# Storage-Class Memory Needs Flexible Interfaces

Haris Volos[†], Sankaralingam Panneerselvam[*], Sanketh Nalli[*] and Michael M. Swift[*]
[†]*HP Labs, Palo Alto and* [*]*University of Wisconsin–Madison*

**Abstract**

With low-latency storage-class memory, software can be a major contributor to access latency. To minimize latency, a file system architecture has to provide flexibility in customizing the file system interface and semantics to application needs so as to cut down generic overheads. We have taken initial steps towards realizing such a design and present preliminary results.

## 1 Introduction

Emerging *storage-class memory* (SCM) [10] technologies, bring the best of two worlds: the low-latency and random-access of memory together with the persistence of disks.

With low-latency storage though, software can be a major contributor of overhead to access latency. To address this issue, previous work has proposed memory mapping files into the kernel [6] or user mode (*e.g.*, [4]) for direct access to data through load and store instructions. However, file system functionality that requires accessing metadata, such as naming, protection, and shared access, is still provided through the file system interface. This introduces two pieces of overhead: (i) cost of calling into the kernel, and (ii) cost to abstract a wide range of resources as files and support generic semantics regarding metadata. For instance, even when an application implements their own synchronization, the file system has to grab locks during writes to support POSIX's atomic-write semantics. For metadata intensive applications, the file-system interface can introduce a substantial overhead.

We propose implementing flexible file-system interfaces as a user-mode library directly accessing file-system data and metadata in memory. Direct access to *metadata* enables the implementation to optimize interface semantics and operations regarding metadata to the specific needs of the class of applications the library targets. This helps cut down generic overheads. We have taken initial steps towards realizing such a design and present preliminary results. Our results are encouraging. We can still implement POSIX with good performance, but with a key-value file interface we can achieve a 86% performance improvement over a kernel file system. This suggests that major performance benefits can come by customizing the interface to the workload.

## 2 Motivation

### The storage landscape today

Emerging storage-class memory (SCM) technologies promise to change many assumptions about storage. They have the persistence of storage but the fine-grained access of memory, and can be attached to the memory bus and accessed through load and store instructions [10]. Four recent technologies provide SCM capabilities: phase-change memory (PCM) [16], spin-transfer-torque MRAM [13], flash-backed DRAM [1], and memristors [19]. While the performance and reliability details differ, they all provide byte-granularity access and the ability to store data persistently across reboots without battery backing. SCMs are currently commercially available in modest sizes of up to 8GB, with Viking Technology and Micron planning to deliver up to 32GB-size flash-backed DRAM DIMMs within the coming year [16, 9, 1].

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. Four features of past storage technologies require this layered design in the kernel:

1. **Protection:** Disks and other block storage devices do not implement a protection mechanism to limit access by a user or process.
2. **Scheduling:** Disks have variable latency from seek and rotational delays and benefit significantly from scheduling to reorder requests.
3. **Caching:** Slow disks benefit from shared caches that allow processes to re-use data fetched by another process.
4. **Drivers:** Disks implement a variety of interfaces and therefore require a driver to present a standard block interface.

Due to the slow speed of disks and the high benefits of scheduling and caching, a kernel implementation of file systems provides many performance benefits at little additional cost. For low-latency SCM devices, however, the cost introduced by this kernel-level layered design is relatively more expensive and can drastically limit the performance of SCM.

In fact, several of the services provided by today's organization are even unnecessary as SCM has none of the features that require a kernel implementation of file systems. As memory, it can be protected by existing 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 provides speeds near DRAM, shared caching may not be as necessary. Finally, SCM does not require a driver for data access as it can implement a standard load/store or protected DMA interface [4]. Previous work has proposed removing some of the kernel layers providing these services, such as scheduling and drivers, by memory mapping files directly into the kernel or user-mode processes for fast access to data [6, 4, 8, 5, 21]. However, they still rely on the POSIX file-system interface for everything else, including protection, global naming, and shared access.

**The abstraction cost of a file**

The file-system interface introduces two pieces of overhead. First, there is a cost of changing modes and cache pollution from entering the kernel [18]. For example, PCM read latencies can be as low as 70 nanoseconds, while the time for a `stat` system call in Linux on a modern processor is approximately 800 nanoseconds.

Second, there is a cost due to the generality of the file-system interface that abstracts every system resource as a file. This generality offers programming convenience but comes at the cost of supporting interoperability between logically different resources, such as disk files and network sockets. For disks, where the I/O cost dominates the abstraction cost, abstraction comes at a relatively low cost. However, for low-latency SCM, similarly to fast networking [12], abstraction becomes expensive as the I/O cost is substantially lower. Abstraction requires the file-system interface to associate all resources with common generic data structures, including reference-counted file descriptors, virtual inodes and dentry objects.

| Operation | Latency ($\mu$s) | Operation | Lat. ($\mu$s) |
|---|---|---|---|
| stat | 0.80 | chmod | 1.4 |
| open | 2.0 | fdup | 0.15 |
| unlink | 3.1 (2.3) | | |

**Table 1: Latency of common POSIX operations** *Unlink cost in parentheses is the cost of unlinking an open file.*

Moreover, generality does not manifest only across different resources but also within a single class of resources. For our particular class of interest, that is storage files, the file-system interface has to support generic semantics regarding file metadata, which can have substantial overhead. For example, POSIX sharing semantics allows files to be unlinked while open. These are useful semantics, for example, for applications that require creating anonymous files but costly for all other applications that do not as it requires synchronizing access to inode metadata to reference count inodes. As shown in Table 1, unlinking an open file is 34% faster than unlinking a non-opened one. Since the open file is not really deleted until it is closed, this difference suggests that about one third of the time of unlink is spent in managing underlying inode and dentry structures.

Overall, while the file-system interface comes with a significant performance cost for accessing SCM, it also provides useful features, such as organizing data under a global logical namespace for easy access and protecting data for secure sharing between applications. Moving forward, we believe that such features will remain relevant. However, limiting ourselves to accessing these features via a single interface and semantics, as we do today with the POSIX interface to the kernel, will unacceptably limit performance by their semantics.

## 3 Key Idea: Flexible Library Interfaces

We propose implementing flexible file-system interfaces as a user-mode library directly accessing file-system data and metadata in memory. This provides two key benefits: (i) low-latency access to data and metadata by removing layers of the storage stack and avoiding the costs of trapping into the kernel [18], and more importantly (ii) flexibility by enabling applications to define their own file-system interfaces and implementation.

An application can optimize file system policies but more importantly optimize the interface. Having *direct access to shared metadata* is what enables the implementation to optimize interface semantics and operations

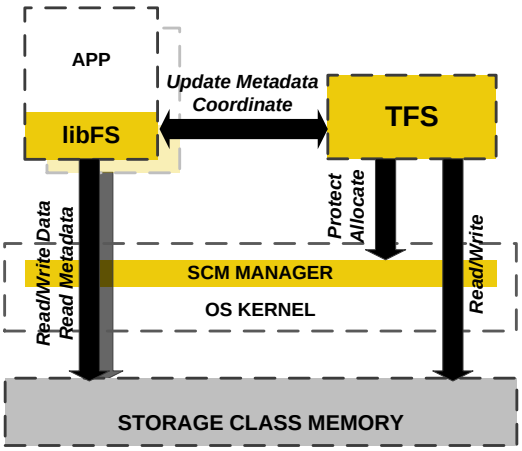


**Figure 1: Decentralized Architecture.** *Functionality is distributed between application processes, a trusted service, and the kernel.*

regarding metadata to the specific needs of the class of applications the library targets. This can be quite beneficial for metadata intensive applications, such as photo stores [3] and IMAP mail servers [7], that use many small files and could benefit from a get/put file interface that optimizes for whole access to small files.

Designing a file system for flexibility and direct access is at odds with one of the fundemental roles of file systems, which is supporting sharing between mutually distustful programs. Mutual distrust requires programs to be defensive against possibly malicious actions by a foreign process. For example, a malicious program may corrupt metadata invariants in parts of the file system it has access to such as inserting two files with the same name in a directory. Fundamentally, addressing these concerns requires a trusted entity [17]. We want to avoid resorting to a purely centralized design though; such a design would require frequent invocation of a kernel- or server-based trusted file system [15] and would eliminate the benefits of direct access.

### 3.1 Architecture

Figure 1 illustrates our proposed architecture. We distribute functionality between an untrusted library and a trusted service to efficiently support protection, synchronization, and integrity. We reduce the kernel role to just multiplexing physical memory via the SCM manager.

**libFS Client Library.** Applications link against a libFS library for each file-system interface they use. Our design relies on hardware protection to enforce access control over file system data. This allows the client library to service most file-system operations directly from SCM without contacting the trusted service. For example, when an application opens a file, the library accesses directory contents in SCM to locate the file and can then read or write file data directly from SCM. The library also implements logic to invoke a trusted service when needed.

**Trusted File System (TFS) Service.** Functionality that requires a trusted third party, including integrity for metadata updates and concurrency control between processes, execute in the TFS service. First, for metadata integrity, clients do not apply metadata updates directly but instead they create a log of their updates, similar to a file system journal, and periodically ship the journal to the TFS. TFS validates the entries in the journal are legal and preserve invariants, such as link counts and free/allocates status. Then, it applies them as a transaction by forcing the log to SCM and then updating data structures. Second, to ensure that clients do not journal and batch conflicting operations, TFS provides a distributed hierarchical lock service that issues leases to clients. Leases can be organized hierarchically to reduce calls into the service. So for example, a client can do a single call to acquire a lease on a directory, which implicitly locks the files of the directory.

Putting it all together, with this architecture, when an application starts, it calls the library to mount the file system, which memory maps the whole file system into the process and relies on hardware protection to enforce file-system permissions. With direct access, a program reading a file can read metadata directly to locate the file contents and then read the data directly without calling the kernel.

### 3.2 Comparison with related work

Achieving performance improvements by matching application needs to storage-system design has been a recurring theme in the systems community. For example, Google's GFS optimizes for web data [11] and Facebook's Haystack optimizes for images [3]. Exokernel [14] and Nemesis [2] have explored exposing storage to user-mode for application performance and flexibility. However, they still maintain protection of the block device within the kernel, so storage access still requires invoking a kernel-mode device driver. Thus, accessing metadata still requires expensive calls into the kernel. Moreover, their focus is primarily on flexibility to optimizing data structure layout for efficient disk access, while our focus is on flexibility to optimizing the interface to cut down overheads due to generic interface semantics.

Recent work has explored high-performance file-system designs targeted for SCM. BPFS [6] memory-maps files directly into the kernel, and Moneta-D [4] and Quill [8] map files into user-mode processes. Memory-mapping files enables fast access to data, which in turn enables the construction of high performance persistent data structures and stores [5, 21, 20]. However, they

all still rely on the POSIX file-system interface for everything else, including protection, global naming, and shared access. Thus, file-system operations continue paying overheads due to generic interface semantics and for accessing metadata through the kernel.

## 4 Open Research Questions

There are several open research questions from the above architecture. We next discuss several of them.

**How to support shared access through different interfaces?** Supporting different library implementations that co-exist and provide optimized interfaces on top of the same memory layout can enable high performance access to SCM without sacrificying the sharing features of file systems. This requires however interfaces to be compatible. For example, a key-value-store interface and a POSIX-like interface have compatible concurrency semantics because whole-file access via get/put is atomic and does not run into any concurrency issues with POSIX operations such as read, write, or unlink. We are in the process of formalizing what constitutes two interfaces compatible, that is what semantics, properties and operations they need to have compatible, and identify such interfaces. For example, is a graph-store interface compatible with POSIX?

**How limiting is a library interface?** Moving the file system into user mode may preclude interoperability of the file system with other kernel functionality such as networking. The main challenge arises from our library's lack of file descriptors and system-wide inodes that allow interoperability between different types of resources. File descriptors are deeply engraved into the kernel interface today with many applications depending on their functionality. For example, process creation allows a child process created through fork() to inherit file descriptors from its parent. Making a library compatible with the kernel, and creating file descriptors on demand when needed [12] could work but in general is hard and inefficient. We are interested in exploring alternative approaches to supporting file-descriptor functionality.

**What are the security implications?** We see two challenges. First, our architecture may be vulnerable to resource denial of service attacks where a malicious application locks too many files in the file system and does not relinquish such locks. One strawman solution would be to terminate such applications, but this approach requires identifying a malicious application from a benign application that just happens to need to lock a large number of files. Second, file systems need to avoid leaking stored data when recycling pages between users without inefficiently zeroing pages. Kernel file systems do not suffer from this problem as they mediate access to each page and therefore control what content is available.

**How can we enable scalable crash consistency?** When clients need the equivalent guarantee of an fsync(), our current design requires a synchronous call into TFS to durably perform outstanding metadata operations. However, workloads with frequent *fsyncs* (e.g. mailserver) can cause frequent calls into TFS, which can greatly degrade performance due to the latency overhead of synchronous communication. We are currently exploring crash-consistency protocols that do not require synchronous calls into TFS by leveraging the fact that the per-client journal of metadata operations can be durably stored in client local SCM. This property alone though does not guarantee recovery as the recovery process has to deal with clients that after failure may no longer hold locks for the updates logged in their journal.

**How flexible and efficient is virtual memory hardware?** Hardware memory protection helps us decentralize and optimize access control, but it introduces three challenges. First, memory and file systems do not have perfectly compatible protection models: memory typically grants read or read/write access, while files may have write-only access. In addition, metadata may have semantically richer permissions, such as directory list and search. Second, permission changes on a file must be efficiently relfected as changed protection on memory pages. Currently, it may require touching the page-table entry for every page of a file. Finally, memory mapping terabytes of SCM into an address space requires a large page table and increases the pressure on TLBs when using 4KB pages. Larger pages avoid this problem but increase internal fragmentation. Exploring new hardware techniques for efficiently protecting and mapping SCM is an open problem.

## 5 File-System Interfaces on Aerie

A major goal of Aerie is to provide a substrate for flexible file-system design. To demonstrate this capability, we implemented two file system interfaces on top of the same memory layout. The first, PXFS, shows how to use the storage abstractions to implement a POSIX-style file system interface for compatibility with existing code. The second one, KVFS, shows how to optimize the interface for a specific workload.

### 5.1 PXFS: POSIX-style File System

PXFS provides most POSIX semantics for files and directories, including moving files across directories and retaining access to open files after they are unlinked.

PXFS implements file objects as a radix tree of memory pages and directory objects as a linear hash table mapping string names to the object identifiers of files and directories. Each object is identified by its virtual memory address.

PXFS supports concurrent file access. When a client

opens a file, it creates a volatile *shadow file object* and acquires a lock on the file, which it holds until it closes the file. The shadow object provides a way for the client to continue accessing the underlying file object when that file is concurrently unlinked or renamed. When another client requests the lock on an open file, clients with the file open notify the service that the file is open when releasing the lock. The service then adds the file into a list of currently open files. The client can still obtain explicit locks on the file object to read or write data, and when the client terminates or notifies the service that it has closed the file, the service reclaims the file's memory. This design guarantees the client can directly access the file even if other clients unlink or rename it.

With this design, read-only access to files only communicates with the TFS service to acquire locks, and if there are no conflicting accesses, a coarse grained lock high in the file system tree suffices. The client can write to file data locally, including writing new data to files, but must communicate with the service for metadata changes such as creating or appending to a file.

### 5.2 KVFS: Key-Value File System

In order to demonstrate how an application can use Aerie's facilities to improve performance, we designed KVFS to provide a (i) simple storage model and (ii) a key-value store interface targeting applications that store many small files in a single directory, such as an email client or wiki software. Clients have a shared consistent view to files through a flat key-based namespace and access files through a simple put/get/erase interface. In addition, all files have the same permissions. In contrast to PXFS, KVFS does not support POSIX semantics, such as a hierarchical namespace and unlinking open files.

KVFS files are implemented with a contiguous, fixed-size region holding the entire file contents, thus placing an upper bound on the supported file size. The file objects store no other metadata, such as permissions or access time. The file system does not have a hierarchical namespace, so all files are stored in a single hash table that maps file names to file objects. Thus, KVFS and PXFS use the *same memory layout* and differ in the policies the interface layer uses to allocate and synchronize data.

We enable scalable concurrent access to the flat key-based namespace through hierarchical locks. A single lock covers the whole hash table and multiple locks under the single lock cover the memory pages that comprise the hash table. Each page's lock also covers the files linked from the key-value pairs stored in the page.

Finally, the get/put interface opens a file and accesses the data in a single operation, which removes the need to maintain state about open files in memory.

| Benchmark | Latency ($\mu s$) | | |
|---|---|---|---|
| | **RamFS** | **ext3** | **PXFS** |
| Sequential read | 0.77 | 0.83 | 0.7 |
| Sequential write | 2.1 | 1.9 | 1.5 |
| Random read | 1.2 | 4.5 | 1.2 |
| Random write | 1.4 | 3.8 | 1.6 |
| Open | 2.0 | 4.7 | 3.7 |
| Create | 9.2 | 113.4 | 13.4 |
| Delete | 3.1 | 11.6 | 2.7 |
| Append | 5.4 | 7.5 | 4.0 |

**Table 2:** Latency of common file system operations. All read/write operations use a 4096-byte buffer.

## 6 Preliminary Results

The goal of Aerie is flexibility and performance. We demonstrate performance of our initial prototype with a mix of application workloads and compare against traditional file systems. In addition, we show the benefits of specializing file system design to a workload.

**Methodology.** We performed our experiments on a 2.4GHz Intel Xeon E5645 six-core (twelve thread) machine equipped with 48GB of DRAM running x86-64 Linux 3.2.2 kernel. We emulate SCM using DRAM by adding delays to model SCM's performance and limit our model to the most important aspect of performance: slow writes. We model PCM by accounting for its slower writes relative to DRAM and introducing a 150ns delay after each write and limiting write bandwidth to 4GB/s.

We compare Aerie against two Linux file systems. RamFS uses the VFS page cache and dentry cache as an in-memory FS and does not provide any consistency across crashes; it thus serves as a best-case kernel FS. To compare against file systems that provide crash consistency, we constructed a block-device emulator, *SCM-disk*, and mounted an ext3 FS. In both cases, we add delays to emulate the write performance of SCM.

### 6.1 Microbenchmark Performance

Table 2 shows the latency of common file system operations on PXFS, RamFS, and ext3. The sequential tests operate on a 1GB file in 4KB blocks, and the random workloads randomly access 100MB out of a 1GB file in 4KB blocks. Open/create/delete are measured by opening/creating/removing 1024 4KB files. Because Aerie batches updates, we report average latency.

As expected RamFS performs consistently better than ext3 except for sequential write. Writes in RamFS are performed directly to SCM whereas in ext3 they are staged in RAM. PXFS performs close to RamFS for all operations but `create` and `open`, where PXFS latency is 45% and 85% higher respectively. Opening a file takes longer for PXFS because pathname resolution walks the persistent directory structure for each path component,

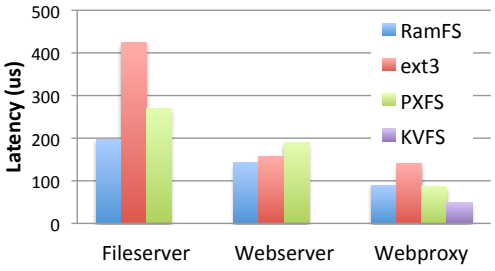


**Figure 2: Average latency to complete one workload iteration.**

and creates a shadow object for POSIX semantics. Of the 3.7$\mu$s to complete an open call, 0.85$\mu$s is spent in lookup and 1.5$\mu$s in creating a shadow. In constrast, RamFS already has the objects in memory so it only pays the overhead of looking up directory entries in the dentry cache, which is highly optimized for lookups.

Compared with ext3 in the kernel, PXFS is between 15% to 90% faster (average 35%) for all operations. Open is faster for PXFS because ext3 has to bring the file into the inode cache. PXFS benefits by not calling into the kernel, which helps all writes and random reads.

### 6.2 Application Workload Performance

Figure 2 shows the average latency to complete one workload iteration for three FileBench workloads: *Fileserver*, *Webserver*, and *Webproxy*. The Fileserver workload emulates file-server activity and performs sequences of creates, deletes, appends, reads, and writes. The Webserver workload performs sequences of open/read/close on multiple files and appends to a log file. Webproxy performs sequences of create/write/close, open/read/close, and delete operations on multiple files in a single directory plus appends to a log file. Each workload is broken up into individual iterations, and we report the latency of an iteration. The Fileserver and Webserver benchmark use 10,000 files, mean directory width of 20, and a 1MB I/O size. The mean file size was 128KB for the Fileserver and 16KB for the Webserver. The Webproxy benchmark was run with 1000 files, mean directory width of 1500, mean file size of 16KB, and 1MB I/O size.

Compared to RamFS, PXFS is 35% slower for Fileserver and Webserver, but matches the performance for Webproxy. Compared to ext3, which provides crash consistency, PXFS achieves 36% and 39% lower latency for the Fileserver and Webproxy workloads. Surprisingly, PXFS performs poorly compared to RamFS and ext3 for the Webserver workload. Since the workload is a read-mostly workload (open/read/close), we expected that direct access will greatly benefit this workload. The reason for low performance is that the workload is dominated by the cost to open a file, which for PXFS is higher compared to RamFS and ext3. This happens due to the lack of caching, which forces PXFS to re-walk persistent structures on every access. In particular, PXFS spends 68% of the total workload time in re-walking persistent directories during path-name traversal. For this reason, we are currently exploring adding a path-name cache to PXFS to reclaim much of the performance difference.

### 6.3 Workload-Specific Performance

A key motivator for direct access is the ability to create workload-specific file system interfaces. We compare the performance of KVFS with a get/put interface in a single directory on the Webproxy workload, whose usage fits the KVFS interface. Figure 2 shows the performance of KVFS for the Webproxy workload. KVFS is 86% faster than RamFS and 79% faster than PXFS. The biggest benefit comes from using a get/put interface instead of open/read/write/close. With the standard interface, PXFS must create a temporary in-memory object representing an open file and record the file offset on every read. With get/put, KVFS can locate the file in memory and copy it directly to an application buffer.

## 7 Conclusion

New storage technologies promise high-speed access to storage directly from user mode. We have presented a decentralized file-system architecture that represents a new design targeting SCM, and reduces the kernel role to just multiplexing physical memory. Applications can achieve high performance by optimizing the file-system interface for application needs without changes to complex kernel code.

Initial results with implementing a POSIX and key-value file-system interface to SCM on top of our architecture have been encouraging. We can still implement POSIX with good performance but the performance improvements really come from customizing the interface to the workload. Moving forward, we plan to further explore interface designs and address the open questions we outlined.

## References

[1] Memory that never forgets: non-volatile DIMMs hit the market. ArsTechnica, Apr. 2013.

[2] P. R. Barham. A fresh approach to file system quality of service. In *NOSSDAV*, May 1997.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *OSDI 9*, Oct. 2010.

[4] 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 *ASPLOS 17*, Mar. 2012.

[5] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making

persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS 16*, Mar. 2011.

[6] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP 22*, Oct. 2009.

[7] Dovecot. Mailbox formats. `http://wiki.dovecot.org/MailboxFormat/`.

[8] L. A. Eisner, T. Mollov, and S. Swanson. Quill: Exploiting fast non-volatile memory by transparently bypassing the file system. Technical report, UCSD, 2013.

[9] Everpsin. Everspin debuts first spin-torque mram for high performance storage systems. `http://www.everspin.com/PDF/ST-MRAM_Press_Release.pdf`, Nov. 2012.

[10] R. F. Freitas and W. W. Wilcke. Storage-class memory: the next storage system technology. *IBM Journal of Research and Development*, 52(4):439–447, 2008.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP 19*, Oct. 2003.

[12] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: a new programming interface for scalable network i/o. In *OSDI*, 2012.

[13] Y. Huai. Spin-transfer torque mram (STT-MRAM): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, Dec. 2008.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP 16*, Oct. 1997.

[15] D. Mazières. A toolkit for user-level file systems. In *USENIX ATC*, June 2001.

[16] Micron. Mobile LPDDR2-PCM. `http://www.micron.com/products/multichip-packages/pcm-based-mcp`, July 2012.

[17] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. PhD thesis, Massachusetts Institute of Technology, 1972.

[18] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *OSDI 9*, Oct. 2010.

[19] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, 2008.

[20] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST 9*, Feb. 2011.

[21] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS 16*, Mar. 2011.