NVMFS: A Hybrid File System for Improving Random Write in NAND-flash SSD

Sheng Qiu and A. L. Narasimha Reddy
Department of Electrical and Computer Engineering
Texas A&M University
herbert1984106@neo.tamu.edu, reddy@ece.tamu.edu

***Abstract*—NAND Flash-based Solid State Drives (SSDs) have been widely used as secondary storage for their better performance and lower power consumption compared to traditional Hard Disk Drives (HDDs). However, the random write performance is still a concern for SSDs. Random writes could also result in lower lifetimes for SSDs.**

In this paper, we propose a hybrid file system – NVMFS, to resolve the random write issue of SSDs. First, NVMFS distributes data dynamically between NVRAM and SSD. Hot data can be permanently stored on NVRAM without writing back to SSD, while relatively cold data can be temporarily cached by NVRAM with another copy on SSD. Second, NVMFS absorbs random writes on NVRAM and employs long term data access patterns in allocating space on SSD. As a result, NVMFS experiences reduced erase overheads at SSD. Third, NVMFS explores different write policies on NVRAM and SSD. We do in-place updates on NVRAM and non-overwrite on SSD. We exploit the maximum write bandwidth of SSD by transforming random writes at file system level to sequential ones at SSD level.

We have implemented a prototype NVMFS within a Linux Kernel and compared with several modern file systems such as ext3, btrfs and NILFS2. We also compared with another hybrid file system Conquest, which originally was designed for NVRAM and HDD. The experimental results show that NVMFS improves IO throughput by an average of 98.9% when segment cleaning is not active, while improves throughput by an average of 19.6% under high disk utilization (over 85%) compared to other file systems. We also show that our file system can reduce the erase operations and overheads at SSD.

I. Introduction

For many years, the performance of persistent storage (such as hard disk drives) has remained far behind that of microprocessors. Although the disk density has improved from GBs to TBs, data access latency has increased by only 9X [7, 4]. Compared with HDDs, SSDs have several benefits. An SSD is a purely electronic device with no mechanical parts, and thus can provide lower access latencies, lower power consumption, lack of noise and shock resistance. However, SSDs also have two serious problems: limited lifetime and relatively poor random write performance. In SSDs, the smallest write unit is one page (such as 4KB) and can only be performed out-of-place, since data blocks have to be erased before new data can be written. Random writes can cause internal fragmentation of SSDs and thus lead to higher frequency of expensive erase operations [13, 11]. Besides performance degradation, the lifetime of SSDs can also be dramatically reduced by random writes.

Flash memory is now being used in other contexts, for example in designing nonvolatile DIMMs [1, 6]. These designs combine traditional DRAM, Flash, an intelligent system controller, and an ultracapacitor power source to provide a highly reliable memory subsystem that runs with the latency and endurance of the fastest DRAM, while also having the persistence of Flash (data on DRAM will be automatically backed up to flash memory on power failure). The availability of these nonvolatile DIMMs can simplify and enhance file system design, a topic we explore in this paper.

In this paper, we consider a storage system consisting of Nonvolatile DIMMs (as NVRAM) and SSD. We expect a combination of NVRAM and SSD will provide the higher performance of NVRAM while providing the higher capacity of SSD in one system. We propose a file system NVMFS for such a system that employs both NVRAM and SSD in one system. Our file system exploits the unique characteristics of these devices to simplify and speed up file system operations.

Traditionally, when devices of different performance are used together in a system, two techniques are employed for managing space across the devices. When caching is employed, the higher performance device improves performance transparently to the layers above, with data movement across the devices taken care of at lower layers. When migration is alternately employed, the space of both slower and faster devices becomes visible to the higher layers. Both have their advantages and disadvantages.

In our file system proposed here in this paper, we employ both caching and migration at the same time to improve file system operations. When data is migrated, the address of the data is typically updated to reflect the new location whereas in caching, the permanent location of the data remains the same, while the data resides in higher performance memory. For example, in current file systems, when data is brought into the page cache, the permanent location of the file data remains on the disk even though access and updates may be satisfied in the page cache. Data eventually has to be moved to its permanent location, in caching systems. In systems that employ migration, data location is typically updated as data moves from one location to another location to reflect its current location. When clean data needs to be moved to slower devices, data cannot be simply discarded as in caching systems (since data always resides in the slower devices in caching systems), but has to be copied to the slower devices and the

metadata has to be updated to reflect the new location of the data. Otherwise, capacity of the devices together cannot be reported to the higher layers as the capacity of the system.

In our system, we employ both these techniques simultaneously, exploiting the nonvolatile nature of the NVRAM to effectively reduce many operations that would be otherwise necessary. We use the higher performance NVRAM as both a cache and permanent space for data. Hot data and metadata can permanently reside in the NVRAM while not-so-hot, but recently accessed data can be cached in the NVRAM at the same time. This flexibility allows us to eliminate many data operations that would be needed in systems that employ either technique alone.

When data is accessed from SSD, initially that data is cached in the NVRAM, and the file system retains pointers to both locations. If this data becomes a candidate for eviction from NVRAM and it hasn't been updated since it is brought into NVRAM, we discard the data from NVRAM and update the metadata to reflect the fact that data resides now only on the SSD. If the data gets updated after it is brought into NVRAM, we update the metadata to reflect that the data location is the NVRAM and the data on the SSD is no longer valid. Since NVRAM is nonvolatile, we can retain the data in NVRAM much longer and get forced to flush or write this data back to SSD to protect against failures. This allows us to group the dirty data together and write the dirty data together to SSD at a convenient time. Second, this allows us to group data with similar hot-cold behavior into one block while moving it to SSD. We expect this will improve the garbage collection process at SSD in the longer term.

In order to allow this flexibility that we described above, where data can be cached or permanently stored on the NVRAM, we employ two potential addresses for a data block in our file system. The details of this will be described later in section III.

The primary contributions of this paper are as following:

- This paper proposes a new file system – NVMFS, which integrates Nonvolatile DIMMs (as NVRAM) and a commercial SSD as the storage infrastructure.
- NVMFS trades-off the advantage and disadvantage of NVRAM and SSD respectively. In our design, we utilize SSD's larger capacity to hold the majority of file data while absorbing random writes on NVRAM. We explore different write policies on NVRAM and SSD: in-place updates on NVRAM and non-overwrite on SSD. As a result, random writes at file system level are transformed to sequential ones at device level when completed on SSD.
- NVMFS distributes metadata and relatively hot file data on NVRAM while storing other file data on SSD. Unlike normal caching or migration scheme, our design can permanently store hot data on NVRAM while also temporarily caching the recently accessed data. To track the hotness of file data, we implement two LRU lists for dirty and clean file data respectively. Our file system will dynamically adapt the number of pages distributed between dirty and clean LRU lists. When the dirty file data are not hot enough we will collectively flush them (grouped into SSD blocks) to SSD and put them to the end of clean LRU list which may be quickly replaced whenever the space of NVRAM is not enough (we always replace LRU clean pages).
- We show that NVMFS improves IO throughput by an average of 98.9% when segment cleaning is not active, while improving IO throughput by an average of 19.6% when segment cleaning is activated, compared to several existing file systems.
- We also show that the erase operations and erase overhead at SSD are both effectively reduced.

The remainder of the paper is organized as follows: We discuss related work in Section II. In Section III we provide design and implementation details on our proposed file system which are then evaluated in Section IV. Section V concludes the paper.

II. RELATED WORK

A number of projects have previously built hybrid storage systems based on non-volatile memory devices [28, 23, 32, 35]. [28] and [32] proposed using a NVRAM as storage for file system metadata while storing file data on flash devices. FRASH [23] harbors the in-memory data and the on-disk structures of the file system on a number of byte-addressable NVRAMs. Compared with these works, our file system explores different write policies on NVRAM and SSD. We do in-place updates on NVRAM and non-overwrite on SSD.

Rio [16] and Conquest [35] use a battery-backed RAM in the storage system to improve the performance or provide protections. Rio uses the battery-backed RAM and avoids flushing dirty data to disk. Conquest uses the nonvolatile memory to store the file system metadata and small files. WSP [31] proposes to use flush-on-fail technique, which leverage the residual energy of the system, to flush registers and caches to NVRAM in the presence of power failure. Our work here explores nonvolatile DIMMs to provide a highly reliable NVRAM that runs with the latency and endurance of the fastest DRAM, while also having the persistence of Flash. In the eNVy storage system [37], the flash memory with a battery-backed RAM buffer is attached to the memory bus to implement a non-volatile memory device. Our work assumes that nonvolatile memory is large enough for both data and metadata and uses dynamic metrics to determine what data is retained in NVRAM. Moreover, our file system transforms random writes to sequential ones at SSD level which can effectively reduce SSD's erase overhead and improve SSD's lifetime.

The current SSDs implement log-structured like file systems [33] on SSDs to accommodate the erase, write operations of the SSDs. Garbage collection and the write amplification resulting from these operations are of significant interest as the lifetime of SSDs is determined by the number of program/erase cycles [21]. Several techniques have been recently

proposed to improve the lifetime of the SSDs, for example [15, 20]. The recent work SFS [12] proposed to collect data hotness statistics at file block level and group data accordingly. However, they were restricted to exploit this information within a relatively short time slice, since all the dirty data within page cache have to be flushed to persistent storage shortly. Our work here exploits the NVRAM to first reduce the writes going to the SSD and second in grouping similar pages into one block write to SSD to improve garbage collection efficiency.

Several recent studies have looked at issues in managing space across different devices in storage systems [14, 18]. These studies have considered matching workload patterns to device characteristics and studied the impact of storage system organizations in hybrid systems employing SSDs and magnetic disks. Our hybrid storage system here employs NVRAM and SSD. Another set of research work proposed different algorithms for managing the buffer or cache for SSD [36, 26, 22, 34]. They all intended to temporally buffer the writes on the cache and reduce the writes to SSD. Our work differs from them since our file system can permanently store the data on NVRAM, thus further reducing writes to SSD.

Much research has been focused on FTL design to improve performance and to extend lifetime of SSDs [27, 19, 30, 29]. Three types of FTL schemes are proposed including block-level mapping, page-level mapping and hybrid mapping that trades-off the first two. The block-level FTL maps a logical block number to a physical block number and the logical page offset within that block is fixed. This scheme can store the entire mapping table in memory since it is small. However, such coarse-grained mapping results in a higher garbage collection overhead. In contrast, a page-level FTL manages a fine-grained page-level mapping table, thus has lower garbage collection overhead. While page-level FTL requires a large mapping table on RAM which cost more on hardware. To overcome such technical challenges, hybrid FTL schemes [27, 30] extend the block-level FTL. These schemes logically partition flash blocks into data blocks and log blocks. The majority of data blocks are using block-level mapping while the log blocks are using page-level mapping. Our work can reduce the erase overhead during GC (Garbage Collection) which benefits various FTL schemes.

III. Design and Implementation

NVMFS improves SSD's random write performance by absorbing small random IOs on NVRAM and only performing large sequential writes on SSD. To reduce the overhead of SSD's erase operations, NVMFS groups data with similar update likelihood into the same SSD blocks. The benefits of our design resides on three aspects: (1)reduce write traffic to SSD; (2)transform random writes at file system level to sequential ones at SSD level; (3)group data with similar update likelihood into the same SSD blocks.

A. Hybrid Storage Architecture

In NVMFS, the memory system is composed by two parts, one is the traditional DRAM, the other is the Nonvolatile DIMMs. Figure 1 shows the hardware architecture of our system. We utilize the Nonvolatile DIMMs as NVRAM which is attached to the memory bus, and accessed through virtual addresses. The actual physical addresses to access NVRAM are available through the page mapping table, leveraging the operating system infrastructure. All the page mapping information of NVRAM will be stored on a fixed part of NVRAM. We will detail this later in section III-B. If the requested file data is on NVRAM, we can directly access it through load/store instructions. While if the requested file data is on SSD, we need to first fetch it to NVRAM. It's noted that we bypass page cache in our file system, since CPU can directly access NVRAM which can provide the same performance as DRAM based page cache. To access the file data on SSD, we use logical block addresses (LBAs), which will be translated to the physical block addresses (PBAs) through FTL component of SSD. Therefore, NVMFS has two types of data addresses at file system level – virtual addresses for NVRAM and logical block addresses for SSD. In our design, we can store two valid versions for hot data on NVRAM and SSD respectively. Whenever the data become dirty, we keep the recent data on NVRAM and invalidate the corresponding version on SSD. We will introduce how we manage the data addresses of our file system in section III-B.

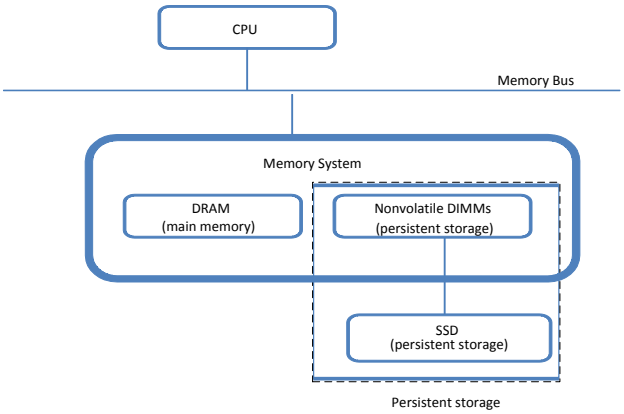


Fig. 1: Hybrid Storage Architecture

The benefit of building such a hybrid storage is that we can utilize each device's advantages while offsetting their disadvantages. Since SSD has poor random write, out-of-place update and limited write cycles, we absorb random writes on NVRAM and only perform large sequential writes on SSD. For metadata and frequently accessed file data, we permanently store them on NVRAM, while distributing other relatively cold, clean data on SSD. We will detail how NVMFS distributes the file data between NVRAM and SSD in section III-C.

B. File System Layout

The space layout of NVMFS is shown in figure 2. The metadata and memory mapping table are stored on NVRAM. The metadata contains the information such as size of NVRAM and SSD, size of page mapping table, etc. The memory mapping table is used to build some in-memory data structures when mounting our file system and is maintained by memory management module during runtime. All the updates to the memory mapping table will be flushed immediately to NVRAM.

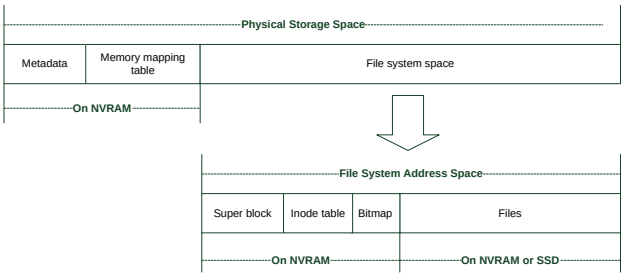


Fig. 2: Storage Space Layout

In the file system address space, the layout of NVMFS is very simple. The file system metadata which includes super block, inode table and block bitmap are stored on NVRAM while the file data are stored either on NVRAM or SSD based on their usage pattern. The block bitmap indicates whether the corresponding NVRAM or SSD block is free or used. In NVMFS, we always put hot file data on NVRAM and cold file data on SSD. In our current implementation, the total size of virtual memory space for NVRAM addresses is 2^{47} bytes (range: ffff000000000000 - ffff7fffffffffff), which is unused in original Linux kernel. The space can be larger if we reorganize 64-bit virtual address space. We modified the Linux kernel to let the operating system be aware of the existence of two types of memory devices – DRAM and NVRAM, attached to the memory bus. We also added a set of functions for allocating/deallocating the memory space of NVRAM. This implementation is leveraged from previous work in [38].

In NVMFS, the directory files are stored as ordinary files, while their contents are lists of inode numbers. To address the inode table, we store the pointer to the start address of inode table in the super block. Within the inode table, we use a fixed size entry of 128 bytes for each inode, and it is simple to get a file's metadata through its inode number and the start address of the inode table. The inode will store several pieces of information including checksum, owner uid, group id, file mode, blocks count of NVRAM, blocks count of SSD, size of data in bytes, access time, block pointer array and so on. The block pointer array is similar as the direct/indirect block pointers used in EXT2. The difference is that we always allocate indirect blocks on NVRAM so that it is fast to index the requested file data even when the file is large which requires retrieving indirect blocks. The block address is 64 bits and the NVRAM addresses are distinct from the SSD block addresses. To build our file system, we can use the command like "mount -t NVMFS -o init=4G /dev/sdb1 /mnt/NVMFS". In the example, we attached 4GB Nonvolatile DIMMs as the NVRAM, and tell NVMFS the path of the SSD device, finally mount it to the specified mount point.

C. Data Distribution and Write Reorganization

The key design of NVMFS relies on two aspects: (a)how to distribute file system data between the two types of devices – NVRAM and SSD; (b)how to group and reorganize data before writing to SSD so that we can always perform large sequential writes on SSD.

File system metadata are small and will be updated frequently, thus it's natural to store them on NVRAM. To efficiently distribute file data, we track the hotness of both clean and dirty file data. We implemented two LRU (Least Recently Used) lists — dirty and clean LRU lists, which are stored as metadata on NVRAM. Considering the expensive write operations of SSD, we prefer to store more dirty data on NVRAM, expecting them to absorb more update/write operations. Whenever the space of NVRAM is not sufficient, we replace file data from clean LRU list. However, we also do not want to hurt the locality of clean data. We balance this by dynamically adjusting the length of dirty and clean LRU lists. For example, if the hit ratio of clean data is 2X more than that of dirty data, we increase the NVRAM pages that are allocated for clean data. In other words, we increase the length of clean LRU list. To achieve this, we maintain two performance counters which keep track of the hits on clean and dirty data (on NVRAM) respectively. We periodically measure and reset the counters. The total number of pages within clean and dirty LRU lists is fixed, equalling to the number of NVRAM pages.

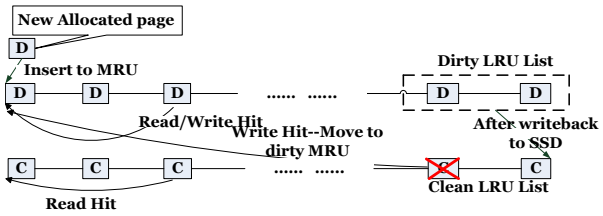


Fig. 3: Dirty and Clean LRU lists

Figure 3 shows the clean and dirty LRU lists as well as the related operations. When writing new file data, we allocate space on NVRAM and mark them as dirty, then insert to the MRU (Most Recently Used) position of dirty LRU list. Read/write operations on dirty data will update their position to MRU within dirty LRU list. For clean data, read operations update their position to MRU of clean LRU list, while write operations are a little different since the related data become dirty afterward. As shown in figure 3, writes on clean data will migrate the corresponding NVRAM pages from clean LRU list to the MRU of dirty LRU list.

Unlike existing page cache structure which flushes dirty data to the backed secondary storage (such as SSDs) within short period, our file system can store dirty data permanently on

NVRAM. NVMFS always keeps the pointer to the most recent data version. We can choose when and which data to flush to SSD dynamically according to the workloads. We begin to flush dirty data to SSD whenever the NVRAM pages within dirty LRU list reaches a high bound (i.e. 80% of dirty LRU list is full). Then we pick dirty data from the end of dirty LRU list and group them into SSD blocks. We allocate the corresponding space on SSD with sequential LBAs and write these data sequentially to SSD. This process continues until the NVRAM pages within dirty LRU list reaches a low bound (i.e. 50% of dirty LRU list is full). The flushing job is executed by a background kernel thread.

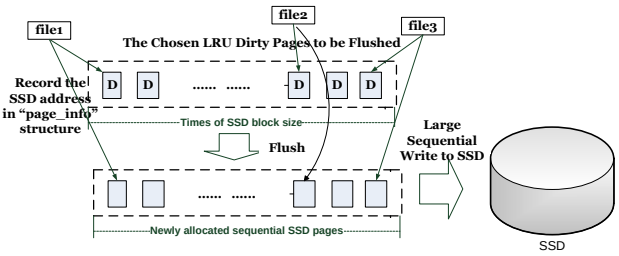


Fig. 4: Migrate Dirty NVRAM Pages to SSD

Figure 4 shows how our file system migrates dirty data from NVRAM to SSD. The dirty NVRAM pages will become clean after migrating to SSD and will be inserted to the LRU position of clean LRU list. As a result, we have two valid clean data versions on NVRAM and SSD respectively. We can facilitate the subsequent read/write requests since we still have valid data versions on NVRAM. Moreover, we can easily replace those data on NVRAM by only reflecting their positions on SSD. In our file system, the file inode always points to the appropriate data version. For example, if file data have two valid versions on NVRAM and SSD respectively, the inode will point to the data on NVRAM. We have another data structure called "page_info" which records the position of another valid data version on SSD. It is noted that we won't lose file system consistency even if we lose this "page_info" structure, since file inodes consistently keep the locations of appropriate valid data version. We will discuss file system consistency in section III-E

D. Non-overwrite on SSD

We employ different write policies on NVRAM and SSD. We do in-place update on NVRAM and non-overwrite on SSD, which exploits the devices' characteristics. The space of SSD is managed as extents of 512KB, which is also the minimum flushing unit for migrating data from NVRAM to SSD. Each extent on SSD contains 128 normal 4KB blocks, which is also the block size of our file system. When dirty data are flushed to SSD, we organize them into large blocks (i.e. 512KB) and allocate corresponding number of extents on SSD. As a result, random writes of small IO requests are transformed to large write requests (i.e. 512KB).

To facilitate allocation of extents on SSD, we need to periodically clean up internal fragmentation within the SSD. For example, when clean data are fetched from SSD to NVRAM because of write accesses, the corresponding data on SSD will become invalid, since the updated data now reside on NVRAM. The fetched data can be smaller than one SSD extent or across several SSD extents, as a result, there will be invalid portions within SSD extents. During recycling, we can integrate several partial valid SSD extents into one valid SSD extent and free up the remaining space. This ensures that we can always have free extents available on SSD for allocation, which is similar to the segment cleaning process of log-structured file systems. It's noted that the FTL component of SSD still manages the internal garbage collection of SSD. We will show how NVMFS impacts it in section IV-C.

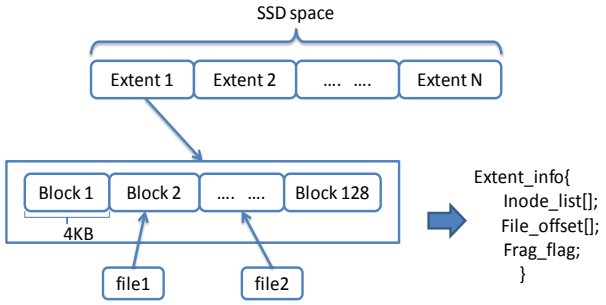


Fig. 5: Space management on SSD

Figure 5 shows the space organization of SSD. As we see, each extent is 512KB which contains 128 normal 4KB blocks. Given the logical block number, it's easy to get its extent's index and offset within that extent. To facilitate extent recycling, we need to keep some information for each block within a candidate extent, for example, the inode and file offset each valid block belongs to. We also keep a flag which indicates whether this extent is fragmented (contains invalid blocks). This information is kept as metadata in a fixed space on NVRAM. The space overhead is small, 64 bits (32 bits inode number, 32 bits file offset) for each 4KB block. Whenever extent recycling is invoked, we choose LRU (Least Recently Used) fragmented extents and move the valid data blocks into NVRAM, update their inodes, finally release the extents' space. It's noted that we only free the recycled extent whenever the associated inodes are all updated. In our current design, two conditions have to be satisfied in order to invoke the recycling: (a)the fragmentation ratio of SSD is over a configurable threshold (ideal extent usage/actual extent usage); (b)the number of free SSD extents is fewer than a configurable threshold. The first condition ensures that we do get some free space after recycling whenever the free extents are not sufficient.

E. File System Consistency

File system consistency is always an important issue in file system design. As a hybrid file system, NVMFS stores metadata and hot file data on NVRAM and distributes other relatively cold data on SSD. When the space of NVRAM is not sufficient, LRU dirty data will be migrated from NVRAM

to SSD. When the segment cleaning process is activated on SSD, valid data will be migrated from SSD to NVRAM. If system crash happens during the middle of these operations, what is the state of our file system? How do we ensure file system consistency? We will discuss this in detail within this section.

As described in section III-C, NVMFS invokes flushing process whenever the dirty LRU list reaches a high bound (i.e. 80% of dirty LRU list is full). The flushing process chooses 512KB data each round from the end of dirty LRU list and prepares a new SSD extent (512KB), then composes the data as one write request to SSD, finally updates the corresponding metadata. The metadata updating involves inserting the flushed NVRAM pages into clean LRU list and recording the new data positions (on SSD) within "page_info" structure that mentioned in the previous section. It's noted that the inodes (unchanged) still point to valid data on NVRAM until they are replaced from clean LRU list. If system crashes while flushing data to SSD, there is no problem, because inodes still point to valid data versions on NVRAM. We simply drop previous operations and restart migration. If system crashes after data flushing but before we update the metadata, NVMFS is still consistent since inodes point to valid data version on NVRAM. The already flushed data on SSD will be recycled during segment cleaning. If system crashes in the middle of metadata update, the LRU list and "page_info" structure may become inconsistent, NVMFS will reset them. Since the inodes still point to the valid data version on NVRAM, our file system is consistent. To reconstruct the LRU list, NVRAM scans the inode table, if the inode points to a NVRAM page, we insert it to dirty LRU list while keeping clean LRU list empty. It's noted that if file data have two valid data versions on both NVRAM and SSD, the data version on SSD will be lost since the "page_info" structure are reset now. The corresponding space will be recycled during segment cleaning since no inodes point to those blocks (invalid blocks within extent).

Segment cleaning is another point prone to inconsistency. The cleaning process chooses one candidate extent (512KB) per round and migrate the valid blocks (4KB) to NVRAM, then updates the inodes to point to the new data positions, finally free the space on SSD. If system crashes during data migration, NVMFS inodes still point to the valid data on SSD. If system crashes during the inodes update, NVMFS maintains consistency by adopting transaction mechanism (inodes update and space freeing on SSD are one transaction) similar to other log-structured file systems.

Another issue is that NVMFS stores metadata and hot data permanently on NVRAM which creates a new challenge: unsure write ordering. The write ordering problem is caused by CPU caches that stand between CPUs and memories [17]. To make the access latency as close to that of the cache, the cache policy tries to keep the most recently accessed data in the cache. The data in the cache is flushed back into the memory according to the designed data replacement algorithm. Therefore the order in which data is flushed back to the memory is not necessarily the same as the order data was written into cache. Another reason that causes unsure write ordering is out-of-order execution of the instructions in the modern processors. To address the problem of unsure write ordering, we use a combination of the instructions MFENCE and CLFLUSH to ensure modification of the critical information, including "metadata", "superblock", "inode table", "bitmap" and "directory files", are in consistent ordering. This implementation is leveraged from previous work in [38].

IV. Evaluation

To evaluate our design, we have implemented a prototype of NVMFS in Linux. In this section, we present the performance of our file system in three aspects: (1)reduced write traffic to SSD; (2)reduced SSD erase operations and erase overhead; (3)improved throughput on file read and write operations.

A. Methodology

We use several benchmarks including IOZONE [5], Postmark [24], FIO [3] and Filebench [9] to evaluate the performance of our file system. The workloads we choose all have different characteristics. For IOZONE, it creates a single large file and perform random writes on it. For Postmark, the write operations are in terms of appending instead of overwriting. For FIO, it performs random updates on randomly opened files chosen from thousands of files. For Filebench, it does mixed read and write on thousands of files which simulate a file server.

In the experimental environment, the test machine is a commodity PC system equipped with a 2.8GHz Intel Core i5 CPU, 8GB of main memory. We also attached 4GB Nonvolatile DIMMs [1, 6] as the NVRAM. The NAND flash SSD we used is Intel's X25-E 64GB SSD. The operating system used is Ubuntu 10.04 with a 2.6.33 kernel. We also tested with other SSDs and got similar results. For space efficiency, we did not include them.

In all benchmarks, we compare the performance of NVMFS to that of other existing file systems, including EXT3, Btrfs, Nilfs2 and Conquest (also a hybrid file system) [35]. The first three file systems are not designed for hybrid storage architecture. Therefore we configure 4GB DRAM-based page cache for them. While for Conquest file system, we implemented it according to the description in [35]. In [35], it is said that a larger threshold for small file will keep more files on NV-memory and achieve better performance. In our implementation of Conquest, we define small file as the one with file size less than 128KB. The reason we made this decision is that we found that with a threshold larger than 128KB, we will not be able to allocate all the small files on NVRAM under our workload. The reason we choose these file systems is as following. EXT3 is a popular file system used in Linux operating system. Btrfs [8] implements some optimizations for SSDs (we mount btrfs with "nodatacow" option to get the best available performance). Nilfs2 [10] is a log structure file system which is designed for NAND flash. Finally, Conquest is also a hybrid file system which utilizes both NVRAM and HDD/SSD as the storage.

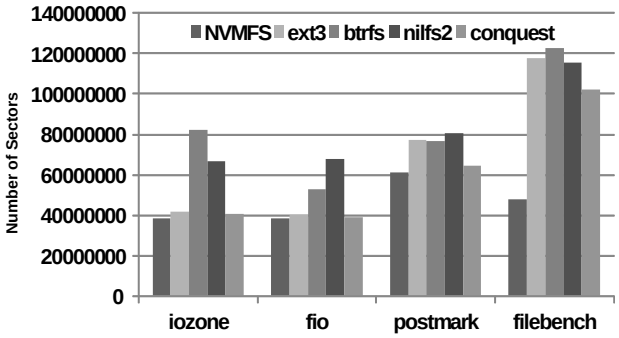


Fig. 6: Write traffic to SSD under different workloads and file systems

B. Reduced IO Traffic to SSD

In this section, we calculated how much IO data are written to SSD while running different workloads for our NVMFS and other file systems. As explained in section III, our NVMFS persistently keeps metadata and hot file data on NVRAM without writing to SSD. However, other file systems have to periodically flush dirty data from page cache to SSD in order to keep consistency. Therefore, NVMFS is expected to reduce write traffic to SSD.

Figure 6 shows the write traffic to SSD (number of sectors) across different workloads. For all the workloads, the IO request size is 4KB. We can see our file system has less write traffic to SSD across all the workloads. For Filebench workload, the reduction is about 50% compared to other file systems. To further explore this, we calculated how many IO requests are satisfied by memory (NVRAM or Page Cache) under different file systems. Figure 7 shows the hit ratio across different workloads, which is the number of IO requests satisfied by memory divided by the total number of IO requests. We can see our file system has higher hit ratio across all the workloads except Postmark. As a result, other file systems have to evict more data from page cache to SSD due to cache replacements. For Postmark workload, although our hit ratio is slightly lower than ext3 and btrfs, we still write less data to SSD. This is because we permanently store metadata on NVRAM, which saves many writes to SSD.

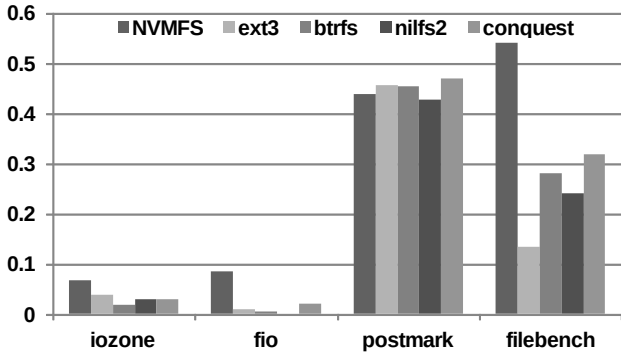


Fig. 7: Hit Ratio on Memory

Conquest only stores small files on NVRAM while keeping large files on disk. Therefore, for Conquest, IO requests to large files still go through page cache and need to be synchronized with SSD. However, our file system can store the frequently accessed portion of large files on NVRAM permanently. Figure 8 shows the write traffic to SSD while running the original and our modified IOZONE workloads. Both the workloads create a large file, then perform random writes on it. The difference is that the modified IOZONE writes randomly only to the first 3GB of the large file. We can see our file system further reduced write traffic by keeping parts of that large file on NVRAM. Because our file system will permanently store the frequently accessed portion of the large file on NVRAM, while Conquest and other file systems need to periodically flush dirty data from page cache to SSD for consistency, NVMFS achieves better performance.

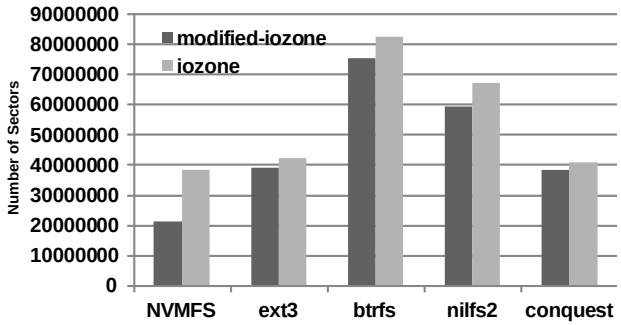


Fig. 8: Write Traffic to SSD under modified iozone

C. Reduced Erase Operations and Overhead on SSD

The erase operations on SSD are quite expensive which greatly impact both lifetime and performance. The overhead of erase operations are usually determined by the number of valid pages that are copied during the GC (Garbage Collection).

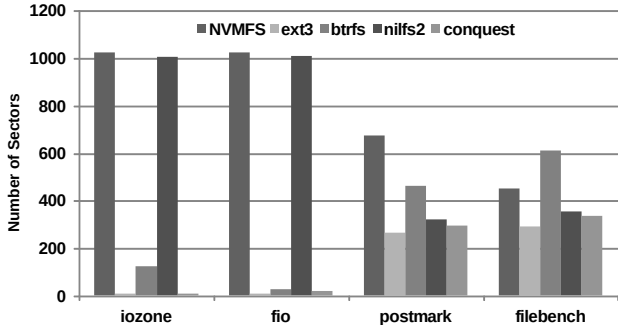
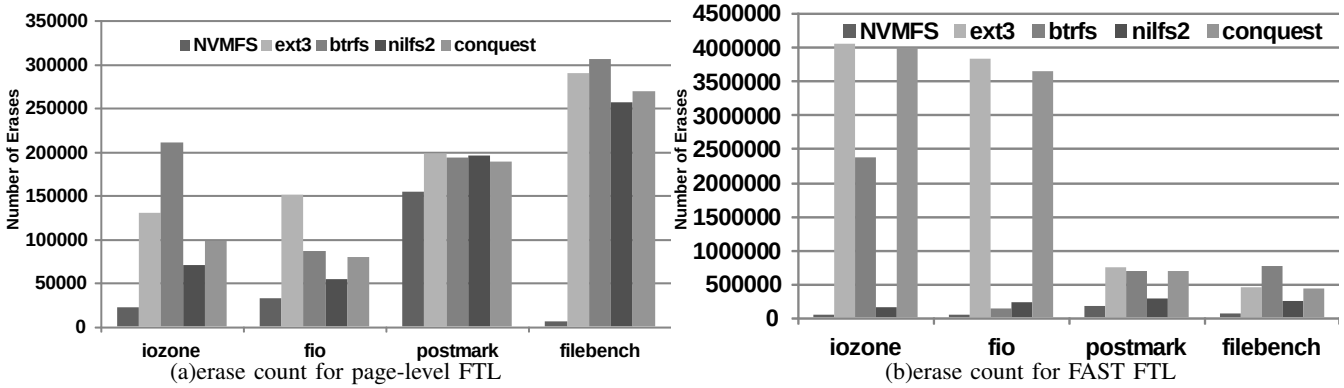


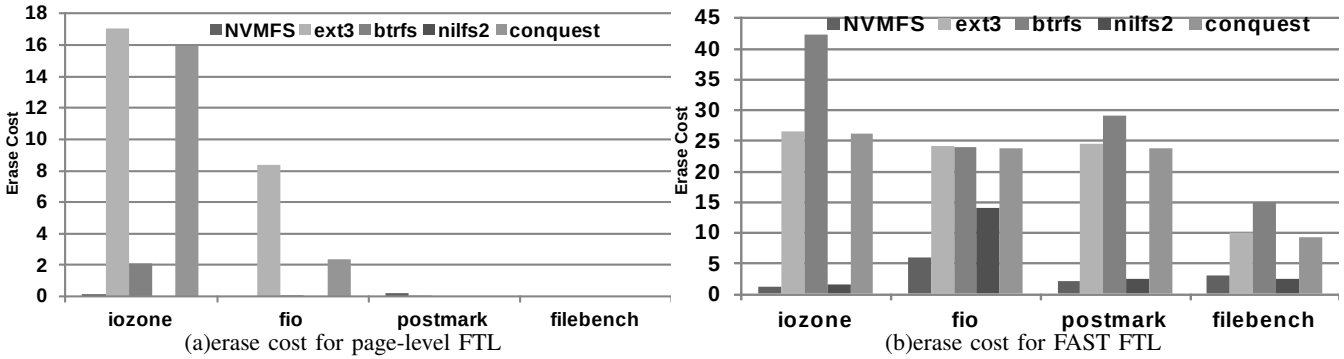
Fig. 9: Average IO Request Size issued to SSD under different workloads and file systems

To evaluate the impact on SSD's erase operations, we collected I/O traces issued by the file systems using blktrace [2] while running our workloads described in section IV-A, and the traces were run on an FTL simulator, which we implemented, with two FTL schemes -(a)FAST [29] as a representative hybrid FTL scheme and (b)page-level FTL [25]. In both schemes, we configure a large block 24GB NAND



(a)erase count for page-level FTL (b)erase count for FAST FTL

Fig. 10: Erase count for page-level and FAST FTL



(a)erase cost for page-level FTL (b)erase cost for FAST FTL

Fig. 11: Erase cost for page-level and FAST FTL

flash memory with 4KB page, 256 KB block, and 10% over-provisioned capacity. Figure 10 and 11 show the total number of erases and corresponding erase cost for the workload processed by each file system.

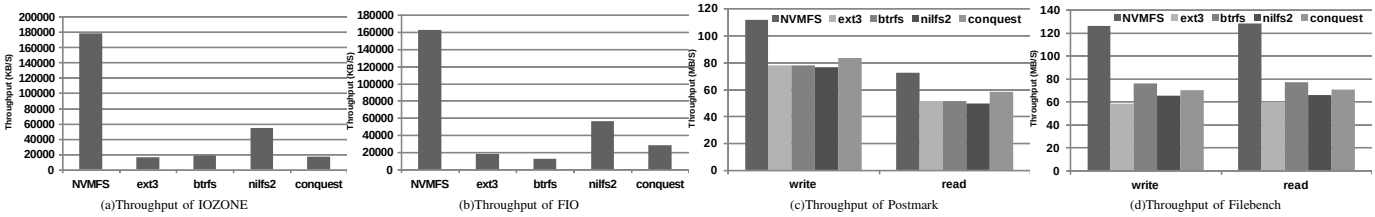
We can see that NVMFS has fewer number of erases under all situations. Our benefits come from two aspects. For Filebench workload, we saved 50% write traffic to SSD as described in section IV-B, thus, we see much fewer erases on SSD. For IOZONE and FIO workloads, we transformed random writes to sequential ones at SSD level, which is similar to log structured file system (such as nilfs2). This design also helps reduce erase overhead. As shown in figure 9, for IOZONE and FIO workloads, our file system and nilfs2 transform random writes to sequential ones at SSD level, thus we observed larger IO request size compared to other file systems. Postmark and Filebench workloads have some locality in accesses. As a result, OS scheduler can merge adjacent IO requests into large ones before issuing to SSD, thus, we see relatively large IO request size across all the tested file systems.

To explore the erase cost, we calculated the average number of pages (valid pages) copied during GC. Figure 11 shows the erase cost which is the average number of pages migrated during GC. For page-level FTL, both NVMFS and nilfs2 have very few valid pages within the erase blocks for most cases. For hybrid FTL scheme, NVMFS also performs better than other file systems.

D. Improved IO Throughput

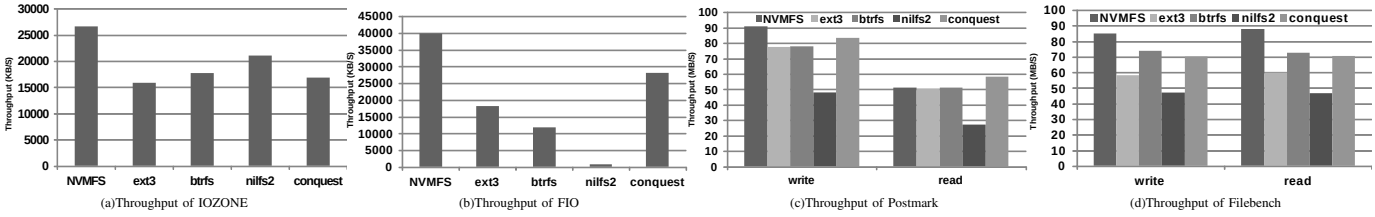
In this section, we evaluate the performance of our file system in terms of IO throughput. We use the workloads described in section IV-A. For our file system and nilfs2, we measure the performance under both high (over 85%) and medium disk utilizations (50%-70%) to evaluate the impact of segment cleaning overhead. The segment cleaning is activated only under high disk utilization. For other file systems that do in-place update on SSD, there is little difference for varied disk utilizations.

Figure 12 shows the IO throughput while the segment cleaning is not activated with our file system and nilfs2. We can see our file system performs much better than all the other file systems across all the workloads. Compared with in-place update file systems, NVMFS transforms small random writes to large sequential writes on SSD, therefore improves the write bandwidth significantly. Compared with log-structured file system such as nilfs2, NVMFS stores hot data on NVRAM and better groups dirty data before flushing to SSD, as a result, the erase overhead of SSD is reduced. We also noticed that Conquest, another hybrid file system, did not perform well under IOZONE and FIO workloads. For IOZONE benchmark, Conquest permanently stored the single large file on SSD and used page cache to temporarily buffer the write accesses, similar as EXT3. Compared with our file system and nilfs2, Conquest still performed random writes on SSD. For FIO benchmark, Conquest put small files (smaller



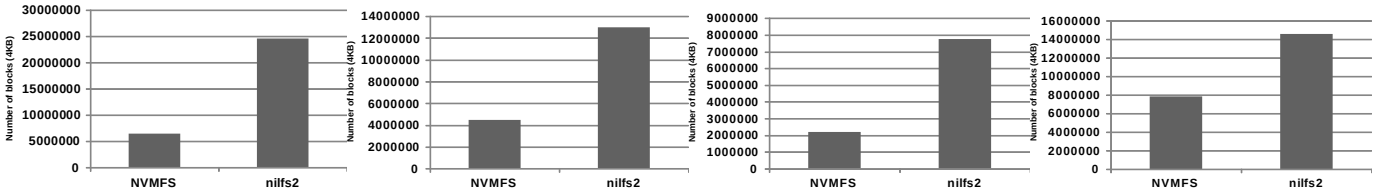
(a)Throughput of IOZONE (b)Throughput of FIO (c)Throughput of Postmark (d)Throughput of Filebench

Fig. 12: IO throughput under different workloads for 50% - 70% disk utilization



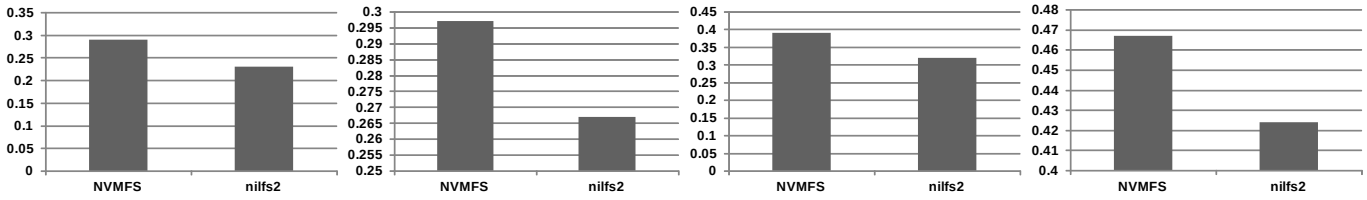
(a)Throughput of IOZONE (b)Throughput of FIO (c)Throughput of Postmark (d)Throughput of Filebench

Fig. 13: IO throughput under different workloads for over 85% disk utilization



(a)IOZONE (b)FIO (c)Postmark (d)Filebench

Fig. 14: Total number of recycled blocks while running different workloads under NVMFS and nilfs2



(a)IOZONE (b)FIO (c)Postmark (d)Filebench

Fig. 15: Cleaning efficiency while running different workloads under NVMFS and nilfs2

than 128KB) on NVRAM, however, for large files, random writes were still performed on SSD.

To evaluate the impact of segment cleaning on our file system and nilfs2, we also measured the performance under high disk utilization (over 85%). Figure 13 shows the throughput when disk utilization is over 85% for all the tested file systems and workloads. We can see obvious performance reduction for both NVMFS and nilfs2, while other file systems have little change compared with that under 50%-70% disk utilization. Compared with nilfs2, our file system performs much better across all the workloads, especially under FIO workload. To further explore this, we calculated the number of blocks (4KB) recycled and the cleaning efficiency while running different workloads under NVMFS and nilfs2. For cleaning efficiency, we measure it using the formula "1 - (moved_valid_blocks / total_recycled_blocks)".

Figure 14 and 15 show the total number of recycled blocks and the cleaning efficiency respectively while running different workloads under NVMFS and nilfs2. We can see for all the workloads NVMFS recycled much fewer blocks compared with nilfs2, which means we generate much fewer background IOs. This is because NVMFS absorbs many small IOs on NVRAM and avoids many writes to SSD which relieves the pressure of segment cleaning on SSD. As a result, forefront IO workloads are less impacted by NVMFS compared to nilfs2. Another benefit of NVMFS is that when we move valid blocks from SSD to NVRAM due to recycling, we can free the corresponding space on SSD once the data reside on NVRAM. However, nilfs2 has to wait until the valid blocks are written back to new segments on SSD, otherwise they may lose consistency for power failure. As shown in figure 15, we also see NVMFS has higher cleaning efficiency relative to nilfs2. This is benefit from our grouping policy on dirty data before flushing to SSD.

V. Conclusions

In this paper, we have implemented a new file system — NVMFS, which integrates NVRAM and SSD as hybrid storage. NVMFS dynamically distributed file data between NVRAM and SSD which achieved good IO throughput. To improve the random write performance of SSD, our file system transformed random writes at file system level to sequential

ones at SSD level. As a result, we reduced the overhead of erase operations on SSD and improved the GC efficiency. We also show that our file system executed segment cleaning more efficiently than nilfs2 and impacted much less on forefront IOs. In future, we plan to explore more efficient scheme to distribute data between NVRAM and SSD, employ better grouping policy on dirty data to further improve the GC efficiency and segment cleaning on SSD.

REFERENCES

[1] "Agigaram ddr3 nvdimm." [Online]. Available: http://www.agigatech.com/ddr3.php

[2] "Btrfs: a linux file system." [Online]. Available: http://linux.die.net/man/8/blktrace

[3] "Fio benchmark." [Online]. Available: http://freecode.com/projects/fio

[4] "Ibm 3340 disk storage unit." [Online]. Available: http://www-03.ibm.com/ibm/history/exhibits/storage/storage3340.html

[5] "Iozone filesystem benchmark." [Online]. Available: http://www.iozone.org/

[6] "Non-volatile dimm." [Online]. Available: http://www.vikingtechnology.com/non-volatile-dimm

[7] "Wd velociraptor: Sata hard drives." [Online]. Available: http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701284.pdf

[8] "Btrfs: a linux file system," 2011. [Online]. Available: https://btrfs.wiki.kernel.org/

[9] "Filebench: a file system and storage benchmark that allows to generate a large variety of workloads," 2011. [Online]. Available: http://sourceforge.net/apps/mediawiki/filebench/index.php

[10] "Nilfs2: New implementation of a log-structured file system version 2," 2011. [Online]. Available: http://en.wikipedia.org/wiki/NILFS

[11] L. Bouganim, B. T. Jónsson, and P. Bonnet, "uflip: Understanding flash io patterns," in *CIDR*, 2009.

[12] H. C. Changwoo Min, Kangnyeon Kim and Y. I. E. Sang-Won Lee, "Sfs: Random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX conference on File and storage technologies*, 2012.

[13] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '09, 2009, pp. 181–192.

[14] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: making the best use of solid state drives in high performance storage systems," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 22–32.

[15] F. Chen, T. Luo, and X. Zhang, "Caftl: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST'11, 2011, pp. 6–6.

[16] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The rio file cache: surviving operating system crashes," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VII, 1996, pp. 74–83.

[17] 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*, ser. SOSP '09, 2009, pp. 133–146.

[18] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST'11, 2011, pp. 20–20.

[19] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09, 2009, pp. 229–240.

[20] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST'11, 2011, pp. 7–7.

[21] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09, 2009, pp. 10:1–10:9.

[22] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "Fab: flash-aware buffer management policy for portable media players," *Consumer Electronics, IEEE Transactions on*, vol. 52, no. 2, pp. 485 – 493, may 2006.

[23] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, "Frash: Exploiting storage class memory in hybrid file system for hierarchical storage," *Trans. Storage*, vol. 6, no. 1, pp. 3:1–3:25, Apr. 2010.

[24] J. Katcher, "Postmark: A new file system benchmark," technical Report TR3022. Network Applicance Inc. October 1997.

[25] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95, 1995, pp. 13–13.

[26] H. Kim and S. Ahn, "Bplru: a buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08, 2008, pp. 16:1–16:14.

[27] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *Consumer Electronics, IEEE Transactions on*, vol. 48, no. 2, pp. 366 –375, may 2002.

[28] J. K. Kim, H. G. Lee, S. Choi, and K. I. Bahng, "A pram and nand flash hybrid architecture for high-performance embedded storage subsystems," in *Proceedings of the 8th ACM international conference on Embedded software*, ser. EMSOFT '08, 2008, pp. 31–40.

[29] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, 2007.

[30] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "Last: locality-aware sector translation for nand flash memory-based storage systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, Oct. 2008.

[31] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, 2012, pp. 401–410.

[32] Y. Park, S.-H. Lim, C. Lee, and K. H. Park, "Pffs: a scalable flash memory file system for the hybrid architecture of phase-change ram and nand flash," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08, 2008, pp. 1498–1503.

[33] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[34] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending ssd lifetimes with disk-based write caches," in *Proceedings of the 8th USENIX conference on File and storage technologies*, 2010, pp. 8–8.

[35] A.-I. A. Wang, G. Kuenning, P. Reiher, and G. Popek, "The conquest file system: Better performance through a disk/persistent-ram hybrid design," *Trans. Storage*, vol. 2, no. 3, pp. 309–348, Aug. 2006.

[36] G. Wu, X. He, and B. Eckart, "An adaptive write buffer management scheme for flash-based ssds," *Trans. Storage*, vol. 8, no. 1, pp. 1:1–1:24, Feb. 2012.

[37] M. Wu and W. Zwaenepoel, "envy: a non-volatile, main memory storage system," in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VI, 1994, pp. 86–97.

[38] 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*, ser. SC '11, 2011, pp. 39:1–39:11.