An Empirical Study of File Systems on NVM

Priya Sehgal, Sourav Basu, Kiran Srinivasan, Kaladhar Voruganti*
NetApp Inc.
Email:{priya.sehgal, sourav.basu, kiran.srinivasan}@netapp.com, kaladhar_voruganti@yahoo.com

***Abstract*— Emerging byte-addressable, non-volatile memory like phase-change memory, STT-MRAM, etc. brings persistence at latencies within an order of magnitude of DRAM, thereby motivating their inclusion on the memory bus. According to some recent work on NVM, traditional file systems are ineffective and sub-optimal in accessing data from this low latency media. However, there exists no systematic performance study across different file systems and their various configurations validating this point. In this work, we evaluate the performance of various legacy Linux file systems under various real world workloads on non-volatile memory (NVM) simulated using ramdisk and compare it against NVM optimized file system -- PMFS. Our results show that while the default file system configurations are mostly sub-optimal for NVM, these legacy file systems can be tuned using mount and format options to achieve performance that is comparable to NVM-aware file system such as PMFS. Our experiments show that the performance difference between PMFS and ext2/ext3 with execute-in-place (XIP) option is around 5% for many workloads (TPCC and YCSB). Furthermore, based on the learning from our performance study, we present few key file system features such as in-place update layout with XIP, and parallel metadata and data allocations, etc. that could be leveraged by file system designers to improve performance of both legacy and new file systems for NVM.**

Keywords—*file system; non-volatile memory; performance*

I. INTRODUCTION

The emergence of flash has been a disruptive force that has led to dramatic changes in how storage systems are designed. However, there are newer forms of non-volatile memory (NVM) technologies (e.g. PCM, STT-MRAM, ReRAM) in the pipeline that promise to be even more disruptive than flash. These new non-volatile memory technologies are byte addressable, unlike flash that is block addressable, and have latencies close to DRAM and densities better than DRAM [20, 28, 29]. Thus, NVM can augment DRAM on the memory bus, allowing applications to persist their working sets close to memory latencies.

Whenever a new media arrives there is a lot of research on designing and developing new file systems that cater to the characteristics of this media. For example, for non-volatile memory, new file systems such as PMFS [16, 33], SCMFS [19], and BPFS [10] have been proposed. These file systems leverage byte-addressability and random access features of NVM to gain maximum performance benefit. Moreover, there has been a lot of research on persistent memory abstractions such as Mnemosyne [6], NV-Heaps [9], PMem-Lib [30][8]. While these approaches seem optimal, there exists no work that evaluates existing block-based file systems on NVM. Since multiple decades worth of work has gone into these file systems, it is important to examine whether these legacy file systems can be fine-tuned using file system configuration options for NVM.

In this paper, we present a study that evaluates the performance of various legacy file systems under various real-world workloads on NVM. We selected commonly used server-class workloads such as webserver, fileserver, webproxy, database and key-value stores as they differ from each other in terms of data access patterns, metadata-data ratios, etc. We evaluated and compared the results of above workloads on NVM-aware file system – PMFS, flash-aware file system – F2FS [3, 25], and five traditional Linux file systems: Ext2 [22], Ext3 [23], Ext4 [12, 24], XFS [34, 36], and NILFS2 [32] on NVM. As we wanted to determine whether legacy file systems could be fine-tuned to perform comparable to PMFS, we assess the performance of these file systems under different mount and format options. Some of the options that we varied include different journaling modes, allocation policies such as delayed allocation, mechanisms to bypass buffer cache using execute-in-place (XIP), etc.

Our study shows that existing file systems, with little reconfiguration or slight changes, perform close to NVM-aware file systems, for various real-world workloads. Moreover, we have identified a few file system features that boost the performance of file systems on NVM. Here is a summary of some key findings from our study:

1. **In-place update vs. log-structured layout:** File system designers prefer a log-structured file (LFS) layout [13] for different types of media as it mitigates media limitations such as rotational latency in case of hard disk and block erasure granularity and write amplification in case of Flash. LFS is also preferred for memory allocation in complete DRAM-based systems that are backed by disks (RAMCloud [18]). Although LFS provides useful features such as continuous snapshots, our study shows that pure LFS-based file system (NILFS2) perform much worse compared to in-place update file systems, such as ext2, ext3, ext4 and XFS in workloads involving read-write mix. F2FS, which is a hybrid file system and treats most of its metadata in-place and data as log-structured, performs much better than NILFS2 and within 15-25% of some of the default configurations of in-place file systems. Moreover, NILFS2 and F2FS employ garbage

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

* The author was in NetApp when the paper was written

collection that utilizes CPU and memory resources for cleaning obsolete or deleted segments, depriving the active file system from these useful resources. This impacts the performance of LFS or hybrid file systems adversely. Therefore, we conclude that in-place update of both data and metadata is a preferred layout for NVM file systems.

2. **Execute-in-place vs. Buffered file system:** Buffer cache is useful and helps improve performance when a file system exists on slow media such as hard disk and is front-ended with fast DRAM. However, on NVM that has access latencies similar to DRAM, buffer cache introduces copy overheads rather than being useful. Execute-in-place (XIP) [21] helps bypass this extra copy and improve file system performance running on NVM by directly performing read and write from/to NVM media. It also bypasses block level I/O scheduling. PMFS adopts this feature by default. Our experiments show that if we only enable XIP feature on existing file systems (ext2 and ext3), it helps existing file systems to perform at par with PMFS – about **5-20%** performance difference in many workloads. Opening the file in direct mode (O_DIRECT) also helps bypass buffer cache, but it only works for data updates and not for metadata updates. Moreover, the block layer overhead of scheduling I/O remains in case of O_DIRECT. Hence, we do not evaluate this feature in our study.

3. **Simple and parallel allocation/de-allocation:** We found that allocation strategies impact the performance of certain workloads, which involve multiple file creations/deletions or file size increase (e.g., fileserver and webproxy). Firstly, features such as allocation groups [34] (XFS) or block groups (ext3, ext4) help scale the performance as it allows parallel allocations. In workloads involving parallel data (read, write) and metadata (create, delete) operations on a large number of files (~500K or more), we found that PMFS performed the worst amongst all the file systems, around **5x** worse compared to ext3 with XIP enabled. This is because it uses only one singly linked list for data and metadata allocation (Section IV B) inhibiting its scalability. Secondly, because of fast random access speed of NVM, optimization such as delayed allocation (XFS and ext4) is of no use in data intensive workloads. Nevertheless, if the workload involves multiple data allocations, delayed allocation boosts the performance even on high speed NVM as it amortizes allocation cost.

4. **Fixed vs. variable block size (extent):** Our experimental results reveal that the performance across traditional file systems that manage data using fixed sized blocks versus variable sized extents is close. Using fixed sized data blocks simplifies the data structures required to maintain the free space information and indexing information about files and directories i.e., inodes. This in turn reduces the CPU path length of the code required to perform lookups and maintain these data structures. Hence, it is advisable to use fixed sized data blocks for NVM file systems such as in ext2 and ext3 vs. variable sized extents as in XFS and ext4.

The rest of this paper is organized as follows. Section II presents the related work; Section III describes our experimental methodology. We present the evaluations and analysis in Section IV. Finally, we provide some recommendations and conclude in Section V.

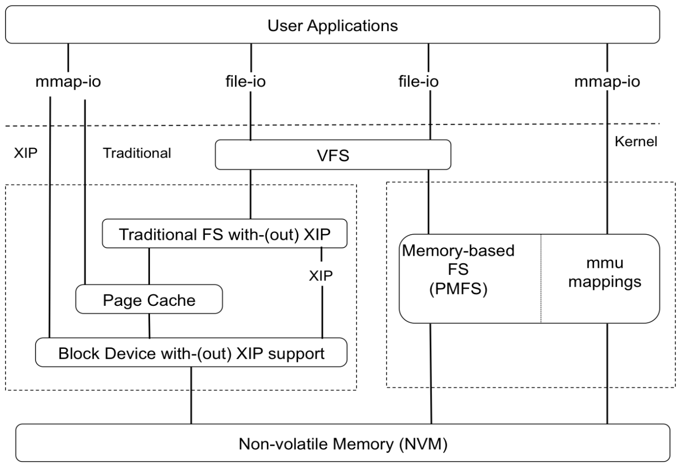

Fig. 1. Traditional vs. optimized POSIX file systems

II. Background And Related Work

The emergence of low speed, byte-addressable storage class memory or non-volatile memory (NVM) on the memory bus has led to a lot of research on methods to access and manage data stored on this fast media. Prior research related to NVM primarily falls into two buckets: (i) exploring file system design alternatives and (ii) exploring interfaces or programming model alternatives presented to applications. While (i) is needed for coarse-grained space management and protection of the NVM media, (ii) is required for efficient access to data and fine-grained data management by application. Fig. 1 summarizes the combinations of available POSIX interfaces: *file-io* (open, read, write) and *mmap-io* (open, mmap and load/store), and file systems: traditional and NVM optimized, for managing and accessing data from the NVM media. Existing research can be classified as:

1. Use of POSIX interface (open/read/write or mmap/load/store) but re-designing the file system for NVM (category i). PMFS [16, 33], SCMFS [19], and BPFS [10] fall in this category. These file systems avoid traditional storage stack overheads and leverage NVM media features, such as byte-addressability, atomic-updates, and fast random access speeds.

2. POSIX library interposers bypass the file system during data path, thereby reducing the latency to access data (category ii). It accesses kernel file systems only during the control path i.e., for access control, allocating space from physical NVM media, etc. Moneta-D [1, 2] and Bankshot [14] adopt this technique and move some file system functionality into hardware. Aerie [7] and [17] also fall in this category but depend on user mode file systems.

Workload	Average file size	Average directory depth	No. of files	I/O Size (r/w)	Threads	R/W ratio	Fsync	Metadata operations
fileserver	128K	3.8	100K	16K	50	1:2	No	Yes (C/D/S)
webproxy	32K	0.7	500K	1M/32K	50	5:1	No	Yes (C/D)
webserver	32K	2.5	500K	1M/8K	50	10:1	No	No

TABLE I. Filebench workload characteristics (Metadata operations – C: Create, D: Delete, S: Stat)

3. Use of new programming models to simplify direct use of NVM by applications (category ii). Examples include Mnemosyne [6], NV-Heaps [9], PMem-Lib [30], and [8]. These libraries provide fine-grained data management, allocation, de-allocation, persistence, and transaction management while coarse-grained space management resides with the file system. NV-Tree [11] and CDDS [15] propose a consistent and durable data structure (B+Tree variant) for non-volatile memory.

4. Use of existing POSIX interface and traditional block-based file systems, but adjust operating system, storage stack or file system configurations to improve performance for NVM. To our knowledge there is only one paper [5] that falls in this category and is closely related to ours.

Lee et al. [5] explore various I/O configuration parameters such as buffer cache, read-ahead, synchronous, direct I/O, etc. on NVM and compare it with results on HDD. They use Ext4 file systems for all their evaluations. Our work differs from this previous work: we concentrate on evaluating performance of the following file system configurations under different workloads on NVM: (a) default access to traditional file system, (b) fine-tuned access to traditional file systems through mount and format options, and (c) NVM-optimized file system, such as PMFS. We use various in-place-update and log-structured file systems to evaluate traditional file systems. Section III provides details on the different workloads and file system configurations that we evaluate.

III. Methodology And Limitations

In this work, we wanted to determine whether it is possible to reconfigure or fine-tune traditional file systems such that they perform close to NVM-optimized file systems (PMFS). Further, we wanted to summarize key file system features that boost their performance on NVM. To achieve this goal we evaluated different Linux file systems under varying workloads on simulated NVM. This section details the experimental hardware and software setup for our evaluations. We describe our testbed in Section A. In Section B and C we discuss the two important dimensions of our evaluation: workloads and file systems, respectively.

A. Experimental Setup

Our experimental setup consisted of a commodity server with 8 Intel Xeon 2.40GHz processors. It consisted of 66GB DRAM, out of which we configured 58GB as ramdisk to simulate NVM. It is important to note that our experiments focus on file systems and do not evaluate the different types of NVM (e.g. PCM, STT-MRAM, ReRAM etc). Thus, we conduct experiments on ramdisk carved from DRAM, which does not simulate NVM slower than DRAM or have asymmetric latencies. Since all performance numbers are relative, the same observations should be valid for NVM, which would be slower than DRAM. In order to pin down the memory pages belonging to ramdisk (NVM), we perform *dd* on the entire 58GB ramdisk. In case of PMFS we reserved 58GB of memory using the grub option *memmap*. User processes and buffer cache used the rest of the free DRAM space (6GB). The swap daemon is switched off in every experiment to avoid swapping off pages to disks (in case of mmap). We conduct all our experiments on Ubuntu 12.04 using Linux Kernel 3.11.

B. Workload Categories

We wanted to evaluate the performance of file systems under different workload parameters: file size, directory depth, read-write ratio, metadata vs. data activity, access patterns (I/O size, sequential vs. random vs. append). We selected five common workloads: webserver, fileserver, webproxy, OLTP database and key-value stores. We used Filebench [27] benchmark suite to emulate the first three workloads, TPC-C [35] for the OLTP database workload, while YCSB [4] benchmark for key-value stores. Table 1 summarizes workload properties of the first three workloads. We discuss all the workloads below in more detail.

Fileserver: This workload emulates a server hosting home directories of multiple users (threads). Each thread picks up a different set of files based on its ID and performs sequence of create, delete, append, read, write and stat operations. This workload exercises both data and metadata activities. The ratio of metadata to data operations is 1:1.

Webproxy: This emulates a simple webproxy server. It generates a mix of create, append, read, and delete operations, simultaneously, from a large number of threads. This workload is characterized by fairly flat namespace hierarchy, with a directory depth of 0.7 i.e., all the 500K files are contained within one large directory. The ratio of metadata to data operations is 1:3.

Webserver: The webserver workload is characterized by a read-write ratio of 10:1, consisting of full file sequential reads by all the threads, emulating web page reads. All the threads append 8K to a common log file. This workload is primarily read-intensive. It not only exercises fast lookups and small file reads, but also concurrent data and metadata updates into single, growing log file.

OLTP Database: The TPC Benchmark C [35] is intended to model a medium complexity online transaction processing (OLTP) workload. The benchmark represents a generic wholesale supplier workload consisting of 9 tables and 5 stored procedures. In our evaluation, we ran TPC-C over MySQL database v5.5. We used 400 warehouses, generating a database size of 38GB. Since our server consisted of 8 CPUs, we used 8 concurrent threads to generate the input load. Note that after every transaction commit, MySQL calls `fsync` on the logfile.

Work load	Read	Update	Insert	Scan	RMW	Dist
A	50	50	0	0	0	Zipf
B	95	5	0	0	0	Zipf
C	100	0	0	0	0	Zipf
D	95	0	5	0	0	Latest
E	0	0	5	95	0	Zipf
F	50	0	0	0	50	Zipf

TABLE II. YCSB workload characteristics

Key-Value Stores: The Yahoo! Cloud Services Benchmark (YCSB) [4] is a workload that is representative of large-scale services provided by web-scale companies. It is a key-value workload. We ran YCSB on a NoSQL database – MongoDB v2.6.7 [31], which performs memory-mapped I/O on its database files and file-io (`write` system call) to its journal. MongoDB calls `fsync` on its journal every 120 ms and `msync` on its database files every 60 seconds. In YCSB, each tuple consists of unique key and 10 columns of random string data of 100 bytes each. Thus, the total size of a tuple is approximately 1KB. YCSB is composed of six workloads – A to F. Table II describes the percentage of different operations and distribution of each YCSB workload. For all the workloads, we set the record count to 16 million and the operation count to 10 million. This test generated a working set of 36GB. We used 8 threads to generate the input load.

C. *File Systems*

We wanted to determine the file system configuration parameter and feature sets suitable for NVM environment under different workloads. Based on varying properties, we ran our workloads on seven different file systems: PMFS, Ext2, Ext3, Ext4, XFS, NILFS2 and F2FS. The distinguishing features across all the file systems are:

- Inode data structures: B-Tree vs. linear fixed size
- Block Size: Fixed vs. variable-sized extents
- Layout or update style: In-place update vs. log structured vs. hybrid
- Allocation strategies: Delayed vs. immediate, parallel allocation
- Journal modes: None vs. ordered vs. writeback vs. data
- Other features (e.g., atomic updates, XIP) designed for NVM

We evaluated the above file systems not only in their default modes but also using diverse mount and format options. Some of the options that we varied include different journaling modes, allocation policies such as delayed allocation, mechanisms to bypass buffer cache using execute-in-place (XIP) and some more options relevant to specific file systems. Table III compares the different properties of these seven file systems based on the factors given above. The last row of this table provides abbreviations of the file system variants used in our evaluation.

PMFS: PMFS [16, 33] is a lightweight POSIX file system that has been explicitly designed for NVM. It is an in-place update file system that bypasses the buffer cache and block layer (see Fig. 1). PMFS supports an important feature, called as *execute-in-place (XIP)* that allows direct I/O from NVM media. XIP [21] is a method of executing programs directly from storage media like ROM or flash memory rather than copying the data into DRAM. As XIP allows direct access to media bypassing page or buffer cache (shown in Fig. 1), it appears as an attractive option for NVM media. PMFS is characterized by atomic in-place updates to metadata, fine grained undo logging for consistency, large page support, and low overhead scheme of protecting the NVM from stray writes, called *write-protect*. We do not enable write-protect feature on PMFS for fair comparison across PMFS and traditional file systems on ramdisk, which lack this feature.

Ext2 and Ext3: Ext2 [22] and Ext3 [23] had been the default file system on Linux for years. There is a lot of similarity between ext2 and ext3 in terms of layout, inode structures, and free space management. Both ext2 and ext3 divide the underlying storage (disk or ramdisk) into fixed size block groups (BG). Each group manages its own free data block bitmaps, and inodes. The two file systems try to increase reference locality by keeping files contained within a single parent directory in the same block group. The maximum block group size is constrained by block size (4K). For our experimental setup, the `mkfs` utility sets the default number of block groups to 464, based on the ramdisk size and block size. We report all the numbers for this default block group value. Ext3 adds journaling support, whereas ext2 has no journal. Ext3 supports three types of journaling modes: data, ordered and writeback, with ordered mode being the default. We evaluate the performance of both ordered and data journal mode of the file system. As XFS supports writeback journal mode, by default, we do not experiment writeback mode in ext3.

We evaluate the performance of ext2 and ext3 when mounted with XIP feature enabled. In Linux, XIP is implemented by adding support to block device operations, and file system operations. A block device operation named `direct_access` is used to retrieve a reference to block on storage. The reference is supposed to be cpu-addressable physical address. The XIP-enabled file system needs to implement a special address-space operation named `get_xip_mem` that is used to retrieve reference to the page frame number (of the underlying media) and a kernel address (virtual address). The file system also implements special read, write function and page fault handler that make use of `get_xip_mem`. Currently, Linux ramdisk block driver and ext2 support XIP. We have added XIP feature support to ext3 file system (ordered mode) and used it for our evaluations. Note that XIP feature in ext2 and ext3 is only limited to data

	Ext2	Ext3	Ext4	XFS	NILFS2	F2FS	PMFS
Inode Structure	Linear	Linear	Hashed B-Tree	B+Tree	B-Tree	Linear	B-Tree
Block Size	Fixed	Fixed	Variable Extent	Variable Extent	Fixed	Fixed	Fixed (Page Size)
Layout / update style	In-place	In-place	In-place	In-place	Log structured	Hybrid	In-place
Allocation Strategy	Immediate	Immediate	Delayed	Delayed	Immediate	Immediate	Immediate
Parallel Allocation	Yes	Yes	Yes	Yes	No	Yes (multi-head logs)	No
Journal	None	Ordered, writeback, data	Ordered, writeback, data	Writeback	Not Applicable	Not Applicable	Fine-grained undo logging (only metadata)
Other Feature	XIP	XIP (added by us)			Continuous Snapshot	Multi-head log, adaptive cleaning	XIP, atomic update, large blocks
Variants evaluated (abbreviations)	ext2-buf, ext2-xip	ext3-buf, ext3-xip, ext3-data	ext4-buf, ext4-no-del	xfs-buf-4, xfs-buf-464	nilfs	f2fs_2, f2fs_4, f2fs_6	pmfs

TABLE III. File system feature set

operations i.e., while performing copy across user and kernel data buffers. Unlike PMFS, this feature and atomic updates is not applicable to metadata operations such as updates to inode or journalling in traditional file systems as it involves more changes in the file system code.

Ext4: Ext4 [12, 24] is an advanced level of ext3 with more scalability (maximum file size, number of files), and more features. In contrast to ext3 file system, ext4 is an extent-based file system that helps reduce metadata overhead. Further, ext4 employs improved allocation strategies such as multi-block and delayed allocation. While these features are attractive in the disk world, we wanted to determine their efficacy in case of non-volatile memory. Hence, we report the results of not only the default configuration of ext4 but also *without* delayed allocation feature.

XFS: XFS [34, 36] was designed for scalability: support terabyte sized files, unlimited number of files and large directories. XFS stores its data and metadata in variable sized extents. It adopts B+ Tree data structure for file/directory inodes, free space management and dynamic allocation of inodes. Similar to ext2/3 and ext4, XFS file system is divided into a number of equally sized chunks called as Allocation Groups (AG). Each AG manages the free space and inodes of its group independently. Thus, increasing the AG count scales up the parallel file system operations, improving its performance. In our setup, we found that the default AG count was set to 4. We increased the AG count of XFS to 464 – same as block group count of ext2 family file systems, and compare it against the default configuration. Similar to ext4, XFS employs delayed allocation policy to obtain large contiguous extent. It supports writeback journaling by default. XFS tracks AG free space using two B+Trees: (a) based on block number, and (b) the size of the free space block. While the free space management and support for unlimited files (using dynamic inode allocation through B+Tree) seem attractive for the disk environment, we guage its affect on NVM.

NILFS2: NILFS2 [32] is a pure log-structured (LFS) file system that supports continuous snapshotting. As NILFS2 creates checkpoints every few seconds or per synchronous writes, users can recover from any inconsistency or data loss quickly. Like any log-structured file systems data and metadata blocks once written, are not updated in place, until they are erased or garbage collected. NILFS2 volume is divided into a number of segments of 8MB (default) size, where each segment is a container of logs. Each log is composed of summary information, payload blocks and an optional superblock. The payload block consists of the file data and metadata (inode BTree). NILFS2 has a `segctord` kernel thread that is responsible for constructing these segments that are cached in memory and flushing them to the underlying media. We used the default configuration of NILFS2 and kept the garbage collector (`nilfs_cleanrd`) on.

F2FS: F2FS [3, 25] is a file system designed for flash media. The file system builds on top of LFS mechanism but solves some of its major problems. Firstly, F2FS solves the wandering tree problem i.e., propagation of index updates recursively from leaf nodes, to direct nodes to indirect nodes, and so on. This recursive propagation leads to a lot of copy and cleaning overhead. Secondly, in LFS the garbage collection process is expensive; under high disk utilization it impacts performance of the actual data access. F2FS solves the first problem with the help of *node address table (NAT)*, which involves an in-place update of metadata, while only writing the data in log-structured manner. Hence, F2FS can be considered as a hybrid file system. To solve the garbage collection problem, F2FS separates hot and cold data during block allocation. It runs multiple active log heads concurrently and appends data and metadata to separate logs based on their anticipated update frequency. Moreover, at high storage utilization, F2FS changes the logging strategy to threaded logging, where new data is written to free space in dirty segments without requiring actual cleaning process. By default the number of active logs is 6. We evaluate the performance of F2FS by changing the number of active log heads to 2 and 4 as well.

D. Limitations

In this work, we perform our experiments on ramdisk carved from DRAM. We do not simulate NVM slower than DRAM. We evaluate various file systems on NVM only from the perspective of performance. We do not examine other features such as instant durability, consistency, recovery point objective, and recovery time objective. According to us, all the file systems in our study, except ext2, can provide file system consistency, irrespective of the underneath media. This is because of the journal or log-structured nature of the file systems. As the buffer cache is not persistent, the data is not persisted to NVM media immediately, even if the write is acknowledged. This could result in data loss if there is a power outage. On the contrary, PMFS and other XIP configurations of file systems bypass buffer cache and write the data to NVM immediately. They also flush the hardware caches before acknowledging the user write. Thus, PMFS and XIP file system configurations (ext2-xip, ext3-xip) have a better recovery point objective or they cannot have data loss once the write is acknowledged, whereas buffered file system configurations lack this feature.

We use PMFS as a baseline for our comparison as it is the only open source NVM-aware file system available. We acknowledge that PMFS is not a production file system. As the observations and conclusions are tied to the features of different file systems, the insights from this work can be used to determine the set of features that can help improve the performance of a file system on NVM under the influence of different workloads.

IV. Evaluation

This section details our results and analysis of various workloads when executed on different file systems on simulated NVM environment. We abbreviated the default configurations of ext2, ext3, ext4, and XFS as ext2-buf, ext3-buf, ext4-buf and xfs-buf-4, respectively, in all our figures. XIP configurations of ext2 and ext3 are denoted as ext2-xip and ext3-xip, while data journal mode of ext3 is denoted as ext3-data. We denote ext4 without delayed allocation as ext4-no-del. The figures show XFS with 464 AG counts as xfs-buf-464, and F2FS as f2fs_2, f2fs_4 and f2fs_6, where 2,4 and 6 stand for the number of active logs. As we do not vary any options for NILFS2 and PMFS they are denoted as nilfs and pmfs, respectively. The last row in Table III summarizes all the file system variants as used in the figures. We ran all the experiments atleast three to five times and report the average readings.

A. Fileserver

Fig. 2 shows the results of fileserver workload on NVM. The y-axis denotes resultant operations per second in units of 1000. We see that PMFS performs the best amongst all file systems, while NILFS2 performs the worst. PMFS outperforms all file systems because of its fundamental design

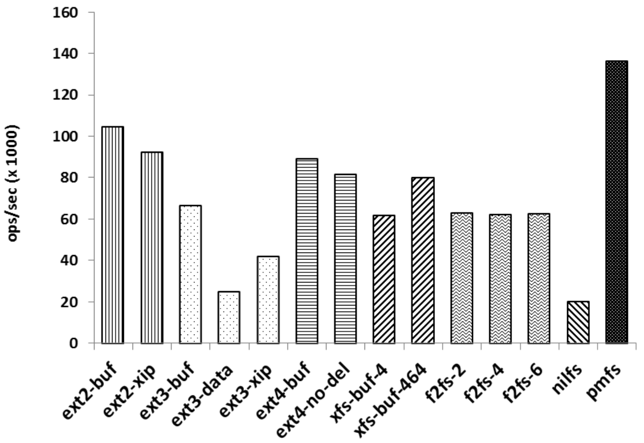

Fig. 2. Performance of file systems under fileserver workload

around non-volatile media. It's XIP support, atomic-updates, and fine-grained logging help reduce the latency of the read and write code paths. Moreover, PMFS adopts simple allocation and de-allocation policies, where it searches through a common linked list of unused blocks. Thus, fileserver workload, which is characterized by data to metadata ratio of 1:1, benefits from PMFS. It outperformed buffered file systems such as ext2-buf, ext3-buf, ext4-buf, xfs-buf-4 (default) by a factor of 1.3, 2, 1.5 and 2.2, respectively. Buffered configurations suffer because of double copy: (1) ramdisk to buffer cache, and (2) buffer cache to the user buffer. Ext2 performs the best amongst other buffered counterparts, as it does not have journal overhead.

As shown in the figure, the difference between ext3-buf and ext4-buf reduces from 34% to 23% if we repeat the same experiment with `nodelalloc` mount option, which disables delayed allocation. Since fileserver has create and append operations, delayed allocation helps improve the performance of ext4. To prove our point, we ran a micro-benchmark that was metadata intensive i.e., create-append-close, delete and stat operations only. Note that the fileset and directory depth for the micro-benchmark was same as that of fileserver. We found that in the metadata intensive benchmark, ext4-buf outperformed ext3-buf by 43%. However, after disabling delayed allocation feature, ext4-no-del performed only 25% better than ext3. Thus, delayed allocation is a helpful feature if workload involves significant data allocations, as it amortizes the allocation cost even on fast media such as NVM.

On increasing the allocation group (AG) count of XFS from default value of 4 to 464, improved the performance by 31%. This is because as the AGs increase, XFS' parallelism improves too, boosting the performance of operations involving data and metadata allocations. As discussed in Section III.C, we choose the value of 464 for AG count from the default BG count of ext4. The impact of AG count is more apparent in workloads consisting of metadata mix and writes (appends) involving allocation because these operations access and modify the AG descriptors frequently.

One thing to note is that after configuring AG count in XFS to 464 (optimal), it performed 10% worse compared to

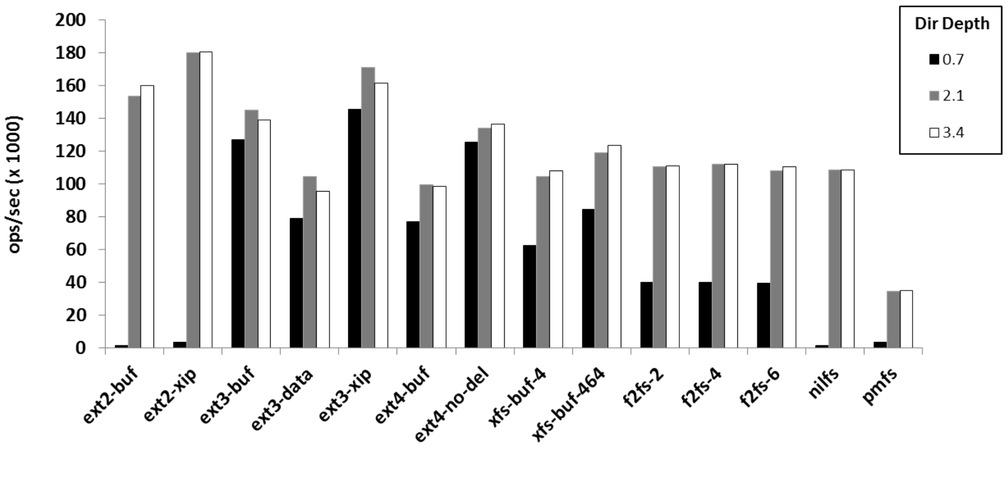

Fig. 3. Performance of file systems under webproxy workload

ext4-buf, whereas the default XFS (4 AG counts) performed 44% below ext4-buf. We found that although XFS employs features similar to ext4 such as delayed allocation strategy and extent-based mapping, it performed below ext4, because its inode and data allocation has a longer code path length than ext4. As XFS supports unlimited number of inodes, it employs B+Tree to manage dynamic inode allocation. Further, XFS allocates data blocks after looking up two B+Trees as discussed in Section III. On the contrary, ext4 uses a simple inode bitmap, similar to that on ext3/2 for inode allocation and de-allocation, thereby improving the performance compared to XFS.

Surprisingly, PMFS outperformed ext2-xip and ext3-xip by a large factor – 1.4x and 3x, respectively. This is contrary to results obtained in all other workloads (Sections IVB-IVE). We ran 2 micro-benchmarks: 1) complete data-intensive benchmark consisting of only reads and writes (appends) and 2) metadata-intensive micro-benchmark (mentioned in above paragraph), on ext2 and ext3 buffered and XIP configurations. We found that for data-intensive workload ext2-xip and ext3-xip file systems outperformed their buffered counterparts by 36%, but they perform equally worse compared to ext2-buf and ext3-buf in metadata-intensive micro-benchmark. We looked at the code and found that ext2-xip and ext3-xip are not pure XIP based file systems. XIP is applied only to the data portion i.e., during reads and writes from/to user/kernel buffers. However, metadata updates (inode, free space information), and journaling still follow the block-oriented methods during the time of persistence or reads. As PMFS follows XIP and atomic updates for both data and metadata, it outplays all other XIP- based file systems.

Ext3-data and NILFS2 performed 5x and 6x worse compared to PMFS. In ext3-data, journaling both data and metadata is redundant and affects performance on NVM file system. On analysis we found that `kjournald` consumed around 50% of CPU cycles, thereby impacting performance of ext3-data. Performance of ext3-data is limited by contention to its common journal used for both data and metadata logging. Due to the fast access speeds of NVM media, the journal gets filled up quickly, resulting in frequent flush operation of the journal. This proves that data journaling is not preferred for NVM file systems. Since NILFS2 is a pure log-structured file system, any write to a file or creation of files inside a directory leads to recursive updates to multiple data and metadata/index blocks resulting in a "wandering tree" problem [3]. This results in multiple obsolete data and metadata blocks aggravating garbage collection and adversely impacting performance. In case of NILFS2 we found that `segctord`, the segment constructor, took around 50-60% of the CPU. Thus, multiple fileserver threads tend to be bottlenecked by a single segment constructor kernel thread, when NILFS2 is used on NVM. The garbage collector, `nilfs_clearnd` also ran frequently consuming around 10-15% of CPU.

F2FS, which also follows log-structured approach for data, performed only 5% worse than ext3-buf. This is because F2FS is a hybrid file system – it follows both log- structured and update-in-place approach. It solves the wandering tree problem by using node address table (NAT), which is updated in place. Further, it supports 6 active log heads: separating out hot, cold and warm metadata and data, thereby improving GC efficiency. Due to these optimizations F2FS performs 3x better than NILFS2. Note that the number of active logs has no affect on performance of F2FS for fileserver workload. It performs well above NILFS2 with even just 2 active logs, i.e., by only separating data from metadata and treating latter differently. We wanted to verify if F2FS performs at par with buffered file system even under high utilization of ramdisk space. We reran the fileserver workload, increasing the total number of files to 200K, which increased the ramdisk utilization from 22% (with 100K files) to 45% (with 200K files). We observed that under high utilization F2FS performed 50% worse compared to ext3-buf. On further analysis, we found that the performance degradation was due to background garbage collection. Thus, background operations such as garbage collection and dirty data flush to backing store degrade the performance of file system as they deprive the active file system of CPU and memory resources.

Insights:

It is evident that update-in-place file systems or atleast supporting in-place updates for metadata is suitable for NVM as it helps utilize the CPU and memory resources efficiently. Note that PMFS is also an in-place update file system. While journaling is needed for consistency on NVM file systems, data mode journaling turns out to be expensive for such a fast media. Setting AG count of XFS appropriately for workloads involving multiple allocations (data and metadata) can help improve performance significantly. Delayed allocation is a helpful feature if workload involves significant data allocations, as it amortizes the allocation cost even on fast media such as NVM.

B. *Webproxy*

The black colored bars in Fig. 3 show the performance of various file system configurations under webproxy workload. As shown in the figure, ext3-xip performs the best across all the file systems for webproxy workload. Compared to the fileserver results, we observe a stark difference in relative file system performance for this workload. NILFS2 performs worst (100x) due to slow lookup on flat directories and same reasons discussed in fileserver. Surprisingly, ext2 variants and PMFS also exhibit abysmal performance. Ext3-xip performs 100x better than ext2-buf and 41-45x better than ext2-xip and PMFS. To analyze the reason, we tried changing various parameters of the workload and found that there were two workload features that contributed to the abysmal performance of certain file systems: flat namespace hierarchy or directory depth of 0.7 and large number of files (500K). In a directory depth of 0.7 (<1), all the 500K files are contained within one large directory. The performance of such a fileset depends upon how lookup and other metadata operations are handled in large directories. We found that ext3, ext4 and XFS use hash tree (indexed directory) to store directory entries while ext2, PMFS and NILFS2 do not, and hence they perform badly.

To prove our point we re-ran the Webproxy benchmark by keeping the total number of files same (500K), but increased the directory depth from 0.7 to 2.1 and 3.4 i.e., this reduced the number of files in one directory but increased the subdirectories. Fig. 3 shows the results for directory depths 0.7, 2.1 and 3.4 across different file systems in black, grey and white color bars, respectively. We see that ext2 variants start performing better than ext3 once the directory depth is increased. This is similar to our results in fileserver. The performance of NILFS2 and F2FS also improves by a factor of 80x and 2.5x, respectively, once we increase the directory depth.

The performance of PMFS improves with the increase in directory depth, but it is **5x** and **3x** worse compared to ext3-xip and NILFS2, even for deeper directory depth. PMFS displays the worst performance across all the file systems. We repeated the test but reduced the number of files to 100K and found that PMFS and ext3-xip perform almost the same (results not shown). Thus, it is evident that the performance of PMFS plummets in workloads involving parallel create/deletes and data allocations of large number of files. Unlike ext2, ext3 and XFS, PMFS lacks parallelism in data and metadata allocations. When a new inode has to be allocated, PMFS takes a big lock on the entire inode table and searches for unused inode. Further, for data allocation, it searches through a common linked list of unused blocks with one lock, thereby reducing its scalability. However, ext2 and ext3 employ features such as block groups, similar to allocation groups in XFS, which allows parallel allocations of data and metadata blocks, resulting in better performance than PMFS in this test.

Another important thing to note is that in contrast to fileserver, ext3-buf and ext4-no-del (without delayed allocation) perform similarly, only 15% worse compared to ext3-xip. Surprisingly, ext4-no-del performed 1.6x better than ext4-buf (0.7 directory depth). We analyzed and found that the main reason was large number (500K) of small sized files of 32K. On the other hand, fileserver workload consisted of only 100K files of average size 128K. Thus, delayed allocation becomes an overhead when there are a large number of parallel allocations required for small sized files. It is more beneficial for large sized files, when they are moderate in number.

Similar to fileserver workload, we observed an improvement in performance of XFS if we increase the AG count from 4 to 464; its performance improved by 13-35% for varying directory depths. Moreover, xfs-buf-464 performs around 48%, 12% and 10% worse compared to ext4-no-del for directory depths 0.7, 2.1 and 3.4, respectively. This is because lookup operation in XFS is much slower than ext4-no-del for large/flat directories.

F2FS performs 30x better than NILFS2 (0.7 depth) as the former adopts hashed-directory entries. But, it performance is 3x worse compared to other buffered configurations (ext3-buf and xfs-buf-464). We found that lookup and create operations is much slower in F2FS for large (flat) directories compared to relatively deep directory.

Insights:

It is clear from this experiment that enabling XIP feature on traditional file system helps boost their performance on NVM. Further, features like allocation group are of great importance in workloads involving parallel data and metadata allocations or metadata and data mix as it helps scale performance. Thus, it should be considered in file system design for NVM. While delayed allocation seems useful in creation of large sized files, it turns out to be expensive in workloads involving allocations of parallel large number of small sized files (<=32K).

C. *Webserver*

As shown in Fig. 4, PMFS performed the best amongst all file systems. Although the number of files is same as webproxy (500K), PMFS performs 17% and 20% better than ext2-xip and ext3-xip, respectively (contrary to the observation in webproxy). This is because, webserver is a read-intensive workload involving multiple lookups followed by full files reads and then appends to a common log file.

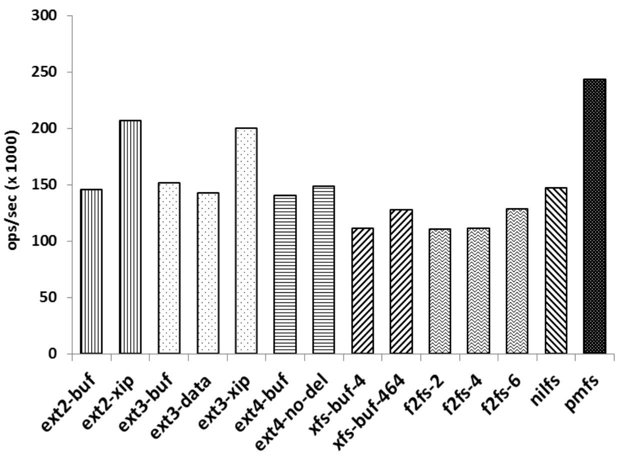

Fig. 4. Performance of file systems under webserver workload

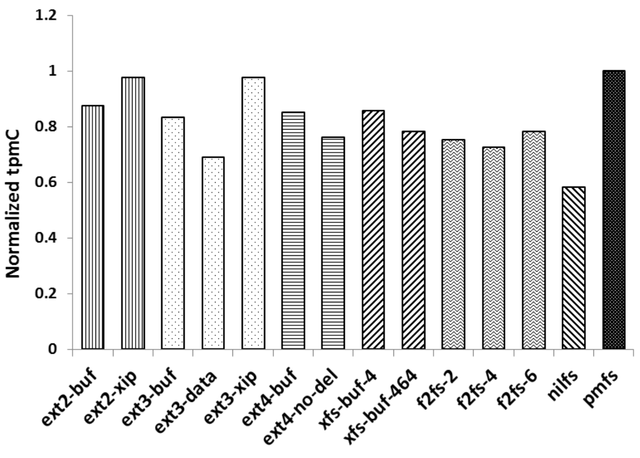

Fig. 5. Performance of file systems under tpc-c workload

Since webserver does not have any metadata operations such as create or parallel data allocations, the deficiency of PMFS with respect to parallel allocations is not exercised, resulting in improved performance. Moreover, ext2-xip and ext3-xip do not support XIP for metadata operations (lookup and updates); hence we see a difference between PMFS and other XIP file systems.

Due to read-intensive workload, buffered configurations of ext2, ext3 and ext4 and ext3-data perform close to each other: 60-70% worse compared to PMFS. Because of the same reason, NILFS2 and F2FS also perform close to buffer and data journal modes of file systems (around 10-16% difference). We see a vast difference between XIP and buffered modes of the file systems because of the extra copy overhead, as discussed in fileserver workload. We see an improvement of 15% when number of active logs is increased in F2FS from 2 to 6. Due to slow lookup operations (discussed in webproxy), xfs-buf-464 performed around 10-15% worse compared to other buffered file system configurations. Further, the performance of ext4-buf and ext4-no-del is close as the workload does not involve significant data allocations or file creations.

Insights:

XIP file system configurations (ext2 and ext3) perform close to PMFS and better than their default counterparts. Compared to the earlier workloads, for read-intensive workloads, the performance difference across various buffered configurations – both update-in-place and log-structured becomes small (10-16% only). Further, delayed allocation has no effect on such read-intensive workloads.

D. OLTP Database: TPC-C on MySQL

Fig. 5 shows the results of TPC-C when executed on MySQL using InnoDB storage engine. It shows the transactions per minute (tpmC) normalized with respect to PMFS. Compared to the workload results mentioned earlier, we observed two differences: (1) PMFS, ext2-xip and ext3-xip perform the best, but *close* to each other (less than 5% difference), and (2) relative performance difference across best file systems (XIP-based) and other configurations reduced significantly – 13-22% across ext2, ext3, ext4 and XFS variants, 32% and 42% compared to ext3-data and NILFS2, respectively.

To better understand the results, we tried to determine how MySQL transformed TPC-C workload to file system I/O requests. We analyzed the I/O and thread pattern of MySQL using `strace` and found that most read and write requests come to mainly three of the nine database files stock.ibd (size=14G), order_line.ibd (size=14G), customer.ibd (size=5G). Moreover, the read-write ratio as seen by the file system is 3:1. For every new client connection a MySQL server thread was created, which performed reads (SELECT) from database files in 16KB units and writes (INSERT/UPDATE) to common log file in 512 to 600KB I/O size. MySQL calls `fsync` after writing transactions to the log file. MySQL also performs regular checkpoint operation where it commits the data buffers from memory to database files in 16KB I/O size units. In summary, TPC-C on MySQL exercises data-intensive workload on the underlying file system and does not perform any allocations or metadata intensive operations.

We emulated the workload pattern detected above using Filebench and compared the performance results with those of TPC-C. We found that there was a performance difference of around 15-20% across PMFS and ext2/3-xip. Although the overall performance pattern looked the same as TPC-C, the relative difference across the file systems increased. On further investigation we found that the latency measured by TPC-C client threads include the data transfer across network socket, which takes around 12-15us. This value is significant compared to the latency of reads and writes in various file systems on NVM, resulting in reduction in relative performance difference.

As discussed in earlier workload sections, XIP-based file systems beat buffered, data journaled and log-structured file systems as in XIP the data is directly read or written from/to memory bypassing the page cache. Moreover `fsync` is

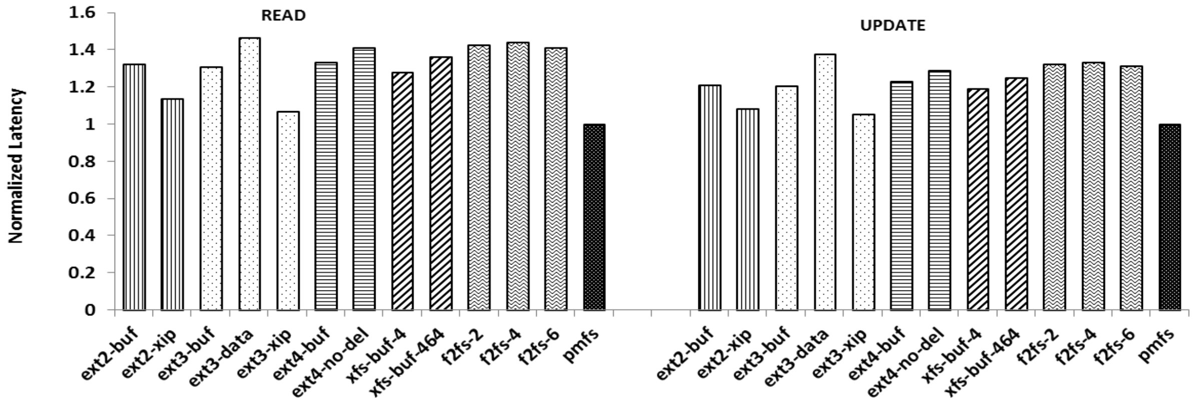

Fig. 6. File System performance for YCSB Workload A

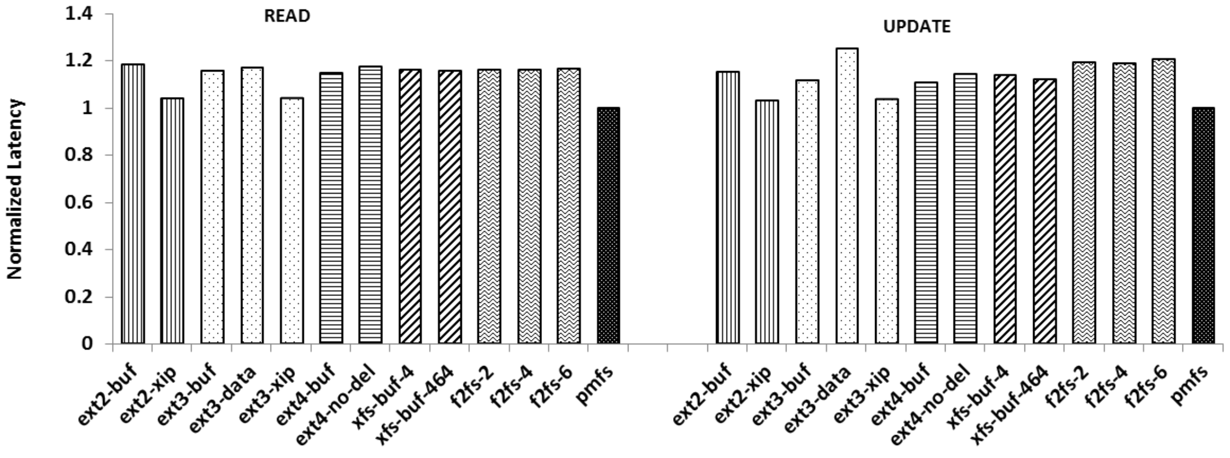

Fig. 7. File System performance for YCSB Workload B

lightweight in XIP file systems as it involves only flushing of the CPU caches, while in other file system configurations the data is actually copied from buffer cache to ramdisk. The reasons for relative performance of NILFS2 and F2FS remain the same as discussed in earlier workloads. Varying active logs from 2 to 6 does not impact F2FS performance. It is 22-25% worse compared to PMFS.

In case of XFS, when the AG count increases from 4 to 464, its performance plummets by 10%. This is contrary to the earlier results. We tried to analyze the reason for this anomaly using `xfs_db` - a tool that aids in debugging XFS. We found that to store a large file of size 5-14GB using 464 AGs, XFS split the file data to multiple extents in different allocation groups. As the size of a single allocation group was only 128MB the size of an extent could not grow beyond 128MB. To address these extents from different allocation groups, XFS inode required more indirections and entries. On the contrary, in case of 4 allocation groups, each AG is big enough to accommodate the whole file in one extent, thereby reducing metadata overhead. Increased indirections impact the lookup latency of a data block, thereby degrading performance.

Insights:

For data-intensive workloads, enabling just the XIP feature on in-place update traditional file systems (ext2/3) bumps their performance and they perform at par with PMFS. Allocation group count should be set appropriately based on the workload – mainly the file sizes, parallelism in data allocations, and CPU cores.

E. *Key-Value Stores: YCSB on MongoDB*

We ran YCSB [4] on NoSQL database – MongoDB, which performs memory-mapped I/O on its database files and file-io (`write` system call) to its journal. Thus, READ and SCAN operation result in memory loads while UPDATE and INSERT result in memory stores to database files and writes to journal. MongoDB calls `fsync` on its journal every 120 ms and `msync` on its database files every 60 seconds.

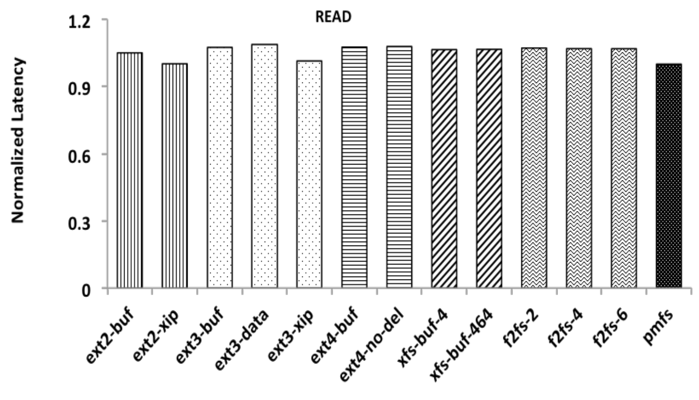

Fig. 8. File System performance for YCSB Workload C

Figures 6 through 11 present normalized latencies of various YCSB transactions under different workloads (A-F) for the maximum sustainable input load (IOPS) across various file systems. Note that contrary to previous graphs, higher is worse. We take the maximum throughput possible for the lowest performing file system and measure the latencies across all file systems for that input load. For example, in workload A, B and C the output throughput starts dropping after an input load of 20K ops/sec for the lowest performing file system ext3-data and F2FS. Thus, we fix the input load for Workload A, B and C to be 20K ops/sec for all the file systems and compare their latencies under this fixed load. Note that we do not report the results for NILFS2. Due to the log-structured nature of this file system, ramdisk formatted with NILFS2 gets filled up completely much before the test finishes. Thus, we could not run YCSB on NILFS2.

Workload A, B, C: Fig. 6 shows the results of Workload A, when READ-UPDATE ratio is 50:50 at an input load of 20K ops/sec. As shown in the figure, ext2-xip, ext3-xip and PMFS perform close to each other (having only 5% difference across them). All the XIP-based file systems perform the best for both READ and UPDATE. In case of READ all buffered file system configurations (ext2-buf, ext3-buf, ext4-buf, xfs-buf-4) experience 27-36% more latency compared to PMFS. F2FS and ext3-data experience an increase in latency by 41% and 46%, respectively. In case of UPDATE buffered file systems perform 19-20% worse whereas F2FS variants and ext3-data perform 31-37% worse compared to PMFS. The performance difference across ext4 and ext4-no-del is insignificant (5% only), as the workload does not involve any allocations. Similarly, we see no difference across XFS and F2FS variants.

These results are contrary to the results obtained in Filebench workloads (webserver), where there was atleast 15-20% difference across PMFS and other XIP file systems and significant difference across other configurations. We investigated and found that the main reason for this difference was the way MongoDB accessed its data i.e., memory-mapped I/O, whereas the Filebench workloads involved file-io i.e., open, read-write calls. As memory-mapped I/O performs loads and stores and bypasses the kernel and file system during data path, ideally the performance of all the file systems should be the same. However, we observe performance difference if (a) there are substantial number of page faults resulting in file system page fault handler, or (b) there is a lot of dirty data due to writes/updates resulting in flush operation to the backing store.

To corroborate this point, we ran two simple micro-benchmarks using fio [26]. The first micro-benchmark involved 100% random loads, while the second performed 100% random stores on a 2GB mmaped file. In both the cases, the I/O size was 1KB (same as I/O size in YCSB/MongoDB). We found that all the file systems performed similarly in case of 100% loads. This is because the cost of page faults amortizes across multiple load requests. In case of 100% stores, we observe no difference in performance across ext2-xip, ext3-xip and PMFS. Buffered file system configurations, ext3-data and F2FS observe a performance degradation of 2.4x-2.5x compared to XIP-based file system. These file system configurations experience such abysmal performance in case of 100% stores because of the flush daemon that tries to sync the dirty pages from buffer cache to ramdisk (block device). Further, in case of ext3-data, extra copy of data in the journal results in more performance degradation compared to ordered mode. On the contrary, XIP-based file systems write directly to the media, bypassing the buffer cache and eliminating this overhead.

Comparing the results of Workload A, B and C, in Figures 6 through 8, we observe that as the percentage of UPDATE decreases in YCSB workloads, the relative difference across file systems also decreases for READ transactions or they perform almost the same. Hence, in Workload C, which is a 100% READ workload we see a maximum of only 8% performance difference across few buffered configurations and PMFS (similar to the results obtained from our micro-benchmark involving 100% loads). However, as we increase the UPDATE percentage – 5% in case of Workload B to 50% in Workload A, we see a performance difference across XIP-based and other file system configurations. Thus, on low latency media such as NVM, UPDATE operation influences the performance of READ as well. Hence, we see a difference in performance in case of READ transactions for Workload A and B. We repeated these experiments on hard disk and found no such co-relation across different types of transactions. Thus, we conclude that in case of low-latency media the slowest performing function or feature can come in the way of other better performing features.

Workload D: Workload D exercises READ-INSERT ratio of 95:5 and follows latest distribution. As seen in Fig. 9, all the XIP and the buffered file systems have similar latencies both for READ and INSERT at an input load of 15K ops/sec. Only ext3-data performed 34% worse compared to other file systems in case of INSERT. We performed two simple YCSB workload experiments – (a) 100% READ, (b) 100% INSERT, with latest distribution. We observed that all the file systems performed similarly for both READ and INSERT, except ext3-data; it performed 35-47% worse in case of 100% INSERT only. This is because MongoDB performs loads and stores, thereby diminishing the effect of various file systems, as discussed earlier. As data journaling introduces extra overhead, ext3-data performs the worst in case of INSERT.

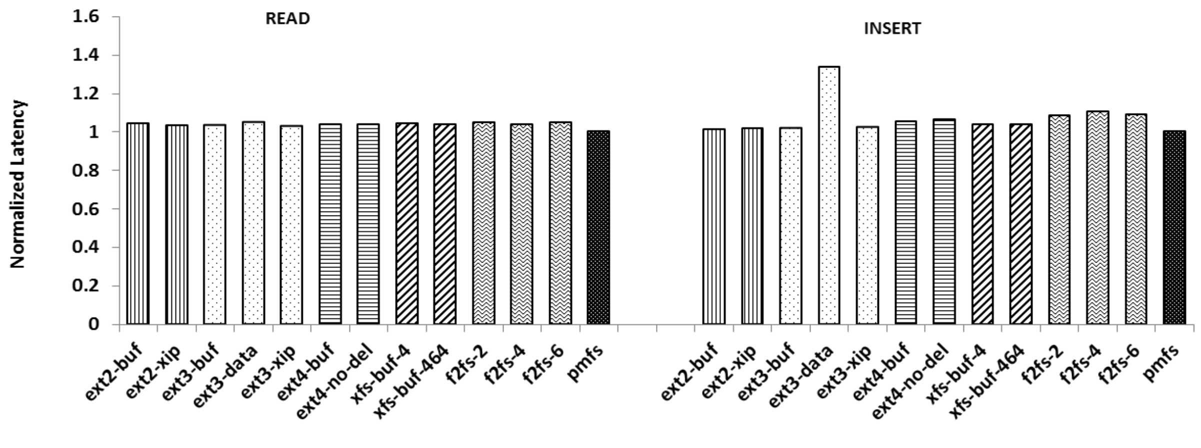

Fig. 9. File System performance for YCSB Workload D

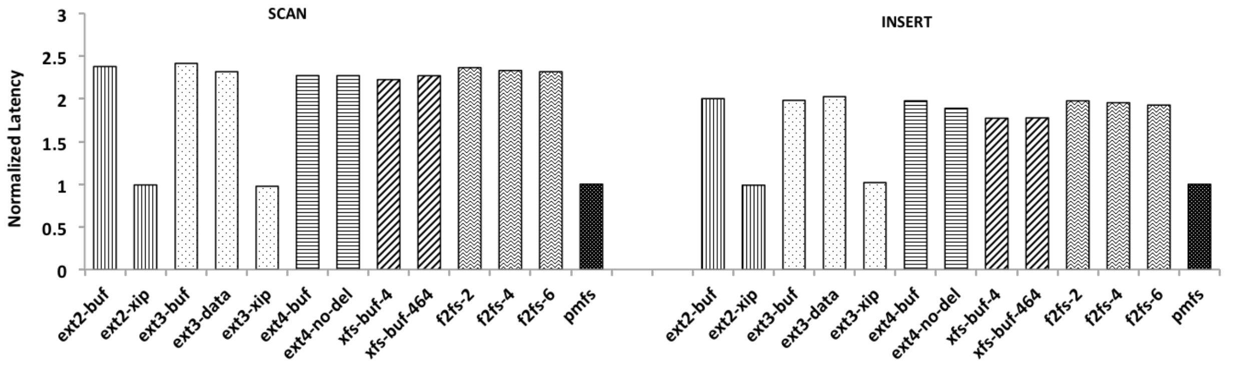

Fig. 10. File System performance for YCSB Workload E

One thing to note is that although both YCSB UPDATE and INSERT operations involve stores, we observe some performance difference across file systems in case of UPDATE (Workload A, B), but absolutely no difference in case of INSERT (except ext3-data). We tried to analyze and found that INSERT on MongoDB results in file appends. But, UPDATE result in read-modify-write type of operation, which consequently increases the number of page faults, around **13x** more than INSERT operation. Thus, we see performance difference across file system configurations for UPDATE but not INSERT.

Workload E: This workload consists of range SCAN and INSERT at 95:5 ratio. We exercise an input load of 5K ops/sec. The operation distribution is zifpian and the scan-length distribution is uniform over a range of 0-100 scan length. Similar to other YCSB workloads, XIP-based file systems perform the best as shown in Fig. 10. To our surprise, all other file systems performed 2.27-2.42x worse in case of SCAN, although it involves loads. This is contrary to our results in Workloads A, B, C and, D, which were dominated by READ operation. In order to determine the difference between YCSB READ and SCAN operations, we reran 100% SCAN and 100% READ workload and measured the number of page faults introduced by each workload. We found that SCAN operation exercised **62x** more page faults than READ operation. In XIP-based file systems page fault results in simply mapping the NVM area to the virtual address space, whereas, in other file systems it involves an extra copy of data from ramdisk to the memory-mapped area or buffer cache. Thus, increased number of page faults contributes to the latency difference across XIP and other non-XIP file system configurations in case of SCAN operation.

Contrary to our observation from Workload D, we see significant difference across XIP and buffered file systems for INSERT transaction. This is because, the dominating operation (SCAN) influences performance of INSERT as well in NVM environment. Similar to Workload A and B, we observe the slowest operation influencing the performance of the other well-performing operation on NVM.

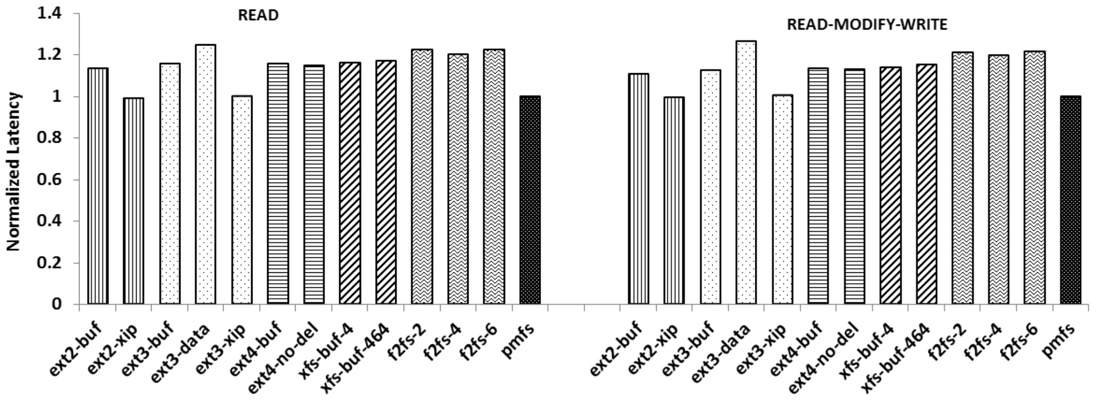

Fig. 11. File System performance for YCSB Workload F

.

Workload F: Fig. 11 shows the results of Workload F. This workload incorporates a new type of operation i.e. READ-MODIFY-WRITES (RMW), where it reads a key, modifies the same and write it back to the database. Note that RMW is similar to UPDATE. The ratio of READ to RMW is 50:50. Due to similarities in read-write ratio between Workload A and Workload F, we see a similarity in latency pattern across these two workloads. As the reasons for performance difference remain the same (as explained in Workload A), we do not discuss it here.

Insights:

It is evident from the six different workloads of YCSB that in case of applications/workloads involving memory-mapped operation simply enabling XIP feature on traditional file systems such as ext2 and ext3 boosts their performance so much so that they perform at par with PMFS, which is solely designed for NVM. The extra copy of data in case of buffered-mode of the file systems and data journaling turns out to be redundant and hence impact performance adversely in memory-mapped applications if they have a mix of loads and stores.

V. Recommendations and Conclusions

Based on our study we found that traditional file systems can be tuned to perform better than their default setting on NVM. In some cases these fine-tuned file systems perform at par with PMFS. Further, we found that features that help improve CPU and memory utilizations turn out to be better performing than others on NVM. PMFS, which is an NVM-aware file system, has most of the features that leverage the byte-addressability and low latency characteristics of the media. But, if one wishes to use a traditional file system with minor modifications or reconfigurations, we recommend few file system features that can help improve its performance on NVM. Few of the features include:

1. **In-place update layout:** In-place update layout is preferred for non-volatile memories as updating in the same place reduces extra copies of data and metadata, thereby obviating the need to garbage collect obsolete blocks of data and metadata (unlike log-structured file systems). Moreover, it also helps leverage the random access feature of NVM. Thus, in-place update layout of file systems result in better utilization of CPU and memory, thereby improving their performance. Although log-structured layout provides with features such as snapshots, it is not a preferred layout from performance perspective on NVM.
2. **Execute-in-place (XIP):** In-place update along with XIP helps leverage the NVM media to its fullest extent. It helps bypass extra layers in the software stack, such as buffer cache, which is redundant in case of NVM. Moreover, as discussed in ext2 and ext3, adding XIP support only to data operations (i.e., read and write) is fairly simple and involves minimal changes in the file system code. One of the functions that need to be implemented is `get_xip_mem`. While accessing both data and metadata using XIP and atomic updates is optimal (PMFS), enabling XIP feature only for data also helps improve performance of traditional file systems and they perform close to PMFS. We have shown the benefits of XIP on ext2 and ext3 only, but it can be extended to other in-place update file systems as well.
3. **Simple and parallel allocation strategy:** For workloads involving metadata and data mix, it is advisable to allow parallel allocations of both data and metadata (inodes). Having a simple linked list protected by a single lock for allocations is not efficient and does not scale beyond a few numbers of files and allocation requests. Instead, a feature such as Allocation Group is favorable to scale the performance of file system on memory. Moreover, the code path for allocation should be simple and short. Optimizations added for the disk world, such as co-locating the data based on earlier data blocks or inodes or size of block (XFS), unnecessarily increase the code path length, thereby degrading performance. Delayed allocation can help improve performance even on NVM for workloads involving multiple parallel data and metadata allocations of large sized files, as it amortizes allocation cost.
4. **Fixed sized data blocks:** Extent-based file systems are favored in disk world, but on NVM these turn out to be expensive; they take more memory and CPU resources for

merging and balancing the extents if the workload involves a lot of writes or appends. The lookup operation (directory search) also takes more time in case of large directories. On the other hand, using fixed size blocks, results in simple index based inodes that are cache friendly, and helps improve the performance of NVM file systems (e.g., PMFS, ext2, ext3).

ACKNOWLEDGMENT

We would like to acknowledge various NetApp ATG members who helped improve the work through many brainstorming sessions and reviewing the paper. We would also like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] A. Caulfield, A. De, J. Coburn, T. Mollov, R. Gupta, and S. Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, 2010.

[2] A. Caulfield, T. Mollov, L. Eisner, A. De , J. Coburn, and S. Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, March 03-07, 2012.

[3] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST'15, pages 273-286, 2015.

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

[5] E. Lee, H. Bahn, S. Yoo, and S.H Noh. Empirical Study of NVM Storage: An Operating System's Perspective and Implications. In IEEE 22nd International Symposium of Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2014.

[6] H. Volos, A. J. Tack, and M. Swift. Mnemosyne: Lightweight Persistent Memory. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2011.

[7] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. Swift. Aerie: Flexible File-system Interfaces to Storage-Class Memory. In Proceedings of the 9th European Conference on Computer Systems, EuroSys, 2014.

[8] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non volatile Main Memory. In TRIOS, 2013.

[9] 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 the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011.

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

[11] J. Yang, Q, Wei, C Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST), 2015.

[12] M. Cao, S. Bhattacharya, and T. Tso. Ext4: The Next Generation of Ext2/3 Filesystem. In Linux Storage & Filesystem Workshop, USENIX Association, 2007.

[13] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. ACM Transactions on Computer Systems, 1992.

[14] M. S. Bhaskaran, J. Xu, and S. Swanson. Bankshot: Caching Slow Storage in Fast Non-volatile Memory. In Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW), 2011.

[15] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Nonvolatile Byte-addressable Memory. In Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST), 2011.

[16] S. R. Dulloor, S. Kumar, A. Keshavamurth, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In Proceedings of the 9th European Conference on Computer Systems, 2014.

[17] S. Peter, J. Li, D. Woos, I. Zhang, D. Ports, T. Anderson, A. Krishnamurthy, and M. Zbikowski. Towards High-Performance Application-Level Storage Management. In 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2014.

[18] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST) 2014.

[19] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[20] EverSpin Second Generation DDR3 Compatible STT-MRAM. http://www.everspin.com/products/second-generation-st-mram.html

[21] Execute-in-place Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/filesystems/xip.txt

[22] Ext2. https://www.kernel.org/doc/Documentation/filesystems/ext2.txt

[23] Ext3. https://www.kernel.org/doc/Documentation/filesystems/ext3.txt

[24] Ext4. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt

[25] F2FS. https://www.kernel.org/doc/Documentation/filesystems/f2fs.txt

[26] Flexible IO (fio) Tester. http://freecode.com/projects/fio.

[27] Filebench. http://sourceforge.net/apps/mediawiki/filebench.

[28] H. Hellmold. Emerging NVM – Enabling Next-generation Data Storage Solutions. In FlashMemory Summit 2014. http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140807_301C_Hellmold.pdf

[29] International Technology Roadmap for Semiconductors, (ITRS): Emerging Research Devices, 2013.

[30] Linux PMem Library. https://github.com/pmem/linux-examples

[31] MongoDB. http://www.mongodb.org

[32] NILFS2. https://www.kernel.org/doc/Documentation/filesystems/nilfs2.txt

[33] Persistent Memory File System. https://github.com/linux-pmfs/pmfs

[34] SGI. XFS Filesystem Structure. http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf

[35] Transaction Processing Performance Council, TPC-C, an online transaction processing benchmark. http://www.tpc.org/tpcc/

[36] XFS. https://www.kernel.org/doc/Documentation/filesystems/xfs.txt