SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **46**:291–318
Published online 19 November 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2299

TridentFS: a hybrid file system for non-volatile RAM, flash memory and magnetic disk

Ting-Chang Huang[1] and Da-Wei Chang[2,*,†]

[1]*Department of Computer Science, National Chiao Tung University, No. 1001, University Road, Hsinchu, 300, Taiwan*
[2]*Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan, 701, Taiwan*

SUMMARY

A hybrid file system with high flexibility and performance, called Trident file system (TridentFS), is proposed to manage three types of storage with different performance characteristics, that is, Non-Volatile RAM (NVRAM), flash memory and magnetic disk. Unlike previous NVRAM-based hybrid file systems, novel techniques are used in TridentFS to improve the flexibility and performance. TridentFS is flexible by the support of various forms of flash memory and a wide range of NVRAM size. The former is achieved on the basis of the concept of stackable file systems, and the latter is achieved by allowing data eviction from the NVRAM. TridentFS achieves high performance by keeping hot data in the NVRAM and allowing data evicted from the NVRAM to be parallel distributed to the flash memory and disk. A data eviction policy is proposed to determine the data to be evicted from the NVRAM. Moreover, a data distribution algorithm is proposed to effectively leverage the parallelism between flash memory and disk during data distribution. TridentFS is implemented as a loadable module on Linux 2.6.29. The performance results show that it works well for both small-sized and large-sized NVRAM, and the proposed eviction policy outperforms LRU by 27%. Moreover, by effectively leveraging the parallelism between flash memory and disk, the proposed data distribution algorithm outperforms the RAID-0 and a size-based distribution method by up to 471.6% and 82.6%, respectively. By considering the data size and performance characteristics of the storage, the proposed data distribution algorithm outperforms the greedy algorithm by up to 15.5%.

Received 13 November 2013; Revised 14 September 2014; Accepted 29 October 2014

KEY WORDS: hybrid file systems; non-volatile memory; flash memory; magnetic disk; file distribution

1. INTRODUCTION

Nowadays, the media for data storage in modern computer systems can be magnetic disk, flash memory, non-volatile random access memory (NVRAM) or a combination of the above media. Magnetic disks have been widely used as major storage devices for decades because of their large capacity and cost effectiveness. However, the mechanical access in disks prevents significant improvement in performance and therefore leads to an increasing performance gap between the disk and memory. In recent years, with the developments in material technology, the emergence of new storage media such as flash memory and NVRAM have provided alternative solutions for data storage. Flash memory provides several advantages over magnetic disks, such as lower access time, lower power consumption and more shock resistance. The growing popularity of using flash memory in multiple domains, for instance commercial servers and portable devices, shows a trend indicating that flash memory has become a viable competitor to magnetic disks. In addition, NVRAM

*Correspondence to: Da-Wei Chang, Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan.
†E-mail: dwchang@mail.ncku.edu.tw

or storage-class memory, such as phase change RAM (PCRAM) [1], Resistive RAM [2] and Magneto-resistive RAM (MRAM) [3], is a kind of random access memory with non-volatility. Their near-DRAM speed and non-volatility are the most attractive features for high performance data storage. However, high density NVRAM is still not fully matured. Current available NVRAM is both expensive and small in size.

On the basis of different material technologies, these three types of storage have different performance characteristics. In the case of a magnetic disk, read and write operations have similar performance. Compared with sequential accesses, random accesses have lower throughput because of more time-consuming mechanical accesses (e.g., disk-head movements). Flash memory has much lower access latency than a magnetic disk, and it achieves performance superior to a magnetic disk under random accesses because no mechanical parts are used. However, write operations are slower than read operations in flash memory. Generally, there are two types of flash memory: NOR flash and NAND flash. NAND flash is widely used as data storage because of its high density. Different forms of NAND flash memory is utilized under different environments, which require different management approaches. For example, in embedded systems, raw NAND flash chips are usually used, and native flash memory file systems (e.g., YAFFS [4]) are required to directly manage the chips. In desktop or server systems, solid state disks (SSDs), which adopt flash translation layers [5] to emulate block devices on the basis of NAND flash memory, are typically used, and block-based file systems (e.g., ext3/4 and NTFS) are used to manage the SSDs. NVRAM has similar characteristics to those of DRAM but provides a non-volatility feature. Specifically, it supports byte granularity accesses and comparable speed to DRAM.

The characteristics of different types of storage have motivated the NVRAM-based hybrid file systems, which manage NVRAM and the other types of storage. Several NVRAM-based hybrid file systems have been proposed for improving performance or extending storage lifetime [6–9]. For example, Conquest [7] and HeRMES [8] hybrid file systems improve storage performance by combining battery-backed DRAM and MRAM, respectively, with magnetic disks. In these file systems, specific data (e.g., metadata or smaller file) are stored in the NVRAM, causing the reduction of time-consuming small-sized accesses to the disk and hence improving storage performance.

Two problems exist when using existing NVRAM-based hybrid file systems for managing a hybrid storage system consisting of NVRAM, flash memory and disk. The first one is a lack of flexibility. Specifically, some assumptions are made on the form or size of the storage media in existing NVRAM-based hybrid file systems. For example, PFFS [9] assumes raw flash memory chips are used, and it cannot support flash translation layer-managed NAND flash memory such as SSDs. Another example is that, with the assumption of large NVRAM size, Conquest forces all the metadata to be stored in the NVRAM. Such enforcement could result in NVRAM depletion problem, which means that the file system is unable to handle file operations requiring NVRAM space when the NVRAM is depleted. Moreover, because of the small size of current available NVRAM, a large amount of space in the disk or flash memory may remain unused when NVRAM depletion occurs, leading to low space utilization of the disk or flash memory. For example, according to a previous study [8], the space consumed by metadata could be up to 2% of the entire file system space. Thus, when 384-MB NVRAM and 2-TB disk are used, over 99% of the disk space may still be unused when the NVRAM is depleted.

The second problem is that the parallelism among different types of storage is not leveraged effectively. Existing NVRAM-based hybrid file systems support a single type of storage except the NVRAM, and therefore, the issue of parallelism among different types of storage is not addressed. However, managing two types of storage (i.e., flash memory and disk) in addition to the NVRAM requires effectively leveraging the parallelism between these two types of storage for achieving high performance.

To address the above problems, an NVRAM-based hybrid file system called Trident file system (TridentFS) is proposed in this paper. Three types of storage, that is, NVRAM, flash memory and magnetic disk, are managed by TridentFS. It is aimed at achieving both high flexibility and high performance. To achieve high flexibility, TridentFS prevents the occurrence of the NVRAM depletion problem by adopting a data eviction scheme to allow data to be evicted from the NVRAM when the NVRAM runs out of its free space. With data eviction, a wide range of NVRAM size

DOI: 10.1002/spe

can be supported in TridentFS. Moreover, on the basis of the concept of stackable file systems, TridentFS can support flash memory in different forms by re-using existing file system implementations. TridentFS achieves high performance by maintaining hot data in the NVRAM and by keeping load balance between the flash memory and disk. Maintaining hot data in the NVRAM reduces the I/O accesses to slower flash memory and disk. To maintain hot data in the NVRAM, an eviction policy is adopted for selecting to-be-evicted data according to the frequency, recency and size of the data. For effectively leveraging the parallelism between the flash memory and disk, data evicted from the NVRAM is distributed to the flash memory and disk according to the runtime storage performance. Data size and performance characteristics of the storage are also considered to speed up data distribution.

We have implemented TridentFS as a loadable module in Linux 2.6.29. For performance evaluation, part of the memory is emulated as NVRAM, and off-the-shelf flash memory-based SSDs are used as the flash memory storage. Six benchmarks are used for performance evaluation. Through evaluation, the flexibility of TridentFS is demonstrated by the following. First, different types of existing file system implementations (ext4 [10], nilfs2 [11] and FAT) can be reused in TridentFS to manage the SSD and the disk. Second, it works well for both small-sized and large-sized NVRAM. In addition, the performance results are as follows. First, maintaining hot data in the NVRAM can achieve performance improvement by up to 115.3%. Second, the proposed eviction policy outperforms LRU policy by 27%. Third, because of effectively leveraging the parallelism between the flash memory and disk, the proposed data distribution algorithm outperforms the RAID-0 method and a size-based distribution method adopted by Conquest [7] by up to 471.6% and 82.6%, respectively. Fourth, by considering the data size and performance characteristics of the storage, the proposed data distribution algorithm outperforms the greedy algorithm by up to 15.5%.

The remainder of this paper is organized as follows. Section 2 describes the related work, which is followed by the description of the design and implementation of TridentFS in Section 3. Section 4 presents the performance results. Discussions and conclusions are given in Sections 5 and 6, respectively.

2. RELATED WORK

The related research can be divided into three categories: NVRAM-based file systems, stackable file systems and file distribution in parallel storage systems. Below, we briefly describe these works.

2.1. NVRAM-based file systems

Many NVRAM-based file systems have been proposed. The common idea is to place critical or hot data in the NVRAM. Briefly, these systems can be divided into two categories: using NVRAM as a write buffer and using NVRAM as storage. In [12–15], NVRAM was used to buffer data (or metadata) written to flash memory or to disk. Because data stored in NVRAM are permanent, periodic flushes of dirty data can be eliminated, and storage writes are needed only when the NVRAM is full. The performance is improved because of the reduction of the amount of storage writes. Instead of using NVRAM as a write buffer, several previous studies have utilized NVRAM as permanent storage [16–18]. MRAMFS [16] and BPFS [17] proposed file systems for managing MRAM and PCRAM, respectively. MRAMFS adopts simplified metadata and online data compression to efficiently use size-limited MRAM. BPFS proposed hardware support for atomic and ordered updates in PCRAM. NVRAM is also used as permanent storage in several hybrid file systems. In these systems, NVRAM is used to store small data (e.g., metadata) for performance improvement. For example, NVMFS [6], Conquest [7], HeRMES [8], PFFS [9], MiNVFS [19] and FRASH [20] store all the metadata in the NVRAM to reduce slower disk or flash memory accesses. Because all the metadata are forced to be stored in the NVRAM, existing NVRAM-based file systems cannot entirely eliminate the NVRAM depletion problem. Most notably, the problem could easily occur in the case of a small NVRAM.

Similar to existing NVRAM-based hybrid file systems, TridentFS also manages NVRAM and the other types of storage in a hybrid manner. However, unlike existing NVRAM-based hybrid file systems, TridentFS eliminates the NVRAM depletion problem by evicting data (including both

DOI: 10.1002/spe

metadata and file data) from the NVRAM to the flash memory or disk when NVRAM is nearly depleted. This allows TridentFS to support a wide range of NVRAM size. In addition, by re-using existing file system implementations on the basis of the concept of the stackable file system, various forms of flash memory can be easily employed in TridentFS. Finally, high performance is achieved in TridentFS by keeping hot data in the NVRAM and allowing data evicted from the NVRAM to be parallel distributed to the flash memory and disk.

2.2. Stackable file systems

To enable the coexistence of multiple file systems, modem operating systems adopt an abstraction layer that maps a set of file system operations from user applications to specific functions of the underlying file system implementation. For example, Virtual File System (VFS) [21] and filter driver [22] are used in Unix/Linux and Windows, respectively. Such abstraction layers also allow developers to easily add new functionality to existing file systems such as traffic monitoring [23] or data encryption [24], by stacking a set of file system operations implementing the new functionality on/below existing file systems [25].

On the basis of the concept of stackable file system, a hybrid file system can be realized by stacking multiple existing file systems (for different types of storage) below a file system, as carried out in UmbrellaFS [26]. UmbrellaFS controls these file systems through the VFS interface. Each newly created file is distributed to one of these file systems according to the rules given by the administrator at mount time. A rule is used to specify the file location according to file level information (e.g., file size or file extension). An example can be that files larger than 1 MB are distributed to the file system of a specific disk.

By giving file-size based rules and employing an individual file system for NVRAM, flash memory, and disk, UmbrellaFS can provide similar functionality to TridentFS. Specifically, both of the file systems are able to place small data in the NVRAM and other data in the flash memory and disk. The differences between TridentFS and UmbrellaFS are as follows. First, TridentFS can keep hot data in the NVRAM and balance the storage loads without prior knowledge of the workloads. However, in UmbrellaFS, keeping hot data in the NVRAM and balancing the storage loads both rely on suitable rules, and a comprehensive understanding of the workloads (which may be obtained from offline profiling) is necessary to build such rules. For example, to keep hot data in the NVRAM, file names of all the hot data have to be known (perhaps through offline profiling) and must be stated in the rules. Second, TridentFS allows metadata and its corresponding file data to be stored separately, which helps to improve performance by allowing more hot data to be kept in the NVRAM. For example, under a metadata-dominated workload, file data tend to be evicted from the NVRAM in TridentFS, allowing more metadata to be kept in the NVRAM. In contrast, UmbrellaFS stores metadata and file data of a file in the same storage. Therefore, for the metadata-dominated workload, part of the NVRAM space in UmbrellaFS is still required for storing cold file data (corresponding to frequently accessed metadata).

2.3. File distribution in parallel storage systems

In parallel or distributed systems, distributing files to multiple storage devices in order to minimize mean response time or storage load variation belongs to NP-complete [27]. Multiple heuristic algorithms have been proposed to find a near-optimal solution [27–31]. Generally, these algorithms can be classified into offline and online algorithms. The offline algorithms find a static distribution on the basis of full knowledge of the files such as access rate and file size. On the other hand, the online algorithms dynamically distribute files at the runtime without prior knowledge of the files. Because prior knowledge about the files is not required, online algorithms are much more practical in cases where files are issued dynamically. In addition, according to the unit of distribution, they can be further classified into partitioned [31] and non-partitioned algorithms [27–29]. In the partitioned algorithms, a file is split into several parts, and different parts can be distributed to different storage devices. In contrast, the non-partitioned file algorithms distribute a whole file to a single storage. In TridentFS, we focus on online non-partitioned file distribution.

DOI: 10.1002/spe

Online non-partitioned file distribution algorithms [27, 29, 30] are typically based on the greedy algorithm, which is derived from the *longest processing time* algorithm [30] and aims to balance the storage loads by distributing each file to the storage with the lowest accumulated load. After a file is distributed, the load of the corresponding storage is increased by the load of that file, which is defined as the time for storing the file to the storage (i.e., the service time). Although these algorithms can balance the storage loads, they could distribute small data to the disk, degrading the disk performance. This is because the storage performance characteristics are not considered in these algorithms. A file distribution algorithm is proposed in TridentFS, which is a variant of the greedy algorithm. However, it differs from existing greedy-based algorithms in that it considers the storage performance characteristics. Specifically, large files tend to be placed in the disk according to the proposed algorithm. The experimental results presented in Section 4 show that the proposed algorithm outperforms the greedy algorithm by up to 15.5%.

ZFS, a file system developed by Oracle Corporation, supports distributing files to multiple storage devices in a storage pool [32, 33]. Different types of storage devices are allowed to reside in the same storage pool. Unlike most existing Linux file systems that rely on the Linux page cache for caching their data, ZFS manages its own cache in the memory. The cache is managed by the ARC algorithm [34] and is thus called the L1ARC (first-level ARC). Data evicted from the L1ARC are distributed to the underlying storage devices.

3. DESIGN AND IMPLEMENTATION

3.1. Overview of Trident file system

TridentFS is a hybrid file system aimed at achieving both high performance and high flexibility. Three types of storage, that is, NVRAM, flash memory and magnetic disk, are managed by TridentFS. TridentFS achieves the goal of high performance by using the following two approaches. First, it places hot data in the NVRAM to allow as much I/O accesses as possible to occur in the high-speed NVRAM, reducing I/O accesses to the slower flash memory and disk. Because NVRAM is an order of magnitude faster than the flash memory and disk, replacing slower accesses to flash memory and disk with faster accesses to NVRAM can lead to a large degree of performance improvement. Note that each file in TridentFS is divided into metadata and file data, and the degree of hotness of the metadata and file data are evaluated separately to determine if they can be stored in the NVRAM. For example, in the case of a metadata-access dominated file, the metadata are more likely to be stored in the NVRAM than in the file data. This separation of hot/cold data evaluation helps to store more hot data in the NVRAM, therefore keeping more I/O activities occurring in the high-speed NVRAM.

The second approach is the fast distribution of the evicted metadata/data to the flash memory and disk. Because of the limited size of the NVRAM, metadata or file data may be evicted from the NVRAM and distributed to the flash memory and disk. Fast distribution is achieved by balancing the service time of the flash memory and disk. Moreover, TridentFS distributes the evicted metadata/file data according to the size of the metadata/file data and performance characteristics of the storage. For example, in TridentFS, large data tend to be distributed to the disk because of the superior sequential-access performance of the disk compared with its random-access performance.

High flexibility is another goal of TridentFS. TridentFS achieves high flexibility in two aspects. First, a wider range of NVRAM size is supported in TridentFS as compared with other existing hybrid file systems. Existing hybrid file systems assume large-sized NVRAM, and they force all the metadata and/or all the file data of the small files to be placed in the NVRAM. In contrast, TridentFS can support both large-sized and small-sized NVRAM because it does not have such enforcement. Metadata/file data originally placed in the NVRAM can be evicted from the NVRAM once the NVRAM runs out of its free space.

Second, various forms of flash memory can be supported in TridentFS. For example, TridentFS can use either raw flash memory chips or SSDs as the flash memory storage. On the basis of the concept of stackable file systems, TridentFS is capable of re-using existing file system

DOI: 10.1002/spe

implementations, called *bottomFS* in this paper, for the management of the flash memory and disk. Therefore, when raw flash memory chips are used, flash file systems such as JFFS2 can be used as the *bottomFS*. When SSDs are used, traditional disk-based file systems such as ext4 and NTFS can be used as the *bottomFS*.

In addition to support various forms of flash memory, using *bottomFS* also has two benefits. First, it reduces the implementation cost of TridentFS. Second, it allows TridentFS to exploit the performance optimization techniques of existing file system implementations. For example, the ext4 file system utilizes the delayed allocation technique to reduce the file fragmentation [10]. Using ext4 as a *bottomFS* in TridentFS can hence enjoy the performance improvement resulting from delayed allocation.

Figure 1 shows the TridentFS architecture. On top of the figure, each request issued from the application to the VFS interface triggers one or more VFS operations, which are implemented by TridentFS. Because data (i.e., metadata or file data) can reside in the NVRAM, flash memory or disk, the *dispatcher* has to locate the data first (i.e., the dotted line from the dispatcher to the *NVRAM manager*) and then dispatch the VFS operations according to the data location (i.e., the solid lines).

Metadata and file data in the NVRAM can be accessed directly by the *NVRAM manager* without passing through the page cache layer of the operating system. Hot metadata and file data tend to be placed in the NVRAM. When the NVRAM runs out of its free space, the *eviction manager* is invoked to evict (relatively) cold data from the NVRAM so as to reclaim free NVRAM space (i.e., the dashed lines). The *eviction manager* selects the to-be-evicted data (metadata or file data) according to the data information tracked by the *access monitor*. After the selection of the to-be-evicted data, the *eviction manager* distributes the evicted data to the flash memory and disk by keeping in mind the goal of maximizing the data-distribution throughput. As mentioned above, the evicted data can be distributed to the flash memory and disk in parallel. TridentFS attempts to effectively leverage the parallelism by balancing the service time of the storage devices. Because the service time of the storage devices depends on the runtime storage performance, the *device performance monitor* is used to measure the runtime performance of the flash memory and disk and the results are reported to the *eviction manager* (i.e., the dotted line from the *eviction manager* to the *device performance monitor*).

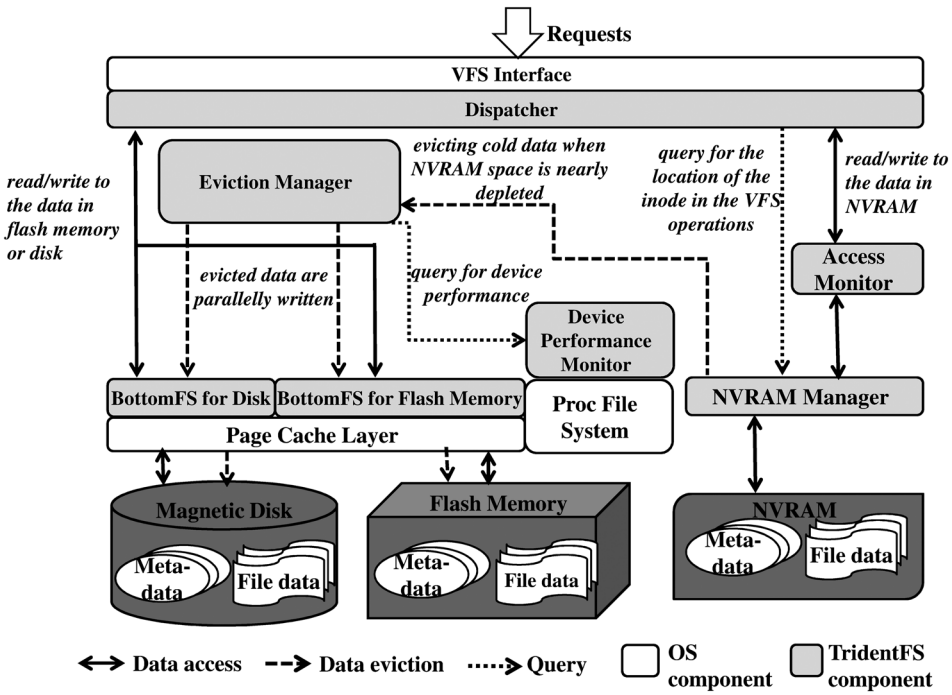

Figure 1. TridentFS architecture.

DOI: 10.1002/spe

Below, we first describe the NVRAM management in Section 3.2, which is followed by the description of the flash memory and disk management in Section 3.3. Data eviction from the NVRAM is described in Section 3.4, and parallel distribution of the evicted data is described in Section 3.5. Finally, the *dispatcher* is described in Section 3.6.

3.2. NVRAM management in TridentFS

Generally, there are two straightforward approaches for managing memory as data storage, ramdisks and memory-based file systems. A ramdisk emulates a memory segment as a disk, allowing existing disk-based file systems such as ext4 to be used on the emulated disk. The main problem with using ramdisk for managing the NVRAM in TridentFS is the addition of an extra layer of caching/buffering, thus reducing system efficiency. Similar to traditional disk accesses, data reads/writes from/to a ramdisk are cached/buffered in the page cache layer of the operating system. Because a ramdisk is itself emulated by memory, this extra caching/buffering leads to unnecessary memory copy operations, thus reducing the system efficiency [35]. Memory-based file systems such as ramfs or tmpfs [35] can avoid such extra caching/buffering. However, these file systems cannot be used to manage NVRAM in TridentFS mainly because they utilize data structures allocated in the volatile memory to manage their data. In order to reduce implementation complexity, these memory-based file systems utilize data structures allocated by the other components in the operating system (e.g., memory manager and VFS). For example, in Linux, the tmpfs relies on the *address_space* structure of the memory manager to locate file data. Because these data structures are allocated in volatile memory, data stored in the NVRAM can no longer be located after system reboot. Fixing the above problem requires a great deal of effort to modify all the related components in the operating system in order to allocate the related kernel data structures in the NVRAM.

Existing NVRAM-based file systems such as MRAMFS or BPFS are not suitable for managing NVRAM in TridentFS either. This is mainly due to the fundamental difference between NVRAM-based file systems and NVRAM management in TridentFS. Specifically, existing NVRAM-based file systems can only refer to data inside the NVRAM. In contrast, in TridentFS, it is required that metadata stored in the NVRAM should be able to refer to metadata or file data distributed to the disk and flash memory.

As a consequence, a new approach is proposed to manage NVRAM in TridentFS. Unlike the ramdisk approach, the proposed NVRAM management approach bypasses the page cache layer in order to prevent the extra memory copy operations. Moreover, it allocates the related data structures in the NVRAM, ensuring that data stored in the NVRAM can be located after system reboot. Finally, the in-NVRAM data structures can refer to metadata or file data distributed to the disk and flash memory.

Figure 2 shows the data structures stored in the NVRAM. The NVRAM is divided into fixed-size blocks (currently, 1 KB) to simplify the management task. From the figure, the superblock includes the information about the NVRAM and *bottomFS*s, such as the block size of the NVRAM, the number of used inodes, and device names of the disk and flash memory. TridentFS adopts simplified and fixed-size inode (i.e., index nodes) structures to index data. Each file (including directory) in TridentFS has a corresponding inode, which consists of the file type (i.e., file or directory), file size, data flag and an array of pointers for locating the file data or directory entries. The 2-bit data flag is used to indicate the location of the corresponding file data. Specifically, the flag is set as 1, 2 and 3 if the corresponding file data are stored in NVRAM, flash memory and disk, respectively. For example, for files with inodes 256 and 257 in Figure 2, the file data are stored in the NVRAM; for file with inode 258, the file data are stored in the disk. The data flag is set as 0 if the file does not have any data (i.e., file size is 0). For each regular file with its data in the NVRAM, the inode contains a number of pointers to *indirect blocks*. Each indirect block in turn consists of 256 pointers, each of which can point to a 1-kB data block. For each directory with its data (i.e., directory entries) in the NVRAM, the inode points to a hash table to efficiently locate a specific directory entry. The file name is used as the hash key, and directory entries hashing to the same hash table bucket (i.e., hash collisions) are chained in the list corresponding to that bucket.

As shown in Figure 2, an inode bitmap (with each bit in the bitmap corresponding to an inode) is used to indicate whether an inode is used in the TridentFS. Because an inode could be stored in

DOI: 10.1002/spe

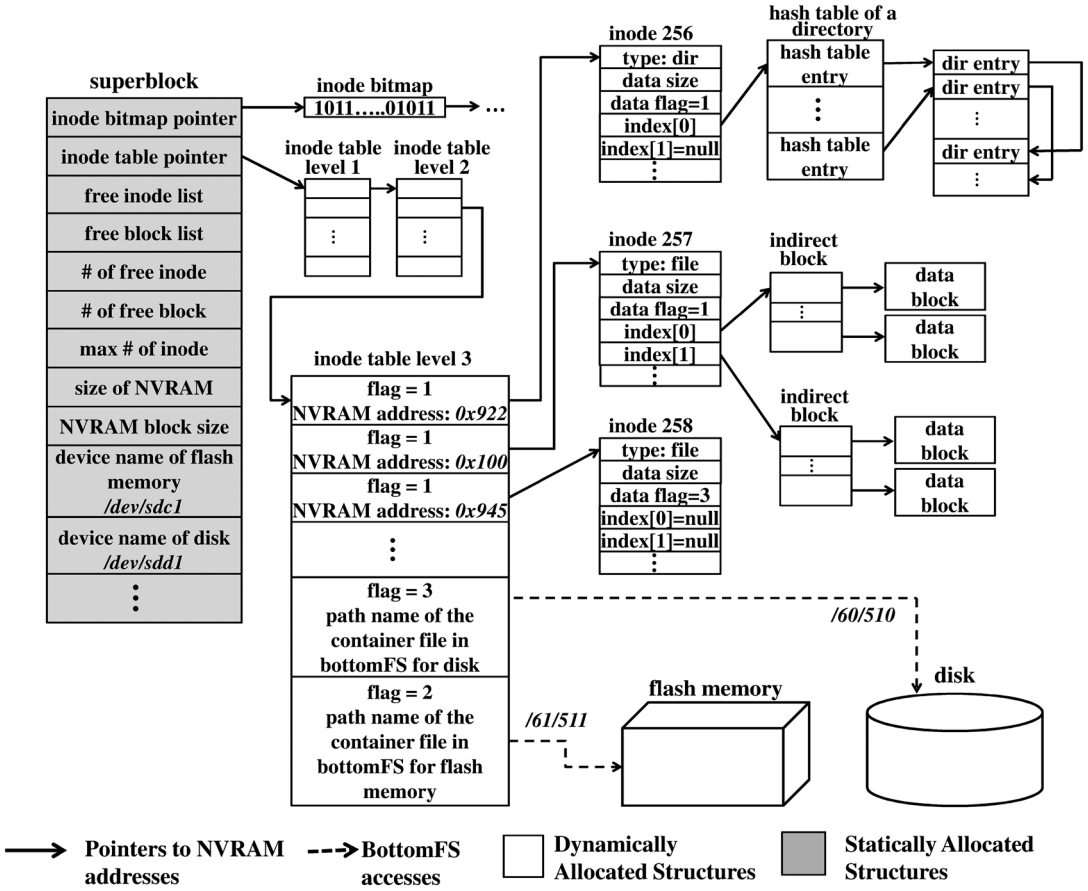

Figure 2. In-NVRAM data structures in TridentFS.

NVRAM, flash memory or disk, an in-NVRAM inode table is adopted to locate the inode. Specifically, given the inode number for a file F, the inode of F can be located by the following steps. First, the corresponding bit in the inode bitmap is checked to see whether the inode is allocated. For an allocated inode (i.e., the corresponding bit in the inode bitmap is set as 1), the inode table is then consulted to locate the inode. A three-level inode table is utilized, akin to the three-level page table used in virtual memory management. To locate an inode, the inode number is divided into three components. The first component is used to index the first-level inode table to locate the corresponding second-level inode table; the second component is used to index the second-level inode table to locate the corresponding third-level inode table, and the third component is used to index the third-level inode table to locate the inode.

Because an inode in TridentFS could be evicted to flash memory or disk, each entry in the third-level inode table includes a flag to indicate the storage where the inode is currently stored, similar to the data flag field of an inode. Specifically, the flag is set as 1 if the inode is stored in the NVRAM. In this case, the corresponding entry in the third level inode table stores the memory address of that inode. As shown in Figure 2, the starting NVRAM address of inode 257 is *0x100*. The flag is set as 2 and 3 if the inode is currently in the flash memory and disk, respectively. In these cases, the inode can be obtained by consulting the corresponding *bottomFS*, as described in Sections 3.3 and 3.6. Note that a flag as 0 indicates an inconsistent state in the file system, and error is reported to the user in this case.

In TridentFS, all of the free NVRAM blocks are chained in a *free block list*, which is used to satisfy block allocation requests such as file append or inode table allocation. A block is returned to the *free block list* when it becomes free. Two techniques are adopted to improve NVRAM space efficiency in TridentFS. First, most of the data structures in the NVRAM are allocated on demand. In TridentFS, only the superblock is allocated (statically) in the NVRAM when building the

DOI: 10.1002/spe

TridentFS. The other structures are allocated on demand (dynamically) at runtime. This helps to store more hot data in the NVRAM, resulting in performance improvement. Taking the three-level inode table as an example, upon the creation of a new inode, the corresponding blocks of the three-level inode table are allocated as needed. These blocks are reclaimed when they become empty (because of inode deletion). Such dynamic allocation in the three-level inode table could save significant space. For example, assuming that each inode table entry requires 34 bits (i.e., 32 bits for NVRAM address and 2 bits to indicate in which storage the inode resides) and the maximum number of the inode is 1 million, the space reserved for the inode table is about 4.1 MB if static allocation is used. When the dynamic allocation technique is used, only three blocks (i.e., a block for each level of the inode table) are required for the inode table to refer to the inode of the root directory after the file system initialization, which consumes 3 KB. Second, a slab allocator is implemented for allocating inodes. Because an NVRAM block space is sufficient for accommodating multiple inodes, allocating an individual block for each inode causes poor space utilization. Therefore, a block is divided into multiple inodes to satisfy multiple inode allocation requests.

In the current implementation, the NVRAM resides in an individual physical memory partition. During the operating system boot process, a range of virtual addresses is set up to map to the NVRAM memory partition to allow data structures in the NVRAM to be accessed by virtual addresses. The starting address of the virtual address range is fixed in the current implementation, and the superblock is allocated at the beginning of the NVRAM partition.

3.3. Flash memory and disk management in TridentFS

As mentioned before, TridentFS employs *bottom file systems* (*bottomFS*s) to manage the flash memory and disk. If the file data or metadata of a TridentFS file *F* are evicted from the NVRAM, a file in the *bottomFS* (called a *container file*) is used to accommodate the file data/metadata of *F*. Each container file is named after the TridentFS inode number for unique identification, which eliminates the effort required to maintain an extra mapping table between TridentFS files and the corresponding container files. In addition, container files are spread over multiple directories in a *bottomFS* to avoid the situation in which a large number of files reside in a single *bottomFS* directory. This is carried out for the following reasons. First, some file systems, for example, ext2, use sequential searches to look up directory entries in a directory. Placing all container files in a single directory in these *bottomFS*s could result in large latency in the directory entry lookup. Second, many file systems have a limitation on the number of files in a directory. For example, the maximum numbers of files per directory are 64K and 32K in FAT32 and ext3, respectively. Spreading container files over multiple directories in these file systems helps to avoid such limitation. For each container file, the corresponding target directory in a *bottomFS* is determined by a hash function, with the file name (i.e., TridentFS inode number) as the key. For example, when a TridentFS file with inode number *N* is evicted, the pathname of the container file would be */100/N* if the hash result of *N* is 100. In the current implementation, container files are spread over 150 directories.

The format of a container file starts with the metadata, followed by the file data. For a container file accommodating a regular file in TridentFS, the metadata is the inode, as shown in Figure 3(a). For a container file accommodating a directory in TridentFS, the metadata includes the inode as well as the directory hash table, as shown in Figure 3(b). Note that the metadata can be empty for a container file. Because the metadata and file data of a TridentFS file can be evicted separately, for a metadata-access dominated file, it is likely that only the file data are evicted from the NVRAM because of the fact that the file data are colder than the metadata. In this situation, a fixed-size space is still reserved for the metadata (as a file hole) in the container file, as shown in Figure 3(c), (d). However, the file data cannot be empty for a container file because TridentFS does not allow evicting the metadata while keeping the corresponding file data in the NVRAM, as mentioned later in Section 3.4.

Note that managing flash memory and disk with *bottomFS*s could potentially result in additional I/O activities. This is because when a container file is accessed, the corresponding *bottomFS* inode also needs to be accessed. Nevertheless, such additional I/O accesses usually do not cause significant performance impact because of the following reasons. First, as mentioned in Section 3.1,

DOI: 10.1002/spe

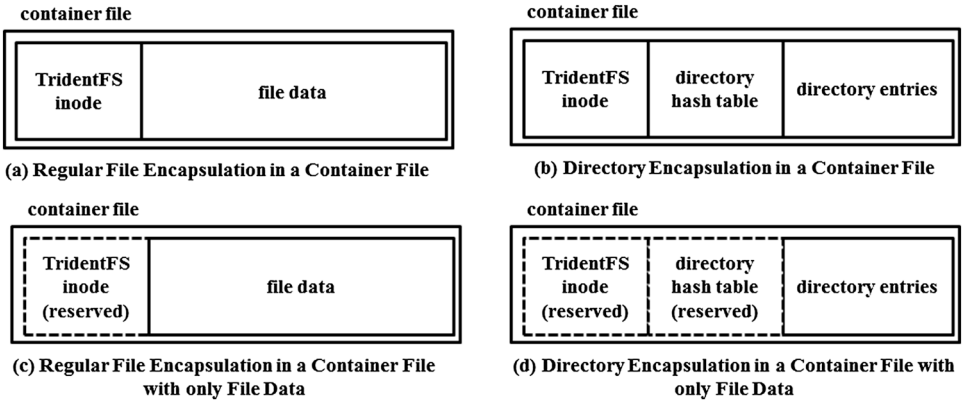

Figure 3. Format of container files in TridentFS.

TridentFS places hot data in the NVRAM to allow as much data accesses as possible to occur in the NVRAM. This reduces accesses to container files and the corresponding *bottomFS* inodes. Second, in practice, accesses to *bottomFS* inodes can usually be served by the memory cache of the operating system (e.g., the inode cache in Linux) because of the large memory capacity of modern computing systems. As mentioned in Section 3.1, using *bottomFS*s improves the flexibility and reduces the implementation cost of TridentFS. Moreover, it allows TridentFS to exploit performance optimization techniques of existing file system implementations. Owing to these benefits, *bottomFS*s are used in TridentFS. Techniques will be developed in the future to further reduce the performance impact of additional I/O accesses resulting from using *bottomFS*s.

3.4. Data eviction

As mentioned before, unlike existing hybrid file systems, which assume large-sized NVRAM and force all the metadata and/or all the file data of the small files to be placed in the NVRAM, TridentFS allows metadata and file data to be evicted from the NVRAM, supporting large-sized as well as small-sized NVRAM. When the NVRAM runs out of its free space, the *eviction manager* is triggered to evict metadata or file data from the NVRAM to the flash memory and disk. Specifically, when the NVRAM space utilization exceeds a pre-defined threshold T_H (i.e., the high watermark), the *eviction manager* starts to perform eviction until the NVRAM space utilization drops below another threshold T_L (i.e., the low watermark). In the current implementation, T_H and T_L are set as 100% and 95%, respectively.

Three factors are considered when selecting data (i.e., metadata or file data) for eviction, that is, recency, frequency and size. TridentFS attempts to keep recently used, frequently used and small data in the NVRAM. As indicated in previous studies [36, 37], frequency and recency are both important factors that indicate the degree of hotness of data. In addition, size is also considered mainly due to the fact that evicting data with large sizes helps to reclaim much NVRAM space, which can be used to accommodate many small-sized hot data.

For each data k (i.e., metadata or file data) in the NVRAM, a score Ω_k is used to evaluate the degree of its hotness. The metadata or file data with the smallest score tends to be evicted. The score Ω_k is calculated whenever the data k is updated, and it can be expressed by

$$\Omega_k = C_k + \omega * \frac{AC_k}{S_k} \tag{1}$$

In (1), the AC_k indicates the access count of the data k, which is increased by 1 each time data k is accessed. For example, assuming that data m and n represent the metadata and file data of a file, respectively, both AC_m and AC_n are increased by 1 when a write request to the file is issued. The AC_m is also increased because the write request causes an update (on the file size, last modification time, etc.) to the metadata. The S_k indicates the size of the data (in bytes). The C_k represents the recency of data k. TridentFS maintains a global and strictly-increasing variable *Clock* to represent

DOI: 10.1002/spe

Ck可能会变得比较大吧，然后导致占主导因素了
好像没问题，毕竟只看差值。

recency, which is increased by 1 each time metadata or file data is evicted. The C_k is assigned to the value of the *Clock* whenever data k is accessed. The ω is a weight parameter between C_k and (AC_k/S_k).

The C_k, AC_k and S_k values for each data are tracked by the *access monitor*. The scores Ω_k for all the metadata and file data are maintained in a min-heap in the memory. When eviction starts, the *eviction manager* repeatedly evicts the data corresponding to the root of the min-heap (i.e., the node with the minimum value) until the NVRAM space utilization drops below T_L.

那这并发读有点低了

For a file, the Ω value of the metadata is typically equal to or higher than that of the file data. This is because metadata needs to be accessed for each file data access request. Therefore, the file data of a file tend to be evicted from the NVRAM before the eviction of the metadata. Although it is still possible that the Ω value of the metadata is smaller than that of the file data, TridentFS does not allow evicting the metadata while keeping the corresponding file data in the NVRAM. This is because, for a file whose metadata is evicted while the file data is kept in the NVRAM, flash memory or disk accesses are still required to obtain the metadata before accessing file data in the NVRAM.

Because data eviction moves metadata or file data from the NVRAM to the container files, the related fields in the inode table, metadata or file data have to be modified accordingly. When an inode is evicted, the flag in the third-level inode table entry corresponding to the inode is modified to indicate whether the inode is stored in either flash memory or disk. Figure 4 shows the modifications of metadata or file data when the file data are evicted. As shown in Figure 4(a), when the file data of a regular file are evicted, the indirect blocks are all freed, and the indirect block pointers in the inode are all invalidated because the indirect blocks are only used to locate the file data in the NVRAM. As shown in Figure 4(b), when the file data of a directory (i.e., directory entries) is evicted, the hash table entries need to be changed to point to the file offsets of the corresponding directory entries in the container file. In Figure 4(a), (b), the data flag in the inode should also be modified to indicate the location of the file data (e.g., set as 2 for flash memory and set as 3 for disk). The file data should be modified when the file data themselves of a directory are evicted. As shown in Figure 4(b), the pointers in the directory entries are modified to point to the file offsets of these entries in the container file.

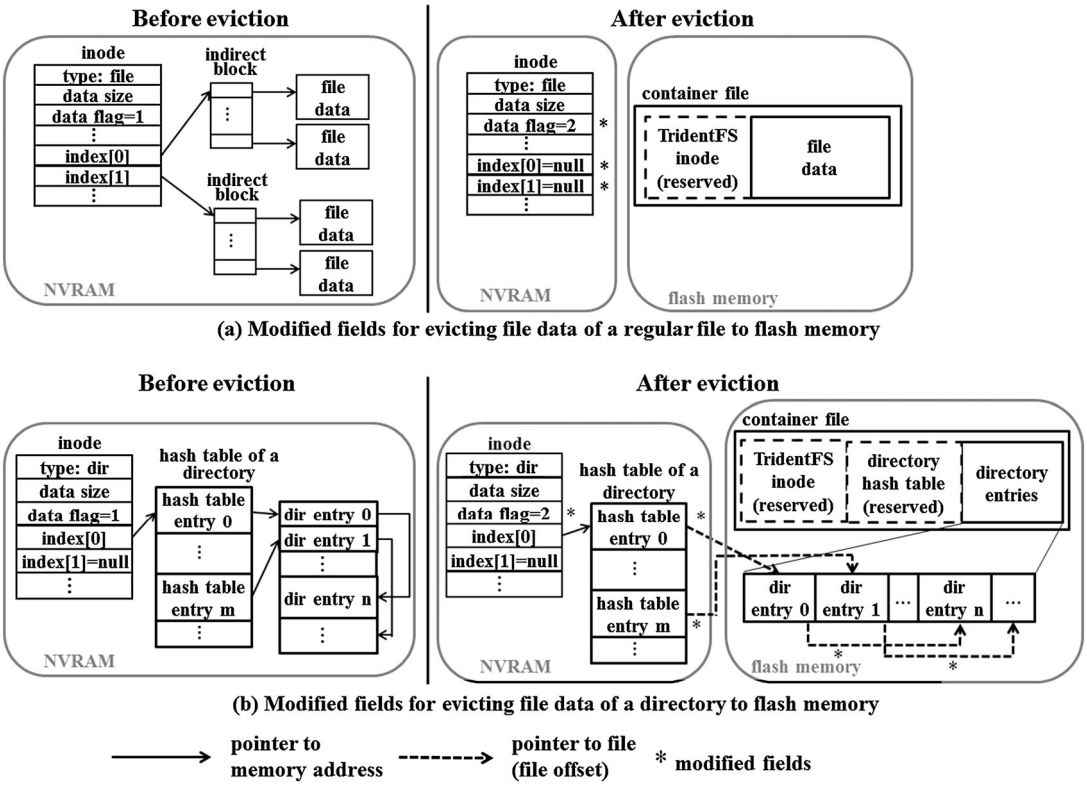

Figure 4. Modified fields for file data eviction.

DOI: 10.1002/spe

It should be noted that not all the data in the NVRAM can be evicted in TridentFS. Specifically, the superblock, inode bitmap and inode table cannot be evicted. The superblock is retained in the NVRAM mainly because of consistency issues. Incomplete update to the superblock could make TridentFS unable to be mounted, and retaining the superblock in the NVRAM helps to reduce the probability of incomplete update to the superblock. The inode bitmap and inode table are retained in the NVRAM for performance consideration. Because almost all the file operations in TridentFS require locating inodes, retaining inode bitmap and inode table in the high-speed NVRAM allows TridentFS to locate inodes efficiently.

3.5. *Parallel distribution of evicted data* 被驱逐数据的并行分布

As mentioned above, evicted file data and metadata are distributed to the flash memory and disk in parallel by the *eviction manager*. Because eviction is adopted to reclaim NVRAM space for further file operations such as file creation and append, fast distribution of the evicted data helps to increase the speed of accepting further file operations. In TridentFS, fast distribution is achieved by two techniques. First, parallelism between the flash memory and disk is leveraged effectively by distributing amount of data to the flash memory and disk according to the runtime storage performance. Distributing an improper amount of data to the flash memory and disk causes service time imbalance between the storage devices, leading to degraded performance. Taking Figure 5 as an example, Figure 5(a) includes 10 to-be-distributed data (total size 185 KB). The eviction order of these data and their sizes are shown in the figure. Assume that the speed of flash memory and disk are 2 MB/s

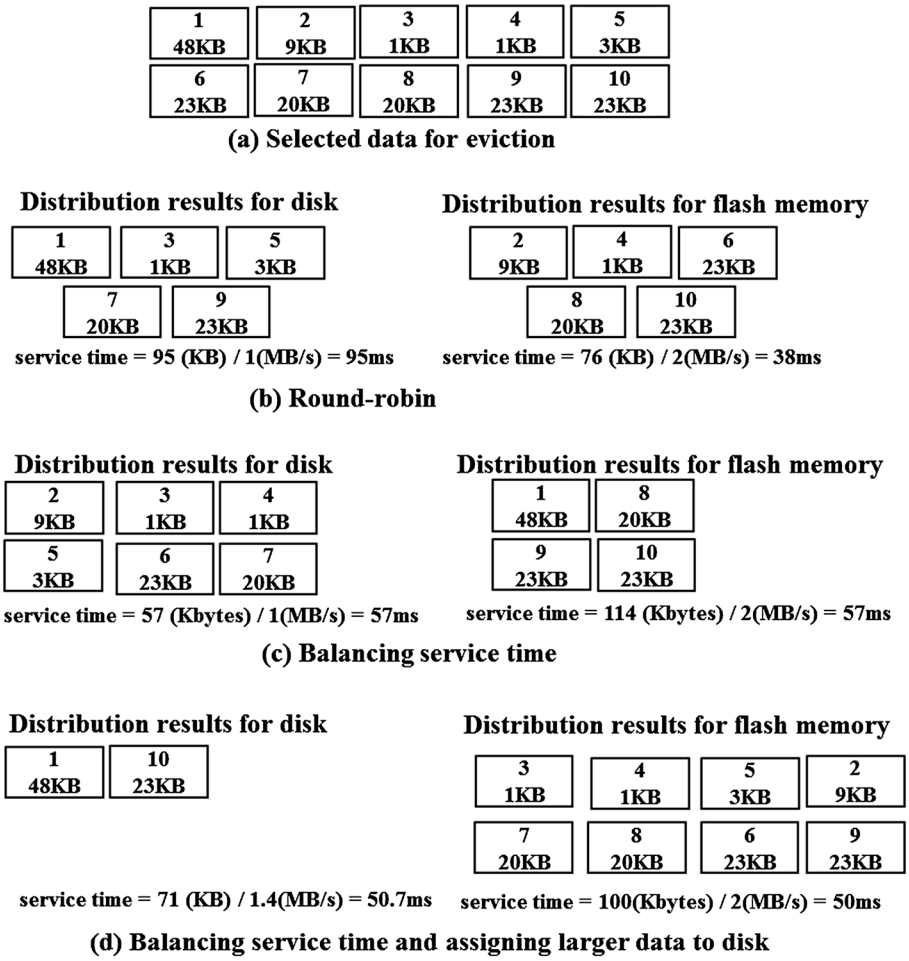

Figure 5. Data distribution results under different distribution methods.

DOI: 10.1002/spe

and 1 MB/s, respectively. In Figure 5(b), distributing these 10 data using a round-robin method leads to imbalance in the service time of the disk and flash memory. The service time of the two storage devices can be balanced by distributing the amount of data to the storage devices according to the storage performance. This can be achieved by the greedy algorithm, which distributes the data to the storage with the lowest accumulated service time, as shown in the Figure 5(c). As seen in the figure, the degree of service time imbalance is reduced.

The second technique to achieve fast distribution is to consider both the size of the evicted data and the storage characteristics. As mentioned in the introduction, the disk performance degrades for small-data accesses because of time-consuming seeks in disks. Therefore, distributing small-size data (e.g., metadata or small file data) to disk would slow down the distribution. As shown in Figure 5(c), many small data are distributed to the disk, decreasing the distribution speed. As a result, large data tend to be distributed to the disk in TridentFS.

Before describing the distribution algorithm in TridentFS, we first define the storage *load* as the accumulated service time of that storage. Whenever data in the storage are accessed or data are distributed to storage, the *load* of the storage is increased by the service time of that data. The service time of the data is calculated by dividing the data size by the runtime performance of the storage, which is reported by a thread (i.e., the *device performance monitor*) that periodically monitors the current performance of the flash memory and disk. The flash memory *load* ($load_{\text{flash}}$) and the disk *load* ($load_{\text{disk}}$) are set as 0 at the mount time.

The distribution algorithm sorts all the to-be-evicted file data and metadata according to their sizes before data distribution. To effectively leverage the parallelism, the distribution algorithm greedily makes the $load_{\text{flash}}$ and $load_{\text{disk}}$ as close as possible. When evicting each file data or metadata, the storage *loads* are compared. If the storage with the lowest *load* is disk, the largest file data or metadata with the distribution destination not yet determined are determined to be distributed to the disk. On the contrary, if the storage with the lowest *load* is flash memory, the smallest file data or metadata with the distribution destination not yet determined are determined to be distributed to flash memory.

Figure 5(d) shows the results of the distribution algorithm used in the TridentFS. In the figure, it is assumed that large disk writes achieve a throughput 40% higher than small disk writes. From the figure, evicting larger data to disk reduces the service time by 12.3% and 11.0%, respectively, for flash memory and disk, when compared with Figure 5(c). The reduction in the service time comes from the improvement in the disk speed (resulting from large writes). The improvement in the disk speed allows a larger amount of data to be distributed to the disk, and thus a smaller amount of data are distributed to the flash memory, leading to service time reduction for the flash memory. In addition, parallelism between the flash memory and disk is still leveraged effectively, as shown in Figure 5(d). Note that fixed storage performance is assumed in Figure 5(c), (d). However, storage performance varies dynamically with the workloads. Therefore, runtime performance of the storage is considered in the distribution algorithm of TridentFS.

Figure 6 shows the pseudo code of the distribution algorithm. The list of file data and metadata selected for eviction, denoted as E, is first sorted in ascending order according to the sizes of the file data and metadata (line 1). The storage with the lowest load is selected as the target storage to store the evicted data in E (lines 4 and 9). If the flash memory is selected as the target storage, the data with the minimum size (i.e., f_{h}, the head of list E) are assigned to the D_{flash}, indicating that the data are distributed to the flash memory (line 7). Otherwise, the data with the maximum size (i.e., f_{t}, the tail of the list E) are assigned to D_{disk} (line 12). After this, the service time of the distributed data are calculated by dividing the data size by the current performance of the storage. The service time is then added to the storage *load* (lines 8 and 13). Finally, when the distribution of all the files finishes, the D_{result} including the D_{flash} and D_{disk} is returned (line 16). Note that when a read or write request from the application is issued to the flash memory or disk, the *load* of the storage is also increased by the service time of that request.

In Figure 6, the current performance of the storage, that is, the value of $P(flash)$ or $P(disk)$, is measured periodically (currently, every 1 s) by the *device performance monitor* thread. In the Linux kernel, this can be achieved by periodically obtaining the I/O statistic information provided by the proc file system of the kernel (i.e., /proc/diskstats) [38] and then calculating the storage performance according to the following equation:

DOI: 10.1002/spe

```
Parallel_Distribution()
Input: eviction list E          // E is a list of n file data or metadata selected for eviction
Output: D_result                // D_result is the distribution results
1   sort the sizes of elements in E in ascending order so that E={f1,f2, ..., fn | S(f1)≤ S(f2)
    ≤ …≤ S(fn) }                //S(fi) = size of fi
2   set D_result, D_flash and D_disk as empty   // D_flash and D_disk are the distribution results for flash
                                                // memory and disk, respectively
3   while E is not empty
4       if load_flash < load_disk
5           f_h = the head of E
6           E = E – { f_h }
7           D_flash = D_flash∪{ f_h }
8           load_flash += S(f_h) / P(flash)   // P(flash) = current performance of flash memory
9       else
10          f_t = the tail of E
11          E = E – { f_t }
12          D_disk = D_disk∪{ f_t }
13          load_disk += S(f_t) / P(disk)     // P(disk) = current performance of disk
14      end if
15  end while
16  return D_result = { D_flash , D_disk }
```

Figure 6. Algorithm of distributing evicted data.

$$P(d_i) = \frac{\Delta S}{\Delta T}, \qquad (2)$$

即吞吐

where ΔS indicates the number of sectors accessed (including read and written) during the previous second and ΔT indicates the time spent for accessing these sectors during the previous second.

3.6. Request dispatcher

Once the VFS layer receives a request from the application, it invokes VFS operations implemented by TridentFS to perform data access. For example, an open() request issued from the application requires pathname lookup, which results in the invocation of a series of inode read and file data read operations to retrieve the inode and the file data of each directory in the pathname given in the request. Because data (i.e., metadata or file data) can reside in the NVRAM, flash memory or disk, the *dispatcher* has to locate the data and then to access it according to the data location.

To locate an inode, the *dispatcher* queries the *NVRAM manager* (with the inode number as the parameter). The location of the inode can be obtained by referencing the flag of the corresponding inode table entry in the NVRAM, as mentioned in Section 3.2. File data can be located by referencing the data flag of the corresponding inode.

Data access can be performed after the data (i.e., inode or file data) has been located. If the data is in the NVRAM, the data is accessed through the *NVRAM manager*. Otherwise, VFS command(s) are generated to the *bottomFS* to access the data. For example, to read a TridentFS inode in the flash memory, the VFS *read*() command is generated and issued to the container file accommodating the inode. As illustrated in Figure 3(a), (b), because the starting offset of an inode in a container file is 0, the read command would be *read*(*cf*, 0, *M*), where *cf* is the file handle corresponding to the

DOI: 10.1002/spe

container file, which is returned from the *open()* command and *M* is the size of a TridentFS inode.[†] Similarly, to write a TridentFS file in the disk, the VFS *write*() command is generated and issued to the container file accommodating the file data. If *L* bytes starting from offset *S* need to be written, the write command would be *write*(*cf*, *M*+*S*, *L*). Note, the VFS *open*() command is generated and issued to the container file if necessary before the issue of the *read*()/*write*() command.

Note that although issuing VFS commands increases the number of function invocations, these function invocations are lightweight. As mentioned above, TridentFS achieves high performance by (1) reducing I/O accesses to the flash memory and disk and (2) effectively leveraging parallelism between the flash memory and disk. Compared with these I/O improvements, the overhead caused by extra function invocations is negligible. Previous studies also show that such kind of function invocations does not cause noticeable overhead [26, 39].

4. PERFORMANCE EVALUATION

In this section, the performance of TridentFS is evaluated. Section 4.1 describes the experimental environment and the benchmarks used for performance evaluation. Section 4.2 presents the performance of various *bottomFS*s in TridentFS. The overall performance results of TridentFS are given in Section 4.3. Sections 4.4 and 4.5 show the performance results of TridentFS under various NVRAM sizes and data eviction policies, respectively. The performance impact of high/low watermarks for data eviction is shown in Section 4.6. Sections 4.7 and 4.8 evaluate different storage performance measurement schemes and distribution algorithms, respectively, in TridentFS. Finally, the code size of TridentFS, the memory overhead and the overhead of dynamic storage performance monitoring are presented in Section 4.9.

4.1. Experimental environment

TridentFS is implemented as a loadable module in Linux 2.6.29. The machine configuration and benchmarks for performance evaluation are shown in Table I. The performance of TridentFS was evaluated on a PC equipped with a 1.86-GHz processor and 2-GB memory. Part of the memory is emulated as NVRAM. For example, for a configuration with 384 MB NVRAM, 384 MB of DRAM is emulated as NVRAM for persistent storage, and only 1664 MB of DRAM is available for the operating system as the main memory. Except from the experiment in Section 4.4, which evaluates the performance of TridentFS under various NVRAM sizes, 384 MB of NVRAM is used in all the other experiments. Note that the current implementation is suitable for the NVRAM with similar characteristics to the DRAM such as MRAM [3, 40] and battery-backed DRAM. Extension of the current TridentFS implementation to NVRAM with different characteristics to DRAM (e.g., PCRAM) is left for future work. Two off-the-shelf flash memory-based SSDs, Intel X25-M and Intel

Table I. Experimental environment.

Hardware	CPU	Intel E6300 dual-core processor 1.86 GHz
	Memory and NVRAM	DDRII 2 GB (4 MB to 384 MB is treated as NVRAM when evaluating TridentFS; 384 MB of NVRAM by default)
	Flash memory	Intel SSD X25-V, 40 GB, max. read/write speed: 170/35 MB/s Intel SSD X25-M, 80 GB, max. read/write speed: 250/70 MB/s
	Disk	Seagate ST3750528AS, 750 GB, 7200 rpm, position time (avg.): 4.2 ms, max. transfer speed: 125 MB/s
	RAID adapter	Adaptec 2405
Operating system	Linux 2.6.29	
Benchmarks	*create-files*, *fileserver*, *webproxy*, *postmark*, *untar*, *kernel-compile*	

[†]To simplify the description, we assume that the prototype of the read/write command is read/write(file, starting_offset, rw_lengh). The parameters in order are the file identifier, the starting offset and the number of bytes needed to be read/written.

DOI: 10.1002/spe

X25-V, are used as flash memory-based storage in the experiments. The Intel X25-M and Intel X25-V SSDs are referred to as *high-speed* SSD and *low-speed* SSD below, respectively.

Six benchmarks were used for performance evaluation: *create-files*, *fileserver*, *webproxy*, *postmark*, *untar* and *kernel-compile*. The *create-files*, *fileserver* and *webproxy* benchmarks are included in the *filebench* file system benchmark suite [41]. The *create-files* benchmark creates files repeatedly, and the file size is based on the gamma-distributed size from 73 KB to 3.6 MB. The *fileserver* benchmark simulates the workload of a file server, which performs a sequence of creation, deletion, append, read and write operations. The initial file set contains files with a mean size of 128 KB. The sizes of each read/write operation and each append operation are 1 MB and 16 KB, respectively. The *webproxy* benchmark simulates the workload of a web proxy, which repeatedly performs a set of file operations (including a file creation, a file deletion and five file read operations). The mean size of the created files is 16 KB, and the initial file set contains 60,000 files with a mean size of 16 KB. The run time of *create-files*, *fileserver* and *webproxy* are all set to 120 s. The *postmark* [42] benchmark simulates the workload of a news or email server. During execution, it creates an initial set of small files and then performs a number of transactions on those files. Each transaction consists of a create/delete operation together with a read/append operation. The initial file set contains 70 K files residing in 150 directories, the file size ranges from 0.5 to 9.8 KB (i.e., the default setting of postmark), and 200 K transactions are performed. The *untar* benchmark extracts files from a bzip2-compressed image. The size of the compressed image is 601 MB, and the number of the decompressed files is 12,680, ranging from 30 KB to 1 MB, in 6608 directories. The total decompressed size is 1.56 GB. The *kernel-compile* benchmark is a CPU-intensive workload, which builds a compressed kernel image from the source tree of the Linux kernel (version 2.6.29). The total size of the object files and the compiled kernel image is about 25 MB.

append 16KB大小，大页的管理应该有小提升

可以对小文件进行特殊的优

4.2. Evaluation of bottom file systems

As mentioned before, *bottomFS*s are employed in TridentFS to manage flash memory and disk. In this section, three file systems (i.e., FAT, ext4 and nilfs) are used as *bottomFS*s, and their performance is evaluated. These file systems represent three different types of file system implementations. Specifically, FAT represents a simple table-based file system, ext4 represents a journaling file system and nilfs2 represents a log-structured file system. Figure 7 shows the performance results of these file systems as the *bottomFS*s. In the figure, the bar corresponding to the *fs1/fs2* configuration shows the throughput of TridentFS when *fs1* is used as the *bottomFS* for the SSD and *fs2* is used as the *bottomFS* for the disk. According to the figure, using ext4 as the

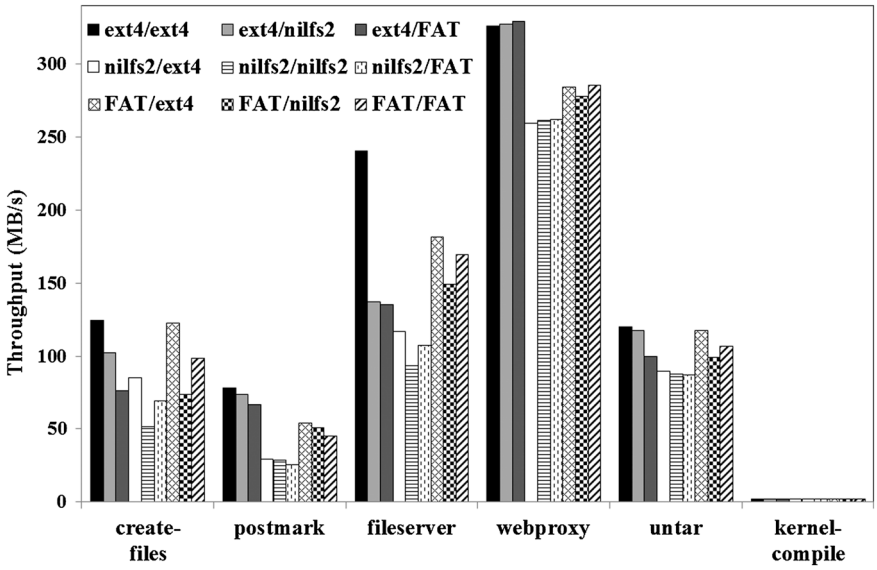

Figure 7. Performance of TridentFS using different *bottomFS*s.

DOI: 10.1002/spe

*bottomFS*s for both the SSD and the disk results in better performance than the other configurations under most of the workloads. Thus, this configuration is used in the following experiments. Note that this figure shows the results under the high-speed SSD. The results under the low-speed SSD are omitted because the ext4/ext4 configuration also outperforms the other configurations under the low-speed SSD.

4.3. Performance results of TridentFS

In this section, the performance results of TridentFS are presented. In addition, to achieve further understanding of the performance improvement in TridentFS, the performance benefit of adopting NVRAM and parallel eviction are included. Eight configurations are used for the performance evaluation: ext4 (ssd), ext4 (disk), ext4 (RAID-0), TridentFS-single (ssd), TridentFS-single (disk), TridentFS-single (RAID-0), TridentFS-size (ssd, disk) and TridentFS (ssd, disk). The first three configurations, that is, the ext4 (ssd), ext4 (disk) and ext4 (RAID-0), do not utilize NVRAM. Therefore, in these configurations, the operating systems can manage 2 GB of DRAM. The configurations ext4 (ssd) and ext4 (disk) represent the single-storage environment that uses ext4 as the file system. The configuration ext4 (RAID-0) represents the environment that uses ext4 as the file system on a RAID-0 storage, which is constructed using the SSD, the disk and the RAID adapter. Data written to the RAID-0 storage are evenly distributed to the SSD and disk with the default stripe size specified by the RAID adapter (currently, 256 kB).

The other five configurations utilize NVRAM. In configurations TridentFS-single (ssd) and TridentFS-single (disk), a variant of TridentFS called TridentFS-single is used as the file system in an environment consisting of NVRAM and a single underlying storage (SSD or disk). In TridentFS-single, data evicted from the NVRAM are distributed to the single underlying storage (SSD or disk). In configuration TridentFS-single (RAID-0), the TridentFS-single is used as the file system on RAID-0. Data evicted from the NVRAM are distributed to the RAID-0 containing the SSD and disk. In configuration TridentFS-size (ssd, disk), another variant of TridentFS called TridentFS-size is used as the file system in an environment consisting of NVRAM, SSD and disk. TridentFS-size is the same as TridentFS except that the former performs data distribution purely on the basis of the size of the evicted data. Specifically, in TridentFS-size, evicted data smaller than a specific size threshold (currently, 1 MB) are distributed to the SSD while the other data are distributed to the disk. In configuration TridentFS (ssd, disk), TridentFS is used in an environment consisting of NVRAM, SSD and disk.

Figures 8 and 9 show the performance of all the configurations with the high-speed and low-speed SSDs, respectively. The benefit of using the NVRAM (and the *NVRAM manager*) can be demonstrated by comparing the performance of ext4 with that of TridentFS-single under the same underlying storage. For example, in Figure 8, TridentFS-single (ssd) outperforms ext4 (ssd) by up to 111.0%, and TridentFS-single (RAID-0) outperforms ext4 (RAID-0) by up to 115.3%. Note that as mentioned in Section 4.1, the same amount of memory is used for both the TridentFS-single and ext4 configurations. The former outperforms the latter because it reduces the amount of I/O to the flash memory and disk. In an operating system, dirty data in the volatile memory need to be flushed to storage when the free memory drops below a specific threshold (i.e., memory pressure) [43]. Compared with the ext4 configurations, the TridentFS-single configurations cause a reduced amount of flushed data because data in the NVRAM do not need to be flushed.

In addition, under write dominated workloads such as *postmark*, *file server* and *untar*, TridentFS (ssd, disk) outperforms TridentFS-single (RAID-0) and TridentFS-size (ssd, disk) by up to 471.6% and 43.6%, respectively, when the high-speed SSD is used. When using the low-speed SSD, TridentFS (ssd, disk) outperforms TridentFS-single (RAID-0) and TridentFS-size (ssd, disk) by up to 310.1% and 82.6%, respectively. Such performance improvements are contributed by effectively leveraging the parallelism between the flash memory and disk achieved by TridentFS. In configuration TridentFS-single (RAID-0), the SSD load and disk load become imbalanced because the same amount of data is distributed to the storage devices (i.e., SSD and disk) that have different performance. Such load imbalance prevents the parallelism between the flash memory and disk from being leveraged effectively. In configuration TridentFS-size (ssd, disk), load imbalance is caused

DOI: 10.1002/spe

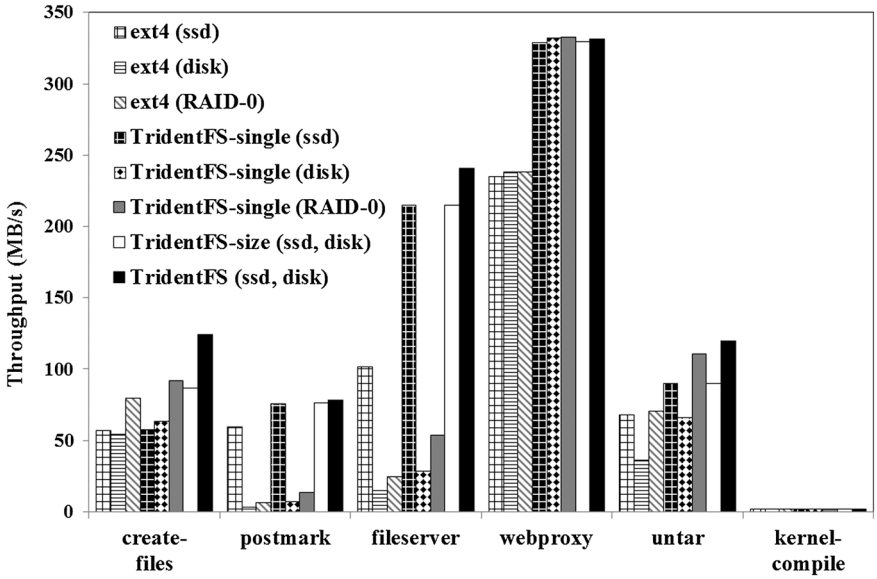

Figure 8. Performance of TridentFS with high-speed SSD.

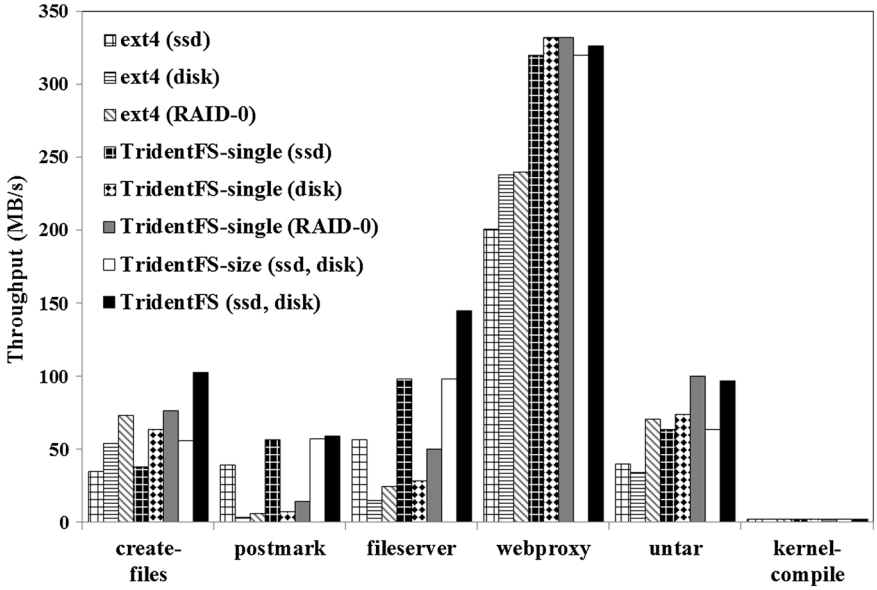

Figure 9. Performance of TridentFS with low-speed SSD.

by the distribution of improper amounts of data to the SSD and disk. Table II shows the percentages of the evicted data distributed to SSD and disk under TridentFS-single (RAID-0), TridentFS-size (ssd, disk) and TridentFS (ssd, disk), respectively. From the table, it can be seen that TridentFS-single (RAID-0) distributes even amount of data to the SSD and disk without considering their performance. In addition, no data are distributed to the disk in configuration TridentFS-size (ssd, disk) under most workloads. This is because in these workloads, all the evicted data are smaller than the size threshold and thus is only distributed to the SSD, causing SSD to be busy but the disk to be idle during data distribution. TridentFS (ssd, disk) distributes the amount of data to both storage devices according to the storage performance. From Table II, TridentFS (high-speed ssd, disk) distributes more data to the SSD, compared with TridentFS (low-speed ssd, disk), because a faster SSD is used.

As shown in Figures 8 and 9, under the read dominated benchmark (*webproxy*), the five configurations utilizing NVRAM have similar performance because most I/O activities occur in the

DOI: 10.1002/spe

Table II. Percentages of evicted data distributed to SSD and disk.

	TridentFS-single		TridentFS-size		TridentFS	
	RAID-0 (high-speed ssd, disk)	RAID-0 (low-speed ssd, disk)	high-speed ssd, disk	low-speed ssd, disk	high-speed ssd, disk	low-speed ssd, disk
create-files	50/50	50/50	23/77	23/77	44/56	33/67
postmark	50/50	50/50	100/0	100/0	91/9	88/12
fileserver	50/50	50/50	100/0	100/0	77/23	67/33
webproxy	50/50	50/50	100/0	100/0	54/46	30/70
untar	50/50	50/50	100/0	100/0	60/40	37/63
kernel-compile	50/50	50/50	100/0	100/0	46/54	18/82

NVRAM and DRAM, and only at most 4% of the I/O activities are issued to the flash memory and disk, leading to unnoticeable performance differences. For the CPU dominated benchmark (*kernel-compile*), similar performance is achieved for all of the configurations.

In the following, the performance of TridentFS is compared with that of ZFS, a file system that can manage hybrid storage devices. In this paper, a Linux port of ZFS (i.e., ZFS on Linux, version 0.6.2) obtained from [33] is used for performance comparison. Figure 10 shows the performance results of TridentFS and ZFS. The disk and one of the SSDs are used as the underlying storage devices. For ZFS, both the disk and the SSD are assigned as simple storage devices in the same storage pool. From the figure, TridentFS achieves higher performance than ZFS. This is mainly because TridentFS results in fewer I/O accesses to the underlying storage devices, as shown in Figure 11. As mentioned above, TridentFS outperforms ext4 because the use of the NVRAM (and the *NVRAM manager*) reduces the amount of I/O to the underlying storage devices (i.e., flash memory and disk). Moreover, as mentioned in Section 2.3, current ZFS implementation has its own L1ARC cache in addition to the Linux page cache. Because of the double-caching problem (i.e., a piece of data can reside in both the L1ARC and the Linux page cache), less data can be cached in the memory. Therefore, ZFS results in more I/O accesses to the underlying storage devices than ext4, which relies on the Linux page cache for caching its data. Figure 11 also shows that ZFS does not effectively leverage the parallelism between the flash memory and disk. Specifically, nearly the same amount of data is distributed to the storage devices (i.e., SSD and disk) that have different performance.

Note that ZFS allows a device in a pool to be used as a read cache, instead of a simple storage device. For example, the SSD can be assigned to the L2ARC (second-level ARC) to act as the

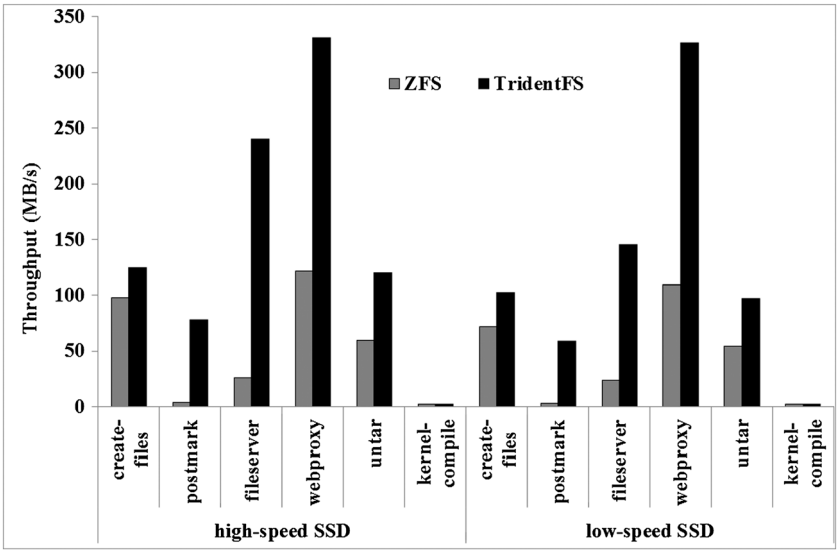

Figure 10. Performance comparison between ZFS and TridentFS.

DOI: 10.1002/spe

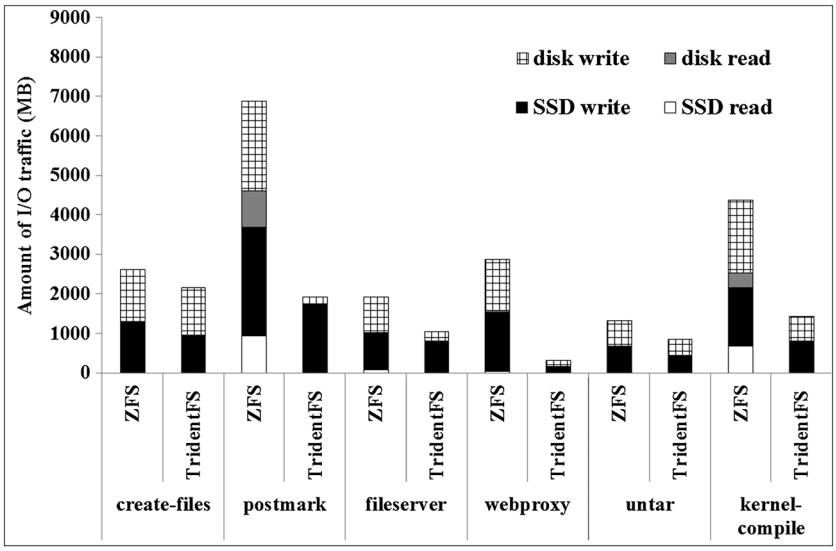

Figure 11. Amounts of I/O in ZFS and TridentFS (with high-speed SSD).

second-level read cache to the disk. Experiments were also conducted for this configuration. However, this configuration does not lead to higher performance than assigning both the SSD and disk as simple storage devices. The reasons are as follows. First, most of the workloads used in this paper are write-dominated, in which a read cache is not helpful for improving the performance. Second, extra I/O accesses are required to copy data from the disk to the L2ARC (i.e., the SSD) for caching the data in the L2ARC. Thus, the performance of TridentFS is compared with the performance of ZFS in which both the SSD and the disk are assigned as simple storage devices.

4.4. Evaluation of NVRAM sizes

In this section, the performance of TridentFS with various NVRAM sizes is evaluated, and the results are shown in Figure 12. For each workload, the results are normalized to the performance when 4 MB of NVRAM is used. From the figure, it is obvious that the performance improves with the growth of the NVRAM size. Specifically, the case of 384-MB NVRAM outperforms the case of 4-MB NVRAM by up to 484% and 625% under the high-speed and low-speed SSDs, respectively. This is due to less I/O traffic to the SSD and disk when larger NVRAM is used.

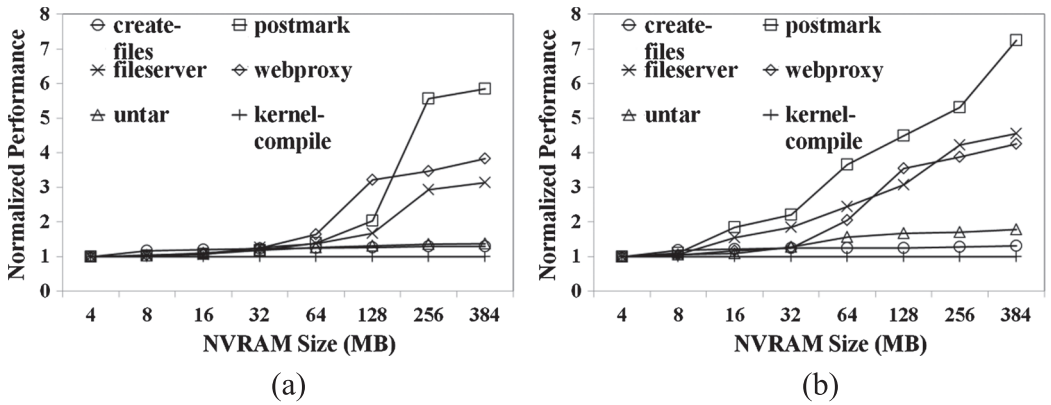

Figure 12. Performance of TridentFS with different NVRAM sizes under high-speed SSD (a) and under low-speed SSD (b).

DOI: 10.1002/spe

4.5. Evaluation of data eviction policies

Keeping hot data in the NVRAM helps to improve performance by reducing I/O accesses to the slower flash memory and disk. In this section, the policy used in TridentFS, as described in Section 3.4, is compared with two commonly used policies, FIFO and LRU. In the FIFO policy, the data first entering the NVRAM is firstly selected for eviction. In the LRU policy, the least recently used data in the NVRAM are selected for eviction. Figure 13 shows the results of these three policies under the TridentFS (ssd, disk) configuration when high-speed and low-speed SSDs are used. For each benchmark, the results are normalized to the throughput of the proposed policy.

As shown in the figure, the proposed policy achieves either better or similar performance than both the FIFO and the LRU policies. Specifically, when the high-speed SSD is used, the proposed policy outperforms the FIFO policy and the LRU policy by up to 23% and 15%, respectively. When the low-speed SSD is used, the proposed policy outperforms FIFO and LRU by up to 30% and 27%, respectively. Therefore, considering frequency, recency and size is helpful for performance improvement in the TridentFS architecture. Further advanced policies can be considered and evaluated in the TridentFS architecture in the future.

4.6. Performance impact of high and low watermarks

As mentioned in Sections 3.4 and 3.5, data eviction starts when the utilization of the NVRAM space exceeds T_H (i.e., the high watermark), and it stops when the utilization drops below T_L (i.e., the low watermark). In this section, the performance of TridentFS under various values of T_H and T_L is evaluated. Figure 14 shows the results with different values of T_H, ranging from 0.6 to 1.0, under the high-speed and low-speed SSDs, respectively. The difference between T_H and T_L is fixed as 0.05, and the results are normalized to the performance of TridentFS when T_H is set as 1.0. According to the figure, setting T_H as 1.0 (100%) achieves good performance under all of the workloads, and therefore, T_H is set as 1.0 in the other experiments.

Figure 15 shows the results with various differences between T_H and T_L under high-speed and low-speed SSDs, respectively. The results are normalized to the performance of TridentFS when the difference between T_H and T_L is 0.05, and T_H is set as 1.0 in this experiment. As seen, setting the difference between the T_H and T_L as 0.05 achieves good performance under all the workloads.

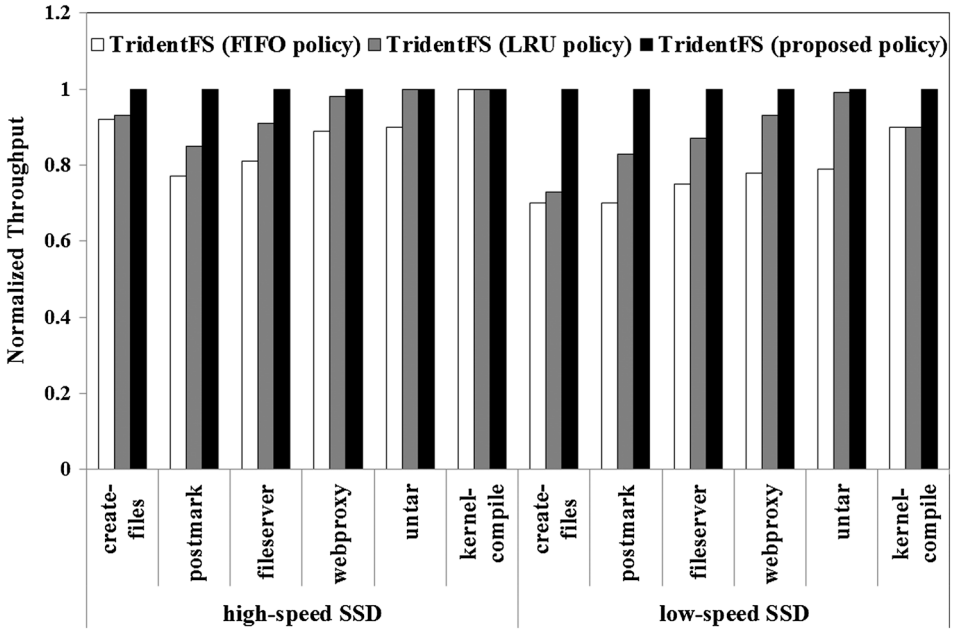

Figure 13. Performance of TridentFS with different data eviction policies under high-speed and low-speed SSDs.

DOI: 10.1002/spe

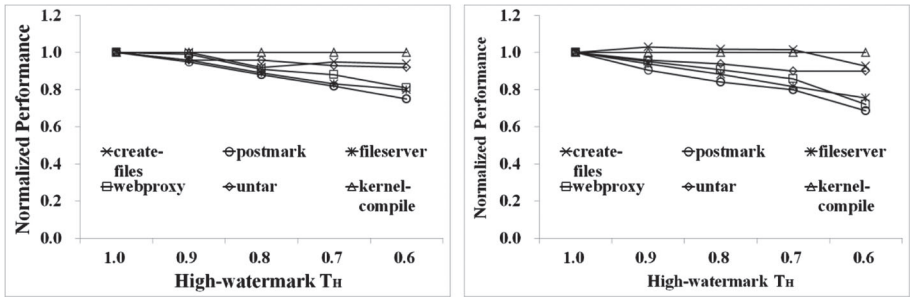

Figure 14. Normalized performance of TridentFS with various values of T_H under high-speed SSD (a) and low-speed SSD (b).

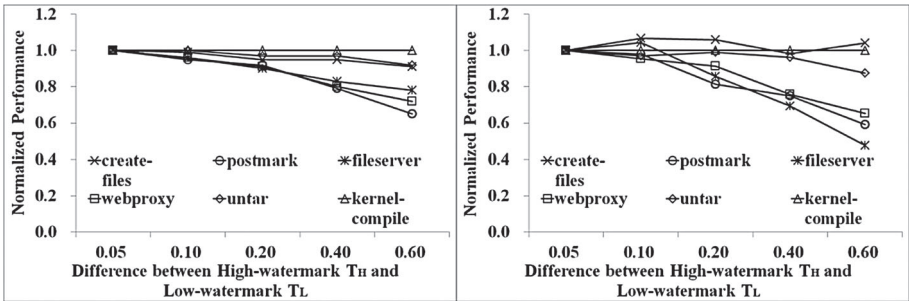

Figure 15. Normalized performance of TridentFS with various differences between T_H and T_L under high-speed SSD (a) and low-speed SSD (b).

To sum up, setting larger values for both T_H and T_L results in superior performance because this allows more data to be kept in the NVRAM. Although such settings could have the side effect of increasing the frequency of file operations blocking for available NVRAM, according to the evaluation results, the benefit of keeping more data in the NVRAM outweighs the overhead of the side effect.

4.7. Evaluation of storage performance measurement schemes

As described in Section 3.5, in TridentFS, data are distributed according to the storage runtime performance, which is measured periodically by the *device performance monitor*. Compared with offline schemes, this online scheme is more practical because it does not require any prior knowledge about the to-be-executed workloads before workload execution. In this section, we show that this online scheme can achieve comparable performance as an offline scheme. Figure 16 shows the throughput of the benchmarks under the high-speed and low-speed SSDs for both the online and offline schemes.

The offline scheme performs a trial run of a workload on the SSD and disk individually, before the (official) execution of the workload, to obtain the performance of the SSD and the disk under the workload. The amount of data distributed to the SSD and disk is proportional to the storage performance. For example, from Figure 8, the performance ratio of high-speed SSD and disk under *create-files* is 1.05, which can be obtained from the results of the TridentFS-single (ssd) and TridentFS-single (disk) configurations under *create-files*. Thus, the offline scheme uses this value to determine the amount of data distributed to the SSD and disk, that is, 51% (1.05/2.05) of the evicted data are distributed to the SSD while 49% (1/2.05) of the evicted data are distributed to the disk. As shown in Figure 16, the adopted online scheme achieves comparable performance with the offline trial-run scheme without prior knowledge (obtained by trial runs) about the workloads.

DOI: 10.1002/spe

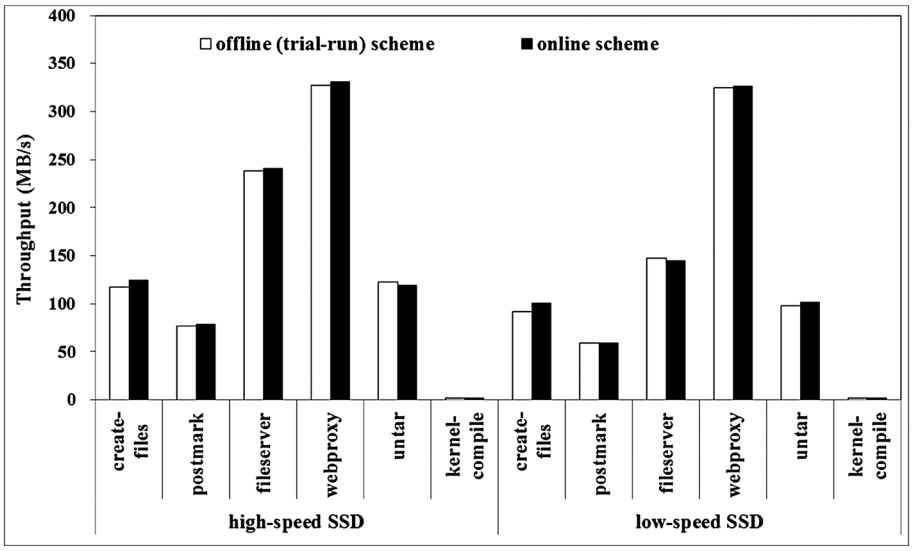

Figure 16. Performance of TridentFS with online/offline performance measurement schemes.

4.8. *Performance of the distribution algorithm*

In this section, the performance of the proposed distribution algorithm (shown in Figure 6) is evaluated. Figure 17 compares the performance of TridentFS when the greedy and the proposed distribution algorithms are used. In the greedy algorithm, each data are distributed to the storage with the lowest load once the data are evicted. Different from the greedy algorithm, the proposed algorithm attempts to distribute large data to the disk, resulting in improved disk performance. From the figure, the proposed algorithm outperforms the greedy algorithm by up to 13.6% and 15.5% when the high-speed and low-speed SSDs are used, respectively. This is due to the improvement in disk performance. To verify this, the performance of SSD and disk is measured by the *blktrace* tool [44], and the results under the proposed algorithm normalized to those under the greedy algorithm are also shown in Figure 17.

As seen in the figure, the proposed distribution algorithm improves the disk performance by up to 70% and 42% under the high-speed and low-speed SSDs, respectively, as compared with the greedy

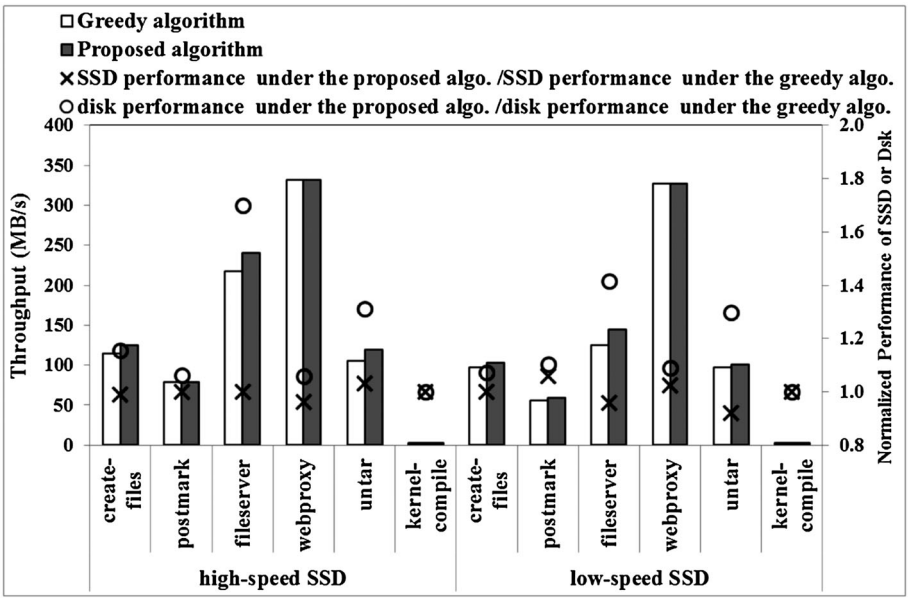

Figure 17. Performance comparison of the greedy and the proposed distribution algorithms.

DOI: 10.1002/spe

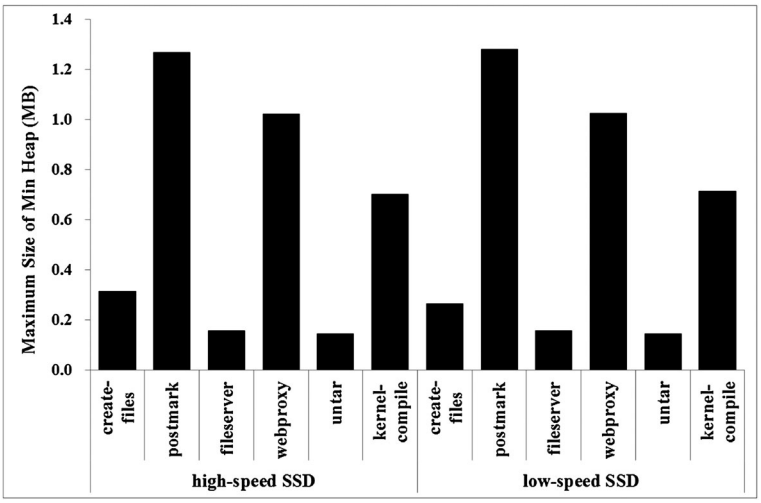

Figure 18. Memory overhead for maintaining the min-heap.

algorithm. The lower improvement of disk performance when using low-speed SSD is because more small data are distributed to the disk (according to Table II). For the SSD performance, similar results are achieved for both the proposed and greedy algorithms.

4.9. Overhead and code size

As mentioned in Section 3.4, a min-heap in the memory is used to maintain the scores Ω for all the metadata and file data in the NVRAM. In the following, the maximum memory overhead required to maintain the min-heap under various workloads are measured, and the results are shown in Figure 18. As shown in the figure, for each workload, the maximum memory overhead under the high-speed SSD is similar to that under the low-speed SSD. This is because the size of the min-heap is determined by the number of data kept in the NVRAM, which depends on the NVRAM size. Because the same size of NVRAM is used for both the SSDs, the memory overhead is similar. From the figure, the maximum memory overhead for all the workloads is less than 1.27 MB, which is insignificant compared with the 1664-MB memory.

As described in Section 3.5, the current performance of the storage is measured periodically by the *device performance monitor* thread. According to our evaluation, the overhead of the dynamic performance monitoring is 1.4 µs for each 1-s interval. Such small overhead is negligible, and it does not have noticeable impact to the system performance.

In the current implementation, the size of TridentFS is 7052 source lines of code (SLOC). Although TridentFS manages three types of storage, its size is smaller than a typical file system that manages a single type of storage. For example, the sizes of ext4 in Linux kernel 2.6.29 and nilfs 2.0.22 are 20,873 and 18,745 SLOC, respectively. TridentFS has a smaller size because it re-uses existing file system implementations to manage the flash memory and disk. As described in Section 3.1, re-using existing file system implementations largely reduces the implementation cost of TridentFS.

5. DISCUSSIONS

5.1. Consistency in Trident file system

File system consistency is critical for file system design. When a system crashes, file operations partially flushed to the storage devices may lead to inconsistency of the file system. In TridentFS, file system consistency can be achieved by maintaining both intra-storage consistency and inter-storage consistency. Intra-storage consistency indicates consistency between data structures stored in the same storage device. Inconsistency between these data structures could occur. For example, in the case of updating a file whose metadata and file data both residing in the disk, inconsistency

DOI: 10.1002/spe

occurs when the system crashes after the flush of the metadata update but before the flush of the file data update. For another example, inconsistency in the NVRAM occurs when a system crash leads to partial update of file data/metadata in the NVRAM. Intra-storage consistency can be ensured by using existing approaches. For example, a journaling file system (e.g., ext4) or a log-structured file system (e.g., nilfs) can be used for a bottomFS. Several techniques were also proposed to address data consistency issues in the NVRAM [17, 45, 46]. Because consistency is not the main focus of the current TridentFS implementation, only a simple technique (i.e., ordered write) is adopted in the current implementation to avoid dangling pointers resulting from system crashes. Although this technique can avoid dangling pointers, it cannot handle NVRAM inconsistency resulting from partial updates or media corruption. Adopting techniques used in [17, 45, 46] will be considered in the future to provide stronger data consistency in the NVRAM.

Inter-storage consistency indicates consistency between structures stored in different storage devices. Taking a TridentFS file with metadata and file data both stored in the NVRAM as an example, assume that the file data need to be evicted from the NVRAM to the disk. If the system crashes before the file data have been completely flushed to the in-disk container file, the metadata in the NVRAM may refer to a non-existing container file or refer to a container file with incomplete file data, leading to inter-storage inconsistency. Such inconsistency could still occur even when intra-storage consistency is maintained. In the above example, the inconsistency still exists even when a journaling file system is used as the bottomFS of the disk. Specifically, the in-NVRAM metadata may still refer to a non-existing container file after the recovery procedure of the bottomFS is performed.

To ensure inter-storage consistency in TridentFS, a cross-storage file system journaling approach can be used. Each file system update that involves multiple storage devices should be logged to the reserved journal space before the update is flushed to the flash memory or disk. Data evictions also need to be logged because they involve multiple storage devices. As in existing journaling approaches [47, 48], the logging is performed in transactions. To effectively leverage the parallelism between the flash memory and disk devices, a separate journal space is allocated on each device, and the logging (i.e., writing of the journal data) is performed on the basis of the distribution algorithm shown in Figure 6. That is, the distribution algorithm is also used to determine the amount of journal data written to each journal space. Once the system crashes, the recovery procedures for ensuring intra-storage consistency are performed first. Then, TridentFS performs the recovery procedure for inter-storage consistency by replaying the completed transactions in the journal spaces. The implementation of this cross-storage journaling approach is planned as future work.

5.2. File-level versus block-level design

TridentFS uses a file-level design, instead of a block-level design. The reasons are as follows. First, compared with a file-level design, a block-level design leads to a larger memory overhead because hot/cold information such as recency and frequency has to be maintained for each block to determine which blocks should be stored in the NVRAM. Second, a file-level design can utilize more information for hot/cold separation than a block-level design. For example, information such as file size and file name extension, which may also be helpful for hot/cold separation, is not available for a block-level design. Third, a file-level design can take advantage of the byte-addressable feature of the NVRAM. For instance, for a file with its metadata stored in the NVRAM, when the modification time (*mtime*) of the file is updated (e.g., because of a *touch* command), a block-level design requires modifying the entire block (1 kB) containing the metadata. By contrast, for the file-level design used in TridentFS, only the *mtime* field (4 B) of the metadata needs to be modified.

5.3. Data migration to the NVRAM

In the current implementation, TridentFS does not migrate evicted data back to the NVRAM. When evicted data become hot, migrating them back to the NVRAM helps to reduce the amount of IO activities to the slower flash memory and disk, improving the system performance. However, a larger memory overhead is required for supporting such data migration. This is because hot/cold

DOI: 10.1002/spe

information should also be maintained for evicted data (including both metadata and file data) so as to determine which evicted data should be migrated back to the NVRAM.

To implement the data migration, the score in Equation (1) can be used to determine the degree of hotness of each evicted data. When the score exceeds a specific threshold, the corresponding data can be migrated back to the NVRAM. The migration can be performed in the background periodically (with a low priority task) or when the file system becomes idle. Implementation of such data migration is left as future work.

6. CONCLUSIONS

In this paper, a hybrid file system with high flexibility and performance, called TridentFS, is proposed to manage three types of storage with different characteristics, that is, NVRAM, flash memory and magnetic disk. Novel techniques are used in TridentFS to improve the flexibility and performance. It provides high flexibility by supporting various forms of flash memory and a wide range of NVRAM sizes. Various forms of flash memory are supported by adopting suitable bottom file systems, and a wide range of NVRAM sizes is supported by allowing data eviction from the NVRAM. High performance is achieved by keeping hot data in the NVRAM and by allowing data evicted from the NVRAM to be parallel distributed to the flash memory and disk. The proposed data eviction policy determines which data should be evicted from the NVRAM when NVRAM runs out of its space according to the recency, frequency and size of the data. In addition, the proposed data distribution algorithm speeds up the data distribution by effectively leveraging the parallelism between the flash memory and disk and by considering the data size and storage performance characteristics.

We have implemented TridentFS as a loadable module on Linux 2.6.29. Six benchmarks were used for performance evaluation. According to the performance results, TridentFS works well for both small-sized and large-sized NVRAM. The proposed eviction policy outperforms LRU by 27%. Moreover, because of effectively leveraging the parallelism between the flash memory and disk, the proposed data distribution algorithm outperforms the RAID-0 and size-based distribution methods by up to 471.6% and 82.6%, respectively. By considering the data size and performance characteristics of the storage devices, the proposed data distribution algorithm outperforms the greedy algorithm by up to 15.5%.

In the future, support of NVRAM with different characteristics to DRAM (e.g., PCRAM) will be considered. In addition, issues related to file system consistency and migration of evicted data will be addressed, and more sophisticated data eviction policies will be evaluated.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments on this paper. This research was supported in part by grants NSC 101-2221-E-006-098-MY3 and NSC 101-2221-E-006-150-MY3 from the National Science Council, Taiwan, Republic of China.

REFERENCES

1. Dong XY, Jouppi NP, Xie Y. PCRAMsim: system-level performance, energy, and area modeling for phase-change ram. *ICCAD '09: Proceedings of the International Conference on Computer-Aided Design*, ACM, NY, USA, 2009; 269–275. DOI: 10.1145/1687399.1687449.
2. Jung M, Shalf J, Kandemir M. Design of a large-scale storage-class RRAM system. *ICS '13: Proceedings of the 27th ACM Conference on International Conference on Supercomputing*, ACM, NY, USA, 2013; 103–114. DOI: 10.1145/2464996.2465004.
3. Zhao W, Torres L, Guillemenet Y, Cargnini LV, Lakys Y, Klein JO, Ravelosona D, Sassatelli G, Chappert C. Design of MRAM based logic circuits and its applications. *GLSVLSI '11: Proceedings of the 21st Great Lakes Symposium on VLSI*, ACM, NY, USA, 2011; 431–436. DOI: 10.1145/1973009.1973104.
4. Manning C. How Yaffs works. Available at: http://www.yaffs.net/documents/how-yaffs-works [last accessed 1 June 2012].
5. Gupta A, Kim Y, Urgaonkar B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. *ACM SIGARCH Computer Architecture News* 2009; **37**(1):229–240.

6. Qiu S, Reddy ALN. NVMFS: a hybrid file system for improving random write in NAND-flash SSD. *MSST '13: Proceedings of the 29th Symposium on Mass Storage Systems and Technologies*, IEEE, CA, USA, 2013; 1–5. DOI: 10.1109/MSST.2013.6558434.
7. Wang AA, Kuenning G, Reiher P, Popek G. The Conquest file system: better performance through a disk/persistent-RAM hybrid design. *ACM Transactions on Storage* 2006; **2**(3):309–348.
8. Miller EL, Brandt SA, Long DDE. HeRMES: high-performance reliable MRAM-enabled storage. *HotOS '01: Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, IEEE, Elmau, Germany, 2001; 95–99.
9. Park Y, Lim SH, Lee C, Park KH. PFFS: a scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash. *SAC '08: Proceedings of the ACM Symposium on Applied Computer*, ACM, NY, USA, 2007; 1498–1503. DOI: 10.1145/1363686.1364038.
10. Mathur A, Cao M, Bhattacharya S, Dilger A, Tomas A, Vivier L. The new ext4 file system: current status and future plans. *OLS '07: Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, 2007; 21–34.
11. Konishi R, Amagai Y, Sato K, Hifumi H, Kihara S, Moriai S. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review* 2006; **40**(3):102–107.
12. Wu M, Zwaenepoel W. eNVy: a non-volatile, main memory storage system. *ASPLOS '94: Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, CA, USA, 1994; 86–97. DOI: 10.1145/195473.195506.
13. Chen PM, Ng WT, Chandra S, Aycock C, Rajamani G, Lowell D. The Rio file cache: surviving operating system crashes. *ASPLOS '96: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, MA, USA, 1996; 74–83. DOI: 10.1145/237090.237154.
14. Baker M, Asami S, Deprit E, Ouseterhout J, Seltzer M. Non-volatile memory for fast, reliable file systems. *ASPLOS '92: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, MA, USA, 1992; 10–22. DOI: 10.1145/143371.143380.
15. Chen JX, Wei QS, Chen C, Wu LK. FSMAC: a file system metadata accelerator with non-volatile memory. *MSST '13: Proceedings of the 29th Symposium on Mass Storage Systems and Technologies*, IEEE, CA, USA, 2013; 1–11. DOI: 10.1109/MSST.2013.6558440.
16. Edel NK, Tuteja D, Miller EL, Brandt SA. MRAMFS: a compressing file system for non-volatile ram. *MASCOTS '04: Proceedings of the 20th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, IEEE, Volendam, Netherlands, 2004; 596–603.
17. Condit J, Nightingale EB, Frost C, Ipek E, Lee B, Burger D, Coetzee D. Better I/O through byte-addressable, persistent memory. *SOSP '09: Proceedings of the 22th Symposium on Operating Systems Principles*, ACM, NY, USA, 2009; 133–146. DOI: 10.1145/1629575.1629589.
18. Wu XJ, Qiu S, Reddy ALN. SCMFS: a file system for storage class memory and its extensions. *ACM Transactions on Storage* 2013; **9**(3):7–7.
19. Doh IH, Choi J, Lee D, Noh SH. Exploiting non-volatile RAM to enhance flash file system performance. *EMSOFT '07: Proceedings of the 7th International ACM & IEEE Conference on Embedded Software*, ACM, NY, USA, 2007; 164–173. DOI: 10.1145/1289927.1289955.
20. Jung J, Won Y, Kim E, Shin H, Jeon B. FRASH: exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage* 2010; **6**(1):3–3.
21. Kleiman SR. Vnodes: an architecture for multiple file system types in Sun UNIX. *Proceedings of the USENIX Summer Technical Conference*, USENIX, CA, USA, 1986; 238–247.
22. Bolosky WJ, Corbin S, Goebel D, Douceur JR. Single instance storage in Windows 2000. *WSS '00: Proceedings of the 4th Conference on USENIX Windows Systems Symposium*, USENIX, WA, USA, 2000; 2–2.
23. Aranya A, Wright CP, Zadok E. Tracefs: a file system to trace them all. *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, USENIX, CA, USA, 2004; 129–145.
24. Zadok E, Nieh J. FiST: a language for stackable file systems. *ACM SIGOPS Operating Systems Review* 2000; **34**(2):38–38.
25. Heidemann JS, Popek GJ. File-system development with stackable layers. *ACM Transactions on Computer Systems* 1994; **12**(1):58–89.
26. Garrison JA, Reddy ALN. Umbrella file system: storage management across heterogeneous devices. *ACM Transactions on Storage* 2009; **5**(1):3–3.
27. Lee LW, Scheuermann P, Vingralek R. File assignment in parallel I/O systems with minimal variance of service time. *IEEE Transactions on Computers* 2000; **49**(2):127–140.
28. Xie T, Sun Y. A file assignment strategy independent of workload characteristic assumptions. *ACM Transactions on Storage* 2009; **5**(3):10–10.
29. Zhu YQ, Yu Y, Wang WY, Tan SS, Low TC. A balanced allocation strategy for file assignment in parallel I/O systems. *NAS '10: Proceedings of the 5th Conference on Networking, Architecture, and Storage*, IEEE, DC, USA, 2010; 257–266. DOI: 10.1109/NAS.2010.10.
30. Graham RL. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics* 1969; **17**(2):416–429.
31. Fisher N, He Z, McCarthy M. A hybrid filesystem for hard disk drives in tandem with flash memory. *Computing* 2012; **94**(1):21–68.
32. Scott W. *Solaris 10 ZFS Essentials*. Prentice Hall: New Jersey, 1998; 1–124.
33. Lawrence Livermore National Laboratory. ZFS on Linux. Available at: http://zfsonlinux.org [last accessed 23 August 2013].
34. Megiddo N, Modha DS. ARC: a self-tuning, low overhead replacement cache. *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, USENIX, CA, USA, 2003; 115–130.

35. Snyder P. Tmpfs: a virtual memory file system. *Proceedings of the Autumn European UNIX Users' Group Conference*, USENIX, London, UK, 1990; 241–248.
36. Jiang S, Ding XN, Chen F, Tan EH, Zhang XD. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. *FAST '05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, USENIX, CA, USA, 2005; 101–114.
37. Gill BS, Modha DS. WOW: wise ordering for writes – combining spatial and temporal locality in non-volatile caches. *FAST '05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, USENIX, CA, USA, 2005; 129–142.
38. Bowden T, Bauer B. The proc file system. Available at: http://www.kernel.org/doc/Documentation/filesystems/proc.txt [last accessed 7 October 1999].
39. Zadok E, Nieh J. FIST: a language for stackable file systems. *SIGOPS Operating System Review* 2000; **34**(2):38–54.
40. Wolf SA, Jiwei L, Stan MR, Chen E, Treger DM. The promise of nanomagnetics and spintronics for future logic and universal memory. *Proceedings of the IEEE* 2010; **98**(12):2155–2168.
41. Kustarz E, Shepler S, Wilson A. Filebench: the New and Improved File System Benchmarking Framework. Available at: http://www.usenix.org/events/fast08/wips_posters/slides/wilson.pdf [last accessed 27 February 2008].
42. Katcher J. PostMark: a new file system benchmark. Available at: http://communities-staging.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp-tr3022.pdf [last accessed 10 August 1997].
43. Huang TC, Chang DW. TESA: a temporal and spatial information aware writeback policy for home network-attached storage devices. *IEEE Transactions on Consumer Electronics* 2013; **59**(1):122–129.
44. Brunelle AD. Blktrace user guide. Available at: http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html [last accessed 18 February 2007].
45. Greenan KM, Miller EL. PRIMS: making NVRAM suitable for extremely reliable storage. *HotDep '07: Proceedings of the 3rd Workshop on Hot Topics in System Dependability*, USENIX, CA, USA, 2007; 10–10.
46. Shivaram V, Niraj T, Parthasarathy R, Roy HC. Consistent and durable data structures for non-volatile byte-addressable memory. *FAST '11: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, USENIX, CA, USA, 2011; 61–76.
47. Huang TC, Chang DW. VM aware journaling: improving journaling file system performance in virtualization environments. *Software: Practice and Experience* 2012; **42**(3):303–330.
48. Prabhakaran V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Analysis and evolution of journaling file systems. *FAST '05: Proceedings of the USENIX Annual Technical Conference*, USENIX, CA, 2005; 8–8.

DOI: 10.1002/spe