

AC-BPFS: A File System for Tiered Storage

Aishwarya Ganesan

ag@cs.wisc.edu

Saket Saurabh

ssaurabh@cs.wisc.edu

Siddharth Suresh

siddharth@cs.wisc.edu

Udip Pant

upant@cs.wisc.edu

Abstract

Fast non-volatile memories (NVM) are byte addressable, offer latencies close to DRAM and have densities better than DRAM that allow applications to persist their data close to memory latencies, but at a higher predicted cost per byte. The I/O latency of a disk-based filesystem is higher than a pure memory-based filesystem, but memory-based filesystems are limited by their storage cost. We propose a hybrid approach, where we use NVM as a primary storage and flash/disk as secondary storage. This paper provides the design and implementation of a file system for a persistent storage tier that includes both NVRAM and Disks.

1 Introduction

Most modern systems have been built with the disk or SSDs as the persistent storage medium and the DRAM for temporary fast byte-addressable memory usage. The data that resides in volatile memory can be lost on crashes or power failure. Moreover, with the emergence of modern storage technologies and new byte addressable persistent memory technologies which offer much lower write and read latencies compared to flash memory at a slightly higher cost, it is imperative that future systems would move towards the usage of byte-addressable non volatile memory (Non Volatile Memory) as the primary storage with the disk as the secondary storage medium. There has always been a trade-off between data durability and data read/write performance when deciding between a fast, volatile storage medium like DRAM versus a slow, persistent storage medium like disk. Persistent memory technologies like phase change spin-torque transfer RAM (STT-RAM), phase change memory (PCM), resistive RAM (ReRAM), and 3D XPoint memory technology can solve both of these problems.

One of the earliest file systems to be proposed for non-volatile memory systems was the byte-addressable persistent file system (BPFS) [3] by a group of researchers working at Microsoft Research. [3] showed that simply running a disk-based traditional file system on top of persistent memory is not enough to provide high performance. It proposes a tree based data structure to store

the file system meta data and also assumes that the hardware is capable of providing in place atomic updates of 8 bytes. A new technique called short-circuit shadow paging has been introduced to perform consistent updates to metadata, which is similar to the copy-on-write. BPFS achieves fine-grained access to persistent data at a much improved performance, while at the same time providing strong reliability and safety guarantees. But since BPFS has been built primarily for NVM as the persistent store and the cost/byte of NVM could potentially hinder such a filesystem from being used in commodity systems that do not have sufficient NVM storage capacity and are built with disks or flash as the persistent data store. Hence, AC-BPFS builds on the existing BPFS implementation thus leveraging the benefits of BPFS, but also provides a modified file system that outperforms existing file system implementations in the presence of a storage tier that consists of both NVM and disk or flash.

Since the capacity of NVM is limited and since all data blocks cannot be stored in NVM, we have explored the usage of Anti-Caching [4] as a mechanism to move data blocks between NVM and disk. The primary motivation of using Anti-Caching is to reduce write latencies, since all writes are write-back and are persisted in NVM, but at the same time does not worsen the read performance as well, due to minimal additional overhead. An Anti-Cache system uses memory as the primary storage and disks as the secondary storage and is designed to evict cold data actively. It has only one copy of the data, which is either in NVM or on disk. It thus entails the storage of the more frequently accessed data blocks in NVM and stores the rest of the blocks on disk or flash. In addition to Anti-Caching, we also move blocks that are read from disk to NVM. This approach is very similar to the virtual memory swapping in operating systems wherein when the amount of data exceeds available memory, cold blocks are written to disk giving an impression to the file system that the amount of memory in NVM is infinite but at the same time provides the best possible performance with the amount of NVM available.

AC-BPFS provides a flexible file system design that scales itself well irrespective of the size of the storage tier, i.e capacity of NVRAM and capacity of disk

or flash. The modular implementation further permits users to modify individual components and apply different policies for anti-caching and disk management depending on the requirement. AC-BPFS has been evaluated thoroughly and evaluated against the original BPFS and also other existing file systems.

2 Background

In this section, we will briefly discuss the relevant background on BPFS [3], a file system for byte addressable persistent memory. BPFS is one of the earliest file systems to be proposed for non-volatile memory systems and the authors showed that simply running a disk-based traditional file system on top of persistent memory is not enough to provide high performance. We will first introduce the BPFS file system layout and also discuss why the design is limited in terms of the cost of storage and volume of storage. We will also briefly describe the techniques proposed in BPFS to achieve consistency and how they achieve atomicity and write ordering. We then describe the changes made to the persistent data structures and file system code to support a disk backed secondary storage in a subsequent section.

2.1 BPFS Layout

BPFS stores all the file system meta data and data in the form of a tree structure consisting of fixed-size blocks of 4KB in the non-volatile memory. Figure 1 depicts the file system organization in BPFS. The BPFS file system consists of three kinds of files namely inode file, directory file and data file with each of these in turn arranged as a tree. The inode file is represented by a tree structure with the root of the inode file serving as the root of the file system. The location of this root node is stored at a known location in the non-volatile memory. Each of the leaves in the inode file contains an array of inode structures with each inode structure representing either a file or a directory. This inode structure contains the pointer to the root of the file or directory that the inode represents. The leaves of a data file tree are the data blocks and the leaves of a directory file tree are the directory entries.

2.2 File System Consistency

BPFS maintains file system consistency by a technique called short-circuit shadow paging. Shadow paging is a copy on write technique, where to modify a node in the tree a copy is made and the data is written to the copy. This requires updates of indirect pointers, which in turn triggers copy on writes on these indirect pointer blocks and this bubbles up to the root of the file system. Short-circuit shadow paging is an optimization, where the byte-addressability and atomic 64-byte write guarantee provided by NVM, makes in-place updates feasible. Hence, in-place updates are performed whenever

possible. Short-circuit shadow paging guarantees that file system operations will commit atomically and in program order. Ordering is provided by hardware primitive called epoch barriers. In a disk based file system, for the writes to be durable, we need to flush the operating system buffer cache. Similarly, for the data to be durable in NVM, we have to flush the CPU caches. BPFS provides the guarantee that data is made durable as soon on *fsync* operation by flushing the CPU caches.

NVM storage cost is higher and so we are limited by the available NVM size. Hence any file system built completely on NVM will be limited by the storage available. Therefore, the storage capacity of BPFS is limited.

3 Design and Implementation

Our goal is to build a file system for a tiered storage hierarchy consisting of disks and NVM that combines the low latency of NVM and low cost of disk-based storage without regressing on performance while offering similar consistency guarantees as BPFS. This seems like a reasonable goal, however, it raises the some non-trivial questions about the overall design of such a tiered filesystem:

- How do we make changes to BPFS data structures so that it is able to locate the data in-memory or disk?
- What policies determine when to move data?
- Do we handle the disks directly or do we go through another file system to store data on disk?
- What is the ideal granularity at which we should move the data?

We provide answers to these questions in this section and discuss at length, the policies and mechanisms required to implement the same. Figure 2 depicts the overall design of our system. We use NVM to store the filesystem metadata and all the filesystem operations are always done within the NVM tier. To preserve the filesystem consistency guarantees, we have opted for a simple design policy where only the data blocks are moved to the disks, while the metadata blocks always reside in the persistent NVM. Such a policy entails that for a given BPFS file, its data may be split across NVM and the disk. However, it is to be noted that even though a data block may be moved between the NVM and the disk, there will always be a single valid copy of that data block- whether in the disk or in the NVM.

The mechanism to determine the exact location of the valid copy of a given data block is through the use of the most significant bit(MSB) of the indirect pointer, pointing to the data block in the BPFS tree structure. This is also being depicted in the Figure 2. If the MSB is not set, then the rest of the 63-bits denote a memory address of the data block within the NVM itself. While on the

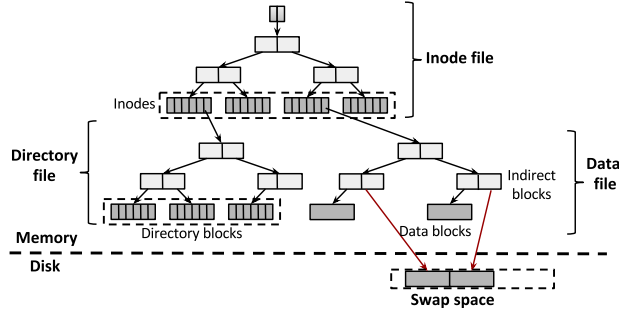


Figure 1: **BPFS tree structure.** The figure shows the file system organization of BPFS, along with our proposed modifications.

other hand, if the MSB is set, then the rest of the 63-bits denote a disk address where the data block was moved to on the disk. This answers the first question that we had raised earlier. In the following subsections, we discuss the design and implementation of Anti-Cache Manager and the Disk Manager that provide answers to the other design questions.

3.1 Anti-Cache Manager

In our design, NVM is the primary data store where all filesystem operations are being performed and hence, it contains the more frequently accessed blocks. We refer to such data residing in NVM as the ‘hot’ data. Since the capacity of the persistent NVM is limited, we have to periodically evict blocks from NVM to disk, so that we can make space for more ‘hot’ data. The evicted data on the disk is referred to as the ‘cold’ data. While it is true that some of the ‘hot’ data may become ‘cold’ over time, it is also possible that some ‘cold’ data will suddenly become ‘hot’ and it has to be brought back in NVM. This is what we term as the ‘anti-caching’ phenomenon- this gameplay between the ‘hot’ and the ‘cold’ data having a reverse caching effect. This anti-caching is at the heart of our implementation that enables us to amortize the cost of read-write operations equivalent to those provided by BPFS. We implement the well-known LRU-1 policy to distinguish the ‘hot’ data blocks from the ‘cold’ data blocks. We maintain a linked list of all the data block references. Whenever a data block is accessed, it is moved to the front of the linked list. Over time, the tail of the list will contain block references which have been least frequently accessed, while the head of the list will point to data blocks being most frequently accessed.

The Anti-Cache Manager component of the AC-BPFS filesystem implements the above proposed design policies. It runs as a separate background thread that continuously receives from the main AC-BPFS thread a stream of NVM data block references, in the order they are being accessed. It maintains an LRU data structure within itself and uses the LRU to evict a number of blocks according to the policy explained above. A soft threshold count of data blocks, known as ‘anti-caching threshold’, is used to determine when and how many blocks to evict.

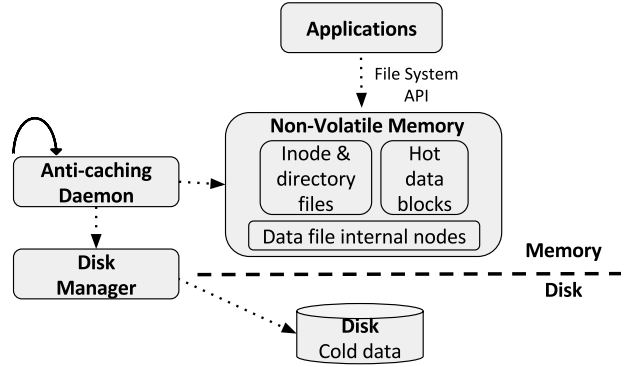


Figure 2: **Design Architecture of AC-BPFS.** The figure shows the overall design of our system.

The Anti-Cache Manager is made to poll arbitrarily every 10 ms on the size of the LRU list. If the size of the LRU list exceeds the specified anti-caching threshold of data blocks count in NVM, those many extra data blocks over the threshold are evicted from the tail of the list. This anti-caching threshold is set based on the latency of NVM writes and the polling time. We found that setting the anti-caching threshold to 80

This subsection answered our second question on the policies required to move data between NVM and the disk. The next subsection gives the answers to the remaining two questions.

3.2 Disk Manager

The Disk Manager component simulates a virtual disk using a file stored on disk in the BPFS partition of a pre-determined size. It exposes APIs for reading and writing blocks into the disk individually and in bulk. It performs prefetching of additional blocks when a read is performed. The lifetime of the prefetched blocks in memory is predetermined to ensure that unused prefetched blocks are cleaned up regularly with the help of a background thread. To avoid fragmentation, a free list of blocks that have been freed is maintained. Since the disk manager writes blocks to a file, the buffer cache is used to buffer writes until all the blocks are written into the file. Our current implementation provides consistency at the expense of performance by notifying BPFS after persisting the blocks, which in turn updates the indirect pointer in the inodes to point to the new blocks through a blocking call.

The disk manager also exposes another write API to the Anti-Cache Manager to facilitate writing one block at a time, for faster random writes since any free block on disk can be used for writing and does not require sequential locality. To further improve performance in the case of bulk eviction of blocks from the anti-cache manager, the disk manager performs an explicit flush only after writing all the blocks on to the disk.

3.3 File System Operations

In this section we describe the workflows for various file system operations.

3.3.1 Read

When a read operation is performed on a file, the BPFS crawls the inode tree to the indirect block containing the pointer to the data block, identifies the location of the data block using the MSB that indicates whether the block is on disk or in NVM, then fetches the block into memory and asynchronously writes it back into NVM. Once the block is persisted in NVM, it frees the block on disk and updates the indirect pointer in the inode tree to point to the new block location in NVM atomically.

When a read operation is performed on a directory, the BPFS crawls the inode tree and the directory file blocks are loaded into memory from NVM and the necessary updates are made into the directory entry cache. It then traverses all the necessary inodes in the directory file and reads them from NVM.

3.3.2 Write

When a write operation is performed on a file, the BPFS traverses the inode tree to find the indirect block containing the pointers to the data block of the file and identifies the location of the data blocks. If the data block is in NVM, the write operation is performed in-place and has no additional overhead. If the data block is on disk, the block is fetched into NVM thus performing a copy on write of the data block and this block is updated and written back to NVM. The block on disk is then freed and the control comes back to the BPFS crawler. The indirect pointer in the inode tree is then updated to point to this new block location in-place along with in-place writes of the other internal nodes. The inode access time is also updated through an in-place atomic write.

3.3.3 Open

When a file is opened, the directory inode tree is parsed to find the file. If the file is not found and a create is requested, a new inode number is generated and the new inode is written to the inode file. Then the directory file is then updated. All these updates are performed in place. Since the metadata updates are in-place and in NVM, they are faster than usual disk writes and the additional overhead of metadata journaling can also be avoided by avoiding disk writes.

4 Evaluation

We compare the performance of this system with that of BPFS and ext4 file systems to evaluate its design and implementation. We use the performance of BPFS as a baseline and focus on following key questions:

- What is the cost and overhead of the anti caching mechanism on top of the BPFS system?
- How does the degree of hotness and coldness of stored data affect the performance of the system?
- How does this file system perform for different cache eviction threshold? (Improve this)

4.1 Method

Making a meaningful performance comparison of this system with both the memory based file system such as BPFS and the disk based file system such as ext4 presents challenges at different levels. We cannot make comparison in isolation since our implementation is a hybrid approach that incorporates both systems. Another difficulty is that for a disk based file systems, such as ext4 that runs in the kernel layer, several parameters and behaviors are opaque to users and we have very limited control on enforcing the pure disk-based behaviors. For instance, ext4 filesystems performs several operations in memory and commit to the disk only periodically.

For the purpose of the evaluation, we run our experiments on a real hardware with 8 CPU cores running RedHat Linux Operating Systems with 16 GB of memory, 500 GB of Solid state Drive and 1 MB cacheline. We simulate the behavior of NVM on the DRAM due to unavailability of the hardware. Unless otherwise noted, both the original and our implementation of BPFS are mounted in the memory with 2 GB image. The anti-caching daemon is set to run at 10 ms by default with caching occupancy threshold of 40 MB. These values were chosen by performing empirical analysis.

4.2 Experimental evaluations

In this section we present the experimental evaluations of this file system using different micro and macro benchmarks and throughput evaluations and offer comparisons with those of BPFS and ext4 file-systems.

4.2.1 Microbenchmark: Various File Operations

In Figure 3b order to ensure the correctness of various file operations and to measure cost of additional overheads due to added behaviors such as anti-caching, we benchmarked 40 different file operations. We used 'timeit' package in Python programming language to measure the time for these operations. Table 5.1 shows a representative sample of those operations for both AC-BPFS and BPFS. The table shows that for the file operations the added overhead is negligible, if any at all. We also performed similar benchmarks for all the file operations in the ext4 file system. Since the time taken for each of the file operations were in the order of one tenth of a millisecond without fsync and in the order of 10 milliseconds with fsync enabled, the benchmarks for

Operations	Anti Cache BPFS(ms)	BPFS(ms)
chmod	5.26	5.19
create	6.49	6.45
link	6.21	6.24
mkdir	6.61	6.55
read	4.67	4.61
readdir	3.24	3.1
rename_dir_intra	8.3	7.61
rename_file_inter	8.6	7.82
rmdir	6.66	5.95
symlink	6.58	6.46
unlink_16M	10.03	10.42
unlink_hardlink	5.5	5.5
write_1M_4k	6.91	6.22

Table 1: **Microbenchmark.**

the operations in the ext4 file systems are not shown in the table.

4.2.2 Microbenchmark: Random write

. We evaluated the performance of making random write operations for various data size for AC_BPFS, BPFS and ext4. For the ACBPFS, to ensure uniform distribution of write operations to data blocks in both the persistent memory and the disk, we pre-generated a list of files of various sizes and random offsets with in those files. We ensured that data chunks written at random offsets exceeded the pre-specified cache threshold to force seek from the disk. We called 'fsync' after every write call to force the data to disk so as to make even comparisons across the filesystems. We used UNIX 'clock_gettime' system call to measure the time for this benchmark.

Figure 3b shows the results for this micro-benchmark. Both AC_BPFS and BPFS are faster than the ext4 in orders of magnitude, as we hoped. It also shows that AC_BPFS, although starts to lag behind the BPFS as the size of the data being written increases, it does not lag significantly, that reinforces our initial intuition and supports our objective.

4.2.3 Microbenchmark: Append

We measured the cost of appending data chunks of various size to an empty file. As with the benchmark for the random write, we called fsync after every write call to ensure that the data gets written to the disk. We used UNIX 'clock_gettime' system call to measure the time for this benchmark.

As with random write, Figure 3a shows that both AC_BPFS and BPFS are faster than the ext4 in orders of magnitude. Since the writing of all the data for the append operation happened entirely in memory, ACBPFS and BPFS performance lines are virtually inseparable. This result justifies the anti-caching mechanism that we have employed and also illustrates its low overhead.

4.2.4 Microbenchmark: Degree of coldness

This microbenchmark demonstrates the effect of the anti-caching and plots the read latency of reading 400

KB data against the degree of coldness. For the benchmark, we read a 400 KB chunk of data, part of which can be hot/cold depending on the degree of coldness. A 20% coldness factor on the X-axis for a 400 KB chunk data size would mean that 80 KB of the 'cold' data resides in the disk, while 320 KB of 'hot' data resides in NVM. So as expected, the read latency for a 20% cold data would be less than the read latency for a 80% cold data.

At a 100% level, the performance will flat-line with the ext4 performance. To implement this benchmark, we create 10 files of 40 KB each and we fix our anti-cache size to be 40 KB, so that at any given point only 1 file can reside in NVM. Then a workload is created, which simulates the reading of certain number of files in the NVM vs the files in the disk, based on the degree of coldness. As it is evident Figure 4, at 0% coldness our performance is exactly what you would expect from BPFS, while at 100% coldness it slightly overshoots the ext4 performance.

This is understandable given the overhead for us, in terms of anti-caching daemon. However, the most interesting part of this graph is that the trend as one increases the degree of coldness. The less the degree of coldness, the more is the performance. This clearly proves the benefit of our anti-caching implementation for a tiered storage.

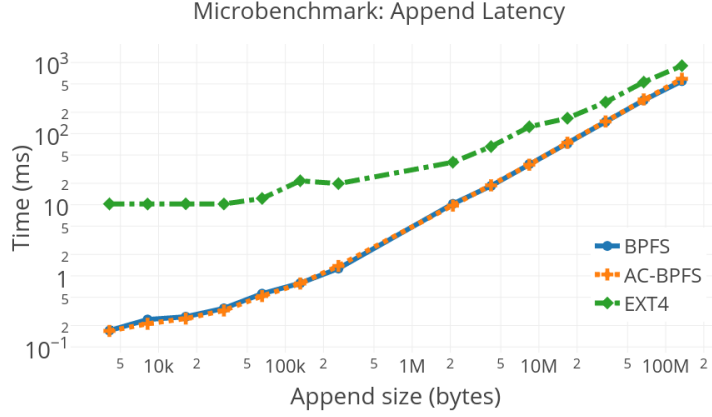
4.2.5 Macrobenchmark: File Bench

For this benchmark, we used FileBench to test and compare the performance of the three files systems by generating a large variety of workloads cite of size 1.25 GB. The workload is comprised of 33% read requests and 67% write requests. For the ACBPFS, we ran the benchmark by varying the (anti) caching thresholds from 20% of the workload size to 100% of the workload size. We did so in order to get insights on the effectiveness and overheads of the anti-caching mechanisms in addition to making comparison of the ACBPFS with BPFS and ext4 file systems.

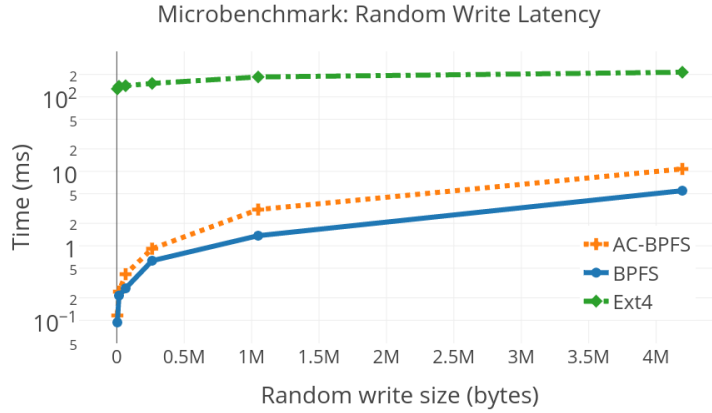
The Figure 5a shows that when the caching threshold is 100% of the workload size (i.e. all the operations happen in the persistent memory), the performance of AC_BPFS is identical to that of BPFS, as we expected. As we decrease the caching threshold, the performance suffers accordingly. On all times, the performance of the AC_BPFS is better than that of ext4 in orders of magnitude.

5 Related Work

There has always been a trade-off between data durability and data read/write performance when deciding between a fast, volatile storage medium like DRAM versus a slow, persistent storage medium like disk. Moreover, re-



(a) **Micro-Benchmark - Append Operation.** The figure shows the append latency of AC-BPFS for various sizes



(b) **Micro-Benchmark - Write Operation.** The figure shows the write latency of AC-BPFS for various sizes

Figure 3: Microbenchmarks

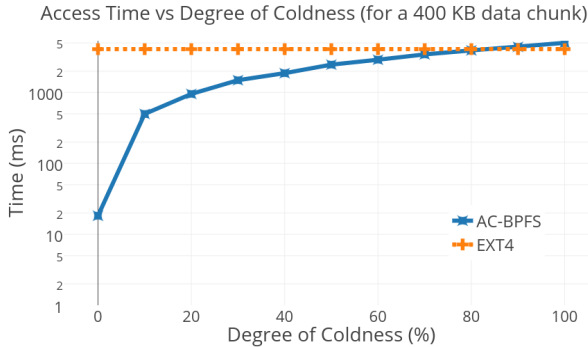
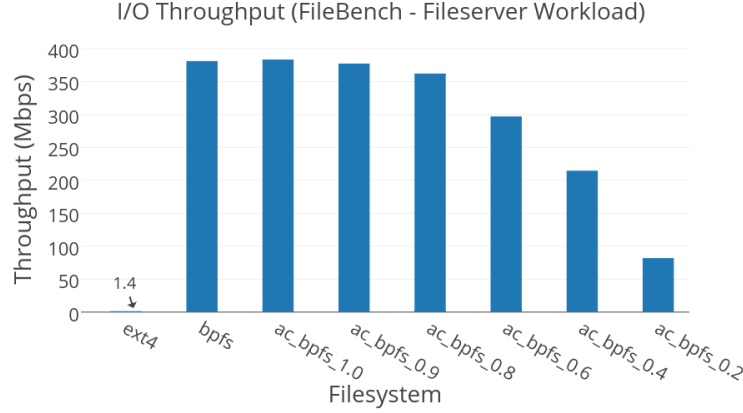


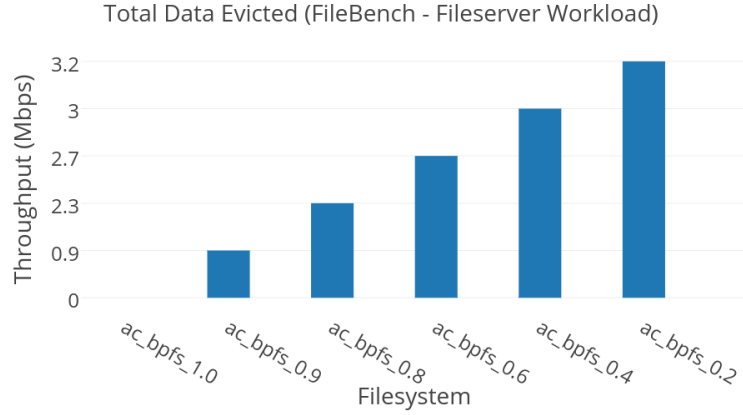
Figure 4: **Impact of Degree of Coldness on Read Latency.** This figure shows the block read latency for various degrees of coldness

searchers have proposed various interfaces and programming models [8, 10] to simplify persistent memory programming and libraries that ensure consistency in the presence of failures. Mnemosyne [10] allows applications to have persistent regions in address space that survive crashes. It provides support for one word atomic updates and lightweight transactions for consistent updates to persistent memory data structures. Mnemosyne handles the case where NVM runs out of space and this is closely related to our work, since our primary assumption is that NVM memory is limited.

Conventional file systems have been optimized for disk and maintaining disk based consistency. Maintaining consistency in memory based systems is different and is still a challenge. One of the earliest file systems to be proposed for non-volatile memory systems was the byte-addressable persistent file system (BPFS) [3]. The BPFS paper showed that simply running a disk-based traditional file system on top of persistent memory is not enough to provide high performance. NOVA [11] on



(a) **FileBench (FileServer) - I/O Throughput.** This figure shows the I/O throughput of ext4, BPFS and AC-BPFS under various configurations for the FileServer workload in FileBench



(b) **FileBench (FileServer) - Block Eviction Count.** This figure shows the number of data blocks evicted from NVM under various configurations for the FileBench file server workload

Figure 5: Macrobenchmarks

the other hand takes a different approach and implements a log structured file system that provides better concurrency by using separate logs for each inode and guarantees consistency by storing the journals in NVM. The overhead of garbage collection in LFS is also reduced in NOVA by exploiting the low random write latency of NVM when compared to the poor sequential write access latencies of SSDs and disks. PMFS [5] is similar to other persistent memory file systems but provides transparent large page support for faster memory-mapped I/O and provides a way for applications to specify file sizes hints causing PMFS to use large pages for files data nodes. Aerie [9] provides a flexible file system architecture that aims to avoid the overhead of trapping into kernel while reading or writing to a file. It achieves this by splitting the functionality across different layers - a kernel layer that handles protection, allocation and addressing, a user library that directly accesses NVM for file reads,

file writes and metadata reads, and a trusted service that coordinates updates to metadata.

The current research around file systems for NVM is focused on maximizing file system performance by exploiting higher I/O throughput guarantees of the byte-addressable persistent NVM [3, 5, 9, 11]. While a majority of the research in this area has tried to solve the problem of consistency, write ordering and atomicity associated with implementing file systems for NVM, there has been some related work in accelerating the performance of existing file systems using NVM [1, 2]. We borrow the idea of Anti-Caching [4] from the database community and propose a radical new approach to file system design in context of main-memory file systems like BPFS [3], [11], where NVM is used as a primary data store for hot data and the disks serve as a backing store for cold data. While these file systems for NVM have been proposed as a replacement for traditional disk

and flash-based file systems, our work proposes to enhance these file systems to function alongside the existing traditional disk and flash-based file systems.

Lv et. al. [7] propose a strategy named Hotness Aware Hit (HAT) for efficient buffer management in flash-based hybrid storage systems by dividing pages into three hotness categories: hot, warm and cold. In general, the hot, warm and cold pages are kept in main memory, flash and hard disk respectively. They employ a page reference queue to record the page reference history. Based on the reference information, the pages are marked with different hotness levels. Pages are allocated to different levels in the storage hierarchy according to the page access sequence and pages hotness. Our approach employs a similar strategy to determine the hotness of data and adapts it with the changes in the usage pattern by employing a multi-tier page reference history. However, our approach differs from this work because we do not use NVM merely for buffering of data. We use it as one of the storage devices in the multi-tier storage system with strategies to make placement decision during the runtime. Ghandeharizadeh et. al. [6] propose a way to use knowledge about the frequencies of read and write requests to individual data items in order to determine the optimal cache configuration given a fixed budget. It employs an offline optimal algorithm that computes: (1) the choice and sizes of the stashes that constitute a cache given a fixed budget and (2) a static placement of data items across the stashes. This paper considers both tiering and replication of data across the selected choices of storage media. Our approach is different in following ways- we only consider tiering approach and do not replicate data across the storage layers, and the decision of placement of data is done online based on the information such as available space in NVM, size of the data and last access time. A separate background process moves cold data from higher storage level (NVM) to lower level (SSD / disk).

6 Conclusion

Fast non-volatile memories (NVM) are byte addressable and offer latencies close to DRAM, but incur a higher predicted cost per byte. Combining the low latency of NVM and low cost of disk-based storage without regressing on the performance maximizes the best of the two worlds. In this paper, we present design, implementation and analysis of multi-tiered storage hierarchy consisting of NVRAM and Disk (or Flash) on File Systems, thereby providing a virtual impression of infinite NVM memory as well as the cheap upgrade in the disk performance with the amount of NVM available through the use of anti-caching mechanism.

7 Future Work

Fast non-volatile memories (NVM) are byte addressable and offer latencies close to DRAM, but incur a higher predicted cost per byte. Combining the low latency of NVM and low cost of disk-based storage without regressing on the performance maximizes the best of the two worlds. In this paper, we present design, implementation and analysis of multi-tiered storage hierarchy consisting of NVRAM and Disk (or Flash) on File Systems, thereby providing a virtual impression of infinite NVM memory as well as the cheap upgrade in the disk performance with the amount of NVM available through the use of anti-caching mechanism.

We identify several opportunities to extend this work in future. In this paper we did not get to do a more thorough analysis on impacts and effects of hotness and coldness of data using multiple workloads and macro-benchmarks. We believe that doing so will reveal several potential areas to improve the system both functionally and architecturally. It is very important that we thoroughly evaluate and analyze crash consistency and reliability issues for this system in depth before extending the system. Lastly, since we simulated the NVM in DRAM, we did not get to evaluate the effects of persistent LRU cache on the performance of the system. Persistent LRU cache could have a significant effect on different aspects of the file system, such as boot time, that remains yet to be studied.

References

- [1] Baker, M., Asami, S., Deprit, E., Ouseterhout, J. and Seltzer, M., 1992, September. Non-volatile memory for fast, reliable file systems. In ACM SIGPLAN Notices (Vol. 27, No. 9, pp. 10-22). ACM.
- [2] Chen, J., Wei, Q., Chen, C. and Wu, L.F., 2013, April. A file system metadata accelerator with non-volatile memory. In Proceedings of Symposium on Mass Storage Systems and Technologies, Monterey, CA, USA (pp. 19-20).
- [3] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D. (2009, October). Better I/O through byte-addressable, persistent memory. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (pp. 133-146). ACM.
- [4] DeBrabant, Justin, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A new approach to database management system architecture. Proceedings of the VLDB Endowment 6, no. 14 (2013): 1942-1953.
- [5] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J. (2014, April). System software for persistent memory. In Proceedings of the Ninth European Conference on Computer Systems (p. 15). ACM.
- [6] Ghandeharizadeh, S., Irani, S., Lam, J. (2015). Memory Hierarchy Design for Caching Middleware in the Age of NVM.
- [7] Lv, Y., Cui, B., Chen, X., Li, J. (2013, October). Hotness-aware buffer management for flash-based hybrid storage systems. In Proceedings of the 22nd ACM international conference on Conference on information and knowledge management (pp. 1631-1636). ACM.
- [8] Persistent Memory Programming. <http://pmem.io/nvml/>
- [9] Volos, H., Nalli, S., Panneerselvam, S., Varadaran, V., Saxena, P., Swift, M. M. (2014, April). Aerie: Flexible file-system interfaces to storage-class memory. In Proceedings of the Ninth European Conference on Computer Systems (p. 14). ACM.
- [10] Volos, H., Tack, A. J., Swift, M. M. (2011). Mnemosyne: Lightweight persistent memory. ACM SIGPLAN Notices, 46(3), 91-104.
- [11] Xu, J., Swanson, S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories.