A New Design of In-Memory File System Based on File Virtual Address Framework

Edwin H.-M. Sha, *Senior Member, IEEE*, Xianzhang Chen, Qingfeng Zhuge, *Member, IEEE*, Liang Shi, *Member, IEEE*, and Weiwen Jiang

Abstract—The emerging technologies of persistent memory, such as PCM, MRAM, provide opportunities for preserving files in memory. Traditional file system structures may need to be re-studied. Even though there are several file systems proposed for memory, most of them have limited performance without fully utilizing the hardware at the processor side. This paper presents a framework based on a new concept, "File Virtual Address Space". A file system, Sustainable In-Memory File System (SIMFS), is designed and implemented, which fully utilizes the memory mapping hardware at the file access path. First, SIMFS embeds the address space of an open file into the process' address space. Then, file accesses are handled by the memory mapping hardware. Several optimization approaches are also presented for the proposed SIMFS. Extensive experiments are conducted. The experimental results show that the throughput of SIMFS achieves significant performance improvement over the state-of-the-art in-memory file systems.

Index Terms—Persistent memory, file system, virtual address space, in-memory file systems, performance

1 Introduction

Increasing timing requirement of data processing calls for new solutions of in-memory computing to overcome the performance penalty caused by the huge performance gap between storage and memory. For example, Spark [1], a cluster computing framework, can be 100 times faster than Hadoop, a well-known implementation of MapReduce [2] for cloud computing, because Spark supports some degrees of in-memory computing. As file systems are the most fundamental infrastructure for storing data, making file systems "in-memory" can significantly benefit applications involving data processing.

In addition to DRAM, emerging persistent memories, such as Storage Class Memory [3], [4], [5] and NVDIMM, also provide opportunities for implementing in-memory file systems. Persistent memories [6], [7] can be directly connected to the memory bus and provide fast byte-addressable data accesses. The design of traditional I/O stack, however, causes unnecessary workload and performance penalty on file systems in memory. Therefore, file system needs re-studying and re-designing in order to effectively exploit the benefits of memory, which is of great importance and also a challenging problem for system designers.

There are two types of file systems for memory. One type is temporary in-memory file systems, such as RAMFS [8] and TMPFS [9]. Temporary in-memory file systems have no sustainable metadata structures and they cannot survive system reboots. Therefore, temporary in-memory file systems are incapable for preserving persistent data. The other type is persistent in-memory file systems, such as PRAMFS [10], EXT4 [11] on Ramdisk, and PMFS [12]. Persistent in-memory file systems have their own sustainable metadata structures that are fixed on known locations of persistent memory. Thus, persistent in-memory file systems can survive system reboots and they are capable for preserving persistent data.

In file accesses, file systems search the metadata structures to find the physical location of file data. Both existing temporary and persistent in-memory file systems use software routines to search the metadata. Or they should build additional mapping tables to map the file into the virtual address space. These methods of file accesses cannot fully take advantages of the memory mapping hardware MMU.

In this paper, we propose a novel framework called "File Virtual Address Space" for high-performance persistent in-memory file systems. Within this framework, each opened file has its own contiguous virtual address space that is organized by a hierarchical page table dedicated to the file. File system can use hardware MMU to locate the physical locations of file data via the contiguous virtual address space of file. A file system, Sustainable In-Memory File System (SIMFS), is designed and fully implemented based on this framework. In SIMFS, the data of a file are organized by a structure called "file page table". The proposed framework and SIMFS can be used on any byte-addressable memory connected to the memory bus.

The file virtual address space of an opened file is embedded into the calling process' address space. This embedding process can be done efficiently by updating few pointers. Once the embedding process is accomplished, data pages can be accessed by load/store instructions directly using virtual addresses, and file data can be read without interrupt such as page fault. There are no conflict between the address spaces of files as each file is incorporated into an appropriative virtual address space.

• *The authors are with the College of Computer Science, Chongqing University, Chongqing 400044, China. E-mail: {edwinsha, xzchen109, qfzhuge, shi.liang.hk, jiang.wwen}@gmail.com.*

Manuscript received 8 May 2015; revised 10 Dec. 2015; accepted 4 Jan. 2016. Date of publication 7 Jan. 2016; date of current version 14 Sept. 2016. Recommended for acceptance by G. Lipari. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2016.2516019

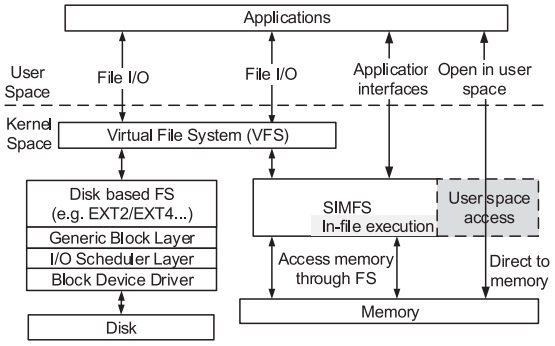


Fig. 1. An organizational view of different kinds of file systems.

SIMFS is able to take advantages of memory mapping hardware in CPU to access file data with significant high throughput. The experimental results show that the throughput of SIMFS can approach the memory bus bandwidth. As Linux, where paging system is its core memory management structure, supports most of existing modern CPUs, our framework can be generally applied to different CPUs. We believe the proposed design framework will have long-lasting impact on the design of persistent in-memory file system.

SIMFS can be implemented in either kernel space or user space with little modification. When it is implemented to support user space file data accesses, the file is opened in the user process directly, i.e., SIMFS inserts the pointers at the top-level file page table into the page table in a user process. Then, an application is able to directly manipulate file data at its own address space with no need of copying data to the user buffer, as shown in Fig. 1. SIMFS can map the whole file into user space directly with a constant time operation regardless of the file size.

We propose a technique called Pseudo-File-Write (PFW) for SIMFS to efficiently achieve the same consistency as that achieved by copy-on-write (COW) [13], [14], [15]. If a file system writes new data in new back-up space by COW, the number of data transfer function called by the file system equals the number of pages for new data. In the proposed framework, PFW calls data transfer function just once to write the whole new data. Therefore, PFW can provide more efficient consistency than standard COW.

We propose a method called in-file execution for application operations. Traditional method may cause large overhead on data copies between buffers and files. SIMFS provides new interfaces for applications, as shown in Fig. 1. With in-file execution method, applications directly manage files in the file system without copying data to any buffers. Furthermore, this method can reduce the number of required system calls. Therefore, in-file execution can bring higher efficiency for applications than traditional method.

SIMFS also implements the generic interfaces of Linux VFS. Moreover, SIMFS supports parallel file accesses of multiple threads and various executables. Thus, it can cohabit with other file systems that use the VFS interfaces, such as EXT4. Besides, SIMFS can also be used as a local file system for distributed file systems, such as HDFS.

We evaluate the performance of SIMFS using both micro-benchmarks, such as FIO [16], and macro-benchmarks, such as Filebench [17], with various sizes of I/O requests. Experimental results show that the performance for both sequential and random reads/writes of SIMFS is significantly improved comparing with typical file systems, including EXT4 [11] on Ramdisk, RAMFS [8], PMFS [12], PRAMFS [10]. The throughput of SIMFS approaches the memory bus bandwidth in best cases. To the authors' knowledge, this is the best known result for file systems in literature.

The main contributions of this paper include:

- We propose a novel framework, file virtual address space, for high-performance, persistent in-memory file system.
- We design and implement a fully functional in-memory file system, SIMFS, based on the proposed framework.
- We propose a technique, PFW, to achieve efficient data consistency in SIMFS.
- SIMFS can be easily extended to support user space file data accesses. SIMFS can map the whole file into user space directly with an O(1) constant time operation regardless of file size.
- We offer a method, in-file execution, for the execution of applications to achieve higher performance by eliminating data copies between user buffer and files.
- Extensive experiments are conducted with both micro-benchmarks and macro-benchmarks to compare the performance of SIMFS with typical file systems, including EXT4 on Ramdisk, RAMFS, PMFS, and PRAMFS, etc. Experimental results show that SIMFS outperforms the state-of-the-art file systems in most cases.

The rest of the paper is organized as follows. We discuss existing file systems and show motivation in Section 2. Then we present the design principles and the framework in Section 3. In Section 4, we describe the implementation of SIMFS. We propose the in-file execution in Section 5. We present experimental results in Section 6. Finally, Section 7 concludes the paper.

2 Background and Motivation

2.1 An Overview of File Systems

With the objective of designing a new in-memory file system for memory in our mind, we would like to first discuss the issues and problems with three major categories of existing file systems that make it hard to directly apply the existing file systems to memory.

The disk-based file systems, e.g., Linux file systems EXT2 and EXT4 [11], are designed for the traditional architecture with fast volatile main memory and slow secondary storage. Data I/O requests have to travel through a deep stack of software layers, as shown in Fig. 1. Buffering and management block I/Os in the software stack, such as generic block layer, introduce unnecessary workload, causing significant performance penalty [12], [18]. Context switch possibly happening during I/O interrupt is time and energy consuming. Also, the data structure of metadata designed for block-based storage device is inappropriate for byte-addressable memory. The performance penalty cannot be avoided even using tools like Ramdisk which simulates hard disk in memory. The experimental results show that the proposed in-memory file system SIMFS reaches 5 times better performance than EXT4

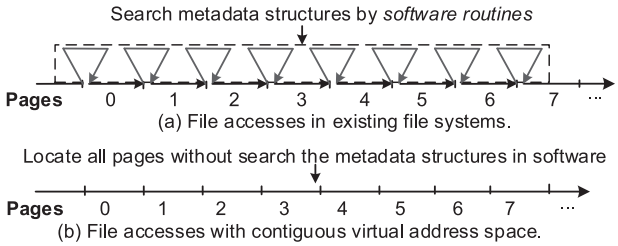


Fig. 2. Illustration of the paths of file accesses.

on Ramdisk. Generally speaking, in-memory file systems can be faster than file systems on hard disk by three orders of magnitude.

In-memory file systems, such as RAMFS [8] and TMPFS [9], [19], are simple implementations using disk caching mechanisms including page cache, inode cache, and directory cache for fast data accesses. Their metadata such as inodes do not have sustainable structures cross system reboots even persistent memory is used. Thus, these file systems cannot be used for preserving persistent data. The experiments show that the proposed SIMFS with data persistency, can be two times faster than RAMFS.

Quite a few research efforts have been taken in designing and developing persistent, in-memory file systems in recent years [10], [12], [15], [18], [20], [21], [22]. The main challenge in designing a persistent in-memory file system is to design an efficient persistent metadata structure. One major purpose of metadata structures is to indexing the physical locations of the requested file data. File systems usually search the metadata structure for mapping the logic addresses of the requested file data into the corresponding physical addresses. Different file systems design different structures to do such a mapping.

For example, PRAMFS [10] uses a two-dimensional array to organize data pages. File data are accessed by searching the array for physical addresses of the pages. Experimental results show that SIMFS is faster than PRAMFS by two orders of magnitude. Another example is PMFS [12], a persistent in-memory file system implemented in Linux kernel. Its file data and metadata are organized in B-tree. Hence, when a process read or write data, PMFS needs to search the B-tree for the physical addresses of the requested data pages. The search process on the metadata structures of these file systems are usually accomplished by going through related software routine, causing significant overhead.

We believe that the most appropriate way to design a persistent, in-memory file system is to incorporate file data into virtual address space and bypass traditional software layers in I/O stack. More importantly, structures in the file system need to be carefully designed to fully take advantage of memory mapping hardware, such as the Memory Management Unit (MMU), for boosting the address mapping of file data. As a result, we may achieve high performance by avoiding the overhead for traversing software routines.

2.2 Motivational Example

File system usually manages two types of data, the metadata that describes attributes of files and the contents of files, i.e., file data. The metadata keeps the reference to the location of each data page in the physical device. To find

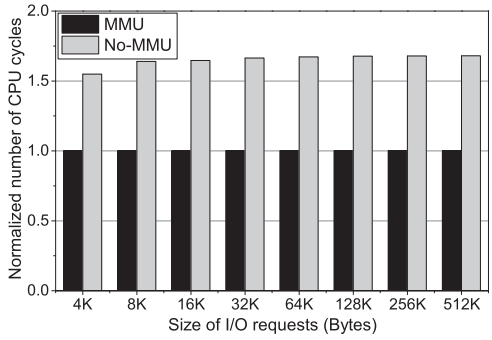


Fig. 3. An example: MMU versus No-MMU.

the location of a data page (or data block) requires traversing the data structure of metadata.

During file accesses, existing file systems, such as EXT4 [11], search the physical location of each page through software routines. Take Fig. 2a as an example. If k pages are accessed in a file, the file system should do the following steps with k times:

1) The file system searches metadata to find the physical location of the next requested page;
2) The file system calls data transfer function to access the data in the current page.

Thus, the file access has overhead for traversing software routines. Such an overhead can be avoided if a contiguous virtual address space for the file is provided.

Considering the Process Management in Linux, when a process is initialized, the process has its own contiguous virtual address space. Then the process can locate the physical addresses of contiguous virtual pages without traversing any software routines. It is because that the virtual address space is translated to the physical address space by MMU and the related process page table.

Similarly, if a file in the memory has a contiguous virtual address space, the physical locations of the file data can be quickly located by the corresponding virtual addresses and MMU. As shown in Fig. 2b, the file system only accesses k pages by one data transfer function without traversing any metadata. Therefore, the performance of file accesses in contiguous virtual address space is much lower than that in existing file systems.

For example, the file data accesses of EXT4 and the approach using contiguous virtual address space are evaluated. The experiments are conducted on SimpleScalar [23]. The normalized number of CPU cycles is presented in Fig. 3. No-MMU denotes the method of EXT4 and MMU denotes the method using contiguous virtual address space. The experimental results show that MMU is 1.65 times faster than No-MMU, on average. The cost of software routines in the real file system can be much higher than that of the simulation experiments, since some subroutines, such as locks, are not taken into account in the simulation.

In this paper, we try to design a novel framework that offers contiguous virtual address space for files for in-memory file systems.

3 Design Principles and Framework

In this section, we introduce the framework of file virtual address space and a new structure of file page table designed

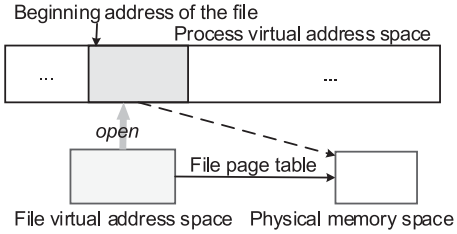


Fig. 4. An illustration of file virtual address space.

for managing file address space with high efficiency. Then, an efficient file access mechanism is presented for applications to directly access files on persistent memory. Finally, we discuss the maximum file size.

3.1 File Virtual Address Space

We propose a new framework based on the concept that each file has its own contiguous "*file virtual address space*." In the new framework, when a file is opened, its file virtual address space is embedded into the virtual address space of process, as shown in Fig. 4. Thus, each opened file has its own contiguous virtual address space. The virtual space of a file is mapped to the physical space by "*file page table*." The file virtual address space is detached from the process virtual address space when the corresponding file is closed.

With contiguous virtual address space, any data in a file can be quickly located by an addition on the beginning address of the file and the corresponding offset. The translation from virtual address to physical address is done by address translation hardware, Memory Management Unit.

Therefore, to read contiguous pages, say k pages, in contiguous virtual address space, the file system merely calculates the beginning address of the contiguous k-page address space, then calls data transfer function only once to read the whole k-page data. A write to an existing file address space behaves the same way as a read. On the contrary, traditional file system should search the beginning addresses of all k pages in metadata and call data transfer function k times. In consequence, once the virtual address space of an open file is set up, the file accesses are performed efficiently as following:

1) File system can read file data without searching for metadata by software routine. Furthermore, as the file virtual address space has been embedded into the process virtual address space, the read can be accomplished with no page fault.
2) For the writes on existing file address space, the situation is same to read, i.e. no software search on metadata.
3) For appending k pages, the file system first allocates k pages and updates the corresponding page tables. Then the situation is similar to 2).
4) File access is accomplished by just one data transfer function call using the contiguous virtual address space. The address translation is done by corresponding MMU hardware.

Therefore, the proposed framework provides an efficient way to conduct file accesses for in-memory file systems.

In order to fully take advantage of address translation hardware and efficiently set up the virtual address space related to file, we propose a new structure, *file page table*, to keep the address mapping information for each data page of a file. File page table is similar to a process page table. From the view of address translation hardware, file page table looks exactly like a section of any normal page table.

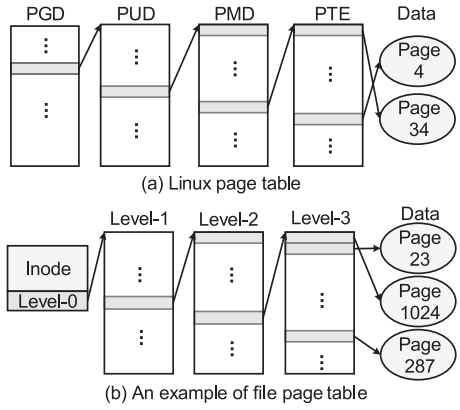


Fig. 5. An illustration of Linux page table and file page table.

Fig. 5 shows an illustration of file page table in Linux. Fig. 5a shows the Linux page table that has four levels, i.e., PGD, PUD, PMD, and PTE. Each entry in the page table may keep the beginning physical address of a page of the next level page table, e.g., a PMD entry keeps the pointer of a PTE physical page.

Fig. 5b shows an example of file page table in Linux that also has four levels; level-0, level-1, level-2, and level-3. The same as Linux page table, each entry in the file page table may keep the starting physical address of a page of the next level file page table, e.g., a level-2 entry keeps a pointer of level-3 physical page. Each entry in the bottom level file page table may store a pointer of a physical data page. The top level of the file page table, i.e., level-0, is stored in the inode structure of the corresponding file. As shown in Fig. 5b, all the data pages of a file are organized in a contiguous virtual address space, but they can actually scatter on physical memory.

Within the framework, each file has a file page table. When a file is created, the file page table is first constructed where all the entries in the bottom level page table are set to be nil. The file page table in the implementation is not necessarily to be four levels, such as three-level file page tables for directory files. The pages of file page table are scattered in data page section of the file system.

In the new framework, file page table is used to set up the virtual address space related to the open file. When the file is opened, the pointer in the top level file page table is written to the corresponding page table entry of the calling process, through which the file virtual address space of the open file is embedded into the calling process' virtual address space. For example, to set up the virtual address space for the file in Fig. 5b, the pointer in level-0 is inserted into the corresponding PGD entry in Fig. 5a.

Based on the new framework, it takes $O(1)$ time to set up file virtual address space. Suppose that the pointers to data pages are stored in metedata as in traditional file systems, inserting all the pointers to process page table literally takes $O(n)$ time where n is the number of data pages in a file.

The proposed framework is different from the concept of SCMFS [18] in two aspects. First, the time to hold the virtual

address space is different. In SCMFS, each file has a segment of kernel virtual address space no matter if it is opened or not. Thus, the management of virtual address space in SCMFS is difficult. Second, the management of memory mapping is different. SCMFS saves a memory mapping table in the persistent memory and maintains the in-memory structures in DRAM. Any modifications to these data structures should be flushed to the persistent memory immediately. Therefore, any updates of the memory mapping information in SCMFS should be performed twice. In the proposed framework, any updates to the file page table are performed only once.

The maximum file size is configurable, which is determined by the implementation of the file page table. For example, the file page table in Fig. 5b contains four levels. This file page table is able to support file size with $2^{12}/2^3 * 2^{12}/2^3 * 2^{12}/2^3 * 2^{12} = 512$ GB. The maximum file size can be expanded to several tera bytes with a couple of level-0 pointers in the inode, where each pointer supports 512 GB. This size is large enough for files of the in-memory file systems. Setting up address space for huge files takes constant cost. It is because the contiguous file page table entries can be written to the process page table by one data transfer function call. Note that for a given implementation of file page table, the maximum file size is fixed.

The number of opened files is limited by the implementation of the systems. For 64-bit systems, if the size of the largest file is set to 4 TB and half of the virtual address space is reserved for the opened files, then the number of files that can be opened simultaneously is more than two million. In the current 48-bit systems, the maximum size of file is set to 32 GB, half of the virtual address space is reserved for opened files, and the number of files that can be opened simultaneously is more than two thousands.

3.2 Efficient File Accesses in User Space

To access file data, general read/write interfaces have to copy data two times: first copy data from file to the kernel buffer, and then copy the data from kernel buffer to the user buffer. Memory copy introduces extra overhead for file accesses. Moreover, such procedures cost large context switch overhead [24]. Compared with the general read/write, user space file access can reduce context switches and data copies [25]. With the user space file access method, an application can directly access file data using the virtual addresses of the corresponding user process.

Most existing file systems use memory mapping (*mmap*) approaches to enable user space file access. *mmap* does two things to map a file into the user space: Copy requested data to page cache; construct the mapping for pages in page cache on the user space page table. Nevertheless, the file system cannot benefit from *mmap* when the size of the transmitted data is small, since *mmap* has overhead in establishing the mapping.

The framework of the file virtual address provides a more efficient way to enable direct file accesses in the user space. To map the file virtual address space into the process space, a special mapping function looks up the inode of the requested file for pointers referencing the file page table and the file size. A segment of contiguous address space equals the maximum file size is allocated in the user process space. The pointer at top level page table in inode is then inserted into the corresponding entry of the user space page table, through which the file virtual address space is embedded into the user virtual address space.

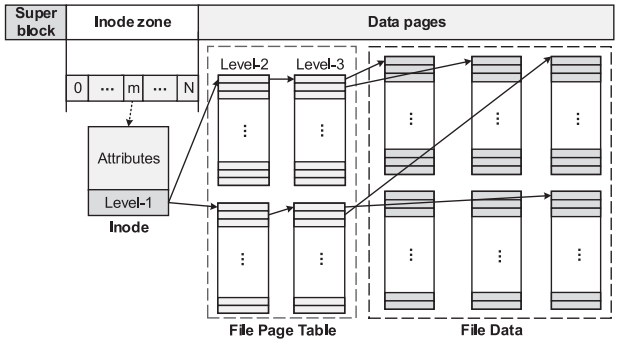


Fig. 6. SIMFS layout.

The user-level application process can now directly access the requested file through its own address space. The time to establish the mapping in the user process space is $O(1)$ for any given file size. When the file is mapped into the user space, file data read operations are executed without copies and page faults. Writes in the existing address space are similar to read. For appending writes, a page fault handler is activated to allocate free pages and update corresponding entries in the process page table.

In summary, the proposed method is an open operation that opens the file in the user space. Compared with the existing memory mapping approach, the propose method has zero overhead in establishing the mapping table for related file data. Therefore, the file system can benefit from the proposed method whatever the size of transferred data.

4 Implementation

In this section, we present implementation details of SIMFS, including the memory layout, the consistency mechanism, the file operations.

The proposed file system, SIMFS, is implemented in Linux on widely used x86_64 architecture. The proposed design framework can be applied on other CPUs such as ARM [26], MIPS [27], PowerPC [28], and SPARC [29]. In the Linux kernel, we reserved a segment of physical memory acting as persistent memory and a chunk of kernel virtual address space for SIMFS.

4.1 File System Layout

Fig. 6 shows the layout of persistent memory provided for SIMFS. The first two memory sections store metadata, namely, superblock and inodes. These two sections have fixed beginning addresses and fixed sizes. Inodes are stored in the inode zone. Each inode has a fixed size and is identified by a unique ID known as *inumber*. There are no insert or delete operations for inodes. Inodes are organized in array. In this case, an inode using the beginning address of the inode zone and the corresponding inumber can be easily located. During system reboot, the metadata pages are mapped to virtual address space through linear mapping to provide data persistency. SIMFS maintains a list of idle inodes in a circular linked list. An idle inode only keeps the inumber of the next idle inode. In order to release the cost of

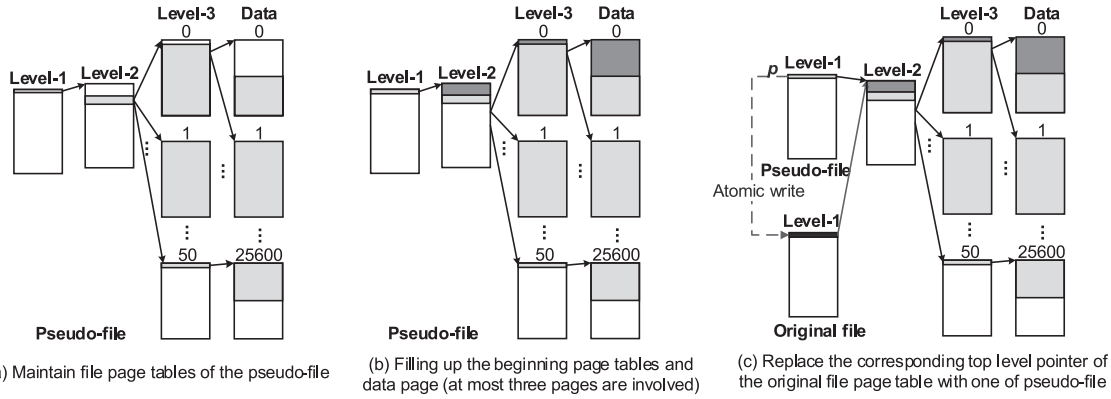


Fig. 7. An example of PFW.

the wear-leveling techniques at the device side [30], [31], free inodes are taken from the head and returned to the tail. In this case, it is expected to alleviate the overhead of the device-side wear-leveling schemes.

As shown in Fig. 6, the memory section *Data pages* stores all the file page tables and file data pages in SIMFS. There are two kinds of files in SIMFS, directory file and regular file. Directory file has its own file page table and a list of directory entries. Each directory entry preserves an inumber and a file or sub-directory name. Each regular file contains a file page table and related file data pages. In this implementation, SIMFS uses a three level file page table for each file, i.e., a pointer in the top level file page table can reference 1 GB file data. If a inode stores 32 pointers to the top level pages in file page table, then a file can store up to 32 GB data. SIMFS manages a list of free memory pages in circular linked list. In order to release the cost of wear-leveling techniques of memory device, free pages are taken from the head and returned to the tail. With this approach, it can help to alleviate the overhead for the device side wear-leveling scheme.

SIMFS uses linear search to locate the directory entries in a directory file. In addition, similar to other file systems, SIMFS also utilizes the Linux VFS' generic directory cache and inode cache.

4.2 Consistency Using Pseudo-File-Write Technique

It is important to guarantee consistency for file systems to survive system crashes or power failures. SIMFS uses a hybrid approach for consistency, switching between journaling [12] and a proposed technique, Pseudo-File-Write.

SIMFS uses journaling for the atomicity and consistency of the file system. SIMFS initiates a new transaction for each atomic file system operation that requires logging, such as *create*, *unlink* and *rename*. In the transaction, SIMFS first records the file data or metadata related to the operation. Then the corresponding file data or metadata are updated. Finally, SIMFS commits the transaction.

For data consistency, we present a technique PFW for the proposed framework. PFW is extended from copy-on-write [13], [14], [15], [32]. However, different from COW, PFW utilizes the contiguous virtual address space of file to write new data and only updates the directly related metadata structure.

Next, we present the details of PFW. A pseudo file has the same structure as a regular file. When a thread is spawned and once this thread initiates a file write operation, the kernel creates a pseudo file for this thread. Note that only one pseudo file is created in the life of a thread. When a thread exits, the corresponding pseudo file, if it exists, will be deleted.

Here we show how a file system uses our PFW technique to conduct "partial-append", i.e., to overwrite the last part of the original file and append some data. Assume the partial-append of k pages begins at offset f of the original file.

1) Given the original file, offset f and the size of new data k, the file page table of the pseudo file is set up correspondingly. The file system allocates free pages for the pseudo file and writes k pages of data to the pseudo file. Because the data are within the contiguous virtual address space, the data transfer function is called only once which is beneficial for less checks and function calls.
2) Before the file system updates the file page table of the original file, if necessary, the file system may need to fill up few beginning pages of the file page table and data page of the pseudo file.
3) Let p be the top-level pointer of the file page table of the pseudo file. The file system replaces the corresponding pointer of the original file page table with p. We can use atomic write to accomplish the replacement of the pointer.

An example is shown in Fig. 7. Assume the size of the original file is 200 MB. We are writing 100 MB data beginning at the offset of 150 MB plus 2 KB. Thus, a file system need to allocate $(100 \times 2^{20})/(4 \times 2^{10}) + 1 = 25{,}601$ pages for storing the new data. A file system using PFW technique first updates the file page table and writes new data to the newly allocated 25,601 free data pages of the pseudo file, as shown in Fig. 7a. Then as Fig. 7b shows, the file system fills up the three beginning pages of the pseudo-file page tables, i.e., level-2, level-3 file page tables, and data page 0. Finally, as Fig. 7c shows, the file system replaces the corresponding pointer in the original level-0 file page table by the single pointer in the pseudo-file level-0 page table.

SIMFS uses the first two steps of PFW to prepare pages for new data in the pseudo file. The third step of PFW and the updates of metadata are accomplished in transaction. When truncating a file, the file size is updated before updating the file page table. When growing a file, the file page table is updated before modifying the file size. For example, if a write crosses the boundary of the regions of multiple

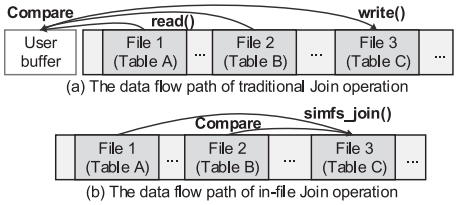


Fig. 8. Two kinds of execution methods.

top-level pointers, SIMFS first updates the new data in the pseudo file. Then SIMFS updates the related top-level file-page-table pointers and modifies the file size in the transaction for metadata updating.

In the concurrent circumstance, the writes using PFW can still correctly update the file because the concurrency of write operations submitted by multiple threads is managed by the file lock in Linux VFS. By using the file lock, a file can only be written by a single thread at any time.

In order to recover the file system to a consistent state and prevent memory leakage, we propose a file-system checker to recover the file system. The file-system checker does as follows: after the system crash or power failure, the file-system checker do two things during recovery. First, it scans the logs of transaction. The uncommitted transactions are fixed and the committed transactions are ignored. For example, the uncommitted *rename* operation will be replayed according to the information stored in the log. Only one directory entry, either the old one or the new one, will point to the inode of the renamed file.

Second, the file-system checker scans the inode zone to rebuild the page allocator and fix the inconsistency of directory files. The residuary pseudo files are deleted, i.e., its inode and pages are reclaimed. For the inumbers in directory files that point to invalid inodes, such as unallocated inode or bad inumbers that larger than the maximum inumber, the corresponding directory entries are removed. If a directory entry for "." does not refer to itself, or a directory entry for ".." itself is the root or it is not refer to its parent directory, then the file-system checker fixes the entry by the correct value. If multiple hard links point to the same directory, the file-system checker makes the first hard link as the real parent directory.

4.3 File System Operations

Now we discuss basic file system operations in SIMFS.

Create operation. The *create* operation is accomplished in the following steps: First, SIMFS obtains a free inode from a list of idle inodes. Second, SIMFS constructs a three-level file page table consisting of three pages for PMD, PTE, and file data page, respectively. Third, a new directory entry is added to the corresponding directory file.

Open operation. The *open* operation performs the following three steps for existing files. First, SIMFS gets a segment of free virtual address space from process address space for the file to be opened. Second, SIMFS maintains the beginning address of the virtual address space in the "inode_info" of the open file. Finally, SIMFS inserts the pointers in the top level file page table of the file into the corresponding entries in the system page table.

Read & write operations. In SIMFS, *read* operations directly read data from the file to user buffer, i.e., no data goes through traditional page cache. Reading multiple pages of data is not stalled by the search of metadata, as the requested data are organized contiguously in the file virtual address space. The SIMFS only have to calculate the beginning virtual address of the requested file position. This address is obtained by adding the beginning address of virtual address space of the file and the offset of the requested position.

In SIMFS, appends write data from user buffer to the original file directly. An append can be done by calling the data transfer function only once, since the data pages are organized in contiguous virtual address space. The in-place updates and partial-appends are accomplished by our proposed PFW technique.

Close operation. Inode keeps a counter for the number of processes that are using the file. In *close* operation, the value of the counter is examined. If it is zero, i.e., no process using the file, SIMFS releases the virtual address space of the file by removing the related entries in the kernel page table. The released virtual address space is retained for later use. If the value of the counter is greater than zero, SIMFS decreases the counter by one.

Delete operation. SIMFS supports hard link through a counter in inode. If the counter is greater than zero, it means that the actual file has multiple hard links and it cannot be deleted. In this case, SIMFS merely removes the directory entry that does not link to the file any more and decrements the counter. Otherwise, SIMFS frees the inode, the file page table, and data pages of the file.

5 In-File Executions

In this section, we introduce a method called "in-file execution", for efficient executions of data processing based on the proposed framework. Using the "in-file execution" method, an application can process file data within the file system directly, i.e., the related file data will not go through any buffers such as user buffer. The method is expected to offer high performance for applications.

Traditionally, a process copies its data from file system to the user buffer, then processes the data, and maybe stores the results back to file system. An example of join operation using traditional method is shown in Fig. 8a. The join operation reads data from File 1 and File 2 to the user buffer via system call *read()*, compares the attributes of the two tables, and stores the join results back to File 3 by system call *write()*. The join operation induces lots of system calls and data copies between files and user buffer, and thus costs large overhead.

Since each opened file in SIMFS has its own virtual address space, we can directly read or write the files in SIMFS by their virtual addresses. With in-file execution method, we can avoid such data copies and avoid related costly *read/write* system calls. For example, an in-file join is shown in Fig. 8b, where the comparisons of attributes and the data copies are directly operated among files, i.e., no data copies between user buffer and files. The experimental results show that the join operation using "in-file execution" obtains more than 60 percent performance improvement on average over that using traditional method.

We can apply in-file execution method in an existing application following three steps: First, we design an in-file

execution interface for the application; second, we open related files to establish their virtual address spaces; third, we process file data within the file system directly using their virtual addresses. Note that the in-file-execution method is a new feature different from the memory mapped I/O (mmap). In addition, the standard mmap function is also implemented in SIMFS.

Memory mapping interface (i.e., *mmap()*) manipulates file data through virtual address space. Nevertheless, in-file execution different from *mmap* in three aspects:

1) *mmap* should transfer data between page cache and files. Differently, in-file execution can directly manipulate file data without copy file data to user buffer or page cache;
2) *mmap* should build additional mapping tables for related file data. Differently, it is not necessary for in-file execution;
3) To store new data, operations on *mmap* should commit many system calls and repeatedly go through long software routines such as VFS checking. For the in-file execution, storing new data is simple. Operations are directly executed within the file system.

In conclusion, the in-file execution method can achieve higher performance than *mmap*. In-file execution induces less data transfer and less software routines. The experimental results show that in-file join of SIMFS improves 28.6 percent performance over mmap-based join on average.

The performance gain one can obtain from such a mechanism depends on both the application workload and the performance difference between DRAM and the persistent memory. In this work, the in-file execution is proposed to provide a method for data processing within the framework of the file virtual address space.

6 Experiments

Since June 2013, we began the design and implementation of an in-memory file system called SIMFS in Linux based on the framework of file virtual address. The current version of SIMFS supports all the standard functions of EXT4, including *create()/delete()*, *open()/close()*, *read()/write()*, *symlink()*, *link()*, *rename()*, *readpage()*, and *mmap()*.

To show the benefit in using the memory mapping hardware facilities, we also implemented a special version of SIMFS, called SIMFS-Soft. Instead of using memory mapping hardware facilities, SIMFS-Soft finds the physical location of each data page by searching the file page table of the corresponding file by software routines.

In this section, we compare SIMFS with typical existing file systems on performance. First, we present the micro-operation results obtained by standard benchmark Flexible I/O (FIO) [16] and Filebench [17]. Then, we show the macro-operation results including application workloads [33] and join operation. The experimental results show that SIMFS outperforms other file systems, including EXT4 [11] on Ramdisk, PRAMFS [10], RAMFS [8], and PMFS [12]. In the experiments, the delayed allocation of EXT4 is enabled. The experiments show that SIMFS almost reaches the memory bus bandwidth for large I/O size.

The experiments are conducted on a system equipped with 32 GB DRAM and a 3.30 GHz Intel i3-2120 dual-core processor. We configure 8 GB DRAM acting as volatile memory for volatile data structures of the system, and use the rest of 24 GB acting as persistent memory for in-memory file systems, including PMFS, PRAMFS, SIMFS, and EXT4 on Ramdisk. Note that there are no special reasons for the proportions. The basic idea is to provide as much memories as possible for in-memory file systems while reserving enough memory for the system.

6.1 Micro-Operation Performance

In this section, we first show the performance of sequential and random read/write requests on various file systems accessed by a single thread. Then, the performance of file systems accessed by multiple threads are presented. Finally, we show the Filebench results of metadata operations.

6.1.1 Throughput with Single Thread

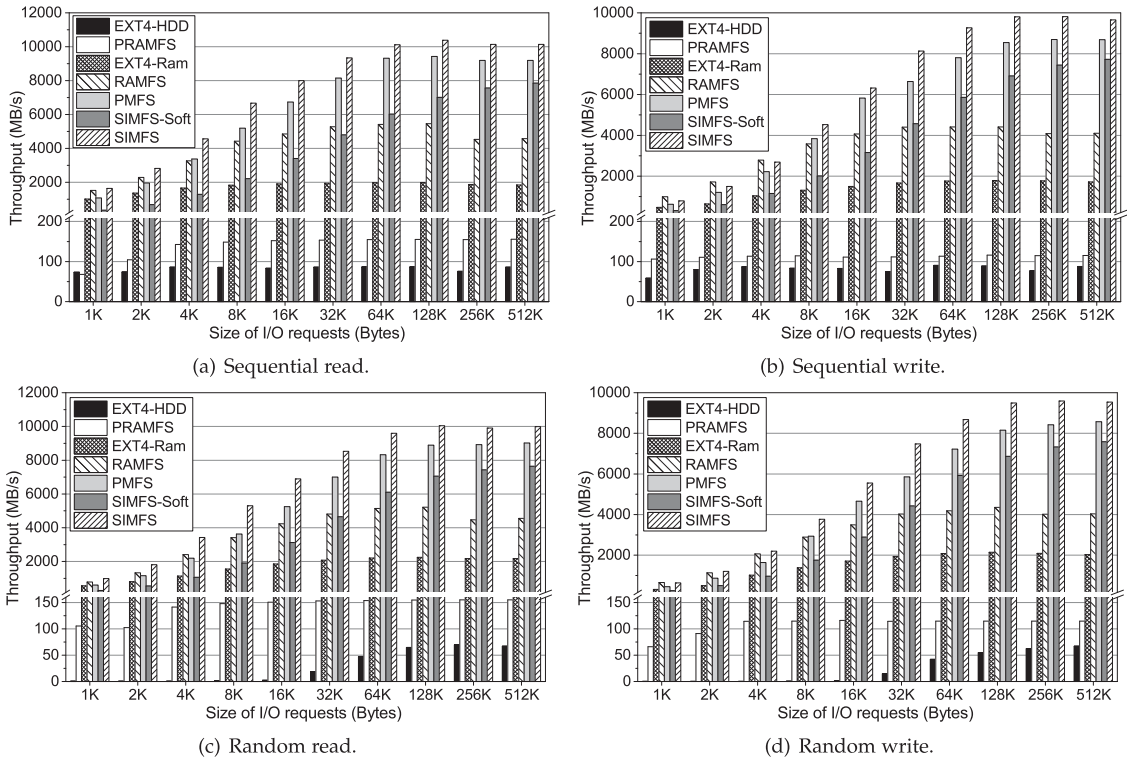
Now, we compare the performance of file systems accessed by single thread. The experimental results are shown in Fig. 9. The "Size of I/O requests" in the figures and tables means the size of data requested in each I/O request issued by the benchmark. Note that from the view of FIO, it does not matter if the target file system is on hard disk or memory. Even 1 KB/2 KB requests are submitted to in-memory file systems, the sizes of I/O requests are still 1 KB/2 KB.

As shown in Fig. 9, for sequential read and sequential write, SIMFS is 2.54 and 1.97 times faster than that of SIMFS-Soft on average, respectively. For random read and random write, SIMFS is 2.16 and 1.83 times faster than that of SIMFS-Soft on average, respectively. The only difference between SIMFS and SIMFS-Soft is that SIMFS-Soft searches file page table by software routines. It is obvious that SIMFS gains high benefit from the memory mapping hardware facilities.

The experimental results in Fig. 9 show that SIMFS constantly outperforms EXT4 on Ramdisk (denoted by EXT4-Ramdisk) on all sizes of I/O requests. The throughput of SIMFS reaches five times faster than that of EXT4-Ramdisk on several cases. The throughputs of sequential read/write and random read/write of SIMFS on average are 4 times and 3.4 times faster than those of EXT-Ramdisk, respectively. The low performance of EXT4 on Ramdisk is because that EXT4 on Ramdisk still goes through all the software layers of the traditional I/O stack, even though file data and metadata are located in memory. The results indicate that I/O stack brings significant performance penalty.

RAMFS [8] uses "page cache" in Linux to store its file data. RAMFS is usually considered efficient as those Linux system functions are well developed. In addition to storing data in page cache, RAMFS keeps metadata in directory entry cache and inode cache temporarily. It neither provides consistency mechanisms nor supports backups of data. Therefore, RAMFS is a "temporary" in-memory file system which cannot survive system reboots. Nevertheless, SIMFS still outperforms RAMFS for most of the cases. In the best cases, the throughputs of SIMFS for reads and writes reach 2.2 times and 2.4 times faster than those of RAMFS, respectively. The average throughput of SIMFS for reads and writes is 68.6 percent higher than that of RAMFS.

We also compare SIMFS with PMFS [12], the state-of-art in-memory file system. The metadata structure of PMFS is



(a) Sequential read. (b) Sequential write.

(c) Random read. (d) Random write.

Fig. 9. Comparing throughput for in-memory file systems using single thread with FIO.

quite different from that of SIMFS. PMFS organizes the file data pages by B-tree. Data accesses to a file in PMFS have large overhead on the software search of the B-tree. On the contrary, a file in SIMFS organizes its data pages by contiguous file virtual address space embedded into process virtual address space. Thus, the data accesses to such a file have little overhead on the search of metadata. The throughput of SIMFS is 1.18 to 1.65 times higher than that of PMFS. Comparing with PMFS, the throughputs of SIMFS are 32.7 and 25.1 percent higher on average for random reads and random writes, respectively.

The performance improvement of SIMFS benefits from the fundamental architecture and file access method. In SIMFS, the logical view of a file is just acted as a segment of contiguous virtual address space. Thus, searching for any file location can be easily done by MMU hardware. SIMFS calls data transfer function only once to accomplish an I/O request. On the contrary, as the virtual addresses of data pages for a file in RAMFS or PMFS may be discontinuous, these file systems have to seek the metadata of the file for finding the corresponding data pages. Note that this search on metadata is done in software. Therefore, RAMFS or PMFS cannot fully take advantages of the MMU hardware as SIMFS does. Moreover, RAMFS and PMFS have to call data transfer function multiple times to access multiple data pages in an I/O request and induce more costs than SIMFS.

Furthermore, we find that for each type of operation, the throughputs of all file systems are increased when the sizes of I/O requests grows larger, as shown in Fig. 9. It is because that when the sizes of I/O requests gets larger, the number of I/O requests issued by benchmarks is decreased and then the overhead of I/O system calls is reduced.

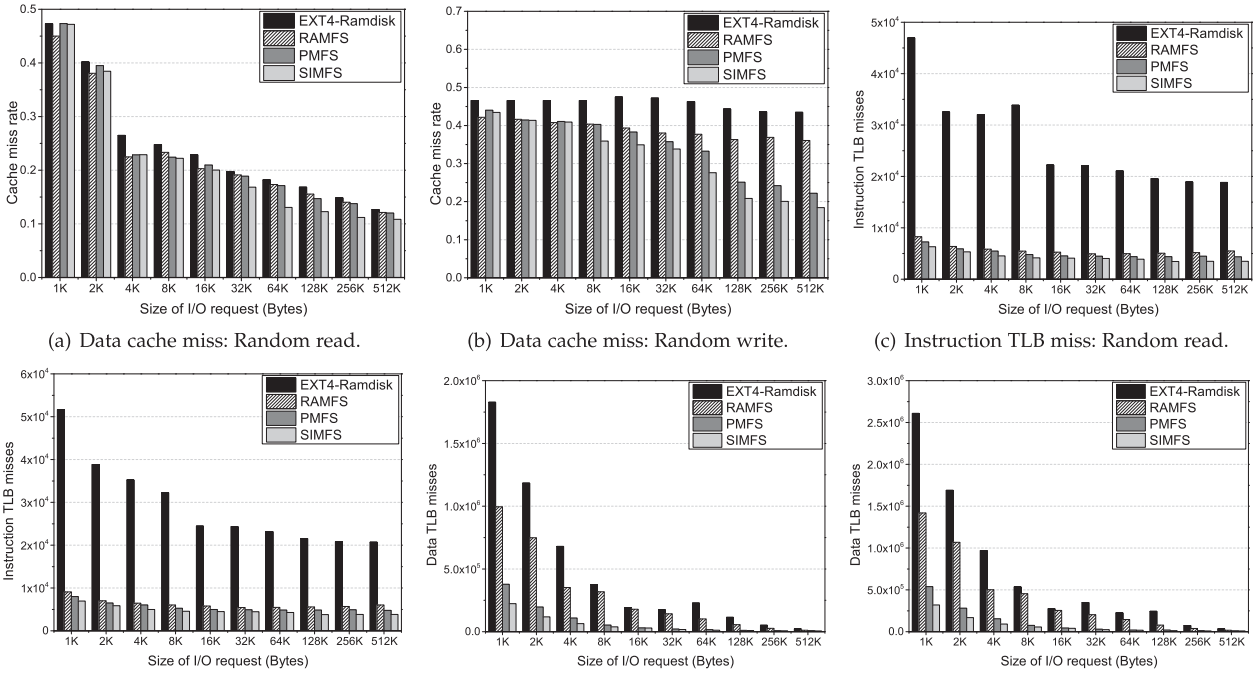
We also compare the throughputs of SIMFS with those of PRAMFS [10], a RAM-base file system, and EXT4 on hard disk (denoted by EXT4-HDD). The experimental results are shown in Fig. 9. Compared to the in-memory file system PRAMFS, SIMFS achieves 60 and 80 times improvement over PRAMFS for sequential/random reads and writes, respectively. It is because that PRAMFS organizes file data pages by a 2-dimensional linked list and accesses them by software procedures. Besides, the throughputs of random read and random write for SIMFS reaches 5,200 and 8,100 times faster than those of EXT4-HDD, respectively.

In order to know the maximum throughput an in-memory file system may achieve, we measure the memory bus bandwidth of the system via the widely used benchmark STREAM [34]. As shown in Table 1, the memory bus bandwidth of different operations are presented. As Fig. 9 shows, SIMFS is able to achieve up to 89 percent of the memory bus bandwidth of the system.

To better understand the detail performance related to processor, the Linux tool *perf* [35] is used to catch the cache misses, instruction TLB misses, and data TLB misses of EXT4 on Ramdisk, RAMFS, PMFS, and SIMFS. The experimental results are presented in Fig. 10. The results show that SIMFS achieves 34, 19, and 9 percent cache miss reductions for EXT-Ramdisk, RAMFS, and PMFS, respectively. SIMFS achieves

TABLE 1
The Memory Bus Bandwidth Measured by STREAM [34]

	Memory bus bandwidth (MB/s)
Copy	11,231.2
Scale	10,288.0
Add	11,572.4
Triad	11,879.9
Average	11,242.9



(a) Data cache miss: Random read. (b) Data cache miss: Random write. (c) Instruction TLB miss: Random read.

(d) Instruction TLB miss: Random write. (e) Data TLB miss: Random read. (f) Data TLB miss: Random write.

Fig. 10. Comparing the cache misses, instruction TLB misses, and data TLB misses of four in-memory file systems.

80, 75, and 24 percent data TLB miss reductions for EXT4-Ramdisk, RAMFS, and PMFS, respectively. SIMFS achieves 60, 25, and 14 percent instruction TLB miss reductions for EXT4-Ramdisk, RAMFS, and PMFS, respectively.

As shown in Figs. 10a and 10b, when the size of I/O requests is larger than 8 KB, SIMFS has lower cache miss rate than other file systems for SIMFS can access multiple pages without searching software routines. As shown in Figs. 10c, 10d, 10e, and 10f, EXT4-Ramdisk has much more TLB misses and data TLB misses. The file accesses in EXT4 go through several software layers designed for disk-based file systems. Similarly, SIMFS has less TLB misses than other file systems. It is because that the path of file accesses in SIMFS is pithier than that of other three file systems.

6.1.2 Throughput with Multiple Threads

In this section, we show the performance of in-memory file systems accessed by multiple threads. The experiments are conducted by FIO benchmarks, where threads may read/write the same file. The sizes of the files are set to 12 GB in the experiments.

Fig. 11 shows the experimental results of EXT4-Ramdisk (denoted by EXT4-R), RAMFS, PMFS, and SIMFS with multiple threads, where the vertical axes show the aggregated throughputs of threads. The numbers of threads in the tests are 2 and 4. As shown in Fig. 11, SIMFS outperforms RAMFS and EXT4-Ram for most cases. The throughputs of SIMFS is 1.3 to 4.8 times faster than those of EXT4-Ram with two running threads. The average throughput improvement of SIMFS over RAMFS and PMFS is 87.9 and 19.5 percent with two running threads, respectively. In the case of four running threads, the average throughput of SIMFS is 2.76, 1.8, and 1.19 times faster than that of EXT4-Ram, RAMFS, and PMFS, respectively. Therefore, SIMFS still shows the advantages of file virtual address space in the tests of multiple threads.

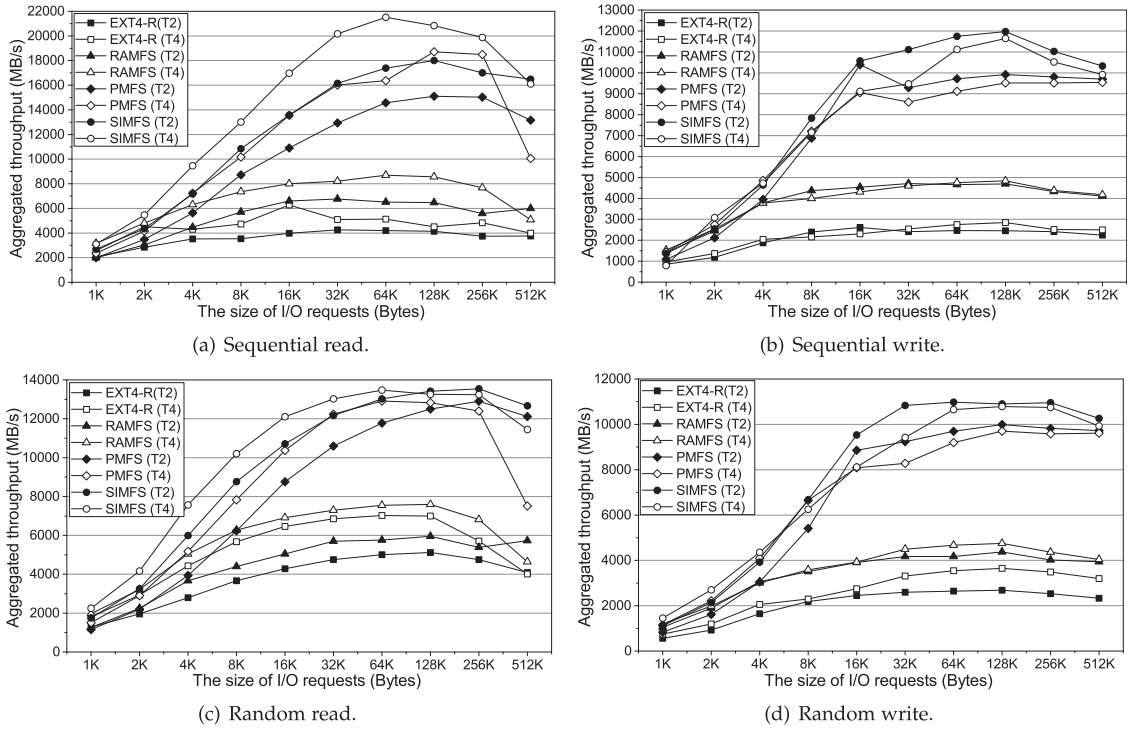
As shown in Fig. 11, the performance of sequential read/write is better than that of random read/write for in-memory file systems. For example, in the cases of four threads reading SIMFS, the maximum aggregate throughput of sequential read is 21,519.6 MB/s whereas that of random read is 13,477.5 MB/s, as shown in Figs. 11a and 11b. It is because that for sequential operations, the data accessed by one thread might be used by another thread while the data are still in the data cache of CPU.

6.1.3 Performance of Metadata Operations

Now we measure the performance of metadata operations of EXT4 on Ramdisk (denoted by EXT4-Ram), PMFS, and SIMFS using Filebench. Fig. 12a shows the relative performance of three in-memory file systems. SIMFS shows the same or improved performance compared with PMFS over all six workloads.

The workload *openfiles* creates a set of empty files. *createfiles* creates a directory tree and fills it with a set of files with specified sizes. The size of files are set via a gamma distribution with a mean file size. *delete* first creates a set of files, where the size of each file is set via a gamma distribution with a median size. The previous three workloads are operated by 16 threads. *statfile* creates a set of files, and loops through them using 20 threads to read the file attributes of each file. *makedirs* creates a directory tree with a number of leaf directories. *removedirs* creates a set of empty leaf directories and then removes them all. We use default settings for all the workloads except that the total size of all files is set as 12 GB to ensure it does not fit in the page cache. In the workloads, the mean number of files in the same directory is set to 100 except for *listdir*, which is 2.

The experimental results of metadata operations are shown in Fig. 12a. We find that SIMFS outperforms PMFS on all cases. Compared with EXT4 on Ramdisk, SIMFS



(a) Sequential read. (b) Sequential write.

(c) Random read. (d) Random write.

Fig. 11. Comparing throughput for in-memory file systems using multiple threads with FIO.

also obtains much better performance on *createfiles*, and achieves about the same performance on other cases. For the case of *openfiles*, SIMFS is 1.07 times faster than PMFS. It is because that SIMFS directly locates the address of inode of a file with an index, i.e., the inumber, whereas PMFS walks through the B-tree to seek an inode. Although SIMFS has to insert the file page table into the process page table in *open* operation, SIMFS is just slightly slower (about 1.5 percent) than EXT4-Ram on *openfiles*. Therefore, the overhead of inserting file page table into the process page table is inconspicuous to the overall performance of SIMFS. For the case of *createfiles*, SIMFS is 2.94 and 1.08 times faster than EXT4-Ram and PMFS, respectively. The performance improvement of SIMFS benefits from less copies and higher throughput of data accesses.

In conclusion, the experimental results show that SIMFS is at least as efficient as other file systems for metadata operations.

6.2 Macro-Operation Performance

In this section, we compare different persistent in-memory file systems by evaluating the performance of macro-operations, including application workloads produced by Filebench and the join operation.

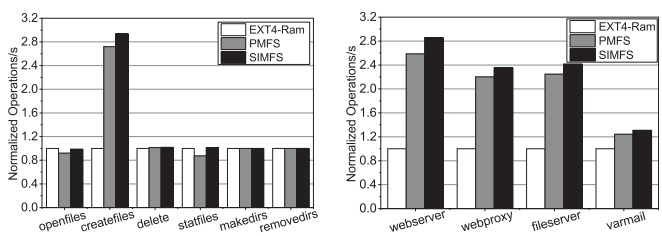
6.2.1 Application Workloads Performance

We evaluate the performance of four multi-threaded application workloads produced by Filebench. We use the default settings for the four workloads, except that we scale the total size of all files as 12 GB at least. The results are shown in Fig. 12b. SIMFS shows performance improvement over EXT4-Ram and PMFS on all workloads.

webserver is a workload that emulates the Web server. 100 threads are used to open/read/close multiple files in a directory tree, along with file appends to simulate the web log. *webproxy* emulates a web proxy server, with a mix of create/write/close, open/read/close and delete operations of multiple files in a directory tree, combining a file append to simulate the proxy log. 100 threads are used simultaneously. *fileserver* emulates a server that hosts home directories of multiple users, which is represented by 50 threads. Each thread performs a sequence of create, read, write, delete, append and stat operations on their own home directories. *varmail* emulates a mail server with three kinds of behaviors including reading mails (open, read, and close), composing (open/create, append and close) and deleting mails. 16 threads are used to mimic such behaviors.

As shown in Fig. 12b, SIMFS outperforms both EXT4-Ram and PMFS for all the workloads. The performance of SIMFS is 1.3 times (for *varmail*) to 2.9 times (for *webserver*) better than those of EXT4-Ram. Besides, the performance improvement of SIMFS over PMFS ranges from 5.2 percent (for *varmail*) to 10.3 percent (for *webserver*). The performance improvements of SIMFS obtained from the simplified metadata structures and the use of file virtual address space and MMU, as shown in Section 6.1.

The variation of performance improvement over different workloads is caused by the characteristics of the workload. For example, the performance improvement of



(a) Metadata operations. (b) Application workloads.

Fig. 12. Experimental results of Filebench workloads.

TABLE 2
Comparing the Execution Time (*ms*) of Nested-Loop Join Operations

Scale	EXT4-D	Imprv.	EXT4-R-MMAP	Imprv.	TMPFS	Imprv.	PMFS	Imprv.	SIMFS	Imprv.	*simfs_join*
10^8	31,295.4	98.1%	878.5	31.0%	1,983.9	69.5%	1,784.8	66.1%	1,734.2	65.1%	605.9
10^9	577,116	98.9%	8,382.7	27.0%	19,695.6	68.9%	17,651.3	65.4%	16,989.3	64.0%	6,115.8
10^{10}	–	–	83,315.3	28.0%	193,934.9	69.1%	178,544.3	66.4%	173,003.6	65.3%	60,012.9
10^{11}	–	–	835,721.5	28.2%	1,963,146.3	69.4%	1,778,038.5	66.2%	1,722,559.4	65.2%	600,216.7
Avg.		**98.5%**		**28.6%**		**69.2%**		**66.0%**		**64.9%**	

webserver is higher than that of *varmail*. It is because that *webserver* commits more data accesses and less metadata operations than *varmail* does, and SIMFS gains significant performance improvement from data accesses.

In conclusion, the performance of SIMFS surpasses those in-memory file systems. The experiments confirm that SIMFS is stable for multiple threads applications. It supports legacy applications. The experiments of workloads show that SIMFS can be used for complex applications.

6.2.2 In-File Join Performance

Next, we evaluate the in-file execution method using nested loop join operation. The standard nested loop join operation joins two tables (files) by using two nested loops. We define the "scale" of an experimental instance as the total iteration times of the two nested loops, which has a key impact on the execution time of the join operation. Assume the number of entries in these two files is N_1 and N_2, respectively. The value of scale equals $N_1 \times N_2$.

We have evaluated the standard nested loop join operation on several file systems, including EXT4 on hard disk (denoted by EXT4-D), EXT4 on Ramdisk using *mmap()* (denoted by EXT4-R-MMAP), PMFS [12], SIMFS, and TMPFS [9]. The DAX feature of EXT4 on Ramdisk is enabled in the experiments to bypass the page cache. We have also implemented in-file nested loop join in SIMFS, i.e., *simfs_join*. With all these implementations of nested loop join, we conduct experiments on various scales, including 10^8, 10^9, 10^{10}, and 10^{11}. The size of each entry in each file is 64 bytes.

The experimental results are shown in Table 2. The time unit of execution time is millisecond. The "Imprv." columns show the performance improvements of *simfs_join* over the corresponding file systems. The nested loop join on EXT4-D for scale 10^{10} and 10^{11} cannot finish in a reasonable time and we will not present it here. The performance improvements of *simfs_join* over EXT4-D for scale 10^8 and 10^9 are 98.1 and 98.9 percent on average, respectively. Even though EXT4 on Ramdisk is applied and *mmap* is used to process the join operation, *simfs_join* still outperforms EXT4-R-MMAP by 28.6 percent on average. It is because EXT4-R-MMAP has to establish mapping tables for files, do more data transfers, and submit more system calls.

For the cases of in-memory file systems, the performance of *simfs_join* achieves 69.2, 66.0 and 64.9 percent improvement on average over TMPFS, PMFS, and SIMFS, respectively. It is because that *simfs_join* cuts down more than half data copies in standard nested loop join by directly access file data within the file system, and also reduces the number of system calls in the execution of join operations.

In summary, the in-file execution of commonly used operations of applications is effective and a good choice to significantly improve performance.

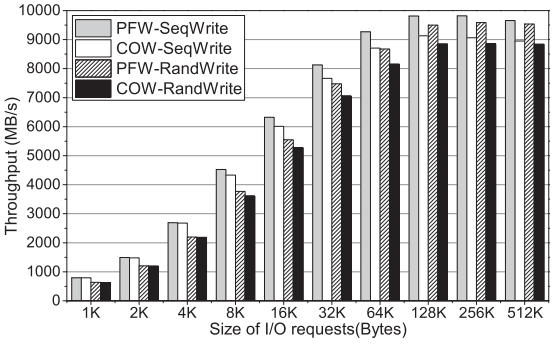


Fig. 13. Comparing the performance of PFW and COW.

6.3 Pseudo-File Write

In this section, we compare the performance of SIMFS with PFW and COW (see Section 4.2), respectively. The experiment results with the benchmark FIO [16] are presented in Fig. 13. The PFW-SeqWrite and COW-SeqWrite represent the throughputs of sequential writes for SIMFS with PFW and COW, respectively. The other two legends have the similar meaning for random write.

The performance of PFW achieves 8 percent performance improvement over that of COW. As shown in Fig. 13, the performance improvement of PFW over COW increases with the increased size of I/O requests. It is because that the more data involved in an operation are, the more PFW can benefit from the contiguous virtual address space. When the sizes of I/O request is smaller than the size of a single page, the performance of COW is close to that of PFW as their activities are nearly the same under this condition.

6.4 User-Space File Access

In this section, we evaluate the performance of the user-space file accesses of SIMFS. In terms of user-space accesses, writing to exiting blocks is similar to reading from a file when a file is mapped to the user space. Thus, the performance of reading is presented in the experiments.

The latencies of four types of read are shown through experiments: (1) the standard read of EXT4 on Ramdisk (denoted by EXT4-Ram), (2) mmap-based read of EXT4 on Ramdisk (denoted by EXT4-Ram-USpace), the standard read of SIMFS, and the user-space read of SIMFS (denoted by SIMFSUSpace). We set the size of each I/O request to be 4 KB and the total size of file accesses ranges from 4 KB to 4 GB in each experiments. The experimental results are presented in Fig. 14. As shown in the results, user-space file

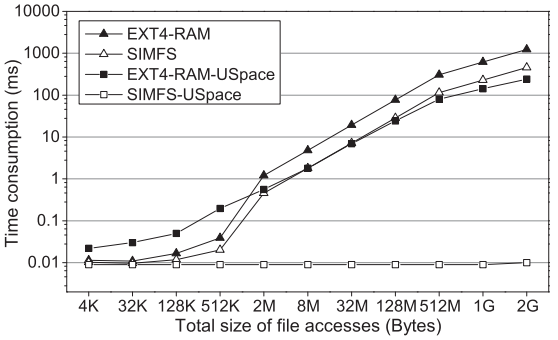


Fig. 14. The time consumption of four types of read over different amount of data.

access can largely reduce the time overhead when read operations are performed. The user-space file access of SIMFS shows about three times higher performance than that of EXT4. Moreover, when the total size of file accesses is less than 512 KB, the user-space file access of EXT4 on Ramdisk using mmap consumed more time than the standard read of EXT4 on Ramdisk. In these cases, mapping a file to the user space consumes more time than the data transfer overhead. Therefore, small file accesses cannot benefit from the mmap method of EXT4. On the contrary, SIMFS does not has such a problem since the SIMFS framework presented in the paper can map a file to the user address space in O(1) time.

7 Conclusion

In this paper, we presented a new design of persistent in-memory file system based on the novel framework of file virtual address, where each open file has its own continuous virtual address space. In this new framework, the virtual address space of an opened file is established by embedding its file virtual address space into the process virtual address space. The file data can be accessed with high throughput using the contiguous virtual address space of the opened file. The data structure and organization of metadata are redesigned for data persistency and highly-efficient file operations in the memory. Our framework can be generally applied to most of existing modern CPUs. We have implemented a fully functional persistent in-memory file system, SIMFS in Linux based on the new framework. The evaluation results show that SIMFS constantly outperforms existing in-memory file systems. The throughput of SIMFS even approaches the memory bus bandwidth on best cases.

Acknowledgments

This work is partially supported by National 863 Program 2015AA015304, NSFC 61472052, Chongqing Research Program cstc2014yykfB40007. The corresponding author is Dr. Qingfeng Zhuge.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, p. 10.

[2] J. Deanand S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM J. Res. Dev.*, vol. 52, pp. 439–447, 2008.

[4] M. Jung, J. Shalf, and M. Kandemir, "Design of a large-scale storage-class RRAM system," in *Proc. Int. Conf. Supercomput.*, 2013, pp. 103–114.

[5] C. Xu, P.-Y. Chen, D. Niu, Y. Zheng, S. Yu, and Y. Xie, "Architecting 3d vertical resistive memory for next-generation storage systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2014, pp. 55–62.

[6] J. Akerman, "Toward a universal memory," *Science*, vol. 308, no. 5721, pp. 508–510, 2005.

[7] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, no. 4, pp. 465–479, 2008.

[8] M. K. McKusick, M. J. Karels, and K. Bostic, "A pageable memory based filesystem," in *Proc. USENIX Summer*, 1990, pp. 137–144.

[9] P. Snyder, "tmpfs: A virtual memory file system," in *Proc. Autumn EUUG Conf.*, 1990, pp. 241–248.

[10] S. Longerbeam, M. Locke, and K. Morgan. (2013). Protected ram filesystem [Online]. Available: http://pramfs.sourceforge.net/

[11] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, 2008, pp. 21–33.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. 9th ACM Euro. Conf. Comput. Syst.*, 2014, pp. 1–15.

[13] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," in *Proc. USENIX Winter Tech. Conf.*, 1994, p. 19.

[14] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: A ZFS case study," in *Proc. 8th USENIX Conf. File Storage Technol.*, 2010, p. 3.

[15] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proc. 22nd ACM Symp. Oper. Syst. Principles*, 2009, pp. 133–146.

[16] (2014). Fio: flexible I/O tester [Online]. Available: http://freecode.com/projects/fio

[17] (2014). Filebench [Online]. Available: http://filebench.sourceforge.net

[18] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 1–11.

[19] H. Kim, J. Ahn, S. Ryu, J. Choi, and H. Han, "In-memory file system for non-volatile memory," in *Proc. Res. Adaptive Convergent Syst.*, 2013, pp. 479–484.

[20] S. Oikawa, "Integrating memory management with a file system on a non-volatile main memory system," in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, 2013, pp. 1589–1594.

[21] J. Gait, "Phoenix: A safe in-memory file system," *Commun. ACM*, vol. 33, no. 1, pp. 81–86, 1990.

[22] E. Lee, S. H. Yoo, and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1349–1360, May 2015.

[23] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE Comput.*, vol. 35, no. 2, pp. 59–67, 2002.

[24] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in *Proc. 4th ACM Int. Conf. Arch. Support Program. Lang. Oper. Syst.*, 1991, pp. 75–84.

[25] D. Stancevic, "Zero copy i: User-mode perspective," *Linux J.*, vol. 2003, no. 105, p. 3, 2003.

[26] (2015). Arm system memory management unit architecture specification [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0062d.a/index.html

[27] (2014). Architecture for programmers volume i-a: Introduction to the mips64 architecture [Online]. Available: https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MD00083-2B-MIPS64INT-AFP-06.01.pdf

[28] (2005). Book iii: Powerpc operating environment architecture [Online]. Available: http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html

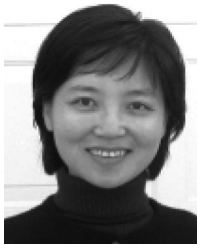
[29] (2011). Oracle sparc architecture 2011 [Online]. Available: http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf

[30] S. Cho and H. Lee, "Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance," in *Proc. 42nd IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 347–357.

[31] M. Zhao, L. Jiang, Y. Zhang, and C. J. Xue, "SLC-enabled wear leveling for MLC PCM considering process variation," in *Proc. 51st ACM/IEEE Des. Autom. Conf.*, 2014, pp. 36:1–36:6.
[32] O. Rodeh, J. Bacik, and C. Mason, "BRTFs: The linux b-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, pp. 317–318, 2013.
[33] P. Sehgal, V. Tarasov, and E. Zadok, "Evaluating performance and energy in file system server workloads," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2010, p. 19.
[34] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Comput. Archit. Lett.*, pp. 19–25, 1995. Available: http://www.cs.virginia.edu/stream
[35] (2015). perf: Linux profiling with performance counters [Online]. Available: https://perf.wiki.kernel.org

Edwin H.-M. Sha received the PhD degree from the Department of Computer Science, Princeton University in 1992. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at the University of Notre Dame. Since 2000, he has been a tenured full professor at the University of Texas at Dallas. Since 2012, he served as the dean of the College of Computer Science at Chongqing University, China. He has published more than 360 research papers in refereed conferences and journals. His work has been cited more than 2800 times. He received Teaching Award, Microsoft Trustworthy Computing Curriculum Award, US National Science Foundation (NSF) CAREER Award, and NSFC Overseas Distinguished Young Scholar Award, Chang-Jiang honorary chair professorship, and China Thousand-Talent Program. He is a senior member of the IEEE.

Xianzhang Chen received the BS and MS degrees in computer science and engineering from the Southeast University, Nanjing, China, in June 2010 and June 2013, respectively. He is currently working toward the PhD degree in the College of Computer Science at Chongqing University, China. His advisor is Dr. Edwin Sha. His research interests include nonvolatile memory, optimization algorithms, and operating system.



Qingfeng Zhuge received the BS and MS degrees in electronics engineering from Fudan University, Shanghai, China and the PhD degree in computer science from the University of Texas at Dallas in 2003. She is currently a professor at Chongqing University, China. She received the best PhD Dissertation Award in 2003. She has published more than 90 research articles in premier journals and conferences. Her research interests include parallel architectures, embedded systems, and optimization algorithms. She is a member of the IEEE.

Liang Shi received the BS degrees in computer science from the Xi'an University of Post & Telecommunication, Xi'an, Shanxi, China, in July, 2008, and the PhD degree from the University of Science and Technology of China, Hefei, China, in June, 2013. He is now a full-temporary teacher in the College of Computer Science at the Chongqing University. His research interests include flash memory, embedded systems, and emerging nonvolatile memory technology. He is a member of the IEEE.

Weiwen Jiang received the BE degree from the Department of Computer Science, Nanjing Agricultural University, Nanjing, China, in 2008. He is currently working toward the PhD degree in the Department of Computer Science at Chongqing University, Chongqing, China. His current research interests include self-timed system optimization, embedded systems, and nonvolatile memory.