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

Sheng Qiu
Texas A&M University
herbert1984106@neo.tamu.edu

A. L. Narasimha Reddy
Texas A&M University
reddy@ece.tamu.edu

***Abstract*—In this paper, we design a storage system consisting of Non-volatile DIMMs (as NVRAM) and NAND-flash SSD. We propose a file system NVMFS to exploit the unique characteristics of these devices which simplifies and speeds up file system operations. We use the higher performance NVRAM as both a cache and permanent space for data. 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. We also reduce the erase overhead of SSD by reorganizing writes on NVRAM before flushing to SSD.**

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

Solid-State Drives (SSDs) have been widely used in computer systems. 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 [7], [9]. 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 Non-volatile 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.

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. 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.

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

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.

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.
- 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.
- 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 [16], [23], [26]. PFFS [23] proposed using a NVRAM as storage for file system metadata while storing file data on flash devices. FRASH [16] 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 [11] and Conquest [26] 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 [22] 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.

The current SSDs implement log-structured like file systems [24] 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 [15]. Several techniques have been recently proposed to improve the lifetime of the SSDs, for example [10], [14]. The recent work SFS [8] proposed to collect data hotness statistics at file block level and group data accordingly. 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 [12]. 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 [19], [25]. 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.

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 by 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 of two parts, traditional DRAM and Nonvolatile DIMMs. Figure 1 shows the hardware architecture of our system. We utilize the nonvolatile DIMMs attached to the memory bus and accessed through virtual addresses as NVRAM. All the page mapping information of NVRAM will be stored at a fixed location in NVRAM. We will detail this later in section III-E. 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) by the FTL at the 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-E.

B. 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

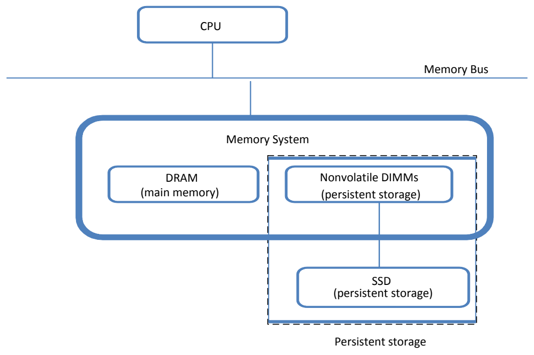


Fig. 1. Hybrid Storage Architecture

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. The total number of pages within clean and dirty LRU lists is fixed, equalling to the number of NVRAM pages.

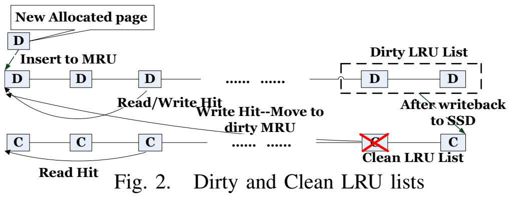


Fig. 2. Dirty and Clean LRU lists

Figure 2 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 at the MRU (Most Recently Used) position of dirty LRU list. Read/write operations on dirty data will update their position to MRU within the dirty LRU list. For clean data, read operations update their position to MRU of clean LRU list, while write operations 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 a 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 the dirty LRU list reaches a high bound (i.e. 80% of dirty LRU list is full). This process continues until the NVRAM pages within the 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.

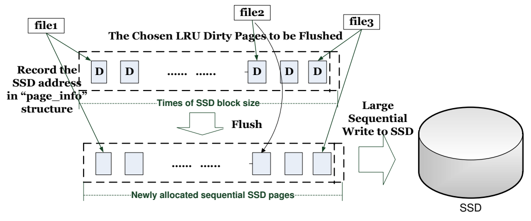

file1
file2
file3
The Chosen LRU Dirty Pages to be Flushed
Record the SSD address in "page_info" structure
D
Times of SSD block size
Flush
Large Sequential Write to SSD
Newly allocated sequential SSD pages
SSD

Fig. 3. Migrate Dirty NVRAM Pages to SSD

As shown in Figure 3, the dirty NVRAM pages will become clean after migrating to SSD and will be inserted to the LRU position of the clean LRU list. 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-D

C. 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 into large write requests (i.e. 512KB).

To facilitate allocation of extents on SSD, we need to periodically clean up internal fragmentation within the SSD. 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. As described earlier, we always write sequentially to SSD in units of 512KB, therefore the procedure of block erase at FTL is expected to benefit from our design. We will show how NVMFS impacts it in section IV-C.

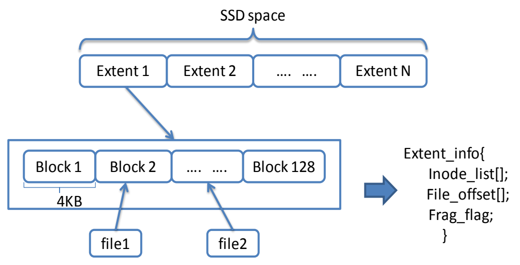


Fig. 4. Space management on SSD

Figure 4 shows the space organization of SSD. 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. This information is kept as metadata in a fixed space on NVRAM. 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.

D. File System Consistency

File system consistency is always an important issue in file system design. In this section, we describe the consistency issue related with data migration and segment cleaning process for our design.

As described in section III-B, 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 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, 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. 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.

Segment cleaning is another point prone to inconsistency. The cleaning process chooses one candidate extent (512KB) per round and migrates the valid blocks (4KB) to NVRAM, then updates the inodes to point to the new data positions, finally frees 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.

E. File System Layout

The space layout of NVMFS is shown in figure 5. 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.

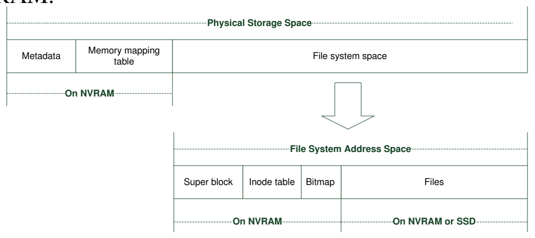


Fig. 5. Storage Space Layout

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. 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 [27].

In NVMFS, the directory files are stored as ordinary files. 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 block count of NVRAM, block count of SSD, 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

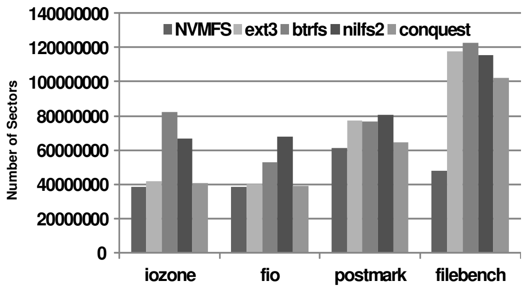


Fig. 6. Write traffic to SSD under different workloads and file systems

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.

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 [17], FIO [4] and Filebench [3] to evaluate the performance of our file system. For IOZONE, it creates a single large file and performs 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 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) [26]. The first three file systems are not designed for hybrid storage architecture. Therefore we configure 4GB DRAM-based page cache for them. The NAND flash SSD we used is Intel's X25-E 64GB SSD.

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.

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).

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 [20] as a representative hybrid FTL scheme and (b)page-level FTL [18]. In both schemes, we configure a large block 24GB NAND flash memory with 4KB page, 256 KB block, and 10% over-provisioned capacity. Figure 7 shows 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 and less erase overhead under all situations. Our benefits come from two aspects: 1)less write traffic to SSD; 2)large sequential writes to SSD.

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 8 shows the IO throughput while the segment cleaning is not activated with our file system and nilfs2. 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 9 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)".

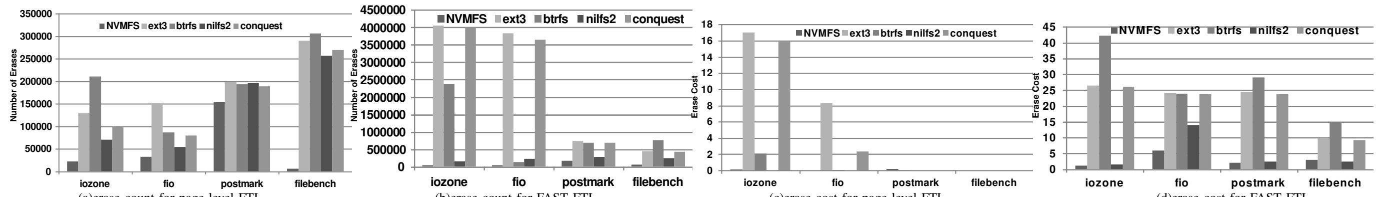
Figure 10 and 11 show the total number of recycled blocks and the cleaning efficiency respectively while running different workloads under NVMFS and nilfs2. We can see for all the workloads NVMFS recycled much fewer blocks compared with nilfs2. As shown in figure 11, we also see NVMFS has higher cleaning 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. 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.

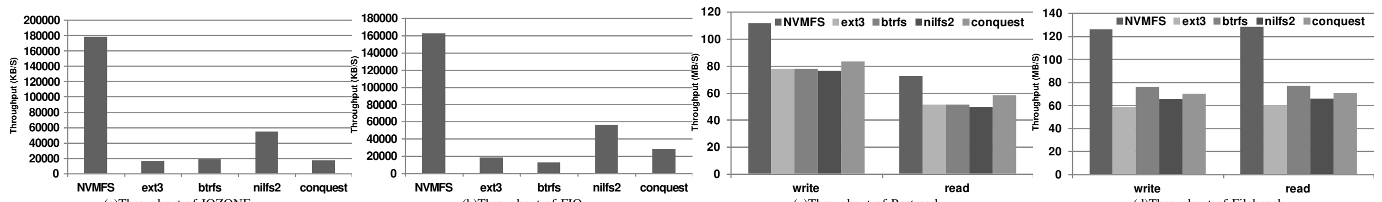
References

[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] "Filebench benchmark." [Online]. Available: http://sourceforge.net/apps/mediawiki/filebench/index.php
[4] "Fio benchmark." [Online]. Available: http://freecode.com/projects/fio
[5] "Iozone filesystem benchmark." [Online]. Available: http://www.iozone.org/
[6] "Non-volatile dimm." [Online]. Available: http://www.vikingtechnology.com/non-volatile-dimm
[7] L. Bouganim, B. T. Jónsson, and P. Bonnet, "uflip: Understanding flash io patterns," in *CIDR*, 2009.
[8] H. C. Changwoo Min, Kangnyeon Kim and Y. I. E. Sang-Won Lee, "Sfs: Random write considered harmful in solid state drives," in *FAST*, 2012.
[9] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *SIGMETRICS*, 2009.
[10] 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 *FAST*, 2011.



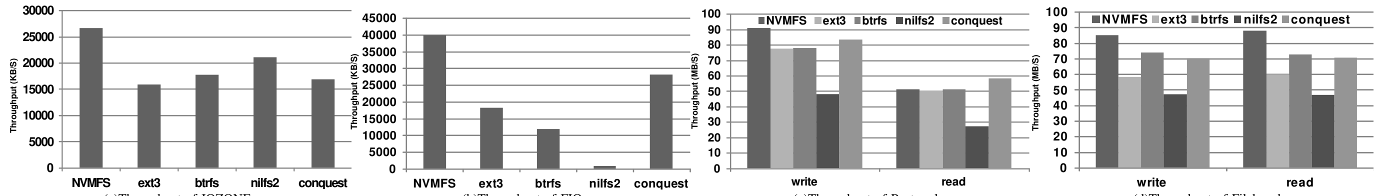
(a)erase count for page-level FTL (b)erase count for FAST FTL (c)erase cost for page-level FTL (d)erase cost for FAST FTL

Fig. 7. Erase count and erase cost for page-level and FAST FTL



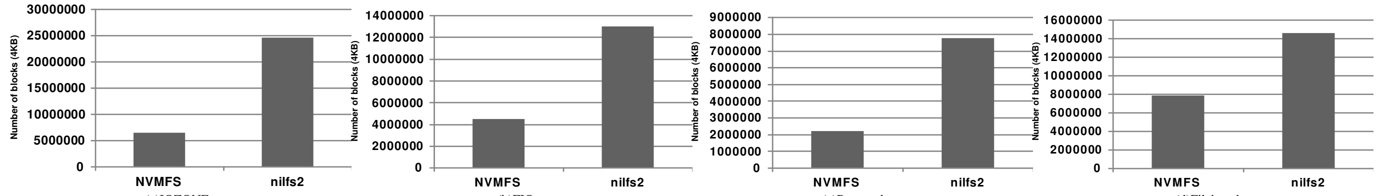
(a)Throughput of IOZONE (b)Throughput of FIO (c)Throughput of Postmark (d)Throughput of Filebench

Fig. 8. IO throughput under different workloads for 50% - 70% disk utilization



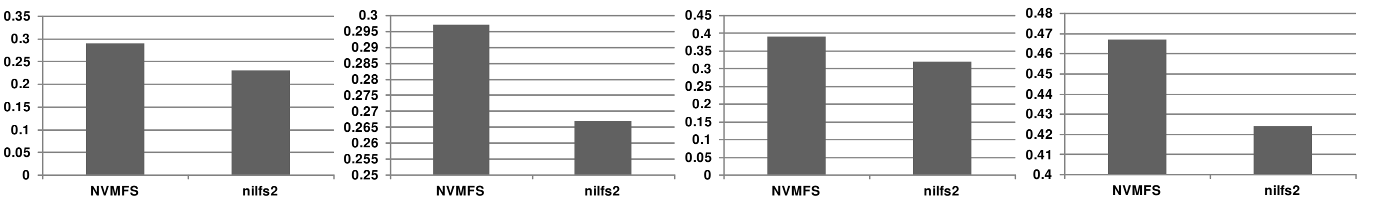
(a)Throughput of IOZONE (b)Throughput of FIO (c)Throughput of Postmark (d)Throughput of Filebench

Fig. 9. IO throughput under different workloads for over 85% disk utilization



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

Fig. 10. Total number of recycled blocks while running different workloads under NVMFS and nilfs2



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

Fig. 11. Cleaning efficiency while running different workloads under NVMFS and nilfs2

[11] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The rio file cache: surviving operating system crashes," in *ASPLOS*, 1996.

[12] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *FAST*, 2011.

[13] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS*, 2009.

[14] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds," in *FAST*, 2011.

[15] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *SYSTOR*, 2009.

[16] 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.

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

[18] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, 1995.

[19] H. Kim and S. Ahn, "Bplru: a buffer management scheme for improving random writes in flash storage," in *FAST*, 2008.

[20] 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.

[21] 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.

[22] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS*, 2012.

[23] 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 *SAC*, 2008.

[24] 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.

[25] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending ssd lifetimes with disk-based write caches," in *FAST*, 2010.

[26] 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.

[27] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *SC*, 2011.