DOI: 10.19729/j.cnki.1673-5188.2018.03.010
http://kns.cnki.net/kcms/detail/34.1294.TN.20180903.1719.002.html, published online September 3, 2018

Persistent Data Layout in File Systems

LUO Shengmei [1], LU Youyou [2], YANG Hongzhang [1], SHU Jiwu [2], and ZHANG Jiacheng [2]
(1. ZTE Corporation, Shenzhen 518057, China;
2. Tsinghua University, Beijing 100084, China)

Abstract

Data layout in a file system is the organization of data stored in external storages. The data layout has a huge impact on performance of storage systems. We survey three main kinds of data layout in traditional file systems: in-place update file system, log-structured file system, and copy-on-write file system. Each file system has its own strengths and weaknesses under different circumstances. We also include a recent usage of persistent layout in a file system that combines both flash memory and byte-addressable non-volatile memory. With this survey, we conclude that persistent data layout in file systems may evolve dramatically in the era of emerging non-volatile memory.

Keywords

data layout; file system; persistent storage; solid state drive (SSD)

1 Introduction

The organization of data stored outside of a core is data layout in file systems. The data layout influences the performance of a file system greatly. Data can be stored in various kinds of storage mediums.

In traditional file systems, disks are employed for storing data out of a core. The performance of a disk is greatly influenced by the time of tracking and locating a sector [1], [2]. To reduce the time of tracking, Fast File System (FFS), ext2, ext3 and other traditional file systems organize the storage space into groups of cylinders, which increases the locality of accesses [1], [3], [4]. To reduce the time of locating a sector, traditional file systems based on disks utilize caching, prefetching, pre-allocating and other technologies to access the data sequentially [4]. Thus, the data layout gathers the data in the successive physical sections as far as possible. Then it can increase the succession and locality of the data accessing, improving the performance of the file system.

In the state-of-the-art file system, solid state drives (SSDs) are used as the external storages [5]–[7]. Because SSDs are electronic memories, they do not have units of mechanical tracking and sector locating. Therefore, the influence of the optimization by gathering data and accessing data sequentially is futile. Nevertheless, the imbalance of read and write latency in SSDs and the lifetime problem are new challenges to SSDs. The random write performances worse than the random read in SSDs. Thus, the data layout in SSDs has the best able data to be written sequentially, and the lifetime problem of SSDs requires slighter write amplification. Then, fine-grained writes to SSDs are employed in file systems to update the data, and this prevents write amplification from page alignment. In addition, the garbage collection of file systems in SSDs should move the valid pages out before wiping the block. In conclusion, the data layout in SSDs not only influences the performance of the file system but also influences the lifetime of SSDs.

This work was supported by ZTE Industry-Academia-Research Cooperation Funds.

Besides the storage performance, the integrity and consistency of data should be taken into consideration. These two properties play an important role in the file system. For example, if a user is updating data in the file system through the cache and a power failure occurs, the data in the cache will not be written back to the external storage yet. Then it is unable for the user to know if the storage stores the latest data. It is crucial that the data will not lose and be inconsistent when there are some failures.

Besides security, the performance of file systems always attracts a lot of attention from researchers. Based on SSDs, we propose a multi-level file system named stageFS. It utilizes the cache to update data not in a granularity of a page but in a granularity of a record. It can decrease the number of writing to a large extent and then improve the performance of the file system.

The rest of this paper is organized as follows. Section 2 shows three kinds of traditional data layout in file systems. Section 3 describes the persistent storage in file systems. Section 4 presents the built-in multi-level persistent file system and the conclusion is given in Section 5.

2 Data Layout in File System

This section will introduce three kinds of traditional data layouts. They are in-place udpdate file system, long-structured file system (LSF), and copy-on-write file system. All the existing data layouts belong to these three kinds.

2.1 In-Place Update File System

In-place update file systems, such as ext2, ext3 and ext4, support overwrite operation. They enable data with continuous logical addresses to be stored continuously in disks, which improves the locality of data when being read. Nevertheless, in-place update file systems may lead data to be written dispersedly, which has impact on the performance of writing data. We will take ext4 as an example to illustrate the in-place update file system.

Ext4 is a representative in-place update file system. The data layout on the disk is shown in **Fig. 1**. An ext4 file system is divided into a lot of block groups. The block allocator always tries to allocate the blocks belonging to the same file into the same block group, which may reduce the time of tracking. In a block group, data are distributed as shown in Fig. 1. The first 1024 bytes in the block group 0 are used to install the boot block, and other block groups have no section like this. The super block describes and maintains the state of the file system, such as the gross of index nodes (inodes) and the used blocks. A portion of block group stores the redundant copies of super blocks and group descriptors. Not all the block groups have the copies. If one has no copy, it will start with the data block bitmap. Reserved Group Description Table (GDT) blocks are used for file system extensions.

In order to ensure that the logical continuous data are stored on the disk continuously, ext4 adopts five kinds of allocating strategy:

1) Multi-block allocator

When a new file is created, the block allocator assumes that it will grow with a high speed, thus allocating 8 KB of continuous disk space to it. When the file is closed, if this space does not be used, the unused part will be recycled; if used, the data are in a physically continuous space.

2) Delayed allocation

This strategy works with the cache. When a file needs more blocks to write due to updating data, the controller will not allocate blocks for it immediately, but until the data in cache must be written back to the disk (such as sync operation occurs and the memory is full). In this way, as many data as possible are stored in the cache, which is beneficial for allocating.

3) Allocating inodes and data blocks in the same block group

When the file system reads inodes of a file system and obtains the locations of data blocks, if the two are in the same block group, the time for tracking will reduce.

4) Inode and its directory in the same block group

When the file system reads the directory and obtains the ID of the inode, if the two are in the same group, the time for tracking will reduce.

5) Dividing the disk into several block groups

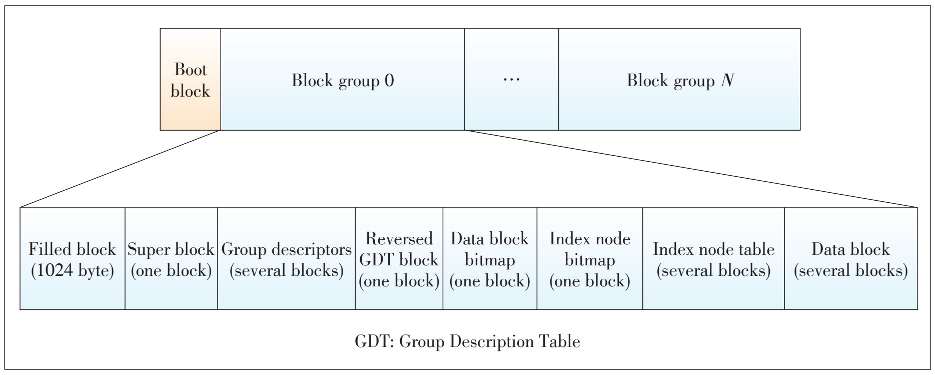
Trying best to allocate the blocks belonging to the same file in the same block group [1], [3], [8] will mitigate the problem of file fragmentation [4], [9], [10].

In in-place update file systems, there are allocation modes that allocating several blocks continuously with the extended segment mode, except the block-level allocation mode.

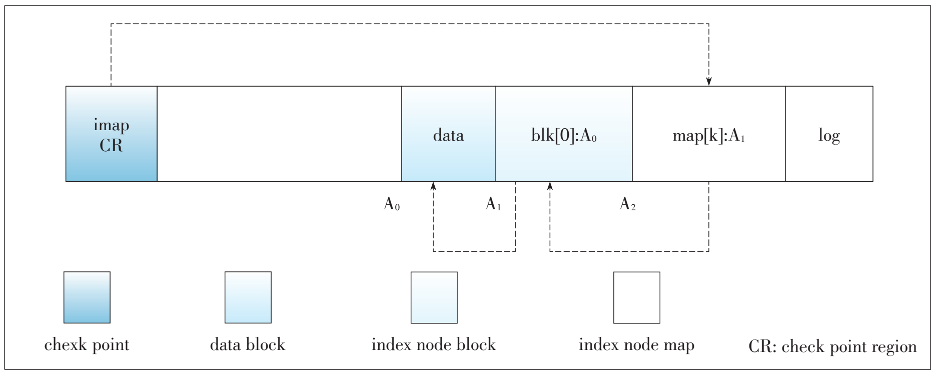
2.2 Log-Structured File System

LSF write data using the mode of append write to the external storage sequentially [11]. This write mode performs well when writing data, while it has a bad performance due to the random read. We can find that the in-place file system is good for reading while LSF is good for writing.

Fig. 2 shows the principles of LSF. It caches the file data in the memory and then writes data using append write to the external memory when the cache has no space. This ensures the sequential write to the storage. LSF also updates inodes sequentially. In order to solve the problem of file location, LSF introduces Inode Map (map of inodes) and check point region


Boot block
Block group 0
…
Block group N
Filled block (1024 byte)
Super block (one block)
Group descriptors (several blocks)
Reversed GDT block (one block)
Data block bitmap (one block)
Index node bitmap (one block)
Index node table (several blocks)
Data block (several blocks)
GDT: Group Description Table

▲**Figure 1. Data layout of the ext4 file system.**



▲**Figure 2. Data layout of a typical log-structured file system.**

(CR). Inode Map records the location of each inode. Through the Inode Map, the controller can locate the corresponding inode quickly and then finds the corresponding file data blocks. The Inode Map is also updated sequentially and the CR can locate the latest version of the Inode Map. The process of file updating is shown as the follow:
1) The file system data is written to the cache.
2) The data is written to the external memory when the capacity of the cache reaches a threshold.
3) During the sequential write process, the file data is updated first, followed by updating the inode; then the Inode Map is updated.
4) The region of CR is updated periodically.

In LSF, the following process of searching the file data according to the inode is different from that in in-place update file system:
1) Locate the latest version of Inode Map according to CR
2) Search the address of the inode in the Inode Map according to the ID of the inode
3) Locate the file data according to the inode.

Take Flash Friendly File System (F2FS) [12] as a typical example to introduce the LSF. F2FS is developed by Samsung based on SSDs. It divides a disk into a number of segments. Each segment has the fixed size: 2 MB. Each section is composed of adjacent segments and several adjacent sections compose a zone. Through the command \emph{mkfs}, one can easily change the sizes of section and zone.

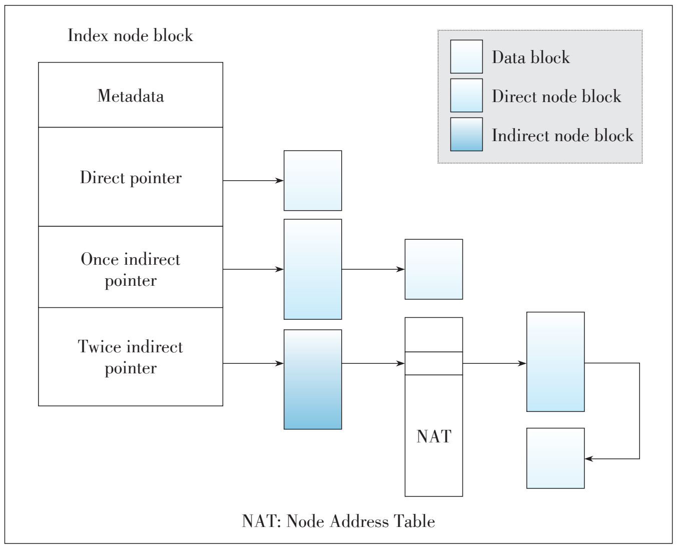
The layout of F2FS is shown in **Fig. 3**. F2FS divides the disk into two regions. One is the metadata region and the other is the data region. Each region is composed of several segments except the super block. The super block is located at the start of the zone, including some information of partition and default parameters. There are two backups of the super block in a system. The check point (CP) includes the states of a file system, the bitmap of Node Address Table (NAT)/Segment Information Table (SIT), the linked list of orphan nodes, the number of current active segments and other information. \emph {Segment Information Table} includes the information of each segment, such as the number of valid blocks and the valid bitmaps of blocks in main memory. \emph{Node Address Table} is used for searching the physical address according to the node ID. \emph{Segment Summary Area} (SSA) stores the owners' information of all blocks in main memory, such as the father node ID of one node and the offset of the node/data. The information in this section is mainly used for garbage collection. The main area includes the data of files and directories and their indexes. It also contains six logs for hot/warm/cold data and metadata.

F2FS mainly solves two following problems based on log-structured file system:

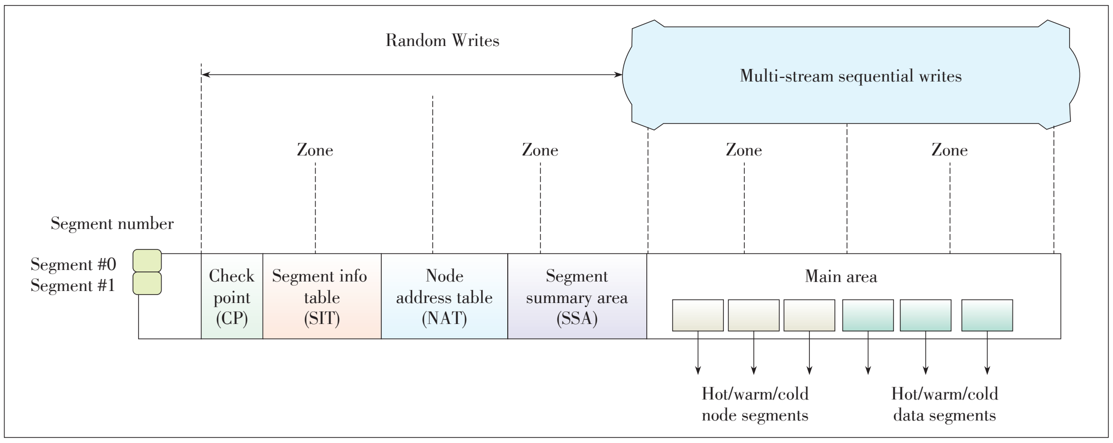
1) Wandering tree problem

The wandering tree problem is that when updating the block, the controller needs to update the pointer pointing to this block,update the pointer of this pointer, and then recur until the pointer pointing to the inode is updated.

F2FS solves this problem by introducing NAT, shown in **Fig. 4**. In traditional LFS, the ID of an inode is transferred into a physical address by Inode Map. F2FS extends this strategy. In F2FS, there are three kinds of node blocks: inode block, di-



▲**Figure 4. Node blocks in the flash friendly file system.**



◀**Figure 3. Data layout of the flash friendly file system.**

rect node block and indirect node block. Inode block includes the metadata of a file, such as the file name, the inode ID, the file size, the update time, the access time, the pointer directly pointing to the block, the pointer pointing to the direct pointer, the pointer pointing to the indirect pointer and other kinds of pointers. Each node has its own unique ID, called node ID. Through this ID, NAT can obtain the physical address of this node. For a file whose size is larger than 4 GB, LFS needs to update three pointer blocks, while F2FS only needs to update the direct node block and NAT. Actually, NAT in F2FS is in-place updated, only data in the main area are updated in the way of log-structured strategy.

2) Garbage collection

F2FS divides the data region into three levels: hot, warm and cold. Then it divides the data region into six levels by combining the division of node blocks and the division of data blocks. Compared with traditional LFS, only the log region is different in F2FS. F2FS maintains six active log regions for data to be written.

2.3 Copy-on-Write File System

The copy-on-write [13]–[15] refers to a new version of the file unit data created in a different location. Typically, an in-place file system updates the data to its original location, while the copy-on-write technology updates the data to a new location and update the file pointer. We take btrfs as an example here.

Btrfs is a Linux file system based on copy-on-write, developed by several companies. It supports a variety of advanced features and is expected to become the next generation of Linux standard file system. Btrfs supports copy-on-write, B-tree metadata management, and dynamic inode allocation.

1) Copy-on-Write

Fig. 5 shows the updating process of traditional file system data. When the file is updated, the data is written to the original location. If the system crashes, it will cause the data block to be in the semi-updated state, and destroy the consistency of the file data.

By using the copy-on-write technology (**Fig. 6**), the file will remain consistent before updating, if the system does not crash. If the crash does not occur and the file pointer is updated after the data block update is completed, the file can keep a consistent state. Therefore, copy-on-write is very effective to maintain file consistency.

2) B-tree metadata management

For ext2, ext3 and other file systems, their directory organization hinders their scalability. In ext2/3, their directories are linearly organized; when there are too many files in one directory, the number of corresponding directory entries increases, which results in increasing lookup times. Btrfs uses B-tree to manage metadata, which solves the problem of time-consuming searching of directory entries, so it has strong expansibility.

3) Dynamic inode allocation

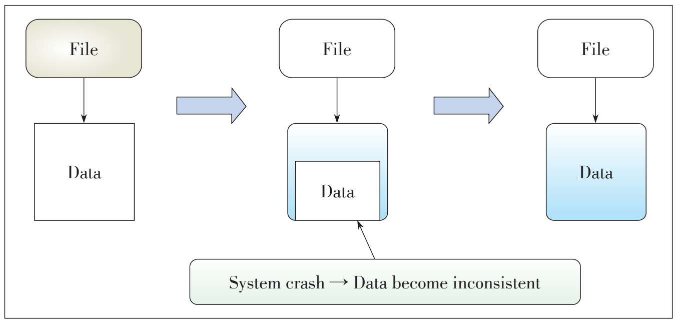
In each block group of ext2, the inode area is allocated fixedly in advance, which means that it can accommodate up to a limited number of inodes. Therefore, each partition creates a limited number of files, which may seriously affect its scalability. In btrfs, the physical storage location of an inode is no longer fixed, so users can create unlimited files anywhere. Therefore, btrfs has better scalability.

3 Persistent Storage in File System

The ultimate goal of a file system is to store a large number of data to a persistent storage in an organized way. These storage devices are different from the memory when an emergency power-off occurs: the persistent storages do not lose data while the memory will. How to realize the persistent storing is a critical issue, which can ensure the integrity and the persistency of data. The following subsections show two main kinds of persistent storages in file systems.

3.1 File System with Journaling

A journaling file system [16]–[21] uses a data structure



▲Figure 5. In-place update of traditional file system data.

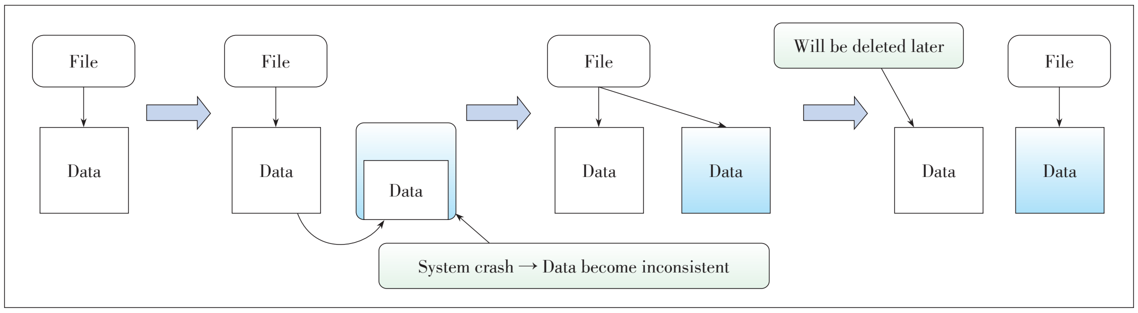


Figure 6.▶
Copy-on-write update.

named journal to record the changes of data which have not been committed to the main part of the file system.

The basic structure of journal is shown in **Fig. 7**. A journaling file system can be recovered more quickly from a system crash or a power failure [18]. It may only keep the track of metadata in the actual implementation. This will improve the performance. A journaling file system may track both the metadata and the corresponding data and some implementations allow users to select the behaviors to use. No matter what the condition is, it needs several separate write operations to reflect changes of data to files when updating file systems. We take deleting a file from a file system as an example to explain why journaling is essential. Deleting a file goes three steps:

1) Remove the directory entry of the file
2) Release the inode and add the released inode to the free inode pool
3) Return all of disk blocks used by this file to the free disk block pool.

If there is a crash between step 1 and step 2, an orphan node occurs and a storage link happens. It is same bad when the crash happens between step 2 and step 3, because the file which has not been deleted yet will be marked deleted and something will be written on the block to cover the undeleted block.

To prevent these problems, a journaling file system provides a journal structure which records changes of data before the change operation occurs [22]. The journal in some systems can change its size dynamically like a regular file, while in other systems it has a fix size and must be allocated in a certain contiguous area. In the second situation, the journal cannot be moved and the file system is mounted. There are also some file systems that allow the journal to be allocated on external separate device, such as SSDs and other non-volatile memories (NVMs). The journal may be distributed on several storages in order to avoid device crash.

When the journal itself is being written to, the journal must guard against crashes. Many journal implementations (such as the JBD2 layer in ext4) gather each change logged with a checksum. If a crash leaves a partially written change with a missing (or mismatched) checksum, the system can simply ignore it when replaying the journal after the recovery from the crash.

There are two kinds of journals, one is physical journal and the other is logical journal. A physical journal is used to log copies of blocks which will be written to the file system latter. If a crash occurs when the blocks are being written to the file system, the system just needs to replay the write in the journal to complete the operation when the file system recovers from the crash. If a crash occurs when the write is being logged to the journal, the partial write will miss or mismatched checksum and can be ignored when the file system recovers from the crash. A physical journal takes a performance penalty because each block changed must be committed twice. However, this may be acceptable when absolute fault protection is required.

A logical journal is used to store changes to metadata in the journal. A file system with a logical journal can recover quickly after a crash, but may allow the inconformity of unlogged file data and logged metadata. For example, appending to a file may involve three separate writes:

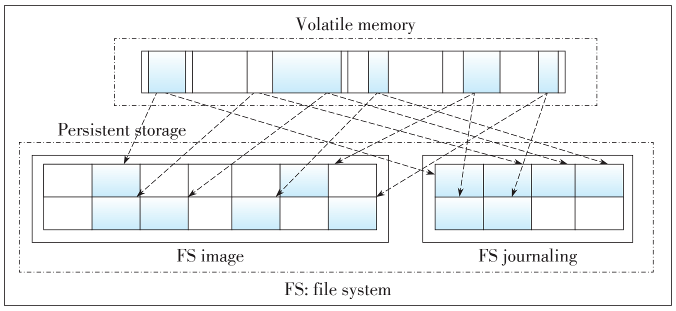
1) Writes to the inode of the file and note in the metadata of the file that its size has increased
2) Writes to the free space map and mark out an allocation of space for the data that will be appended
3) Writes to the newly allocated space and write the appended data actually.

In a metadata-only journal, step 3 is not logged. If step 3 is not done, but steps 1 and 2 are replayed during recovery, the file will be appended with garbage.

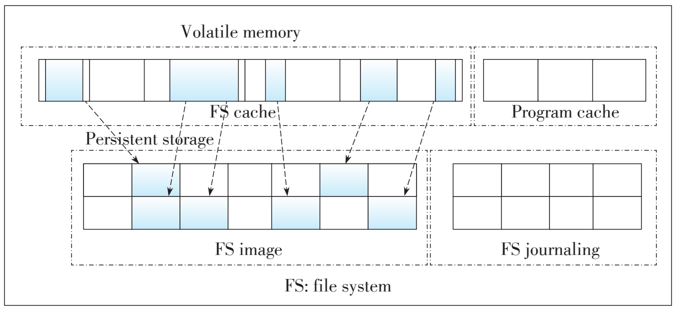
3.2 File System with Virtual Memory

Virtual memory [23]–[26] is a technique to manage memory. It employs both software and hardware to map virtual addresses used by a program to physical addresses. The translation hardware in CPU translates virtual addresses to physical addresses automatically. The software in a file system with virtual memory can extend capabilities by providing a larger virtual address space when more physical storages are added in the system. File systems with virtual memory divide a virtual address space into pages. Pages are blocks whose virtual memory addresses are contiguous. The size of a page is usually at least 4 kilobytes.

The basic structure of file systems with virtual memory is shown in **Fig. 8**. It is expected that applications have a continuous memory space, however, the physical blocks are actually



▲Figure 7. Basic structure of journaling file system.



▲Figure 8. Basic structure of file system with virtual memory.

stored dispersedly. Some blocks may even be stored in external storages; they are swapped into the memory when they will be used.

Virtual memory benefits applications by freeing them from managing a shared memory space, which will improve the security because the memory is isolated well [26]. It can also use more memory conceptually by the paging technique. When the memory is full, it will employ the persistent storage to work as an extended part of memory. In a traditional file system, a part of disk is used as the extended memory. With the rise of SSDs, there are some systems employing an SSD (due to its low cost, power efficiency and so on) as the extended memory [27]–[29], such as NVMalloc [30] and FlashVM [31].

NVMalloc was proposed to employ NVM as a secondary memory partition for applications to allocate explicitly and use memory regions in it. NVMalloc provides an NVMalloc library with a series of services, enabling applications to access NVM storage. With NVMalloc, files can be accessed in a byte-addressable fashion by using the memory mapped I/O interface. The approach in NVMalloc is able to re-energize computations outside of the core on large scale machines. This increases the capacity of the memory. NVMalloc shows that it can compute larger size of problem than the physical memory whose manner is cost-effective manner. In addition, it has better performance and efficiency when computing time or data access locality increases.

FlashVM focuses on high performance, reduced flash wear-out for improved reliability, and efficient garbage collection. It modifies the code paths for allocating/reading/writing pages in order to optimize the performance of flash. FlashVM further uses zero-page sharing and page sampling to reduce the number of page writes. It also makes full use of the discard command and provides fast online garbage collection of free VM pages.

4 Built-in Multi-Levels Persistent File System

The file layout of the flash file system not only affects the performance of the flash storage system, but also has impact on the control of life loss of flash [12], [32]–[34]. However, the data layout of the flash file system has different requirements for different operations, and the file system step-by-step operation limits the optimization of the data layout, which is mainly reflected in two aspects:

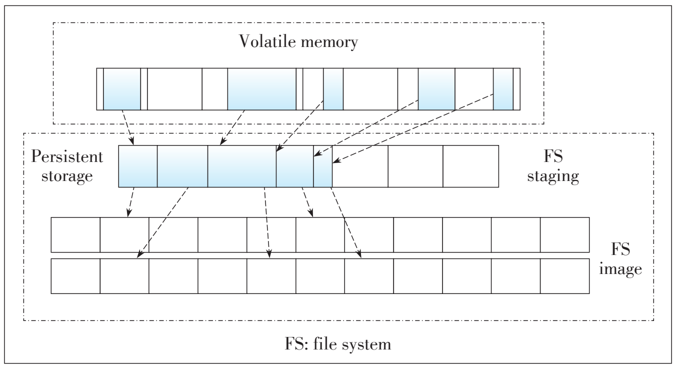
1) Fine-grained writes conflicts with page granularity reads. Flash memory write operations expect fine-grained writes to extend flash memory life. Flash read operations expect a page granularity read to improve read performance.
2) The conflict between synchronization and data layout is optimized. Because of consistency or persistence requirements, the file system provides an application that is explicitly called synchronous connections (such as fsync) or uses operating system background processes (such as pdflush) to synchronize data frequently to external memory. Synchronization reduces the duration between data persistence and data buffering. This reduces the probability of data merging on the same page. The update of valid data in a single shell is small. On the other hand, synchronization reduces the amount of data buffer, resulting in lacking data for effective classification, reducing the accuracy of data packets and affecting the optimization of the data layout effect.

Thus, StrageFS is proposed as a solution to these issues.

The basic structure of stageFS is shown in **Fig. 9**. The SSD managed by stageFS is divided into two spaces: FS Staging and FS Image. FS Staging provides the persistent storage for the recent write to the file system, while FS Image storages the other data in the file system. In the staging phase, only the dirty parts need to be written back into FS Staging in the way of recording. A record that has been marked with a unique ID is a dirty part in a page. In FS Staging, files are deleted in the grain of a page and the delete operation happens when the space of FS Staging is going to be used up. At this time, stageFS merges the pages in FS Staging with the pages in FS Image and patch them to FS Image. After the patching operation, stageFS erases the whole space in FS Staging.

StageFS includes two phases, the first one is staging phase and the second one is patching phase.

The staging phase is designed to provide efficient persistence to file system, for example, to write file system updates to persistent storage efficiently. The goal includes both high write performance and low writes amplification. The staging phase provides data durability to the content updates, including updates to file pages, directory entry pages, and index nodes. StageFS tracks the dirty bytes of each page instead of marking a page dirty. It records the write location of each write request, including the offset and length in each page where the request updates. When I/O synchronization is required, StageFS iterates all dirty files in the file system. For each dirty file, its dirty pages are performed using either full-page steal or record-level logging, according to their dirty granularity, hotness, etc. Full-page steal write is to steal pages from the hidden area in FS image and write dirty pages in full pages. Record-level logging write is to update data in the granularity of record and



▲Figure 9. Basic structure of StageFS.

compact these dirty parts to the staging area. Each record has a logical ID for identification, an offset for marking the start address and a length. In the implementation of StageFS, the dirty parts of a file are tracked in the logical ID tuples in the page cache. These dirty parts are indexed in a hash table, which keeps an ordered linked list to store the tuples for each file.

The patching phase is designed to accumulate data in the input datasets and improve the effect of data layout optimization. In the patching phase, space allocation in the FS image for file system updates is performed lazily. In the staging phase, each update is appended to the staging log with only the logical ID. With the ID, its offset and length in the file are known. In the patching phase, space allocation is performed to reorganize the data into a better layout:

- Page-level indexing that is transformed from the non-indexed record-level logging
- More sequential accesses by merging and reordering random writes.

The updates in the file system are written to FS Image in the granularity of page, and are written back by using the memory copies. As memory pages of the staging data are pinned in the main memory, it does not need to scan and merge the variable length records in the staging area. Therefore, file system updates are written twice: one is to FS Staging in record level for data durability, and the other is to FS Image in page level for data indexing.

StageFS needs to ensure consistency in both the staging phase and the patching phase. In the staging phase, file system updates are written in a log-structured way. StageFS treats each synchronized write as a transaction and uses the padding record as the commit record, which indicates the end of a transaction. For a synchronized write, a new page is allocated to be the padding record. Therefore, every synchronized write has a padding record to indicate its completeness. Though the padding record is used as the commit record, there is no ordering between data/inode record writes and the padding record write. An unwritten page has all ‘0’s, and the partially written page is detected by checking the Error Correction Code (ECC). If any page in one transaction is not written, the transaction is not committed. During recovery after failures, content updates in the staging area need to be merged with corresponding pages in the file system image. StageFS reads the updates of files or directories in the staging area, and marks their inode pages in icache as obsolete by setting theirs obsolete bits. StageFS delays the merge operation to the succeed I/O accesses. Therefore, I/O operations during recovery need to check the obsolete bit in icache before performing read or write operations. If the obsolete bit is set, data pages in the file system image are read to the page cache followed by the updates from the staging area. In the patching phase, StageFS pre-allocates all the space that is needed in the current patching phase when starting the patching operation. Then it writes the bitmap changes to the tail of the staging logging. Only after the bitmap changes are persistently written, the patching writes are performed. If system fails during patching, bitmap changes are read to check the write statuses of the staging data. If the patching fails, StageFS marks all corresponding pages of the bitmap changes as invalid, and then restarts the patching phase by allocating space and writing the staging data to the FS image.

5 Conclusions

This paper introduces the data layout in file systems. First, we give the introduction of disks and SSDs. Their difference requires us to design a suitable data layout for SSDs instead of directly using the data layout in file systems based on disks. Second, we introduce three kinds of traditional data layouts in file systems and analyze their advantages and disadvantages in different circumstances. Third, we take persistent storage into consideration. We introduce journaling file system first, and then we introduce virtual memory. Besides, we give a brief introduction on SSDs used as the extended memory in virtual memory file systems. Finally, we propose a new file system based on SSDs, named as stageFS. It employs FS Staging which likes a cache in the system and each updating only writes the dirty parts of the page into FS Staging. At the moment that FS Staging is almost full, these data are being written back to FS Imaging in the grain of a page. StageFS employs the technologies performing well in SSDs and also has a new multi-level structure, archiving better performance in the file system based on SSDs.

References

[1] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 3, pp. 181–197, 1984.
[2] D. Hitz, J. Lau, and M. A. Malcolm, "File system design for an NFS file server appliance," in *Proc. USENIX Winter*, San Francisco, 1994, vol. 94, pp. 19–19.
[3] S. Tweedie, "Ext3, journaling filesystem," in *Ottawa Linux Symposium*, Ottawa, Canada, 2000, pp. 24–29.
[4] M. Cao, S. Bhattacharya, and T. Ts'o, "Ext4: the next generation of ext2/3 filesystem," in *Linux Storage & Filesystem Workshop (LSF)*, San Jose, USA, 2007.
[5] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *ACM SIGMETRICS/Performance*, Seattle, USA, 2009.
[6] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," in *USENIX Conference on File and Storage Technologies*, San Jose, USA, 2010, pp. 101–114.
[7] N. Agrawal, V. Prabhakaran, T. Wobber, et al., "Design tradeoffs for SSD performance," in *USENIX Annual Technical Conference*, Boston, USA, 2008, pp. 57–70.
[8] R. Card, T. Ts'o, and S. Tweedie, "Design and implementation of the second extended filesystem," in *Proc. First Dutch International Symposium on Linux*, Groningen, Netherlands, 1994.
[9] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: the linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, article no. 9, 2013. doi: 10.1145/2501620.2501623.
[10] R. Y. Wang and T. E. Anderson, "XFS: a wide area mass storage file system," in *IEEE Fourth Workshop on Workstation Operating Systems*, Napa, USA, 1993, pp. 71–78. doi: 10.1109/WWOS.1993.348169.

[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, Feb. 1992. doi: 10.1145/146941.146943.
[12] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, USA, 2015, pp. 273–286.
[13] Z. N. J. Peterson, "Data placement for copy-on-write using virtual contiguity," Ph.D. dissertation, University of California Santa Cruz, USA, 2002.
[14] D. Hitz, M. Malcolm, J. Lau, and B. Rakitzis, "Copy on write file system consistency and block usage," U.S. Patent 6 892 211, May 10, 2005.
[15] W. A. Sawdon and F. B. Schmuck, "Deferred copy-on-write of a snapshot," U. S. Patent 6 748 504, Jun. 8, 2004.
[16] B. J. Fuller, "Single transaction technique for a journaling file system of a computer operating system," U.S. Patent 6 021 414, Feb. 1, 2000.
[17] J. Piernas, T. Cortes, and J. M. Garcia, "Dualfs: a new journaling file system without meta-data duplication," in *ACM 16th International Conference on Supercomputing*, New York, USA, 2002, pp. 137–146. doi: 10.1145/514191.514213.
[18] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *USENIX Annual Technical Conference*, Anaheim, USA, 2005, pp. 105–120.
[19] Z. Zhang and K. Ghose, "yFS: a journaling file system design for handling large data sets with reduced seeking," in *2nd USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, USA, 2003, pp. 59–72.
[20] M. T. Jones, "Anatomy of linux journaling file systems," IBM DeveloperWorks, USA, 2008.
[21] M. I. Seltzer, G. R. Ganger, M. K. McKusick, et al., "Journaling versus soft updates: asynchronous meta-data protection in file systems," in *USENIX Annual Technical Conference*, San Diego, USA, 2000, pp. 71–84.
[22] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *10th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, USA, 2012, pp. 9–9.
[23] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.
[24] K. Li, "Shared virtual memory on loosely coupled multiprocessors," Yale University, New Haven, USA, Tech. Rep., 1986.
[25] P. J. Denning, "Virtual memory," *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970.
[26] A. W. Appel and K. Li, "Virtual memory primitives for user programs," in *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, USA, 1991, vol. 26, no. 4.
[27] A. Badam and V. S. Pai, "SSDAlloc: hybrid SSD/RAM memory management made easy," in *Proc. 8th USENIX Conference on Networked Systems Design and Implementation*, Boston, USA, 2011, pp. 211–224.
[28] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using NVM as virtual memory," in *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*, Boston, USA, 2013, pp. 29–40. doi: 10.1109/IPDPS.2013.69.
[29] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister, "Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2285–2294, Dec. 2012. doi: 10.1109/TVCG.2012.240.
[30] C. Wang, S. S. Vazhkudai, X. Ma, et al., "Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines," in *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai, China, 2012, pp. 957–968. doi: 10.1109/IPDPS.2012.90.
[31] M. Saxena and M. M. Swift, "Flashvm: Virtual memory management on flash," in *USENIX Annual Technical Conference*, Boston, USA, 2010.
[32] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, USA, 2013.
[33] Y. Lu, J. Shu, and W. Wang, "ReconFS: a reconstructable file system on flash storage," in *Proc. 12th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, USA, 2014, pp. 75–88.
[34] J. Zhang, J. Shu, and Y. Lu, "ParaFS: a log-structured file system to exploit the internal parallelism of flash devices," in *USENIX Annual Technical Conference*, Denver, USA, 2016.

Manuscript received: 2017-10-26

Biographies

LUO Shengmei (luo.shengmei@zte.com.cn) received his master's degree from Harbin Institute of Technology, China. He has been working with ZTE Corporation for over 20 years. His research interests include cloud computing and big data. He is a member of CIE and CCF.

LU Youyou (luyouyou@tsinghua.edu.cn) received the B.S. degree from Nanjing University, China in 2009 and the Ph.D. degree from Tsinghua University, China in 2015, both in computer science. He is currently an assistant researcher in the Department of Computer Science and Technology, Tsinghua University. His current research interests include nonvolatile memories and file systems. He received the best paper award at IEEE NVMSA'14 and the best paper runner-up at MSST'15. He is a member of the IEEE, ACM and CCF.

YANG Hongzhang (yang.hongzhang@zte.com.cn) received his master's degree in computer science and technology from University of Chinese Academy of Sciences, China in 2015. He has been working with ZTE Corporation for 3 years. His research interests include distributed file system and cloud computing. His paper on pNFS was received by HPCA'15. He is a member of the IEEE, ACM and CCF.

SHU Jiwu (shujw@tsinghua.edu.cn) received the Ph.D. degree from the Department of Computer Science and Technology, Nanjing University, China. He is currently a professor in the Department of Computer Science and Technology, Tsinghua University, China. His current research interests include nonvolatile memories and file systems. He is IEEE Fellow and CCF Fellow.

ZHANG Jiacheng (zhang-jc13@mails.tsinghua.edu.cn) received the B.S. degree from Harbin Institute of Technology, China, in software engineering in 2013 and is now a Ph.D. candidate student in Tsinghua University, China, majoring in computer science. His current research interests include nonvolatile memories and storage system. His paper on flash-based file system was received by USENIX ATC'16.