Performance Analysis of Aerie under Mail Server Workloads

Sanketh Nalli, Salini Selvaraj Kowsalya, Zuyu Zhang
Computer Sciences Department
University of Wisconsin-Madison, WI
sankey@cs.wisc.edu, salinisk@cs.wisc.edu, zuyu@cs.wisc.edu

Abstract

Inspite of the rapid advancements in storage technology, the fundamental architecture of storage in operating system remains fixed. Applications invoke the kernel to store and retrieve data and kernel invokes the file system. Recent work suggests a new class of memory called storage class memory(SCM) which blurs out the distinction between fast, expensive and volatile memory, and slow, cheap and non-volatile storage. The file system for such a class of memory does not need kernel interaction. Applications can modify the file system and optimize it according to their needs. One such flexible file system architecture is provided by Aerie [13]. It exposes storage class memory to user mode programs which allows direct access to memory from user mode. Aerie[13] evaluates the application level performance on three Filebench profiles: file server, web server and web proxy. We are interested in measuring the performance of Aerie for mail server profiles. Mail servers usually deal with a large number of small files which are in constant flux. It has frequent reads/writes, create/delete and appends. In this paper, we adopt two simple benchmarks which mimic the behavior of a mail server and alter them to use the file system APIs exposed by Aerie. We compare the performance of Aerie with tmpfs and ext3 file system (with and without caching) on linux 3.2.2.

1 Introduction

Storage-class memory is a new class of devices that provide the interface of memory along with the persistence of disks. It combines the benefits of a solid-state memory (high performance and robustness) with archival capabilities of conventional inexpensive HDD [3]. It is created out of flash-based NAND that can provide read performance that is nearly as good as DRAM and write performance that is significantly faster than HDD technology. That said, one may think of using a flash-based storage devices as it can offer compelling economics versus DRAM. In order to be considered as an acceptable alternative to DRAM memory, it must provide near-RAM performance. Emerging device technologies such as phase-change-memory (PCM), spin-torque transfer RAM (STT-RAM) and memristors provide persistent storage near the speed of DRAM. These technologies collectively are termed storage-class memory (SCM) as data can be accessed through ordinary load/store instructions rather than through I/O requests.

Traditionally, persistent storage has resided behind both a bus controller and storage controller. Since the latency of a read or a write is dominated by the access to this device, the overhead of this architecture does not materially affect performance. In contrast, technologies such as storage class memories have access latencies of a few nanoseconds. Thus keeping SCMs behind I/O bus would waste the performance benefits of the storage medium [6]. Instead, SCMs can be attached to the memory bus, thus reducing the latencies to access persistent storage. Since they allow processor to access persistent storage through memory load/store instructions, they can successfully provide simpler and faster techniques for storing persistent data [14]. Thus with two significant features such as high speed and direct access from user mode, SCM exhibits a fundamental shift from the existing storage architectures.

In present operating systems, file systems need kernel intervention for dealing with storage : for data protection and abstraction of storage device through a driver. These kernel file systems significantly mar performance when operating on SCMs. They are designed for storage devices so as to support electromechanical access to data. The existing OS structure of file systems as a kernel-level service may no longer be necessary with SCM. The cost of exposing storage to user-code and enabling direct access to data is far lesser than requesting the kernel to do so. Many file systems designs have been proposed for effectively utilizing SCM. File systems can also be customized

to improve I/O performance and the new interface can be exposed to the application.

Aerie is one such flexible file system that exposes SCM to user-mode processes. It allows users to enable their applications to define their own file system interface and implementation without extending the kernel. It also offers low-latency access to data by removing layers of code otherwise present in the kernel file system. In short, using Aerie, applications can link to a file system library that provides logical access to data and communicates with a service for coordination. The OS kernel provides only coarse-grained allocation and protection.

Aerie exposes POSIX APIs for legacy applications. It can be used to implement many other file system interfaces as well. In Aerie [13], the authors have evaluated the application-level performance with three FilbBench profiles - file server (which emulates file server activity and performs sequence of creates, deletes, appends, reads and writes), web server (which performs sequence of open/read/close on multiple files and appends log to a file) and web proxy (which performs sequence of create/write/close, open/read/close on multiple files in a single directory and appends to a log file). They have compared the performance and reported the latency against traditional and user mode file system.

In this paper, we evaluate Aerie's application-level performance with two mail server benchmarks, PostMark and Bonnie. We modify the source code of these benchmarks to use APIs provided by Aerie. We replace open, read, write and close calls with corresponding libfs calls. We observe that Aerie is close to tmpfs in performance for small reads and small writes and is much faster than ext3 file system without caching. With caching, all three perform alike.

2 Motivation

It is predicted that by 2020, server-room power demands will be too high because of the advancements in computer applications [8]. Compute centric and data centric programs will require more power and storage space for keeping hard disks. The cache/memory/storage hierarchy might soon become the bottleneck for large systems. Though MIPS and MFLOPS will be inexpensive, storing and retrieving data through that will decrease the performance of the system. It is time to identify the root cause of this problem and build a technology that will solve it. Careful studies tell us the root cause of this problem is the access time gap between memory and storage. A solid-state memory like Phase-Change memory (PRAMs), STT-RAMs and memristors that blurs the boundaries between storage and memory by being low-cost, fast, and non-volatile is found to be the solution for this problem.

Despite rapid advancements in storage technology, the fundamental architecture of storage in operating systems has remained stable: applications invoke the kernel to store and retrieve data, which invokes a file system and then a block driver. As a result, the overhead brought by I/O latency is much higher than that of file system layer itself. But SCM has no features that compel for a kernel implementation of file systems. Protection to the data can be given by memory-translation hardware. Furthermore, it has much less need for scheduling to optimize latency, as there are no long seek or rotation delays. Because SCM provide speeds near DRAM, caching data may be unnecessary. Finally, SCM does not require a driver for data access as it can implement a standard load/store or protected DMA interface.

Several file systems [9] [6] [10] [4] [7] [14] [1] [11] have been developed for SCM (see Section 6). In contrast to most of the file systems, Aerie [13] allows users to build a file system as a library, offering great flexibility to applications. In their paper, the authors evaluated the flexibility and performance of different aspects of file system with Filebench profiles(file server, web server and web proxy). These file system benchmarks emphasize on raw throughput or performance on large, relatively long-lived groups of files.

But current file servers provide services such as electronic mail, netnews, and commerce service, which depend on enormous numbers of relatively short-lived files. These systems are inherited from an era predating the exponential growth of the Internet and were never envisioned as being scalable to today's required levels. This encourages us to evaluate Aerie's performance on such short-lived files. Benchmarks such as PostMark and Bonnie were designed to create such a pool of short lived files, continually changing files, to measure the transaction rates for a workload approximating a large Internet electronic mail server.

From our performance analysis, we try to determine if Aerie is suitable for small files which are in constant flux and which are frequently created/read/written/appended/deleted.

3 Design

This section describes the design of our experiments. We compare the mail server perfor-

mance of Aerie against tmpfs and ext3 file system (with/without caching). We use two simple benchmarks which mimics the behavior of a mail server. The benchmarks which we considered are PostMark and Bonnie. We ensured that the benchmark specifies several options allowing us to configure and stress the file systems in interesting ways.

Mail servers usually deal with a large number of small files. Not only are these files relatively small (from one kilobyte to over one hundred kilobytes, depending on content), they are constantly in flux. At any given time, files are being rapidly created, read, written or deleted, all over the disk drives allocated for these tasks. We are interested in observing how the underlying file system performs under heavy workloads. While performing the experiments, we configure our benchmark accordingly.

In the following sections, we explain how PostMark and Bonnie generates workloads.

3.1 PostMark

PostMark [12] generates an initial pool of random text files ranging in size from a configurable low bound to a configurable high bound. This file pool is of configurable size and can be located on any accessible file system. Once the pool has been created (also producing statistics on continuous small file creation performance), a specified number of transactions occurs. Each transaction consists of a pair of smaller transactions:

- Create file or Delete file
- Read file or Append file

Create/Delete files: The incidence of each transaction type and its affected files are chosen randomly to minimize the influence of file system caching, file read ahead, and disk level caching and track buffering. This incidence can be tuned by setting either the read or create bias parameters to produce the desired results. When a file is created, a random initial length is selected, and text from a random pool is appended up to the chosen length. File deletion selects a random file from the list of active files and deletes it.

Read/Append files: When a file is to be read, a randomly selected file is opened, and the entire file is read (using a configured block size) into memory. Either buffered or raw library routines may be used, allowing existing software to be approximated if desired. Appending data to a file opens a random file, seeks to its current end, and writes a random amount of data. This value is chosen to be less than the configured file size high bound. If the file is already at the maximum size, no further data will appended. When all of the transactions have completed, the remaining active files are all deleted. It also produces statistics on continuous file deletion.

3.2 Bonnie

Bonnie [2] performs a series of tests on a file of known size. If the size is not specified, Bonnie uses 100 Mb. Bonnie can work with 64-bit pointers if they are available. For each test, Bonnie reports the bytes processed per elapsed second, per CPU second, and the CPU usage (user and system). The tests are

1. Sequential write character wise
2. Sequential read block wise
3. Sequential write block wise
4. Sequential read character wise
5. Sequential rewrite block wise

Sequential write character wise: The files are written using the *putc()* stdio macro. The loop that does the writing has to be small enough to fit into any reasonable I-cache.
Sequential write block wise: The files are created using *write(2)*. The CPU overhead would be just the OS file space allocation.
Sequential rewrite block wise: Each chunk of the file is read with *read(2)*, dirtied, and rewritten with *write(2)*, requiring an *lseek(2)*. Since space is not allocated, and the I/O is well-localized, this will test the effectiveness of the file system cache and the speed of data transfer.
Sequential read character wise: The file is read using the *getc()* stdio macro. Even in this case, the inner loop is small.
Sequential read block wise: The file is read using *read(2)*. This is a very pure test of sequential input performance.
Random Seeks: This test runs SeekProcCount processes in parallel, doing a total of 4000 *lseek()*s to locations in the file computed using by *random()* in bsd systems, *drand48()* on sysV systems. In each case, the block is read with *read(2)*. In 10% of cases, it is dirtied and written back with *write(2)*.

4 Implementation

We performed our experiments with 64-bit Intel(R) Core(TM) i5-2500K CPU at 3.30GHz

running GNU/Linux Kernel 3.2.2, with 16 GB of physical memory. We altered the source-code of PostMark and Bonnie to incorporate Aerie's APIs along with POSIX APIs and perform file operations using either of them.

Postmark uses $fwrite()$, $write()$, $fread()$ and $read()$ calls for file access. At run time, one can choose between $fwrite()/fread()$ or $write()/read()$ either by typing in *set buffering true* or *set buffering false* respectively. We altered parts of PostMark that use $write()/read()$ to use $libfs\_write()/libfs\_read()$. We also altered the part of code that use $fwrite()/fread()$ to use $write()/read()$. $libfs\_open()$ and $libfs\_close()$ replaced $open()$ and $close()$ respectively.

After successful modification of PostMark source code, we built the benchmark by linking it against client-dependent pxfs libraries provided by Aerie. These libraries are the ones that implement all the libfs APIs.

We then started off with our experiments by creating 20 files of size ranging from 10KB to 15KB. We increased the number of files in steps of 1000 till we observed a significant degradation in the performance of Aerie. The reason behind starting with a very small number(20) is that we wanted to measure the performance of ext3 file system without caching. With even 20 files, ext3 file system (without caching) showed very poor performance. The results are given in the next section.

As for Bonnie, we also encountered $sync()$ and $fsync()$. We omitted these pieces of code as the current PXFS writes synchronously and hence $sync()$ is unnecessary. We replaced $write()/fwrite()/putc()$ calls with $libfs\_write()$ calls. Similarly we followed the same suite for read i.e replace $read()/fread()/getc()$ calls with $libfs\_read()$ calls. $libfs\_open()$ and $libfs\_close()$ replaced $open()/fopen()$ and $close()/fclose()$ respectively. Once the porting is complete, we built Bonnie by linking it against client-dependent pxfs libraries provided by Aerie.

We collected the performance measurements for Bonnie by generating files of sizes in the order of megabytes. Bonnie gives the following performance measures:

- Number of kilobytes of data read per second, if we read one character at a time
- Number of kilobytes of data read per second, if we read one block at a time

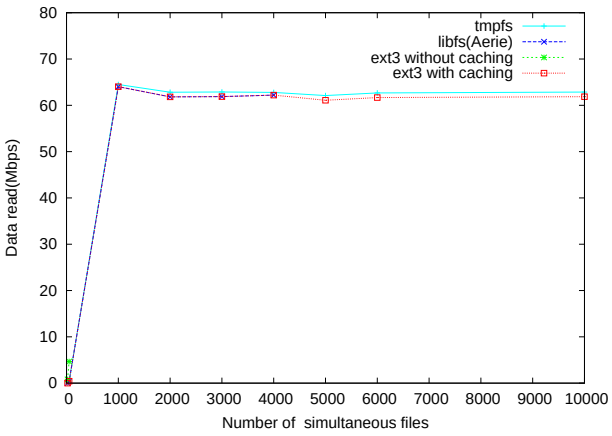

Figure 1: **PostMark: Performance comparison of data read per second of various file systems**

- Number of kilobytes of data written per second, if we write one character at a time
- Number of kilobytes of data written per second, if we write one block at a time

We present the results in the next section.

4.1 Challenges

When conducting our experiments, we were faced with numerous challenges. The most frequent one was the Aerie file server crash during most trials. We are still unsure of the root cause but suspect that the file server is simply unable to accept connections from user-level applications beyond a certain limit. The exact figures are reported in the section below. Other issues were minor and involved claiming a pool of memory to act as SCM, altering source-codes of benchmark to adhere to standards imposed by PXFS etc. These were resolved in time.

5 Evaluation

5.1 PostMark evaluation results

Figure 1 shows the performance comparison of data read in the order of megabytes per second for various file systems. Similarly, Figure 2 shows the performance comparison of data written in the order of megabytes per second for various file systems. From the graphs it is evident that Aerie keeps up with tmpfs in read/write performance. We observed that when we increase the number of simultaneous files greater than 5000, Aerie's file server crashed.

The surprising observation here is that we did not observe a gradual descent in Aerie's performance.

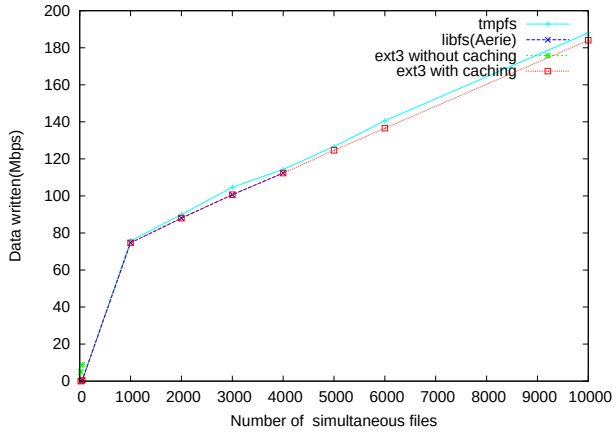

Figure 2: **PostMark: Performance comparison of data written per second of various file systems**

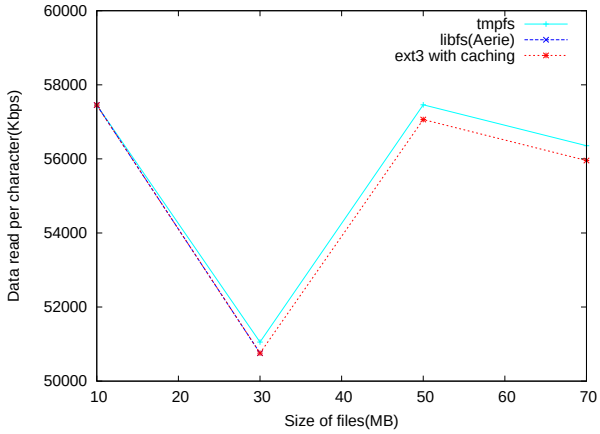

Figure 3: **Bonnie: Performance comparison of data read per second(character wise) of various file systems**

Rather we observed an abrupt crash after a certain limit. We suspect that this might be because of some of implementation bugs in Aerie. We are looking into Aerie's code to find the bug.

We also evaluated the file system performance of ext3 with caching. Our results showed that ext3 with caching performs as good as Aerie, and it continued to perform well even when the number of simultaneous files was increased to 10000. The evaluations results of ext3 file system without caching showed poor performance in comparison with other file systems. This is obvious because it is extrememly time consuming to read/write data in hard disk, synchronously.

5.2 Bonnie evaluation results

Figure 3 and Figure 4 show the performance comparison of data read per second(character wise/block wise) of various file systems. From the experiments we found that, Aerie's read performance is almost as good as tmpfs performance. But as said earlier, file server crashes when we tried to read from a file of size greater than 30 MB. But ext3 file system with caching performed as good as Aerie and continued to perform well even after we increased the file size beyond 30 MB. We are still trying to figure out why there is a steep decrease in amount of data read per second when we increased the file size from 10 MB to 30 MB. As you can see from the graphs that if we read the file block wise, then amount of data read per second is much greater than that of reading the file character wise.

Similarly Figure 5 and Figure 6 show the performance comparison of data written per sec-

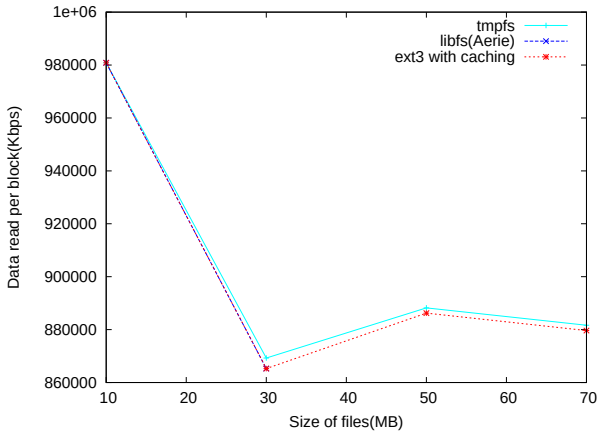

Figure 4: **Bonnie: Performance comparison of data read per second(block wise) of various file systems**

ond(character wise/block wise) of various file systems. Aerie's write performance was also observed to be almost as good as tmpfs performance. But file server crash bug was still present when we increased the file size beyond 30 MB. Our experiments proved that the write performance of ext3 file system with caching was same as that of Aerie and continues to work even after we increased the file size beyond 30 MB. As you can see from the graphs, writing data block wise is much more efficient than writing data character wise.

From the above results, we conclude that Aerie can be used for mail server applications and it can achieve high performance by optimizing the file system without any changes to complex kernel code.

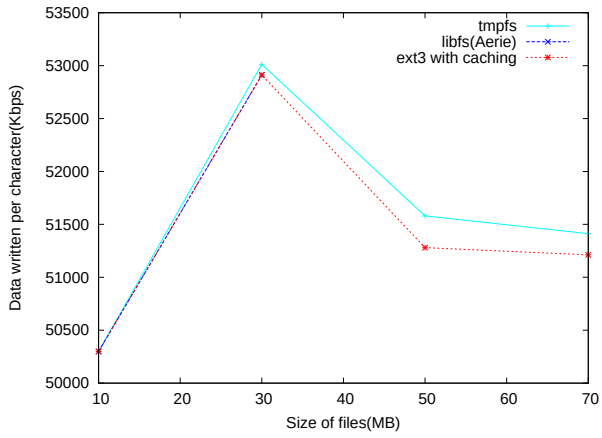

Figure 5: **Bonnie: Performance comparison of data written per second(character wise) of various file systems**

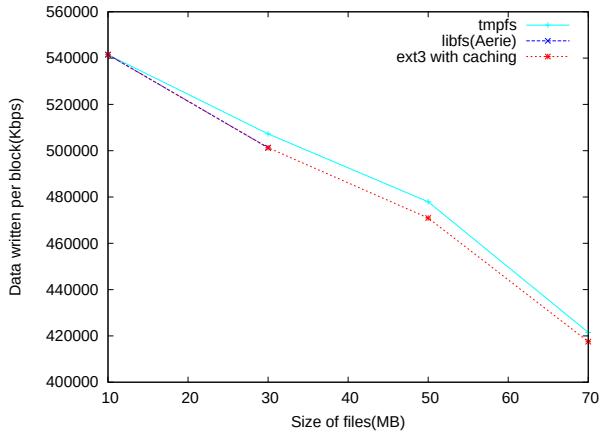

Figure 6: **Bonnie: Performance comparison of data written per second(block wise) of various file systems**

6 Related work

The complexity of kernel file systems is not necessary for SCM. A plenty of studies focus on integrating fast user level access of SCM, into existing file system. In the rest of this section, we present a summary of various file systems developed for SCM.

SCMFS [14] is a file system that aims to minimize CPU overhead of file system operations. It is built on virtual memory space and utilizes the memory management unit (MMU) to map the file system address to physical addresses on SCM. It is designed to reuse the memory management infrastructure, both in hardware and the OS. The space is kept contiguous for each file in SCMFS to simplify the process of handling the read/write requests in file system. Further, frequent allocation and de-allocation can invoke many memory management functions and can potentially reduce performance. To avoid this, space pre-allocation mechanism is adopted. In this mechanism, when a file shrinks or is deleted, it is simply marked as a null file. When space has to be allocated, the system looks for the first null file. SCMFS was implemented in Linux 2.6.33, as a prototype.

BPFS [6] is a file system designed to achieve high performance, strong safety and consistency. It guarantees that all system calls are reflected to the storage atomically. It allows data to be made durable as soon as the caches contents are flushed to persistent storage. Consistency is enforced using short-circuit shadow paging, which means that updates are committed either in-place or using a localized copy-on-write. Further, BPFS does not allow storage of complex data structures such as hash tables. Instead, simple, non-redundant data structures are stored in persistent memory and this data is cached in more efficient volatile structures. BPFS was implemented in Windows driver model and supports open, read, write, close operations.

Hitz et. al [9] designed a file system for an **NFS** file server appliance, which uses Write-Anywhere-File-Layout (WAFL) for NFS access pattern. In order to avoid consistency checks after abnormal power-off, WAFL uses battery backed NVRAM as write cache to store logs of NFS requests. Conquest[10] is a hybrid file system that combines persistent RAM with conventional disks. Conquest stores meta data and small files in NVRAM that emulated by battery-backed DRAM, while still uses disks for large file storage. But both BPFS and NSF doesnt implement file system as a library which can be optimized for specific applications. Implementing filesystem as a library might offer better flexibility to the application program.

Moneta [4] was an initial study that adopted NVRAM as memory for high performance purpose. They proposed a storage array and their I/O scheduler to study the benefits from different kinds of NVRAM, such as PCM, SSD. To eliminate the overhead from file system, Moneta-Direct [5] virtualizes Moneta interfaces and adopts user level library while enforces permission checks in the kernel. Thus, Moneta-Direct has a direct data access from user level. But metadata operations still need the involvement of kernel mode for protection concerns.

DFS [11], a file system for flash storage, stores the files directly in a single very large virtual storage address space considering it as a single level store. And then it leverages the virtual flash storage layer

to perform virtual to physical block allocations.It is a generic kind of file system which any applications can use. Though it reduces the data path to access flash memory, it does not provide application-specific optimization.

Logical Disk [7] and ZFS [1] have built file systems using system primitives higher than block abstraction. Logical disk provides a clean separation of file and disk management by providing an abstract interface to disk storage by using logical block numbers and block lists. The file system writes to the logical block, and the logical disk chooses the physical location on the disk where the block will be written. It is flexible because different logical disks can be implemented according to the different access patterns and different disk can use different implementation of logical disk. On the other hand, ZFS provides abstraction by decoupling file systems from physical storage. Allocation and deallocation functionalities should be provided by storage space allocator by allocating permanent storage space from a pool of storage devices as file system requests it. Our system will also similarly expose the primitive structures to the file system interface layer.

7 Conclusions

From our experiments, we realize that Aerie has the capability to perform similar to tmpfs. Ext3 with caching can act as a good substitute for SCM but is prone to power failure leading to loss of data stored in memory. tmpfs too is prone too such a loss and hence data must be written to disk at regular intervals. These problems can be avoided with the use of SCM. Research on SCM is in its infancy and can see the light of day in near future if we continue to put consistent efforts.

References

[1] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. Technical report, 2003.

[2] T. Bray. Bonnie.

[3] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4-5):449–464, 2008.

[4] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, 2010. IEEE Computer Society.

[5] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. 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*, ASPLOS '12, pages 387–400, 2012. ACM.

[6] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *SOSP*, pages 133–146, 2009.

[7] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *SOSP*, pages 15–28, 1993.

[8] B. Geroffrey.

[9] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, 1994. USENIX Association.

[10] A. i A. Wang, P. Reiher, and G. J. Popek. Conquest: better performance through a disk/persistent-ram hybrid file system. In *In Proceedings of the 2002 USENIX Annual Technical Conference*, pages 15–28, 2002.

[11] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. Dfs: A file system for virtualized flash storage. In *FAST*, pages 85–100, 2010.

[12] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Applicance Inc., 1997.

[13] H. Volos and M. M. Swift. Aerie : Distributing file system functionality for direct access to storage class memory.

[14] X. Wu and A. L. N. Reddy. Scmfs: a file system for storage class memory. In *SC*, page 39, 2011.