DOI: 10.3969/j. issn. 1673-5188. 2017. 01. 006
http://kns.cnki.net/kcms/detail/34.1294.TN.20170227.1114.004.html, published online February 27, 2017

A Survey of System Software Techniques for Emerging NVMs

BAI Tongxin[1], DONG Zhenjiang[2], CAI Manyi[1], FAN Xiaopeng[1], XU Chengzhong[1], and LIU Lixia[2]
(1. Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China;
2. ZTE Corporation, Nanjing 210012, China)

Abstract

The challenges of power consumption and memory capacity of computers have driven rapid development on non-volatile memories (NVM). NVMs are generally faster than traditional secondary storage devices, write persistently and many offer byte addressing capability. Despite these appealing features, NVMs are difficult to manage and program, which makes it hard to use them as a drop-in replacement for dynamic random-access memory (DRAM). Instead, a majority of modern systems use NVMs through the IO and the file system abstractions. Hiding NVMs under these interfaces poses challenges on how to exploit the new hardware's performance potential in the existing system software framework. In this article, we survey the key technical issues arisen in this area and introduce several recently developed systems each of which offers novel solutions around these issues.

Keywords

non-volatile memory; persistent memory; file system; IO system

1 Introduction

Ever since the Internet and mobile computing dominated people's daily life, the continuing supply of big data, which relies on immense computing power to extract the hidden big values, has demanded higher speed of data storage. The big data trend challenges the design of computer systems, on both hardware and software, to sustain the development of new data intensive applications. For example, deep data analytics and in-memory computing demand shorter turn-around time between processing iterations, which translates to faster data transportation between storage and processors. As of its current stage, in-memory computing uses dynamic random-access memory (DRAM) as the main media for hot data storage given its advantage in speed and bandwidth. However, DRAM would soon hit energy wall when the total memory capacity keeps growing in a data center. According to a recent study, 100 petabytes main memory with DDR 3 DRAM would consume 52MW power, which is far beyond the energy budget for building a future exascale data center [1], [2].

The combined requirements on capacity, performance and power have motivated both industry and academia to pursue new technologies and build alternative memory devices to bridge the gap between fast DRAM and slow disks. In recent years, semiconductor manufacturers have invested heavily on non-volatile memory (NVM) devices. As the most mature NVM in its class, Flash is already widely used in commercial servers due to its high density and low static power consumption. However, Flash has its notable downside. Memory wear and block erasure make it ill-fit for random read and write for which DRAM has outstanding performance [3]. Alternative NVM technologies have advanced rapidly, each expected to improve upon some or all of the Flash weaknesses. Among the most influential new types of NVMs are Phase Change Memory(PCM), Spin Transfer Torque RAM(STT - RAM), Resistive RAM (RRAM), Racetrack Memory, and Domain Wall Memory (DWM). Most recently, Intel and Micron jointly announced 3D XPoint and claimed the new NVM delivers a performance of 1000 times shorter latency and longer endurance than of the conventional Flash. The density and performance boost is said to be enabled by a novel structure of memory cell with a stackable data access array.

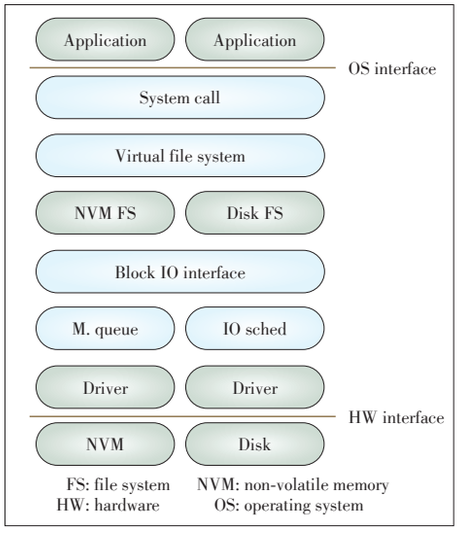
Other than higher read-write performance, most of the new NVMs support some extent of byte-addressable random access. The fine-grained access capability would revolutionize the way data is communicated between on-core and off-core memory. Potentially, CPU could directly address the data on the secondary storage without first sending instructions to a device controller. A secondary storage with byte-addressing capability is usually called Storage Class Memory (SCM). Once SCM is widely deployed, changes must be applied to the IO interface as well as the file system for they are traditionally optimized for slow devices. The fast storage would make many optimizations less effective, or even harmful, and favor a simpler design of the software stack for better performance.

The promising future of new hardware motivates software innovations which promote the lower level improvement to higher level usability. In this article we survey and introduce recent advances on how the NVM adaptation is addressed in sys-

tem software, particularly in IO and file system.

2 IO Subsystem for NVM

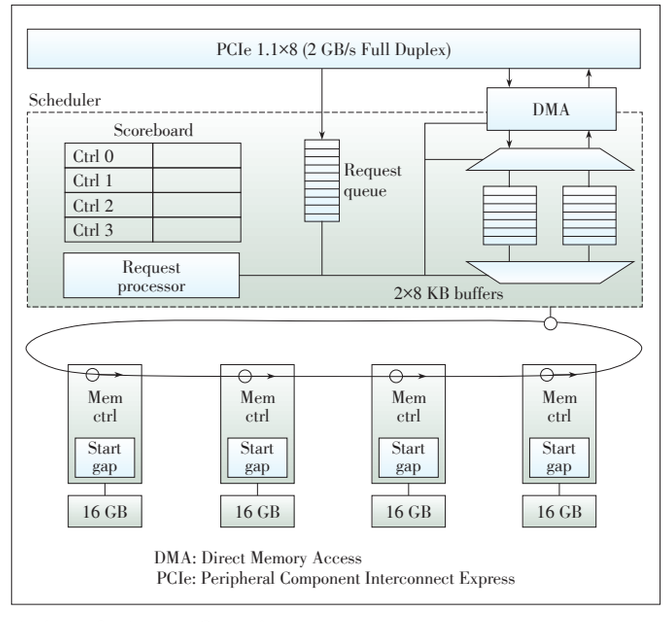
Fig. 1 summarizes a standard hierarchy of system components in which NVM devices shared the same IO interface with other block devices. The usual procedure of reading and writing data on a block device begins with a user program issuing a system call with arguments specifying the location and size into the target file in the file system. After the program is trapped to the kernel mode, the system call request is transitioned through multiple layers of kernel components, including the file system and the block IO interface, until it's finally translated into a sequence of low level instructions to the device driver. Since accessing block devices can pose a long latency, the IO requests are usually processed in an asynchronous way, leveraging Direct Memory Access (DMA) and interrupt handling to complete the data transfer while saving a large amount of CPU time. The software overhead during the request processing is caused by program state transition, file system management, IO scheduling, interrupt handling, buffer cache management and so on. Studies show that the software latency for processing blocked IO request is in the 10 microseconds on modern Intel processors running Linux operating system (OS). In reality, the exact software overhead varies for specific systems. Swanson et al. [4] measured that a single 512 byte IO request incurred about 19 microseconds of software overhead on Intel Nehalem 2.27GHz processor. In the test conducted by Yang et al. [5], a 512 byte IO operation cost 5 to 7 microseconds in software on 2.93GHz Intel Xeon processor. Compared to tens of milliseconds of latency due to disk access, the relative cost of software is very small. However, with NVM storage taking place, the relative cost of software increases significantly. The modern Flash solid-state drive (SSD) offers read-write latency of tens of microseconds. It's projected that newer generations of NVM can further reduce the state-of-the-art by 10 times [6]. With device latencygreatly reduced, the originally small software and interface overhead will dominate the cost of an IO operation. Therefore, in the future NVM systems, software and interface optimizations are the key to better storage performance. In the following we will introduce a number of recent developments on high performance IO interface and processing techniques for NVM storage.



◀**Figure 1. IO system software.**

2.1 Moneta

Caulfield et al. proposed Moneta [7] in 2010, an experimental NVM interface framework. Based on the simulations of Phase - Change Memory (PCM), the experiment results show that the Moneta interface helps random read and write performance with an increment of 18 times than that of the baseline. The software overhead is reduced by 60%. 4KB random read and write throughput can be maintained at the level of the 450k IOPs. Moneta is based on a design comprised of a token ring network and memory controller array, which improves the IO rate by exploiting the latency and parallelism advantage of the hardware. The detailed structure of Moneta IO is shown in **Fig. 2**. The IO scheduler is responsible for coordinating the data transmission and request scheduling. Data is transferred between memory and device through the PCIe interface. The requests are exchanged in the form of a command via a token ring network connecting the NVM memory controller and the request queue. When the system is running, the driver software issues IO requests which are transmitted via the Peripheral Component Interconnect Express (PCIe) interface to the Moneta scheduler and then are inserted into a first - in first - out (FIFO) queue.Requests larger than 8KB need to be decomposed and transmitted in sequence. To streamline data transfer, each Moneta memory controller is equipped with two 8KB



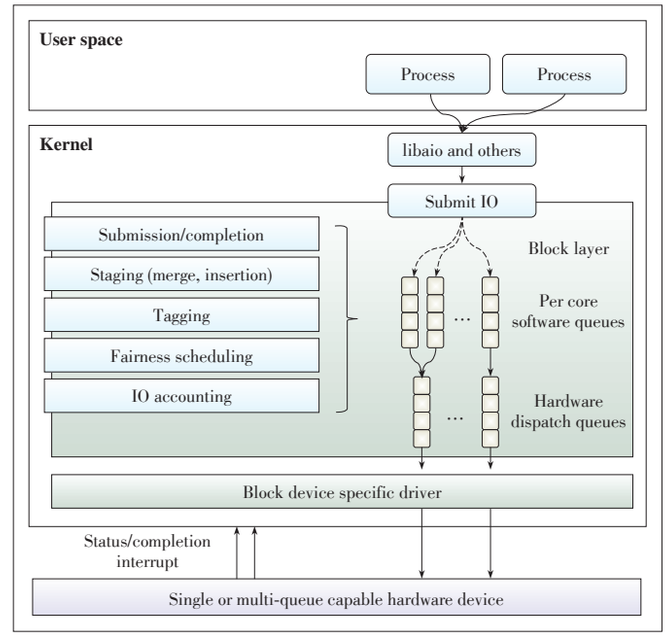
▲**Figure 2. Moneta IO architecture.**

buffers, used for caching the data read from and written to the storage after the requests are successfully processed. To make full use of the 2GB bandwidth of PCIe, it is not enough to expand the capacity of the device interface. As Caulfield et al. [7] have tried, a series of kernel optimization are applied to improve the efficiency of request processing in CPU. These software approaches include 1) avoiding the default IO scheduler, 2) using the Moneta built-in atomic read-write operations in place of locked kernel IO operations, 3) allowing multiple threads to process device interrupts in parallel, and 4) using spinlocks to avoid unnecessary context switch overhead caused by interrupts. Combining the above methods, Moneta reduces the IO access latency to around 1μs, achieving a significant improvement relative to the baseline latency of 10 μs. Correspondingly, the effective bandwidth increases to hundreds of mega bytes per second. Compared to traditional IO, Moneta excels with its comprehensive overhead reduction on interrupts, synchronization and scheduling.

For evaluation, the Moneta prototype was tested against a set of database applications. One interesting observation is that traditional database's optimization against disk storage can produce counter effect on Moneta's own optimization for NVM. In particular, PostgresSQL and MySQL receive less performance gainfrom the new framework than the simpler BerkeleyDB does.

2.2 Linux Multiqueue Block IO

The low latency and good parallelism of NVM storage would be underutilized features if the original Linux IO subsystem remains using a single request queue, which would easily become a performance bottleneck when the IO request rate approaches million per second. In order to solve the scalability problem in an NVM based system, Linux adopts a new block device interface blk-mq starting from kernel version 3.13. The internal structure of blk-mq is shown in **Fig. 3**. In the new IO framework, each IO request is processed in two phases separately. When an IO request arrives at the kernel through a system call, it is pushed onto a software staging queue which is dedicated to the CPU core on which the working thread is running. The request stays in the software queue while the kernel applies scheduling logics; then it's transmitted to a hardware dispatch queue waiting for the hardware to be ready to process it. To achieve high concurrency, a single storage device can be configured with multiple dispatch queues to better utilize the parallel processing capability of CPUs and in-band signaled interrupts of devices. In fact, the hardware queues can be allocated as many as several thousands, a number determined by how many virtual context a device can support. For example, a device supporting MSI-X can allocate 2048 queues for it can register 2048 interrupts. This new design of the IO subsystem promotes a fast and localized IOrequest processing scheme, especially by reducing unnecessary remote memory accesses in an NUMA environment.



▲**Figure 3. Internal structure of Linux blk-mq.**

The blk-mq block device interface successfully separates the software scheduling and the hardware message buffering functionalities which used to share a single request queue. The separation reduces synchronization and buffer congestion and hence leads to a much improved IO scalability.

2.3 Poll or Interrupt?

Despite that a redesign of IO subsystem internal structure fundamentally improves the IO scalability, there remain other system software overheads affecting IO performance. Most noticeably, the overhead comes from interrupt handling and context switch. In a typical OS setting, after an IO request is submitted to the kernel waiting to be processed, the calling thread returns or simply blocks for response. The completion of the request starts with a device interrupt notifying the CPU that the data is ready. After the device driver picks up the interrupt and finishes the handling procedure it will notify the IO subsystem which then will complete the remaining work and wake up the suspended thread. This asynchronous style of IO operation saves valuable CPU time since the program waiting for response can yield the CPU temporarily for other programs to use. In theory, however, the benefit of asynchronous IO only exists if the hardware latency is larger than the combined software overhead. When the device latency is reduced to microsecond level, the benefit will diminish.

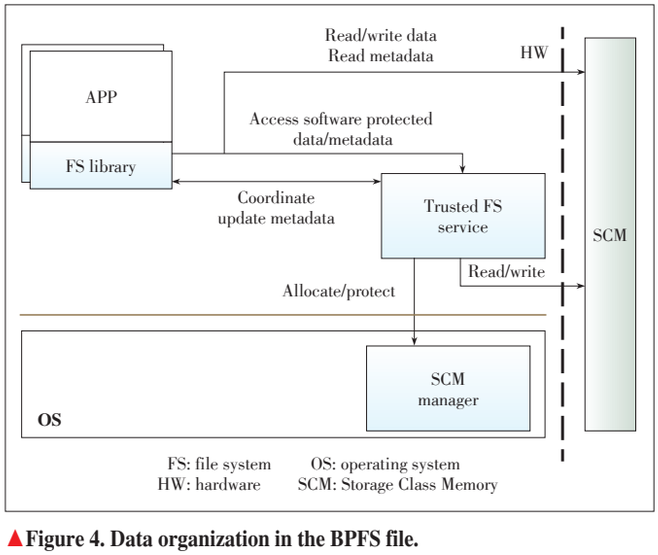
Polling provides a lighter weight means for checking device status than waiting for interrupt. To find out which is better with high performance NVM, Yang et al. compared the throughput and latency results using a simulated environment [8]. The study shows that synchronous completion requires shorter time and induces a better CPU utilization when that de-

vice latency is as low as several microseconds. Another interesting observation is, in the synchronous mode, a better hardware performance can reduce the software cost, whereas there is no such benefit for asynchronous IO. Furthermore, by studying throughput scalability, based on the case of 512 byte random reads, the study finds that the throughput of synchronous IO scales linearly with increased number CPUs. In contrast, asynchronous IO can only achieve 60%–70% throughput of the synchronous IO. Because asynchronous IO is suitable for processing long wait, when the system has a complex device setup, it can be suggested that the IO request be processed by a mixed mode of synchronous and asynchronous IO, achieving a load balance between CPU and the device.

2.4 NVM Express

NVM Express (NVMe) [9], [10] is a new software interface specification for accessing NVM devices attached to the PCIe bus. A working group on NVMe was formed in 2007. Technical work on the specification started in 2007 and the first release was finished in 2011. NVMe was designed from the ground up as an open device interface to exploit the low latency and parallelism of the future NVM devices, leading to an increased capacity of data path between CPU and storage. As a key design goal, the NVMe interface allows the processing power to fully utilize the internal parallelism available in the NVM devices and the bandwidth of the PCIe bus, hence effectively improving the IO performance. An example IO subsystem supporting NVMe is shown in **Fig. 4**.

NVMe can support up to 65,536 request queues, with request submission and completion stages allocating on different sets of queues (SQ and CQ).Separating the two stages reduces the likelihood of IO congestion, an issue often raised when using a single request queue in dealing with a large number of IO requests. The actual IO operations over the NVMe interface involve the software and the device exchanging commands and data on a dedicated memory mapped area in the host program's address space. Moreover, an NVMe device can support up to 2048 virtual contexts. With highly scalable multi-queue based IO scheduling, the NVMe interface supports a high throughput and concurrent data path between CPU and storage.



▲**Figure 4. Data organization in the BPFS file.**

The ecology of NVMe-based systems is collaboratively built by the device manufacturers, chipsets providers as well as software venders. The first NVMe drives came from Samsung [11], LSI [12], Kingston [13] and Intel [14]. Early operating systems supported include Linux 3.3, Windows 8.1 and Windows Server 2012 R2. On chipsets, the major venders have already added NVMe support in their new product lines. To further promote software development using NVMe features, Intel announced the Storage Performance Development Kit (SPDK), a development toolkit providing the user level and poll based IO programming interface.

Performance studies on actual NVMe systems have recently been carried out by a group of researchers. Xu et al. measured and analyzed the performance differences of several database systems using NVMe SSD and Serial Advanced Technology Attachment (SATA) SSD on real machines [15]. The experiment results show that, compared to SATA SSD, the software overhead of NVMe SSD is reduced from 25% to 7% meanwhile 4 KB read throughput is increased from 70k IOPS to 750k IOPS. NVMe helps bring about 8 times performance gain in the tested database applications. This study is a solid proof that a redesigned device interface is necessary for exploiting the potential of new storage hardware. The performance demonstration also assures the system designers that NVM storage is ready to be a major investment for boosting overall service performance.

3 File Systems for NVM

A file system provides a data persistence service over a named space organized in directories. It strives to meet two practical goals: to maintain a consistent view of the data and to guarantee persisted data are reliably stored. In order to ensure the reliability and consistency, the file system needs careful organization of data layout as well as correct implementation of the operation semantics.

In a file system, stored objects are comprised of data and metadata, both of which may be accessed and modified when a single file operation is involved. The latency of accessing a conventional storage device can be long and unstable, which is a major concern of modern file system optimization. Techniques such as grouping metadata based on accessing pattern can improve data locality, leading to a better use of buffer cache. For reliability and consistency purposes, journaling and block level copy - on - write are typically employed to guard against system crash and power outage. These techniques themselves bring up additional operation overhead which may contribute to severe performance degradation. When a file sys-

tem is migrated to an NVM-based system, the change of storage structure can incur a host of new issues meanwhile eclipse the effect of existing optimizations. For example, fast NVMs make data prefetching and caching no longer a key mechanism for latency reduction. Moreover, the software overhead introduced by prefetching and caching can be significant in the setting of new systems.

New NVM storage exposes challenges as well as opportunities to modern file systems. New software structures and optimizations have been proposed and evaluated in several experimental systems. In this section we survey five state-of-the-art NVM file systems, outlining the new techniques around key issues developed in these systems.

3.1 File Operation Interface

Modern file systems usually offer two types of interfaces for reading and writing a file, the standard read() and write() system calls and then map the application programming interface (API). Realizing these two interfaces is affected by which particular hardware interface the NVM device is stalled on. If the device is installed on PCIe bus through NVMe interface, the legacy file system will function well as long the hardware interface is supported. Moreover, any IO system optimization against NVMe will benefit the file system as well. In this setting, data copying between the device and memory is necessary to realize both the read/write and the mmap APIs. In contrast, if the NVM is attached to the system through the memory bus, mmap would not involve extra copying, rendering faster external data operations.

The Byte-Addressable Persistent File System (BPFS) [16] is an NVM file system for the Windows operating system. Implemented using Windows Driver Model, BPFS provides users with the standard file operation interface. In the core, BPFS maintains an independent physical space for the file system, separated from the physical space of the user processes. Read and write operations involve data copying between the two physical spaces. Since the hardware is fast, buffer cache is no longer needed and the reduced software complexity helps improve the performance as a consequence. As an experimental file system, BPFS does not support mmap.

The Storage Class Memory File System (SCMFS) [17], [18] is an NVM file system developed for Linux, providing a compatible interface with common file systems in Linux. SCMFS leverages the processor's VMM features and simplifies the design of the file system to reduce the software overhead.

The Persistent Memory File System (PMFS) [19] is another example file system that exploits the processor's paging and memory ordering mechanisms to reduce the software overhead. PMFS provides both read/write API and mmap interface.

Aerie [20] is more of a file system framework than a single file system. It can be extended and customized based on particular application's requirements. As a framework, Aerie provides a flexible interface for the higher level software to work with. It can be used for implementinga POSIX compatible file system as well as building a user level library allowing applications to access files without going through the OS kernel.

The Non-Volatile Memory Accelerated (NOVA) file system [21] is log structured and designed for DRAM/NVM hybrid memory systems. NOVA is designed with full account of the device's byte addressability as well as the concurrency available in modern multicore systems. Implemented in the Linux kernel, NOVA supports both standard file operations and mmap APIs.

3.2 Internal Organizations and Management

Externally, files and directories are the main objects that a file system manages. Internally, the data in a file system are arranged in a collection of inodes, data files and logs. On the storage level, a data file can be laid out either in a sequence of extents, each of which being a consecutive run of blocks, or in a sequence of indirectly linked blocks [15]. Some data structures specific to file systems are crucial components for correctness and reliability purposes. For example, to guarantee crash consistency, a file system often uses journals to log uncommitted changes. Another widely used structure is the copy-on-write log tree, which provides a foundation for atomic updates of large data blocks.

As mentioned in previous sections, NVM has changed the cost ratio of software and hardware. As a result, the structures and algorithms in the conventional file systems, which were optimized against slow storages, do not work well for the fast NVM devices. To fully utilize the new hardware features, changes must be applied in software, potentially modifying the basic internal structures in the file systems. In the following, we highlight the key techniques and strategies for improved NVM usage introduced in the aforementioned file systems.

BPFS uses shadow paging to ensure reliable data update. Internally, the inodes, the catalog files and the data files are all stored in pages that are organized into a tree structure, as shown in Fig. 4. Unlike standard shadow paging that uses page level copy-on-write, the fine-grained access to NVM allows BPFS to manipulate data on a subpage level, a technical improvement that supports in-place modification of small data and partial copy-on-write, both of which can reduce the chance of page copying. For sake of performance, BPFS retains certain dynamic data structures in DRAM, including the storage management data structures and directory cache, to help speed up metadata querying speed.

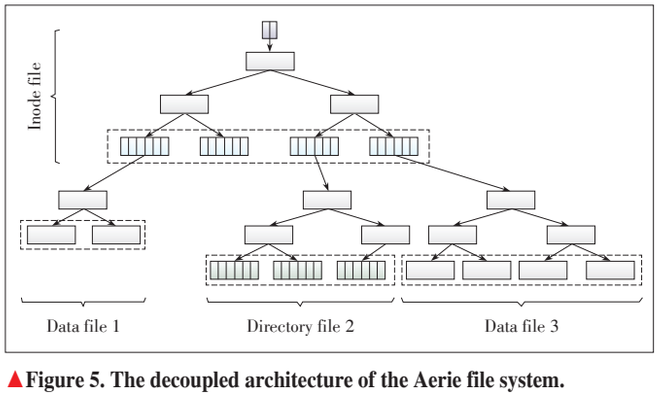
SCMFS leverages the existing virtual memory management (VMM) mechanisms provided by the OS and the hardware to simply the storage management of the new NVM file system.In SCMFS, the metadata and the address mapping table are stored in the physical address space, whereas the inodes, the catalog files and the data files are all mapped into virtual address space. In order to expedite the storage allocation and reclamation, SCMFS pre-allocates plenty of null files so that when

creating a new file it firstly looks for a suitable null file, and when deleting files it only marks them as null files. When the total size of null files is too large, garbage collection is triggered to recycle the storage of null files.

PMFS has the entire file system in the kernel address space. Under this arrangement, programs use the Direct Access (DAX) mechanism of Linux to access files, bypassing the buffer cache and incurring at most one copying between user space and kernel space for every piece of data. Moreover, zero-copy access is made possible to the memory mapped files whose user addresses are directly mapped to their in-kernel storage. Internally, PMFS organizes file storage based on B-tree structure, with 4KB, 2MB and 1GB as units of blocks. PMFS uses logs for basic consistency purpose.

Aerie decouples normal read and write operations from the management of the file system to reduce software overhead of directory lookups, metadata querying, synchronization operations and so on. Thus, different types of services can be assigned to different components which communicate and cooperate through Remote Procedure Call (RPC) as well as distributed lock service. As shown in **Fig. 5**, Aerie's distributed service architecture includes three major components: the storage manager in the OS kernel, the Trusted FS Service (TFS) and the FS Library (libFS). The storage manager is responsible for core functions and services that require privileged operations, such as allocating storage space for users, mapping the address space of the files, and modifying access permissions. TFS provides users with metadata modification, concurrency control and other critical services without special hardware support. TFS runs in an independent process and accepts RPC requests from user programs. LibFS provides ordinary file read-write operations and read-only operations of metadata. When privileged functions are required, LibFS will send RPC requests to the TFS. Aerie adopts an extent based multi-layerstructure for storage management. Seeking in a file is done by mapping the offset to a certain extent using a multi-level index. Moreover, user program scanning modifies an extent directly without going through the TFS.

NOVA augments the basic design of a log structured file system with features optimized against NVM. Based on the observation that logging is fast with NVM yet search is slow, NOVA builds an index in DRAM in addition to the logs in NVM to accelerate search operations. The traditional log structured file system suffers from the complexity of garbage collecting released logs into contiguous free regions. In NVM, random access is cheap so supplying a large contiguous region for logging is no longer necessary. In NOVA, logs are stored as linked lists so they don't need to be allocated in contiguous memory. Logs are chained up under individual inodes, which allows for high concurrency during access and recovery.



▲**Figure 5. The decoupled architecture of the Aerie file system.**

3.3 Consistency and Atomicity Maintenance

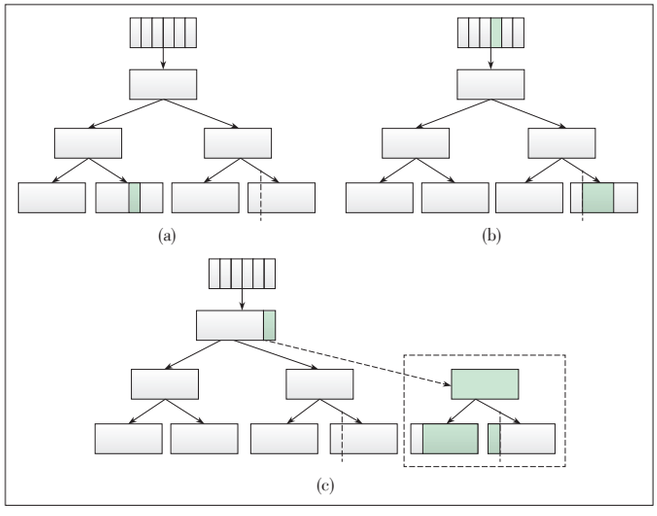
Journaling and shadow paging are techniques commonly used in file systems to achieve crash consistency. However, when implementing these techniques in an NVM system following the traditional way, performance issues may arise.

1) Issues with journaling. When journaling is enabled in a file system, write operations are amplified since every update requires writing into the storage twice, one to add an entry to the log, and the other to commit the change in the file. In terms of performance, the sequential characteristic of log appending is very favorable in the case of disk storage, but the new NVMs support fast random access, which significantly dampens the performance benefit of sequential logging.
2) Issues with shadow paging. With copy-on-write, an update to a logical page by the user program needs to be written to another free page, and only when changes are committed, the reference pointing to different methods for the old page will be replaced to ensure the atomicity of the modification. Since file systems usually organize internal storage in tree-like structures, a page getting modified implies its parent must be modified too. This may lead to a chain effect that an update on a single page triggers a series of page copying, causing severe write amplification.

Compared to traditional storage, NVM is fast and suitable for random access. The following outlines the solutions to the above issues, which take account of the hardware advantage.

BPFS proposes a technique called short-circuit shadow paging for updating persistent data. The new shadow paging scheme consists of three methods, in-place updates, in-place appends, and partial copy-on-write (**Fig. 6**). In-place updates can be applied for writes of 8 bytes or less, using hardware supported primitives. In-place appends refers to writing to the free area immediately beyond the file's end point. Since all the data beyond the file size is ignored, in-place writing to these locations is safe and once the writing is complete the file size is updated atomically. Partial copy-on-write allows atomic updates spanning multiple pages. The page copying only propagates to a point in which a single write suffices to commit the entire change.

In PMFS [19], based on the comparison study over the costs of different consistency techniques, the authors found that journaling at 64 byte granularity was most efficient for metadata



▲Figure 6. Three approaches to updating a file in BPFS: (a) in-place updates, (b) in-place appends, and (c) partial copy-on-write.

updates while it was less desirable than copy-on-write for large updates. Based on this observation, PMFS follows a hybrid strategy for consistency where in-place updates and fine-grained logging are used for metadata updates, and copy-on-write for file data updates. The metadata log update could be implemented by two different methods, namely undo logging and redo logging. In redo logging, the new data is firstly logged before committing to the file system. In undo logging, the old data is firstly logged before the new data is written in-place. In case of a failure, the system can be rolled back using the old data in the undo log. On the one hand, undo logging is costly for writes since a write barrier is required for every log entry in undo logging whereas only one write barrier is needed for a transaction in redo logging. On the other hand, redo logging is costly for reads and more difficult to implement since all the reads within a redo logging transaction have to search the redo log for the latest copy before reading from the file system. PMFS uses undo logging for metadata journaling.

Aerie needs to maintain the consistency of data updates in its particular distributed framework. If each metadata update requires one round RPC request from user program to the TFS, it's bound to hurt the service scalability. To solve this problem, Aerie applies an optimization that the client buffers the requests before periodically sending them to the TFS. In TFS, Aerie uses redo logging to realize atomic metadata updates.

NOVA modifies small metadata atomically in place. For single inode updates, NOVA relies on logs to record the changes. Modifications across multiple inodes resort to lightweight journaling to guarantee atomicity.

3.4 Hardware Primitives for Persistent Memory

Modern processors maintain consistency of memory operations by following certain a memory model. Without affecting the correctness of the program, memory requests could be delivered out of order after they are scheduled and buffered in the memory controller. Even the ordering enforcing instructions only guarantee the memory operations are properly ordered in the processor, disregarding in what order the data updates actually reach the memory. Unlike memory consistency, a file system is strict about when the data updates are safely stored. As previously mentioned, the file system consistency is realized by techniques such as journaling and copy-on-write, mostly implemented in software. For NVM, particularly byte-addressable NVM, pure software approaches to achieve consistency incur large overhead. To reduce the cost, new hardware primitives have been proposed and exploited.

BPFS proposes a write barrier instruction. Using the barriers, a program execution breaks down into a sequence of epochs. The order of persistent memory operations across epochs is strictly maintained. In addition to barriers, a file system needs atomic data updates to help crash recovery. Traditional file systems could verify atomicity by computing checksums. Leveraging hardware features, BPFS proposes a new atomic write primitive for small data updates. It is shown in a related research that such a light weight atomic operation requires merely 300 nanojoules reserved in the capacitor. With the new primitive, all data updates less than 8 bytes could be done in place.

PMFS uses atomic write instructions for modifying data of 8 bytes, 16 bytes and 64 bytes. The following scenarios explain when to apply the atomic instructions: 1) when reading a file, update the access time in the inode with the 8 byte atomic write instruction; 2) when appending to a file, use the 16 byte atomic write to update the size and access time in the inode; and 3) if Restricted Transactional Memory (RTM) [22] is available, use the RTM transactions for atomic updates within a cache line.

Aerie relies on the atomic instructions available in the x86 instruction set to realize three basic atomic primitives: 1) wlflush, which is implemented with the x86 clflush instruction, writes back the entire cache line; 2) bflush, which relies on the x86 mfence instruction, writes back the entire cache in processor to the storage, and 3) fence, which also uses the mfence instruction, enforces orderly writebacks. Based on these hardware primitives, Aerie manages a redo log for metadata updates.

However, it is worth to note that both clflush and mfence have limitations regarding memory writes to NVM. Clflush only flushes the cache line to the memory controller; it is left unknown whether the write eventually reaches the memory. Mfence only guarantees write orders are consistently recorded across CPUs; it has no constraints on the order of arrivals to the memory.

NOVA enforces write ordering upon memory operations by using a set of newly developed x86 instructions that have been proposed to tackle the above issues. These instructions include clflushopt (a more efficient version of clflush), clwb (cache line

write back without invalidation) and PCOMMIT (commit writes to NVM).

4 Conclusions

In recent years, the research on NVM and its software support has been a hot topic in computer systems area. The key advantage of NVM is its capacity to simultaneously achieve high density, low latency and low energy consumption. Hence it can potentially solve the energy scalability issues of large scale computer systems. The current NVM platform technologies, from the device interface to the software support, are not yet fully developed, leaving numerous challenges to be solved. From the software perspective, the most challenging issues arise in several areas, including IO optimization, memory management, file system as well as programming abstraction. To tackle these problems, researchers have explored novel ideas which involve restructuring the system software internals. In this article, we sampled a number of representative results in these areas and believe that new software techniques will emerge in response to the hardware's changing landscape in the future.

References

[1] B. Giridhar, M. Cieslak, D. Duggal, et al., "Exploring DRAM organizations for energy-efficient and resilient exascale memories," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, Denver, USA, 2013, pp. 23:1–23:12. doi: 10.1145/2503210.2503215.
[2] Joel Hruska. (2014, Jul. 14). *Forget Moore's law: hot and slow DRAM is a major roadblock to exascale and beyond* [Online]. Available: http://www.extremetech.com/computing/185797-forget-moores-law-hot-and-slow-dram-is-a-major-roadblock-to-exascale-and-beyond
[3] Wikipedia. (2016, Dec. 19). *Memory* [Online]. Available: https://en.wikipedia.org/wiki/Flash
[4] S. Swanson, and A. M. Caulfield, "Refactor, reduce, recycle: restructuring the IO stack for the future of storage," *Computer*, vol. 46, no. 8, pp. 52–59, Aug. 2013.
[5] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Usenix Conference on File and Storage Technologies*, San Jose, USA, Feb. 2012, pp. 3–3.
[6] Wikipedia. (2016, Oct. 18). *XPoint* [Online]. Available: https://en.wikipedia.org/wiki/3D
[7] A. M. Caulfield, A. De, J. Coburn, et al, "Moneta: a high-performance storage array architecture for next-generation, non-volatile memories," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, USA, Dec. 2010, pp. 385–395.doi: 10.1109/MICRO.2010.33.
[8] A. Huffman. (2012). *NVM express revision 1.1* [Online]. Available: http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf
[9] NVM Express. (2013, Apr.). *NVM express explained* [Online]. Available: http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf
[10] Josh Linden. (2013, Jul. 18). *Samsung announces industry's first 2.5-inch NVMe SSD* [Online]. Available: http://www.storagereview.com/samsung_announces_industry_s_first_25inch_nvme_ssd
[11] Lyle Smith. (2013, Nov. 18). *LSI SF3700 SandForce flash controller line unveiled* [Online]. Available: http://www.storagereview.com/lsi_sf3700_sandforce_flash_controller_line_unveiled
[12] Dave Altavilla. (2014, Jan. 13). *Kingston hyper X predator PCI express SSD unveiled with LSI SandForce SF3700 PCIe flash controller* [Online]. Available: http://hothardware.com/news/kingston-hyperx-predator-pci-express-ssd-unveiled--with-lsi-sandforce-sf3700-flash-controller
[13] Intel. (2016) *Product comparison* [Online]. Available: http://www.intel.com/content/www/us/en/solid-state-drives/intel-ssd-dc-family-for-pcie.html
[14] J. Condit, E. B. Nightingale, C. Frost, et al, "Better IO through byte-addressable, persistent memory," in *ACM Symposium on Operating Systems Principles*, Big Sky, USA, Oct. 2009, pp. 133–146. doi: 10.1145/1629575.1629589.
[15] Q. Xu, H. Siyamwala, M. Ghosh, et al., "Performance analysis of NVMe SSDs and their implication on real world databases," in *8th ACM International Systems and Storage Conference*, Haifa, Israel, May 2015, article no. 6. doi: 10.1145/2757667.2757684.
[16] X. Wu and A. L. N. Reddy, "SCMFS: a file system for storage class memory," in *Conference on High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, Nov. 2011, article no. 39. doi: 10.1145/2063384.2063436.
[17] X. Wu, S. Qiu, and A. L. Narasimha Reddy, "SCMFS: a file system for storage class memory and its extensions," *ACM Transactions on Storage (TOS)*, vol.9, no.7, Aug. 2013. doi:10.1145/2501620.2501621.
[18] D. S. Rao, S. Kumar, A. Keshavamurthy, et al., "System software for persistent memory," in *the Ninth European Conference on Computer Systems*, Amsterdam, The Netherlands, Apr. 2014, article no. 15. doi: 10.1145/2592798.2592814.
[19] H. Volos, S. Nalli, S. Panneerselvam, et al., "Aerie: flexible file-system interfaces to storage-class memory," in *the Ninth European Conference on Computer Systems*, Amsterdam, The Netherlands, Apr. 2014, article no. 14. doi: 10.1145/2592798.2592810.
[20] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *14th USENIX Conference on File and Storage Technologis (FAST 16)*, Santa Clara, USA, Feb. 2016, pp. 323–338.
[21] R. Arpaci-Dusseau and A. Arpaci-Dusseau. (2015, Mar.). *Operating systems: three easy pieces* [Online]. Available: http://pages.cs.wisc.edu/~remzIOSTEP
[22] Intel Corporation, "Intel64 software developer's manual," vol. 1, ch. 14, 2013.

Manuscript received: 2016-10-14

Biographies

BAI Tongxin (tx.bai@siat.ac.cn) obtained a PhD degree of Computer Science from the University of Rochester, USA. He is an associate professor at the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interest concerns the interplay of programming language technology and big data systems.

DONG Zhenjiang (dong.zhengjiang@zte.com.cn) graduated from Harbin Institute of Technology, China. He is the vice president of Cloud Computing & IT Institute of ZTE Corporation, and the executive director of China Artificial Intelligent Association and of the Standing Committee of CCF on Service Computing. He is responsible for research and development of intelligent networks, customer services, IPTV and cloud computing, and has presided a number of national key R & D projects. His research achievements have been used in dozens of countries. His research interests are big data, AI and network security.

CAI Manyi (my.cai@siat.ac.cn) is currently a master student at the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. Her current research interests include database systems and big data systems.

FAN Xiaopeng (xp.fan@siat.ac.cn) received his B.E. and M.E. degrees in computer science from Xidian University, China in 2001 and 2004 respectively. He received his Ph.D. degree in computer science from Hong Kong Polytechnic University, China in 2010. He is currently an associate professor at the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include big data analytics, mobile cloud computing and software engineering. His recent research has focused on big data analytics in urban computing. He has published more than 30 papers in conferences and journals, and served as a TPC member for several conferences. He is a member of IEEE and ACM.

XU Chengzhong (cz.xu@siat.ac.cn) received his Ph.D. degree from the University of Hong Kong, China in 1993. He is currently a professor of the Department of Electrical and Computer Engineering of Wayne State University, USA. He also holds an adjunct appointment with the Shenzhen Institutes of Advanced Technology of Chinese Academy of Sciences as the Director of the Institute of Advanced Computing and Data Engineering. His research interest is in parallel and distributed systems and cloud computing. He has published more than 200 papers in journals and conferences with more than 7000 citations.

LIU Lixia (liu.lixia@zte.com.cn) received the M.S. degree from Ocean University of China in 2008. She is a senior engineer in the System Architecture Department of ZTE Corporation. Her research interests include text mining, bid data analysis and mining, machine learning, mathematical statistics, and cloud computing.