PTE, the page table where it belongs to is spin-locked. Thus, our map-ahead mechanism synchronously processes pages in the same page table. If pages belong to other page tables, they are handled asynchronously. When we have pages to process across multiple page tables, asynchronous map-ahead can be processed in parallel.

4.2 *Mapping cache*

When `munmap` is called, the kernel releases the virtual memory area (VMA) to which the file is mapped and removes its corresponding page table entries. In traditional systems, it is reasonable to delete them immediately because the memory mapping overhead is negligible compared to the cost of data read from the secondary storage (such as HDDs). Moreover, they will be reclaimed soon from the main memory. However, the memory mapping overhead in NVM storage systems is a huge burden, and the file pages of in-memory file systems remain in the main memory. Thus, new management techniques



(a) fio: Sequential Read (b) fio: Random Read (c) YCSB: Load

Figure 2: Performance of *map-ahead* and *extended madvise*

are needed for in-memory file systems. We propose a *mapping cache* technique to minimize the overhead associated with memory mapping by reusing page table entries. When `munmap` is called, our mechanism does not release VMAs. Instead, the kernel caches the VMAs in the mapping cache to reuse them later.

The mapping cache consists of two parts: a hash table and a LRU list. When the VMAs are cached in the mapping cache, the VMAs are added to the hash table and the LRU list, respectively. The hash table is used to quickly find the cached VMAs, where its hash key is generated with the file path. When `mmap` is called, the kernel checks if the VMA of the file requested for memory mapping exists in the mapping cache. If there is a reusable VMA, the kernel does not allocate a new VMA, but reuses the cached VMA and its page table entries. The LRU list is then used to release the cached but not used VMAs. If the kernel continues to allocate new VMAs but not releases them, the process's address space may become scarce even in 64-bit address space. The kernel checks the threshold whenever VMAs are cached in the mapping cache. If the total amount of cached VMAs exceeds the threshold, excess VMAs are released from the tail of the LRU list. Those are the oldest and presumably have the least chance of being used in the near future. In our experiments in §5, we experimented with two thresholds: 2GB and no-limit.

4.3 *Extended madvise*

The existing `madvise` system call allows a process to give advice to the kernel about the access pattern of the specific memory range. For example, if a process gives a hint to the kernel that a specific memory range will be accessed sequentially (`MADV_SEQUENTIAL`), the kernel aggressively performs read-ahead on the given memory range. If the `madvise` is used suitably, the performance of the applications will be greatly improved. Since `madvise` mainly helps paging between the main memory and the secondary storage, it is unrelated to in-memory file systems.

We have extended the `madvise` system call to take advantage of user hints in the in-memory file systems. In our mechanism, if the kernel receives the `MADV_SEQUENTIAL` or `MADV_WILLNEED` hint, the kernel performs an asynchronous map-ahead on a given range. The map-ahead predicts the access patterns through page fault patterns, but if the extended `madvise` is used, the kernel can predict the access patterns without tracking the page fault patterns. In the case in which the kernel receives the `MADV_RANDOM` hint, the kernel decreases the map-ahead window size to one. In the case the kernel receives the `MADV_DONTNEED` hint, the VMA of the memory mapped file is not cached in the mapping cache. With these extensions to `madvise`, we can manage the memory-mapped file I/O more effectively.

5 Evaluation

Our experiments were performed on a system with an Intel Xeon E5-2620 Processor, running Linux kernel 4.4 extended with our techniques. The main memory and file system settings are the same as the one described in §2 (32GB DRAM, 32GB NVDIMM-N, Ext4-DAX file system on NVDIMM-N).

We measured the performance of *map-ahead* and *extended madvise* by using fio benchmark [4]. It repeatedly reads a 4GB memory mapped file for 30 seconds in a single thread. Figure 2(a) shows the results of a sequential read. When the record size is 4KB, *populate-mmap* improves the bandwidth by 46%, compared to *default-mmap*. This is achieved by the reduced page faults; *populate-mmap* makes about 13 thousand page faults, while *default-mmap* does about 16 million. *Map-ahead* makes about 14 thousand page faults, which is slightly larger than *populate-mmap*, but the achieved bandwidth is far better than *populate-mmap*. Since the overhead of the page table entry construction is mostly hidden by asynchronous kernel threads, the bandwidth is improved by 38% from *populate-mmap*. Applications can provide file access pat-

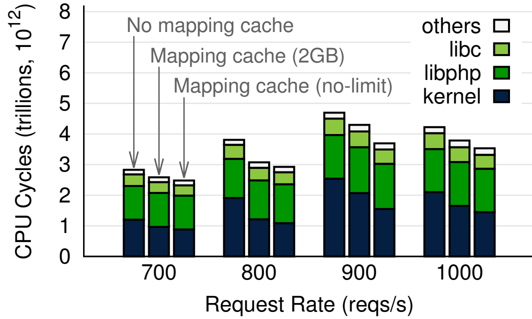


Figure 3: Performance of *mapping-cache*

terns via the `madvise` system call. Fio benchmark also provides `MADV_SEQUENTIAL` on sequential file I/Os to pass the kernel user hint. Our *extended madvise* technique is designed to perform an asynchronous map-ahead based on the user hint. Thus, it improves the performance further by 23%, compared to *map-ahead* technique, which is a 70% improvement compared to *populate-mmap*. The performance of *map-ahead* and *extended madvise* still improve for bigger record sizes (8, 16, 32, 64KB).

Figure 2(b) shows the results of a random read. The performance of *populate-mmap* is the worst for random read. Random read only accesses a partial pages of files, but *populate-mmap* pays the overhead for all pages of the entire file. As the record size increases, the bandwidth of *map-ahead* increases due to short sequential page accesses within a record. Since fio benchmark invokes `madvise` with `MADV_RANDOM` for random access, *extended madvise* shows nearly the same performance as *default-mmap*.

We used MongoDB [17] with MMAPv1 to evaluate the performance of *map-ahead* and *extended madvise*. MMAPv1, which is a storage engine provided by MongoDB, is implemented with memory mapped file I/O. For the workload, we used a database load workload in YCSB benchmark suite [12]. The load workload was used to insert data into MongoDB. The total data size is 20GB, which is a collection of multiple files with various sizes (64MB–2GB). Figure 2(c) shows the result of the YCSB load on MongoDB. Compared to *default-mmap*, *map-ahead* and *extended madvise* improve the performance by 14% and 17%, respectively. Compared to *populate-mmap*, their improvements are 6% and 9%, respectively. One thing to note is that *populate-mmap* may cause a large latency for the load operation if it is the first load operation for a database file. Since *populate-mmap* possesses a prefault overhead at the beginning, the latency of the first load transaction tends to be large. As for *map-ahead*, the performance improvement is rather small, even if the access pattern of the load workload is completely sequential. Since the data was divided into multiple files, aggregated performance improvements become small overall.

We evaluate the performance of Apache HTTP server [2] with *mapping cache* technique. The server executes an Apache HTTP server to get requests from clients. The server has 10 thousand 1MB-size HTML files obtained as copies of Wikipedia page. The total size is about 10GB. In this experiment, we used eight additional machines for the clients. Each machine contains an X5550 Xeon CPU and 64GB of DRAM. Ten client threads from eight machines make requests to the server simultaneously. Each client thread executes httperf, a testing tool used to measure web server performance [18]. The httperf tool requests several URLs in order of appearance in a file which follows a zipf-like distribution.

Figure 3 shows the breakdown of the CPU cycles for five-minute runs of Apache HTTP server. In this experiment, *mapping cache* reduces the number of page faults by 31–51%. As a result, it reduces the overhead involved in the page fault and page table construction. While the portions labeled as *libphp*, *libc*, and *other* do not change much, the CPU cycles used in the kernel are greatly reduced by 24–35% on *mapping-cache*. Compared to *populate-mmap*, the overall performance is improved by 12% for *mapping cache* with a 2GB limit and by 18% for *mapping cache* without limit.

6 Conclusion

The overhead in the software layer is becoming larger than the storage latency as low-latency NVM storage devices become available. The main reason is that existing OSs do not reflect the characteristics of them. Thus, memory mapped file I/O is receiving attention as a way to avoid software overhead. The memory mapped file I/O still incurs expensive additional overhead, and this is mainly caused by the fact that the current mechanism in memory mapped file I/O involves resource management mechanisms that are designed for slow block devices. To exploit the benefits of memory mapped file I/O on low-latency NVM storage devices, we propose map-ahead, mapping cache, and extended madvise techniques that can effectively alleviate the overhead of the mapping files. In our experiments, map-ahead improved the performance by up to 38% over the prefault method (populate-mmap) and an additional 23% performance improvement could be achieved by using extended madvise. In addition, mapping cache improved the performance by up to 18% compared to the prefault method (populate-mmap).

Acknowledgements

This research was supported by Samsung Electronics and Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Plannig (NRF-2016M3C4A7952587).

References

[1] Ahn, J., Kwon, D., Kim, Y., Ajdari, M., Lee, J., and Kim, J. DCS: A Fast and Scalable Device-centric Server Architecture. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48, ACM.

[2] Apache HTTP Server. `https://httpd.apache.org/`.

[3] Arulraj, J., Pavlo, A., and Dulloor, S. R. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, ACM.

[4] Axboe, J. Fio: Flexible IO Tester, 2014. `https://github.com/axboe/fio`.

[5] Basu, A., Gandhi, J., Chang, J., Hill, M. D., and Swift, M. M. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA '13, ACM.

[6] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J., and Swanson, S. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, ACM.

[7] Cho, S., and Cho, Y. Page fault behavior and two prepaging schemes. In *Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications* (1996), IPCCC '96, IEEE.

[8] Choi, J., Ahn, J., Kim, J., Ryu, S., and Han, H. In-memory file system with efficient swap support for mobile smart devices. *IEEE Transactions on Consumer Electronics 62*, 3 (2016), 275–282.

[9] Chung, S. W., Kishi, T., Park, J. W., Yoshikawa, M., Park, K. S., Nagase, T., Sunouchi, K., Kanaya, H., Kim, G. C., Noma, K., Lee, M. S., Yamamoto, A., Rho, K. M., Tsuchida, K., Chung, S. J., Yi, J. Y., Kim, H. S., Chun, Y. S., Oyamatsu, H., and Hong, S. J. 4Gbit density STT-MRAM using perpendicular MTJ realized with compact cell structure. In *2016 IEEE International Electron Devices Meeting* (2016), IEDM '16, IEEE.

[10] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, ACM.

[11] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09, ACM.

[12] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, ACM.

[13] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, ACM.

[14] Intel and Micron's 3D XPoint Technology. `https://www.micron.com/about/our-innovation/3d-xpoint-technology`.

[15] Kim, H., Ahn, J., Ryu, S., Choi, J., and Han, H. In-memory file system for non-volatile memory. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems* (2013), RACS '13, ACM.

[16] Leung, A. W., Pasupathy, S., Goodson, G., and Miller, E. L. Measurement and Analysis of Large-scale Network File System Workloads. In *USENIX 2008 Annual Technical Conference* (2008), ATC '08, USENIX Association.

[17] MongoDB. `https://www.mongodb.org`.

[18] Mosberger, D., and Jin, T. Httperf - a Tool for Measuring Web Server Performance. *ACM SIGMETRICS Performance Evaluation Review 26*, 3 (1998), 31–37.

[19] Narayanan, D., and Hodson, O. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, ACM.

[20] Netlist NVvault DDR4 NVDIMM-N. `http://www.netlist.com/products/vault-memory-storage/nvvault-ddr4-nvdimm`.

[21] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The RAMCloud Storage System. *ACM Transactions on Computer Systems 33*, 3 (2015), 7:1–7:55.

[22] Pappas, J. Annual Update on Interfaces, 2014. `https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140805_U3_Pappas.pdf`.

[23] Park, S., Kelly, T., and Shen, K. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, ACM.

[24] Roselli, D., Lorch, J. R., and Anderson, T. E. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2000), ATC '00, USENIX Association.

[25] Song, N. Y., Son, Y., Han, H., and Yeom, H. Y. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Transactions on Storage 12*, 4 (2016), 19:1–19:27.

[26] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, ACM.

[27] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, ACM.

[28] Wilcox, M. Add support for NV-DIMMs to ext4. `https://lwn.net/Articles/613384`.

[29] Wu, X., and Reddy, A. L. N. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), SC '11, ACM.

[30] Xu, J., and Swanson, S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies* (2016), FAST '16, USENIX Association.

[31] Yang, J., Minturn, D. B., and Hady, F. When Poll is Better than Interrupt. In *10th USENIX Conference on File and Storage Technologies* (2012), FAST '12, USENIX Association.