FSMAC: A File System Metadata Accelerator with Non-Volatile Memory

Jianxi Chen[*Ŧ], Qingsong Wei[*✉], Cheng Chen[*], Lingkun Wu [*]
[*]Data Storage Institute, Agency for Science, Technology and Research, Singapore
[Ŧ] Wuhan National Laboratory for Optoelectronics
[Ŧ] School of Computer, Huazhong University of Science and Technology, China
[✉]Corresponding author: WEI_Qingsong@dsi.a-star.edu.sg
{CHEN_Jianxi, CHEN_Cheng, WU_Lingkun}@dsi.a-star.edu.sg

Abstract— **File system performance is dominated by metadata access because it is small and popular. Metadata is stored as block in the file system. Partial metadata update results in whole block read and write which amplifies disk I/O. Huge performance gap between CPU and disk aggravates this problem.**

In this paper, a file system metadata accelerator (referred as FSMAC) is proposed to optimize metadata access by efficiently exploiting the advantages of Non-volatile Memory (NVM). FSMAC decouples data and metadata I/O path, putting data on disk and metadata on NVM at runtime. Thus, data is accessed in block from I/O bus and metadata is accessed in byte-addressable manner from memory bus. Metadata access is significantly accelerated and metadata I/O is eliminated because metadata in NVM is not flushed back to disk periodically anymore. A light-weight consistency mechanism combining fine-grained versioning and transaction is introduced in the FSMAC. The FSMAC is implemented on the basis of Linux Ext4 file system and intensively evaluated under different workloads. Evaluation results show that the FSMAC accelerates file system up to 49.2 times for synchronized I/O and 7.22 times for asynchronized I/O.

Index Terms— **File system, Non-volatile Memory, Metadata, Consistency**

I. INTRODUCTION

The performance gap between CPU and disk-based storage is a sophisticated challenge for decades. Flash memory is faster than disk and widely used in recent years, but its performance is still too far away from that of CPU to eliminate the gap. As emerging technology, next generation Non-Volatile Memory (NVM) is storage class memory, such as Spin-Transfer Torque MRAM (STT-MRAM), Phase Change Memory (PCM) and Memristor, which combines the features of persistency as disk and byte-addressability as DRAM. NVM can be used as either persistent storage or memory. It is a good candidate to eliminate the I/O bottleneck in current computer systems.

Using NVM as block device through I/O interface such as SATA, SCSI or PCIe allows hosts to access NVM in the same way as conventional disk. No modification is required for operating system and application software. However, putting NVM on I/O bus as block device limits its performance and byte-addressability. An ideal way to utilize NVM is attaching it on memory bus as a memory device.

On the other hand, file system performance is dominated by metadata access because it is small and popular. Metadata and data are stored as block in file system. Partial metadata update results in whole block read or write which amplifies disk I/O greatly. Research work in [17] reported that only 21% of requests are file I/O and about more than 50% of them are metadata operations. Improving metadata I/O is desirable. Previous researches proposed to put metadata into DRAM for high performance. But, metadata will lost if system crashes. In-memory metadata has to be flushed back to disk periodically. Emerging NVM makes it possible to store metadata in NVM without risk of data lost caused by system crash.

Since NVM is expensive and its capacity is small, we argue that real system still needs block device such as disk, or SSD to store large amount of data. As a memory device, NVM can be used to optimize file system performance.

In this paper, a file system metadata accelerator (FSMAC) is designed by utilizing the advantages of NVM to optimize metadata access. FSMAC does not change the file system layout on disk, but it decouples data and metadata I/O path, putting data on disk and metadata on NVM at runtime. Thus, data is accessed in block from I/O bus and metadata is accessed in byte-addressable manner from memory bus. Metadata stored in NVM is updated in-place and never synchronized to disk at runtime. Thus, metadata access is significantly accelerated and metadata I/O is eliminated because metadata in persistent NVM is not flushed back periodically to disk anymore.

Metadata in NVM becomes physically persistent once updated. In the legacy system, metadata update in DRAM page buffer is not persistent, and becomes persistent only after it is flushed back to disk. In FSMAC, the latest version of metadata is in NVM. If system crashes in the middle of metadata update, the metadata will lost after reboot. Since NVM is much faster than disk, data update is behind metadata update. The speed mismatch of NVM and disk results in challenges in designing consistency mechanism for FSMAC because metadata is required to be updated after data update. Journaling technology is widely used in the conventional file systems. But simply applying or moving journaling from disk to NVM does not solve this problem perfectly for performance and space overhead. A light-weight consistency mechanism

978-1-4799-0218-7/13/$31.00 ©2013 IEEE

combining fine-grained versioning and transaction is introduced in the FSMAC.

The rest of this paper is organized as follows. Section II provides an overview of background and motivations. In Section III, we present the design of file system metadata accelerator (called FSMAC). Section IV presents its implementation details. Performance evaluation is presented in Section V. Section VI gives related work. Conclusion is summarized in Section VII.

II. BACKGROUND AND MOTIVATIONS

A. Non-volatile Memory

Flash memory is the first generation Non-volatile Memory. It is faster than HDD. But it is still much slower than DRAM. NAND flash memory can incur only a finite number of erases for a given block. To overcome the limitations of flash memory, next generation Non-volatile Memory is developed to combine both features of HDD and DRAM. It is non-volatile, fast, byte-addressable and power efficient. PCM, STT-MRAM and Memristor are candidates of next generation non-volatile memory. NVM refers to next generation non-volatile memory in this paper. The main attributes and technique parameters of these NVM technologies are summarized in Table 1.

PCM uses distinct phase change states (crystalline or amorphous) with different resistance to store values of 1 or 0. Due to change the state of materials by heating and cooling, its write performance is not perfect and cells are limited to as few as 10^8 writes. PCM write consumes more energy about one order of magnitudes than read [1]. Compared to flash memory, PCM is byte-addressable and has better endurance. Since its projected density is better than DRAM, PCM is a potential candidate to replace DRAM. PCM chip is commercially available and applied in some mobile digital devices.

STT-MRAM has advantages of lower power consumption over DRAM, unlimited write cycles over PCM and better scalability over conventional MRAM which uses magnetic fields to flip the active elements. Some prototype chips were developed in recent years and demonstrated that access latencies are 32 ns for read and 40 ns for write, in the same level as that of DRAM [2]. Company Everspin officially announces that the commercial STT-MRAM chip with DDR3 interface will be available in 2013 [33]. However, cell size is the biggest barrier for STT-MRAM.

Memristor is a more wonderful non-volatile memory technology with DRAM comparable performance and write endurance of 10^{10} cycles. However, this technology is still in early development and no commercial chip is available [34].

NVM is rapidly becoming a promising technology for the next-generation memory and attracts increasing attention in both academia and industry. The availability of NVM product with DDR3 interface will make it possible to attach NVM on memory bus directly and replace DRAM without modification.

As alternative, Non-Volatile DIMM (NVDIMM) offers a practical NVM option for systems integrators [3]. It combines DRAM and NAND flash technology to deliver a high speed and low latency non-volatile memory module, which can be integrated into the main memory of an industry standard server, perform workloads at DRAM speeds, yet be persistent and provide data retention in the event of a power failure or system crash. The NAND flash which has the same capacity of DRAM is invisible to host system and does not read or write during normal operation. Host will signal the NVDIMM to save or restore the DRAM contents to/from the NAND flash when power is down or up. NVDIMM can be viewed as the first commercially viable NVM in the memory market [4]. In this paper, the file system metadata accelerator is designed and evaluated on the basis of the NVDIMM with DRAM-like performance and disk-like persistency.

Table 1: A Summary of NVM Characteristics

Technology	Read latency (ns)	Write latency (ns)	Density (um^2/bit)	Endurance (writes/cell)
PCM	48	150	0.0058	10^8
STT-MRAM	32	40	0.0074	∞
Memristor	100	100	0.0058	10^{10}
DRAM	15	15	0.0038	10^{18}

Table 2: Metadata I/O rates in general workloads

Workload	Metadata I/O(%)	Data I/O(%)
Varmail	85	15
Webserver (Mean file size: 64KB)	56	44
Webserver (Mean file size: 16KB)	71	29

B. Metadata I/O in File System

File system performance is dominated by metadata I/O. Metadata is accessed and updated more frequently than file data in a file system. Research work in [17] shows that about 21% of requests are file I/O and about more than 50% of them are metadata operations. In particular, system calls to *file stat* comprise 42% of all file system related calls and even reach 71% in some workloads [17].

We have analysed the real enterprise workloads. Table 2 shows the proportions of data I/O and metadata I/O for enterprise workloads on Linux Ext4 file system. The percentage of metadata I/O in *Varmail* workload reaches 85%. In *Webserver* workloads, metadata I/O rates are 56% and 71% for mean file size of 64KB and 16KB respectively. Our metadata characterization shows the prevalence of metadata I/O in enterprise file system, aligning to previous study in [17].

Metadata is organized as block in file system. Partial metadata update results in whole block read and write which amplifies disk I/O. Since metadata I/O is so popular and most of them are small and random I/O, it is desirable to accelerate metadata access to improve file system performance.

C. File System Consistency

File system consistency includes metadata consistency, data consistency and version consistency in different levels [5]. Metadata consistency is a low level consistency which guarantees metadata consistency but does not provides any guarantees on data. Compared to metadata consistency, data consistency is a higher level file system consistency. It guarantees the consistency of both metadata and data. The version consistency provides stronger consistency, in which an old version of the data is possible to be retrieved.

Different approaches to guarantee file system consistency have been introduced. File system check such as *fsck* is used to check file system consistency after system crash [6]. It scans the entire storage space to check the consistency and try to correct the file system errors if possible, which is time consuming. Recently, some new checking methods, such as Recon [8], is introduced to check the inconsistency caused by file system bugs at runtime.

Journaling is another approach for file system consistency. It uses the idea of write-ahead logging to solve the consistent problem. Lots of famous file systems such as Ext3 [9], Ext4 [10], JFS [11], XFS [12], ReiserFS [13] and NTFS [14] use journaling technique to maintain consistency.

Journaling can convert small random writes into a sequential appending write, but updating on metadata block incurs both log write and original write. Furthermore, metadata blocks are written to journal zone frequently even only a few bytes are modified in those blocks. In this scenario, if metadata is stored in NVM and updated in-place, it has to wait till the file data is written out before update the metadata in NVM, which will shackle the performance of the hybrid file system. So, it is unacceptable to maintain consistency of a hybrid file system with metadata stored in NVM by traditional journaling approach.

Duplication technique is another approach to solve the consistency problem in file system [5]. Copy-on-write and Write-on-redirect are two common implementations of duplication. Write on a block (metadata or data) would incur two copies of the block. Once the write is completed on disk, the new version of data is added to the file system tree. Some latest file systems, like ZFS [16] and btrfs [15], use the duplication technique to maintain consistency.

Duplication can maintain file system consistency in metadata, data and even version level. Its drawbacks are the complexity of implementation and extra space cost for block-based duplication. For file systems with metadata in NVM which is byte-addressable but expensive, it is a high cost solution to maintain consistency with block-based duplication.

This motivates us to design a new mechanism to maintain file system consistency when NVM is used to accelerate metadata accesses.

III. Design Of the FSMAC

In this section, we present the design of file system metadata accelerator (FSMAC).

A. Overview

We design the FSMAC with two considerations. 1) Do not change data layout on disk so as to keep the file system compatible and portable across machines. 2) Do not change existing interfaces between different layers and minimize the modification of kernel, which makes the metadata accelerated file systems transparent to applications.

Fig. 1 shows the overview of the FSMAC. FSMAC decouples data and metadata I/O path, putting data on disk and metadata on NVM at runtime. Thus, data is accessed in block from I/O bus and metadata is accessed in byte-addressable manner from memory bus. To do this, following changes are made. First, a dedicated NVM management module is designed by modifying current memory management. Second, the metadata management is modified in current file system to pin metadata in NVM. Third, a light-weight file system consistency mechanism is designed. Finally, the policy of metadata fetch and flush is redesigned. Metadata could be loaded into NVM when file system is mounted or on-demand. Whether metadata in NVM is flushed back to disk or not depends on unmount options (file system mount and unmount is modified).

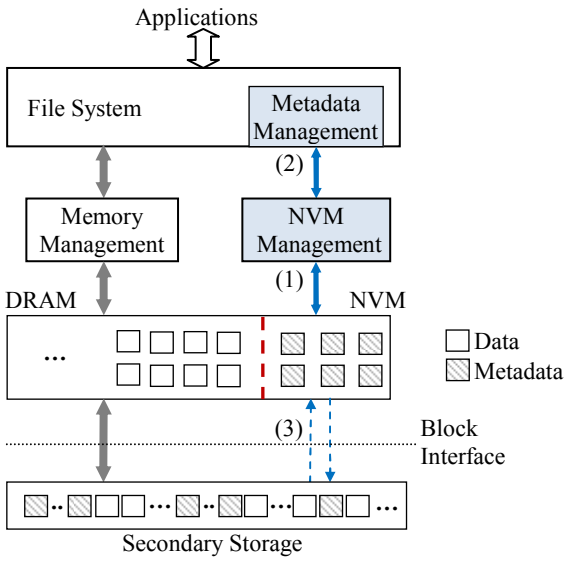


Fig.1: Architecture of file system metadata accelerator

B. NVM and Metadata Management

Metadata is stored in NVM and is not writen back to disk at running time. Since memory is allocated dynamically, we need to protect the metadata from being overwriten after system reboot. To address this problem, a special memory zone is reserved physically for NVM (called NVM Zone), as shown in Fig.2. A fixed access entrance is defined at the beginning of the NVM Zone to store the key data structures such as system status and file systems information. This entrance is access point of the FSMAC, from which all valid metadata can be retrieved after reboot.

The NVM Zone is managed by ourself. A set of dedicated kernel APIs are designed to allocate/deallocate memory from NVM Zone for FSMAC. Other applications who use legacy memory management functions are not allowed to access the reserved NVM Zone. In this way, the NVM could only be accessed by the FSMAC.

To minimize the modification of file system, page buffer is reused for metadata. Current file system allocates page buffer for data and metadata from DRAM without differentiation. By contrast, FSMAC allocates page buffer for metadata only from NVM Zone through the dedicated APIs. In this way, data path and metadata path is decoupled. Since FSMAC supports multiple file systems, there are metadata belonging to different file systems in NVM Zone. Metadata page buffers belonging to the same file system are organized as a list. The heads of the lists are stored in the fixed entrance in NVM Zone.

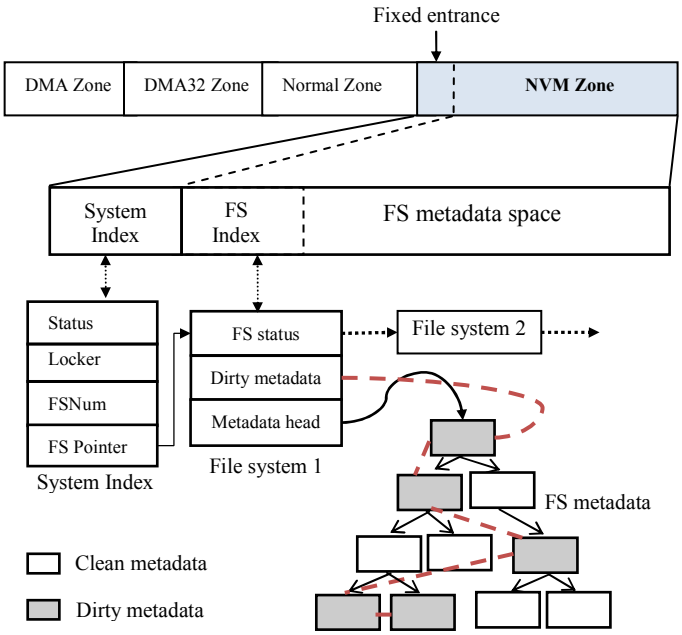


Fig.2: Data layout in NVM Zone

Current system flushes back dirty metadata blocks to disk periodically. To pin metadata in the NVM Zone, metadata management in current file system and flush policy are redesigned. An option is given to user to flush all dirty metadata into disk if necessary.

The system index and file system index are two core structures to maintain system-level status and file system level status respectively (See Fig.2). Accelerating multiple file systems is supported in the FSMAC. The number of accelerated file systems and corresponding file system indexes are maintained in the system index, which stays in the fixed entrance of NVM Zone. From the fixed entrance of the NVM Zone, all metadata blocks stored in NVM Zone can be found when system reboots or recovers from crash. All dirty metadata blocks are organized as a list, which helps to flush dirty metadata to disk quickly in case synchronization is needed (e.g. the disk is ported to another machine).

C. *File System Consistency*

As mentioned before, metadata in NVM is updated in place and becomes physically persistent once updated. In the legacy system, metadata update in DRAM page buffer is not persistent, and becomes persistent only after it is flushed back to disk. In FSMAC, the latest version of metadata is in NVM. If system crashes in the middle of metadata update, the metadata will be corrupt after reboot. We leverage widely used versioning to improve metadata durability in the NVM.

We carefully select the granularity of metadata versioning because it impacts performance and cost. One choice is byte-based versioning to fully utilize NVM byte-addressability. But it is very complex and tedious to implement. Another choice is block-based versioning which is widely used in legacy file system. But this approach results in write amplification and NVM space waste. We use fine-grained versioning, making a trade-off between these two choices. The granularity of fine-grained versioning can be inode size, e.g., 128bytes. Fine-grained versioning is able to maintain consistency at acceptable management cost and space cost.

Write ordering is used in current file systems, which writes file data into disk before metadata. In FSMAC, data is in the disk and metadata is in the NVM. Metadata in NVM is updated in-place, which is much faster than data update. But write ordering constrain requires that metadata update is after data update. If the metadata update in NVM must wait till file data is written to disk, the performance will degrade significantly. The speed mismatch of NVM and disk results in challenges in designing consistency mechanism for FSMAC. To address this problem, we introduce a new method combining transaction and fine-grained versioning.

We need a mechanism to control the status of different version and when to 'commit' the metadata update. Journaling is widely used and implemented mechanism in current file system. But directly using journaling into FSMAC which contains both NVM and disk is not suitable because it maintains too many statuses. To leverage current implementation in file system, we choose to simplify journaling mechanism for FSMAC. Like general journaling file system, FSMAC uses *transaction* to manage different versions of metadata. By contrast, we only maintain two statuses for a transaction: *running* and *committing*, which is much simpler than general journal file system.

Before updating a metadata, *original version* is created. The *original version* will be deleted only after the transaction is successfully committed. If system crash happens before the transaction committing, the *original version* of the metadata is recoverable. When metadata in a committing transaction is going to be updated, a *committing version* is created. In this case, metadata has three versions: *original version*, *committing version*, and *updating version* (the latest version). In most case, there are only one or two versions for a metadata.

Transaction committing writes file data only from DRAM page buffer to disk. At same time, multiple versions of metadata will be deleted. In this way, we can maintain whole file system consistency, as well as overcome speed mismatch between NVM and disk.

D. *Metadata Fetch and Flush*

FSMAC needs to fetch metadata from disk to NVM at initial stage. Metadata fetching is one-time operation for FSMAC. We can classify metadata in disk into location-fixed metadata and location-non-fixed metadata. Location-fixed metadata includes super block, block group descriptor, inode block and bitmap, etc., which reside in fixed address in disk. They could be loaded into NVM at file system mounting time or at the time these blocks are first requested. Location-non-fixed metadata blocks include indirect blocks and directory blocks. These blocks must be loaded into NVM on demanding due to their dynamically allocated locations.

Both location-fixed and location-non-fixed metadata blocks are not flushed to disk at running time. An option is given to user to issue a clean-unmount to flush all dirty metadata into disk if necessary (e.g., porting the disk to another machine), which makes the file system with FSMAC is compatible with original disk file system. Clean-unmount also guarantee file system consistency in case of any NVM

errors. Since all dirty metadata pages are organized as a list in NVM, it is very easy to only flush dirty metadata to disk.

IV. IMPLEMENTATION

We implemented FSMAC on the basis of Linux extended file systems.

A. Function Design

Implementation of FSMAC in Linux involves modifications in VFS, memory management and specific file system, as shown in Fig.3. NVM management module is implemented by modifying memory management to allocate memory from NVM Zone and support NVM page buffer management. NVM metadata management and consistency maintenance are implemented in specific file systems, which are going to benefit NVM metadata acceleration. NVM page buffer and its status are managed by metadata management module.

B. Memory Partition

In order to reuse data in NVM after system reboot, all metadata in NVM must be protected from being overwritten. To address the issue, we partition the physical memory space and reserve a partition named as NVM Zone to store metadata. Further, a set of dedicated memory allocation and deallocation APIs on NVM Zone are provided. These dedicated APIs are only invoked by the FSMAC. Other applications are not allowed to access NVM Zone.

We modify *node_zonelist()* in Linux kernel to add a new zone and make sure that the current memory allocation functions, i.e. *alloc_page()* and *kmem_cache_alloc()* would never allocate memory from NVM Zone. Meanwhile, we add a set of memory management functions, i.e. *alloc_page_nvm(),free_page_nvm(),mem_cache_alloc_nvm(), kmem_cache_destroy_nvm()*, which allocate/free memory only from/to NVM Zone. These functions are implemented on the basis of the current memory management functions with similar parameters.

Protected data in NVM Zone can be reused while system reboots. To do this, a fixed entrance is defined at the beginning of the NVM Zone and by which all structured data can be recognized. A system index is stored in the fixed entrance. From this index, all status and metadata of mounted file systems can be retrieved. Rebooting from a crashed system, both running and committing transactions are checked to recover file system to the latest consistency status. Pointers of these two transactions are stored in FS index structure.

C. Store Metadata in NVM

In Linux kernel, *sb_getblk()* is used to get the buffer header of a file system metadata block. If the metadata block is not cached in the page buffer, a new page buffer is allocated for the block and a pointer of *buffer_header* is returned. Instead of using the original *sb_getblk()*, FSMAC introduces a new function *nvm_sb_getblk()*, which allocates buffer pages from the NVM Zone.

Since these page buffers are used as metadata persistent store in NVM, we added an additional data structure *nvm_meta_bh* to manage metadata in NVM. The *nvm_meta_bh* maintains brief information of the metadata

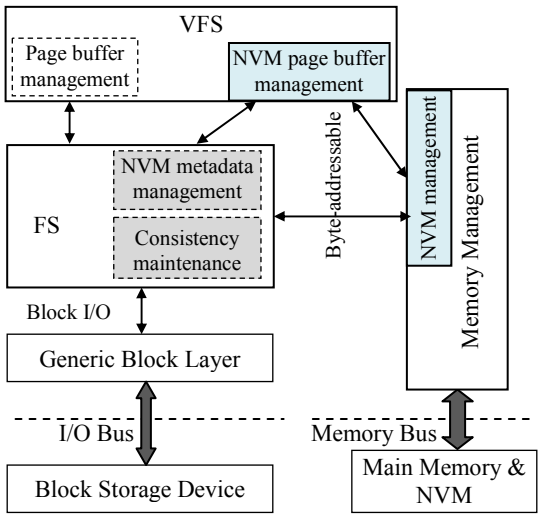


Fig.3: Implementation details of FSMAC

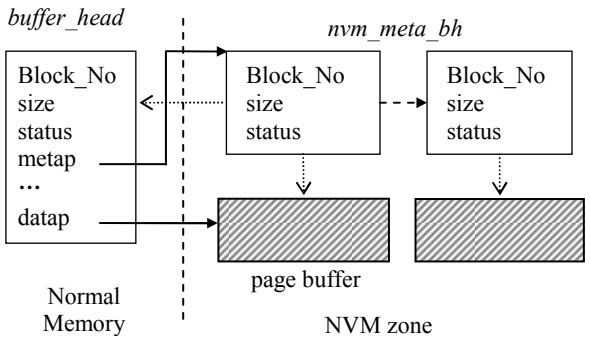


Fig.4: Data structures for managing metadata in NVM Zone

page buffer, i.e., block number, while *buffer_header* maintains detailed information of the page buffer. A pointer is used in *nvm_meta_bh* to link with the corresponding *buffer_header*. All *nvm_meta_bh* are organized as a list. In order to search *nvm_meta_bh* quickly, a pointer pointing to it is added in *buffer_head*. Fig.4 shows the relationship between metadata page buffer *nvm_meta_bh* in NVM and *buffer_head* in normal memory.

When metadata is modified, we change the status of *nvm_meta_bh* as dirty, instead of *buffer_head*. The function of *ext4_handle_dirty_metadata()* is rewritten. In this way, the Linux flush thread will not flush the metadata into disk anymore even though it is dirty in fact.

D. Maintain Consistency

FSMAC uses fine-grained metadata versioning and transaction to maintain file system consistency. We implemented it on the basis of the frame of Linux Journaling Block Device 2 (JBD2) source codes.

There are many transactions in journal system of JBD2. Only one of these transactions is at status of running or committing. Others are waiting for dirty metadata blocks to be written to disk in checkpoint list. But for FSMAC, metadata is stored in NVM and no necessary to be synchronized to disk. Once it is updated in NVM, it becomes persistency. So

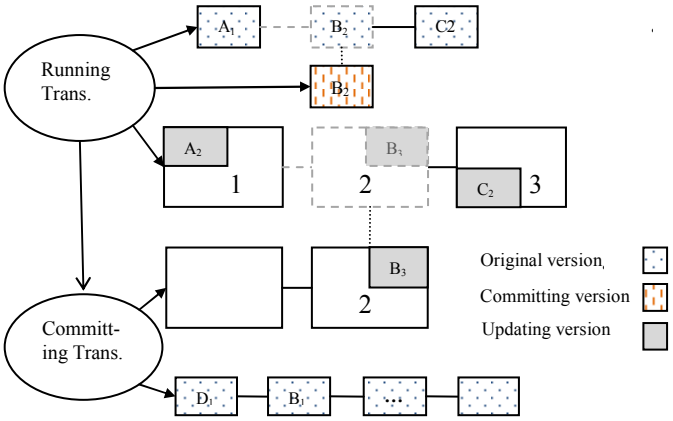


Fig.5: Transaction with fine-grained metadata versioning

FSMAC only maintains two transactions (running and committing), and removes checkpoint.

A metadata block is divided into many fine-grained sub-blocks, whose size can be inode size. Updating sub-block (metadata) results in creating an *original version* for this metadata. In the implementation, a running transaction consists of a metadata block list, a sub-block original version list or a sub-block committing version list. A metadata block to be updated will enter into the metadata block list firstly. When update is taken placed on a sub-block, the original data of this sub-block is copied and inserted into the sub-block original version list. If a sub-block in the committing transaction is updated again, a duplicate is made and inserted into both sub-block original version list and committing version list. The committing version list is used for fast recovery when system crashes during a committing transaction. Once commit executed, the committing version list in the running transaction can be removed.

Fig.5 shows an example of the relationship among the three lists. Metadata block 1 is in metadata block list of running transaction and its sub-block A is updated, so an original version of metadata is maintained in the sub-block original version list. Metadata block 2 is in both running and committing transactions and its sub-block B was updated in both transactions, so this sub-block has three versions: original version B1, committing version B2 which is also an original version for the running transaction, and updating version B3.

E. Synchronize Metadata to Disk

Since every metadata block in NVM has a corresponding *nvm_meta_bh* structure and all dirty metadata blocks are organized as a list, synchronization just goes through the list and flushes all dirty blocks to disk. For a dirty block, a write *bio* is constructed and submitted to general block layer to update it into disk.

The operation of synchronization is controlled by an option of modified file system un-mount. After synchronization, metadata on disk is the latest version of the file system and the disk can be ported to other machines.

V. EVALUATION

A. Evaluation Setup

We have implemented a prototype of the FSMAC on Linux Ext4 with kernel version 2.6.34. The experiment server is configured with an Intel duo-core CPU, 16GB DRAM, Intel 64GB SSD and Seagate HDD. Since NVDIMM has same read/write speed as DRAM, we use DRAM to simulate NVDIMM in our experiment system.

We have tested original Ext4 file systems and metadata accelerated Ext4 file system (Ext4-FSMAC) under different workloads.

B. Macro-benchmark Performance

(1) Filebench

Filebench is a file system benchmark which can generate different workloads to evaluate file system performance. In our experiment, four workloads covering wide features of read dominant and write dominant are generated to evaluate FSMAC. Workload *file create* creates a set of 200,000 files, where average file size is 16KB and 4 threads are running concurrently. Workload *file append* appends 8KB per request to a 1GB file and executes a *fsync* after every 32 appends. *varmail* and *webserver* are workloads to simulate I/O on enterprise environment of mail server and web server. *Varmail* workload consists of 10,000 files and its mean directory width is 50. Workload w*ebserver* has 100,000 files and is generated by 100 concurrent threads.

Fig.6 shows the results of the evaluation on both SSD and HDD. Ext4-FSMAC outperforms Ext4 from 5.4% to 147.4% in different workloads on SSD. Ext4-FSMAC also outperforms Ext4 from 10.9% to 621.7% in different

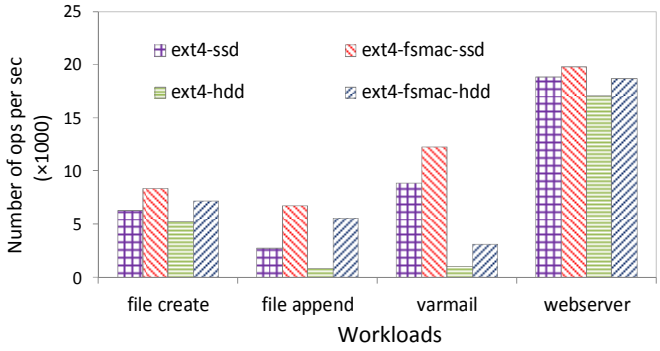


Fig. 6: Performance on workloads of filebench

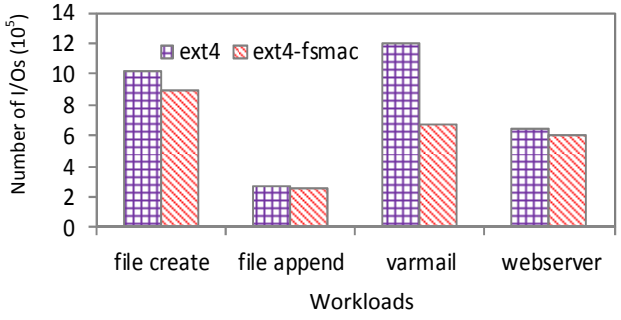


Fig. 7: Number of I/Os submitted to block layer

workloads on HDD. For workload *file append*, FSMAC achieves up to 2.47 times and 7.22 times speedup on SSD and HDD respectively. For workload *varmail*, FSMAC achieves speedup up to 1.38 and 3.08 on SSD and HDD. FSMAC achieves almost the same speedups on SSD and HDD for workloads *file create* and *webserver*. The reason lies in that the former two workloads issue lots of synchronous write operations while workload *file create* mainly operates in memory buffer and workload *webserver* is dominated by read operation. The results show that slower storage device with FSMAC achieves more acceleration, especially for those workloads with more write and synchronization operations.

We have collected the number of I/Os submitted to general block layer. As shown in Fig.7, FSMAC is efficient to reduce disk I/O caused by metadata update. The reduction of disk I/O for workloads *file creates*, *varmail* and *webserver*, directly results in performance improvement, which can be seen in the Fig.6. For workload *file append*, the Ext4-FSMAC only reduces 8.4% of I/Os but improves performance by 621.7% on HDD. The reason is that the workload issues *fsync* for every 32 append writes. After issuing *fsync*, the process must wait the metadata to be written to disk in Ext4 file system, while in Ext4-FSMAC, there is no waiting for metadata update because metadata is updated in-place in NVM.

(2) PostMark benchmark

PostMark is a single-threaded synthetic benchmark which simulates combined workload of email, usenet, and web-based commerce transactions. The workload includes a mix of data and metadata-intensive operations [35]. The PostMark version 1.51 is used in our evaluation and its configurations are descripted in Table 3.

Fig. 8 shows results of our experiments on SSD. From the results, we can see that FSMAC achieves up to 1.67 times performance acceleration on Ext4 file system across different operations.

(3) FFSB benchmark

The Flexible File System Benchmark (FFSB) is a cross-platform file system performance measurement tool. It uses configurable profile to generate different workloads. In our

Table 3: PostMark Configurations

Parameters	value
Number of Files	200,000
Number of Subdirectories	50
File Size	1KB-10KB
Number of Transactions	350,000
Buffered I/O	yes

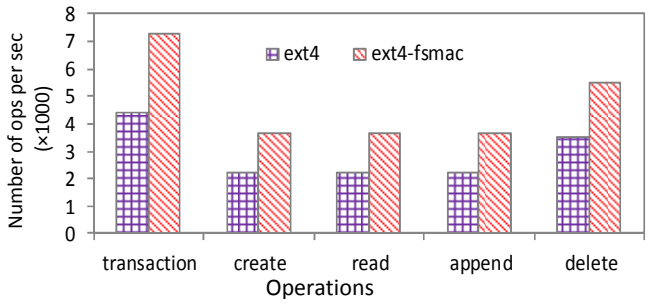


Fig. 8: Performance on workloads Postmark

Table 4: Weight of different file size

File Size	Weight (%)	File Size	Weight (%)
4KB	33	128 KB	5
8KB	21	256 KB	4
16 KB	13	512 KB	3
32 KB	10	8MB	2
64 KB	8	32MB	1

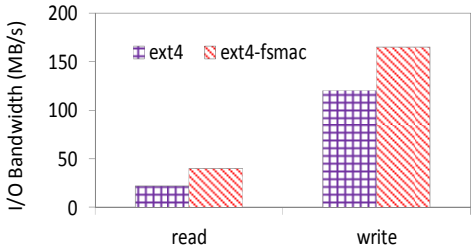


Fig. 9: I/O bandwidth of Ext4 and Ext4-FSMAC for FFSB

experiment, FFSB is configured to generate a synthetic workload on a common workstation. Read/write block size is 4KB. Ten types of file operations are configured with equal weight of 10%. The number of directories is set as 100. Direct I/O, aligned I/O and buffered I/O options are turned off. The weight of different file sizes is configured as Table 4.

Fig.9 shows results I/O bandwidth of Ext4 and Ext4-FSMAC under workload FFSB. Compared to Ext4 file system, Ext4-FSMAC improves read performance by 82% and writes performance by 35%.

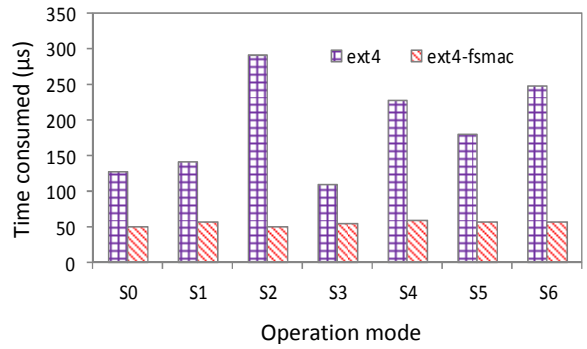
C. *Micro-benchmark Performance*

Fs_mark [32] is a benchmark released by EMC Corporation for evaluating performance of file creation. It offers 7 sync/asynchronous modes with option "*–S x*" (marked as "*Sx*" in this paper) where "*x*" can be number from 0 to 6. If *x* is 0 and 2, there is no synchronous (*fsync*) during file creation. We have evaluated performance of micro-operations including *create*, *write* and *fsync* in different sync/asynchronous modes. In our experiments, 4 threads are running in 4 directories concurrently and each directory has 6 subdirectories with 100 files. File name is as long as 40 chars and file size is 200KB. Files are created and written with I/O size of 16KB.

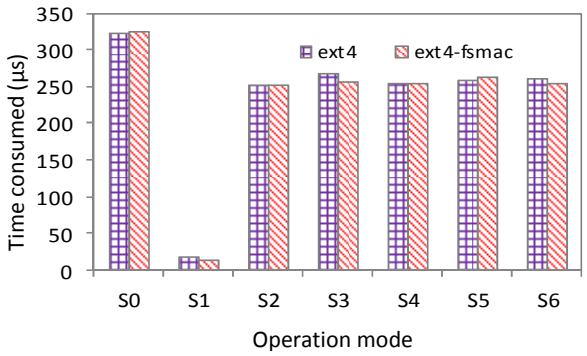
Experiment results are shown in Fig.10. The results indicate that Ext4-FSMAC outperforms Ext4 for *create* and *fsync,* while achieves same performance for *write*. The reason is that *write* is dominated by file data operation while *create* and *fsync* is dominated by metadata operation. Ext4-FSMAC has better performance of metadata operation than Ext4 file system. For *write* in "*S1*" mode, the synchronization is only issues before file closed and written data stays in page buffer, so performance is much better than that other modes. For *fsync* in mode "*S4*" and "*S6*", Ext4-FSMAC outperforms Ext4 by 34 times because *fsync* is executed over each file. Ext4 file system is much more sensitive on synchronous operations than Ext4-FSMAC.

D. *Performance with Varying File System Mount Options*

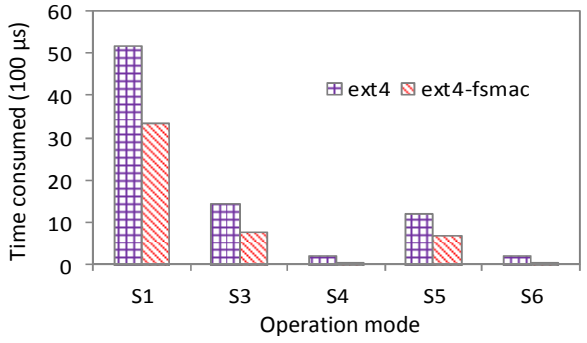
We have compared performance between Ext4-FSMAC and Ext4 with different file system mounting options, which include no-sync (default mount option), dir-sync and sync mounting.



(a) Time for creation



(b) Time for write



(c) Time for fsync

Fig.10: file operation time in different modes

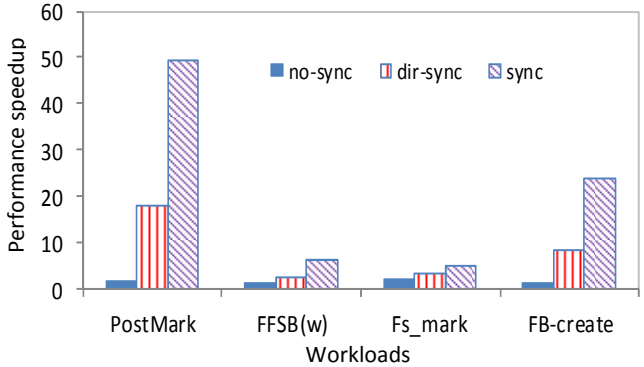
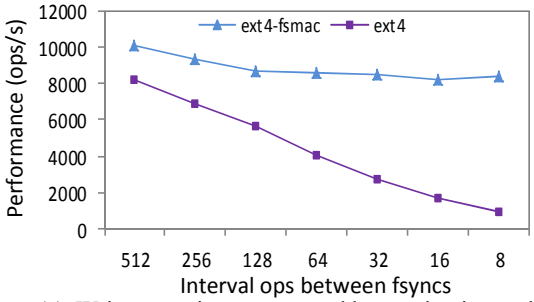
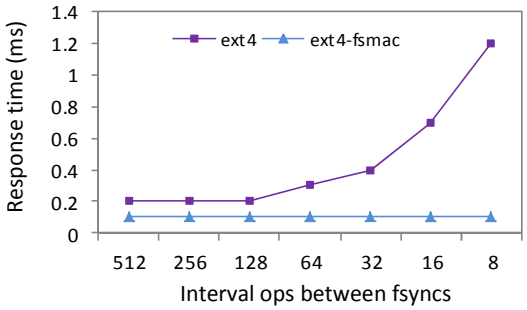


Fig. 11: Performance speedup in vary mounting options



(a) Write operations per second by varying interval operations between 2 fsyncs.



(b) Write response time by varying interval operations between 2 fsyncs.

Fig. 12: Comparison on write performance affected by synchronization frequency on ext4 and Ext4-fsmac file

Fig.11 shows performance results for different mount options. From the results we can see that Ext4-FSMAC achieves 1.55×, 17.6× and 49.2× performance improvement over Ext4 file system for no-sync, dir-sync and sync mounting option respectively.

E. Effect of Synchronization Frequency

We also evaluate the effect of synchronization frequency on Ext4-FSMAC and Ext4 file systems, in which workload *file writefsync* provided by Filebench is used. The workload appends 8KB of data to a 1GB file. After a predefined *N* times of append, file synchronization (*fsync*) is issued. We tested the file system performance by varying the interval of two synchronizations.

Fig. 12 shows the results. We can see that frequent file synchronizations degrade Ext4 file system performance significantly. Its write performance drops almost linearly with increasing synchronization (See Fig.12 (a)). By contrast, increasing synchronization does not affect Ext4-FSMAC so much (See Fig.12 (b)).

F. FSMAC for Different File Systems

As present in section IV, FSMAC implementation includes a shared part for all file systems and a file system specific part. The latter only involves a few lines code for metadata management and is easy to implement. Except the detailed implementation and evaluation of FSMAC on Linux Ext4 file system, we also had a draft implementation and evaluation on Linux Ext2 and Ext3 file system, whose source code is similar with that of Ext4. Since file system of Ext2 does not support journal, FSMAC for Ext2 does not contain function of consistency maintaining for simplifying the implementation.

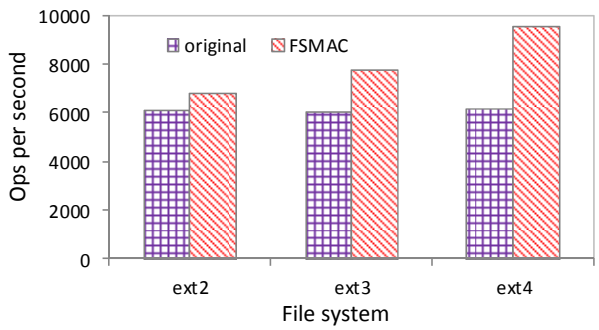


Fig. 13: Performance accelerated for different file systems

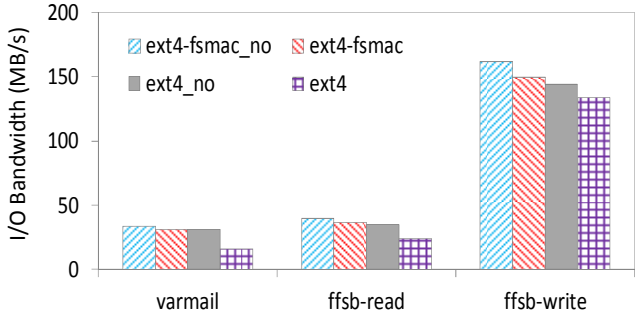


Fig. 14: Overhead of consistency maintenance

File system performance benchmark of FFSB with the same configure as showing in prior section is used to evaluate the performance speed up in these different file systems.

The experiment results (See Fig.13) show that the FSMAC works well on Ext2, Ext3 and Ext4 file system, accelerating their performance. In addition, Ext4 with FSMAC perform best. The reason is that Ext4 employs write-delay method for file data writes. Delaying write makes file data on disk become more sequential and reduces small random I/O. The metadata access in Ext4 file system issues lots of small random I/O, which would impact the performance significantly. By adopting FSMAC in Ext4, the metadata I/O is eliminated completely and the performance of file system is increased more significantly than that of Ext2 and Ext3.

G. Overhead of Consistency Mechanism

To evaluate the overhead of consistency mechanism used in FSMAC, we have test the performance of both Ext4-FSMAC and Ext4 with/without consistency mechanism (referred as Ext4-fsmac, ext4, Ext4-fsmac_no, ext4_no). From the results (See Fig.14), the following observation can be made. (1) Journaling degrades performance of Ext4 file system significantly (up to 50%). (2) FSMAC incurs acceptable overhead. For example, FSMAC performance drops less than 10% for workload *ffsb-write* workload.

VI. Related Work

As emerging technology, NVM attracts increasing research interests in recent years. Some of these researches, such as BPFS [20] and SCMFS [21], used NVM as persistent memory and redesigned a whole file system on non-volatile memory. Both metadata and data of file system are stored in NVM and accessed in byte-addressable manner. File system consistency is maintained in CPU instruction level. Building file system on NVM depends on an assumption that NVM is big enough. However, NVM with high density and large capacity is still far away from the real application. It is also difficult to migrate data between different computer systems. DFS [22] is a file system build on virtual memory space and stores data in PCI-e based flash controller. Different from above mentioned file systems, FSMAC decouples data and metadata path, and only maintains metadata on NVM, requiring much small capacity on NVM. At the same time, disk is used to store large amount of data, which can be terabytes. FSMAC can be deployed to accelerate local block-based file system and remote block storage such as SAN.

NVM was also used as block storage device to build NVM storage system or hybrid storage system. The Conquest file system [23] divides the file system into two parts. One part including metadata and small files is stored in NVM and the other part including big files is stored in common block storage device, which is a reasonable way to use NVM in current. Since data is stored separately in two different devices, data migration is very complex and difficult. All data must be copied for a migration because it is incompatible with other file system. In work [24], file system metadata is stored in MRAM in a special designed structure and layout to accelerate the performance. A similar work is done by using NVM and Flash memory store metadata and file data respectively [28]. FRASH is also a hybrid file system [25]. It redesigns an in-memory metadata structure as well as an on-disk structure. In-memory structures are stored in NVM and copied into memory during the mount phase. Same problems of compatibility and migration with Conquest are not solved in these hybrid file systems, which is also the differences compared to FSMAC. Above mentioned works did not consider consistency issues.

Pairwise-FS [26] and Shortcut-FS [27] are file systems for PCM. Both of them use PCM as block storage device and try to optimize write access on PCM. As a block storage device, NVM is accessed by a relatively long path of block I/O stack and its performance is limited.

Replacing traditional DRAM with NVM is another research issue in recent years. A consistent and durable data structure is designed on NVM in [29]. Mnemosyne [30] and NV-Heaps [31] were developed to provide transaction mechanism to update data in NVM at user-level. They have the same target with this paper to explore ideal way for NVM based computing system in the future but in different approaches.

VII. Conclusion

Byte-addressable non-volatile memory is a promising technology that is likely to be the next big milestone in computer hardware. In this paper, we presented a file system metadata accelerator (called FSMAC) by exploiting NVM as memory device to optimize metadata accesses. FSMAC decouples the data and metadata path, putting data on disk and metadata on NVM at runtime. Thus, data is accessed in block from I/O bus and metadata is accessed in byte-addressable manner from memory bus. Metadata stored in NVM is updated in-place and never synchronized to disk at runtime. Therefore, metadata access is significantly accelerated and

metadata I/O is eliminated. FSMAC does not change the file system layout on disk so that the accelerated file system is compatible with traditional disk file systems after a clean-unmount. A light-weight consistency mechanism combining fine-grained versioning and transaction is proposed in the FSMAC.

We have implemented FSMAC in Linux kernel and conducted intensive evaluations on enterprise workloads. Experiment results demonstrate that proposed FSMAC is efficient in accelerating metadata access and reduce disk I/O for different file system.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their feedback and suggestions for improvement. This work was supported by Agency for Science, Technology and Research (A*STAR), Singapore under Grant No. 112-172-0010.

REFERENCES

[1] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. IBM Journal of R. and D., 52(4/5):465–479, 2008.

[2] Takayuki Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. IEEE Design & Test of Computers, 28(1):52–63, 2011.

[3] Dushyanth Narayanan, Orion Hodson. Whole-System Persistence. In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12), March, 2012.

[4] AgigaTech. ArxCis-NV (TM) Non-Volatile DIMM. http://www.vikingtechnology.com/nvdimm-technology.

[5] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In Proceedings of USENIX Conf. on File and Storage Technologies (FAST'12), 2012.

[6] Marshall KirkMcKusick,Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[7] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[8] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In Proceedings of USENIX Conference on File and Storage Technologies (FAST'12), 2012.

[9] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In Proceedings of the Fourth Annual Linux Expo, Durham, North Carolina, May 1998.

[10] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier. The new ext4 filesystem: current status and future plans. 2007 Linux Symposium, Volume Two. pp.21-33, 2007.

[11] Steve Best. JFS Overview. http://www.ibm.com/developerworks/library/l-jfs.html, 2000.

[12] Adan Sweeney, Doug Doucette,Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In Proceedings of the USENIX Annual Technical Conference (USENIX'96), January 1996.

[13] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[14] Rajeev Nagar. Windows NT File System Internals: A Developer's Guide. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[15] Wikipedia. Btrfs. en.wikipedia.org/wiki/Btrfs, 2009.

[16] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/ os/community/zfs/docs/zfs last.pdf, 2007.

[17] Andrew W. Leung. Measurement and Analysis of Large-Scale Network File SystemWorkloads. In Proceedings of the USENIX Annual Technical Conference (ATC'08), 2008.

[18] Nitin Agrawal. et al. A Five-Year Study of File-System Metadata. In Proceedings of USENIX Conference on File and Storage Technologies (FAST'07), 2007.

[19] Drew Roselli. A Comparison of File System Workloads. In Proceedings of the USENIX Annual Technical Conference (ATC'2000), 2000.

[20] Jeremy Condit, Edmund B. Nightingale, Christopher Frost. Better I/O Through Byte-Addressable, Persistent Memory. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'09), Pages 133-146, 2009.

[21] Xiaojian Wu, A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Anaglsis (SC'11), pp. 1498-1503. 2011.

[22] WILLIAM K. JOSEPHSON, LARS A. BONGO, and KAI LI. DFS: A File System for Virtualized Flash Storage. In Proceedings of the Eighth USENIX Conference on File and Storage Technologies (2010), USENIX Association. Pp.85-100, 2010.

[23] An-I A. Wang, Geoffrey H. Kuenning, Peter Reiher and Gerald J. Popek. The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design. ACM Transactions on Storage, Vol.2, No.3, Pages 309–348, 2006.

[24] Kevin M. Greenan and Ethan L. Miller. Reliability Mechanisms for File Systems Using NonVolatile Memory as a Metadata Store. In Proceedings of the International Conference on Embedded Software (EMSOFT'06), 2006.

[25] Jaemin Jung, Youjip Won, Eunki Kim, et al. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. ACM Transactions on Storage, Vol.6, No.1, 2010.

[26] Eunji Lee, Jee Eun Jang, Kern Koh, Hyokyung Bahn. Pairwise-FS: A Novel File system Structure for Phase Change Memory. In Proceedings of The 18th IEEE Real-Time and Embedded Technology and Applications Symposium Work-in-Progress (WiP) , Pp5-8. 2012.

[27] Eunji Lee, Seunghoon Yoo, Jee-Eun Jang, Hyokyung Bahn. Shortcut-JFS: A Write Efficient Journaling File System for Phase Change Memory. In Proceedings of MSST'12. 2012

[28] In Hwan Doh, Jongmoo Choi, Donghee Lee, Sam H. Noh. Exploiting Non-Volatile RAM to Enhance Flash File System Performance. Pp.164-173, In Proceedings of the International Conference on Embedded Software EMSOFT'07, 2007.

[29] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan and Roy H. Campbell. Redesigning Data Structures for Non-Volatile Byte-Addressable Memory. In Proceedings of USENIX Conference on File and Storage Technologies (FAST'10), 2010.

[30] Haris Volos, Andres Jaan Tack, Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), 2011.

[31] 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 Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), 2011.

[32] Fsmark. http://sourceforge.net/projects/fsmark/.

[33] http://www.mram-info.com/everspin-officially-announces worlds-first-st-mram-chip-will-be-available-2013.

[34] Iulian Moraru, David G. Andersen, Michael Kaminsky, Nathan Binkert, Niraj Tolia, Reinhard Munz, Parthasarathy Ranganathan. Persistent, Protected and Cached: Building Blocks for Main Memory Data Stores. Technique report of CMU-PDL-11-114, 2011.

[35] A. Traeger, E. Zadok, N. Joukov and C. P. Wright. A Nine Year Study of File System and Storage Benchmarking. ACM Transactions On Storage, Vol.4, No.2, May 2008.