The Design and Implementation of an Efficient User-Space In-memory File System

Edwin H.-M. Sha, Yang Jia, Xianzhang Chen, Qingfeng Zhuge, Weiwen Jiang, and Jiejie Qin
College of Computer Science, Chongqing University, Chongqing, China.
Email: {edwinsha, jiayangxx1, xzchen109, qfzhuge, jiang.wwen, qinjie6839}@gmail.com

***Abstract*—The file accesses of existing in-memory file systems have additional costs for traversing the software stacks of the kernel, such as VFS. To avoid such costs, the existing file systems generally enable user-space file accesses using the memory-mapped file (*mmap*) techniques. The *mmap* approaches, however, are add-ons of the file systems in the kernel level that have large overhead for mapping the files into the user space. In this paper, we propose the design of a genuine user-space in-memory file system. A file system, User-space in-Memory File System (UMFS), is designed and implemented. The software routines of the file operations in UMFS are re-designed to enable user-space file accesses and be compatible with the POSIX interfaces. A file can be mapped into the user space in constant time regardless of the file size. The file data of UMFS are accessed in the user space with consistency. UMFS achieves high performance utilizing the contiguous virtual address space of the user process and the address translation hardware. Extensive experiments are conducted. The experimental results show that UMFS surpasses any of the existing file systems. To the authors' knowledge, UMFS is the first genuine user-space file system in the literature.**

I. Introduction

The demand for high performance becomes even stronger in data processing. New in-memory file systems [1]–[7] are proposed to boost the performance of data processing as file systems are the fundamental infrastructure for storing data. The in-memory file systems achieve high throughput by exploiting the characteristics of the emerging Storage Class Memory (SCM) [8], such as byte-addressability and near-DRAM speed.

Nevertheless, the file systems mostly access the file data in the kernel space with large overhead for traversing the software stacks of the kernel, e.g, the Virtual File System (VFS) layer. Several approaches are proposed to provide user-space file accesses [4], [9]–[11], such as memory-mapped file (*mmap*) [4] and FUSE [11]. The existing user-space file access approaches, however, are add-ons of the file systems in the kernel-level that have large overhead in supporting the user-space file accesses. For example, the FUSE-based user-space file systems employ a driver in the kernel to handle the requests of the user.

In this paper, we propose the design of a genuine user-space in-memory file system for achieving high performance file accesses. A file system, User-space in-Memory File System (UMFS), is implemented in Linux based on the proposed design. The file system operations of UMFS are re-designed to support file accesses without traversing the software stacks of the kernel. A POSIX-compatible library *libumfs* is designed and implemented for supporting the re-designed software routines of the file operations in UMFS. Thus, UMFS can cohabit with other file systems and programs that use the standard POSIX interfaces. The file data of UMFS can be exposed to the user space in $O(1)$ time that is independent from the file size. The file accesses in UMFS take advantages of the contiguous virtual address space of the user process and the address translation hardware (e.g., MMU).

This work is partially supported by National 863 Program 2015AA015304, Chongqing High-Tech Research Program cstc2014yykfB40007, NSFC 61472052.

The data consistency is ensured by recording the updates in a new structure, "U-Log". U-Log is located in the SCM but mapped to the virtual address space of the user process when the file is opened. In the cases of system failure, UMFS can be recovered by scanning through the U-Logs of the files during the next system reboot.

In addition to the persistent data structures, each process maintains a temporary data structure *U-info* in DRAM for the opened files in the user space, as shown in Figure 4. The U-info is an array for storing the beginning virtual address spaces of the files of UMFS. The U-info is initially created when a process opens its first file and destroyed when the process is terminated. The size of the array equals the maximum number of the file descriptors that a process can use. The kernel generally sets the number to 1024. Each item of the array is one-to-one corresponding to a file descriptor. The items of U-info are initialized as nil. When a file of UMFS is opened by a process, the related element in the U-info of the process is updated with the beginning virtual address of the file. The number of the file descriptors is used as the index of the element. When a file of UMFS is closed by a process, the value of the related element in the U-info of the process is reset to nil. Thus, UMFS can easily verifies if the requested file belongs to UMFS by directly checking if the related element of the file in the U-info is nil. The overhead of such a judgement is negligible for a file system operation.

We use the standard tool, Flexible I/O (FIO) [12], to measure the performance of UMFS. UMFS is compared with typical in-memory file systems, including SIMFS [5], PMFS [4], and EXT4 [13] on Ramdisk. Experimental results show that UMFS achieves significant performance improvement over the existing file systems for sequential and random read/write. For example, the throughput of UMFS achieves 4 times, 4 times, and 7 times that of SIMFS, PMFS, and EXT4 on Ramdisk for sequential write by one thread. To the authors' knowledge, UMFS achieves the best known result for file systems in literature. The main contributions of this paper are as follows:

- We analyze and measure the costs of the software stacks of the kernel and the existing user-space file accesses.
- We propose the new design of a genuine user-space file system considering interface compatibility, data consistency, and the overhead of user-space file accesses.
- We implement a functional user-space file system, UMF-S, in Linux based on the proposed design of genuine user-space file system.
- Experiments are conducted with standard benchmarks to compare UMFS with the existing in-memory file systems, including SIMFS, PMFS, and EXT4 on Ramdisk. The experimental results show that UMFS surpasses the state-of-the-art file systems.

II. Motivation and design principles

A. Motivation

In-memory file systems. With the development of SCM [8], [14], [15], several in-memory file systems [1], [3]–[5], [16] are proposed to achieve high performance file accesses taking advantages of SCMs. The file accesses of these in-memory file systems, however, bypass the traditional I/O stacks for block-based devices [17]. The data are directly transferred between files and the user buffer without going through the page cache of the system. Thus, the performance of in-memory file systems are higher than that of traditional block-based file systems [13] (even deployed on Ramdisk) for less overhead on software stacks and less data copies.

Nevertheless, the in-memory file systems in Linux systems still work with the Virtual File System (VFS) layer of the kernel. We measured the cost of the software stacks of SIMF-S [2], [5] in the kernel. The experimental results are shown in Fig. 1. The size of file accesses (i.e., the x-axis) is the size of each file access committed by the application. "Data transfer" represents the time for copying data from the file to the user buffer. "Software stacks" represents the time for traversing the software stacks. As shown in Fig. 1, the overhead of the software stacks in the kernel ranges from 10% to 50% of the total file access time. Especially, suppose the file accesses avoid the software stacks, the performance of the file accesses can be improved two times when the size is less than 8KB.

File access in user space. Beyond the kernel-space file accesses, previous works also have proposed approaches for user-space file accesses. One approach is to provide an interface for bridging the user-space file system code with the kernel routines. FUSE [10], [11] is the most widely adopted interface. A FUSE-based file system works like a standalone application. The interface of FUSE itself adopts two components to achieve user-space file accesses. On the user side, a multi-threaded daemon maintains the main logic of a file system and interacts with the in-kernel driver of FUSE. On the kernel side, the in-kernel driver handles the requests from the user applications. This in-kernel driver, like other file systems in kernel space, accomplishes its requests with VFS.

Another widely adopted technique is memory-mapped file, i.e., *mmap*. The traditional block device based file systems, such as EXT4 [13], enable user-space file accesses by *mmap*:

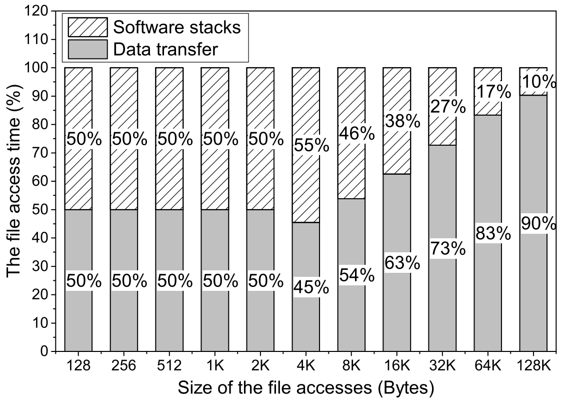


Fig. 1. The overhead of the software stacks in the kernel.

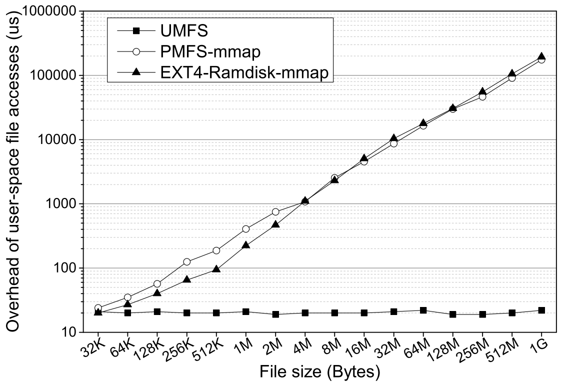


Fig. 2. The overhead of *mmap*.

First, the file system copies the requested file data into the page cache [18] of the kernel; then, the corresponding page cache is mapped into the virtual address space of the user process. The *mmap* of in-memory file systems, such as PMFS [4] and DAX file system, is different. They directly map the file data into the user-space. Thus, *mmap* may cause file data inconsistency since the updates of the mapped file may be incomplete and unrecoverable in a system failure. Furthermore, *mmap* approaches have large overhead in establishing the mapping of file data and limited size of mapping.

Compared with FUSE, *mmap* truly provides user-space file accesses when the file is mapped into the user space. The main overhead of *mmap* is the time for establishing the mapping of the file. Therefore, we compare the time for mapping a file by *mmap* with that by the proposed UMFS. The experimental results are shown in Fig. 2. UMFS can be 8700 times faster than *mmap*, as shown in Fig. 2. The time for *mmap* in PMFS and EXT4 on Ramdisk are dependent on the size of the mapped file. It is because that *mmap* should build a new memory mapping table for the mapped file data apart from the metadata structure of the file.

In conclusion, the existing user-space file accesses are all add-ons of the file systems in the kernel space. They all have drawbacks in providing user-space file accesses.

B. Design Principles

In this paper, we aim to design a genuine user-space file system with high performance following three principles.

The file should be directly exposed to the user space. We try to expose the file to the user space since the software stacks in the kernel cause large overhead for data accesses. Unlike the traditional *mmap*, we would like to map the file into the user space directly without going through the page cache . The file data can be accessed with zero copy using the virtual address space of the user process.

The file mapping should provide stable performance. The most unstable issue in the existing *mmap* is the time for mapping. In the design of UMFS, we establish the mapping of a file by updating only several entries of the process page table when opening the file. Thus, the time for mapping any file should be a constant regardless of the file size as the sizes of the files can be various.

The data consistency and concurrency of files should be considered. Data consistency and concurrency are the natures of most functional file systems. In the design of UMFS, we consider the file data in the user space as a critical section. The data consistency and concurrency of the critical section is guaranteed from both the user side and the kernel side.

III. Design

In this section, we present the design of the proposed User-space in-Memory File System (UMFS).

Compatible with the POSIX interfaces. On one hand, the standard file system operations using POSIX interfaces, such as *read()*, should go through the VFS layer in the kernel space. In a genuine user-space file system, however, we expect to perform such operations in the user space. Therefore, the software routines of file system operations need re-design. On the other hand, the standard POSIX interfaces are widely used in most existing programs. Hence, we do not modify the interfaces of the file system operations.

In this paper, we design a dedicated library, called *libumfs*, for the file system operations of UMFS. The *libumfs* implemented the basic operations of UMFS with the standard interfaces. The software routines of the operations, however, are redirected to that of UMFS. For example, to open a file A, *libumfs* checks whether the file A belongs to UMFS using the parameter "file path" of *open()*. If the file belongs to UMFS, the *libumfs* opens the file with the dedicated routines of UMFS other than the traditional VFS layer. Similarly, the software routines of the other basic file system operations, such as *read()* and *write()*, are all re-designed in *libumfs* to support the user-space file accesses of UMFS.

***O(1)* time mapping.** In order to expose the physical space of a file to the user space in constant time, we use the framework of file virtual address space [2] to manage the files in UMFS. In the framework, each file has an independent virtual address space. The physical space of a file can be mapped to the user space by embedding the related file virtual address space into the virtual address space of the user process, as shown in Fig. 3. This operation can be accomplished in constant time.

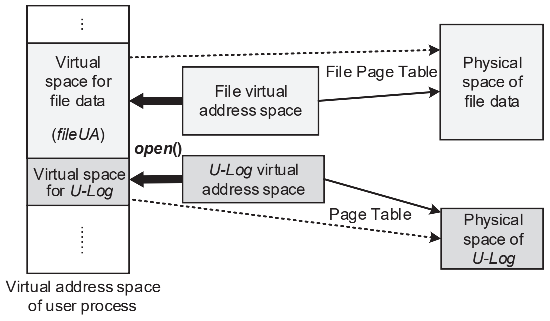

Virtual space for file data
(fileUA)
Virtual space for U-Log
Virtual address space of user process
File virtual address space
U-Log virtual address space
open()
File Page Table
Page Table
Physical space of file data
Physical space of U-Log

Fig. 3. A concept of file data and metadata virtual address space.

In practical terms, the file virtual address space is represented by the "file page table", which is in the same form with the page table of system. The data pages of a file are organized by the corresponding file page table. Thus, the physical space of a file can be exposed to the user space in three steps. First, UMFS applies a contiguous free virtual address space for the file, say *fileUA*, from the user process. Then, UMFS parses the page table of the user process to find out the entry that related to *fileUA*. This can be complete in a short constant time. Finally, UMFS updates the entry found in step 2) with the entry in the top-level file page table related to the file. Obviously, the three steps can be performed in constant time independent from the file size.

Consistency and concurrency. Different from the traditional *mmap* approaches, we still consider the consistency of file operations. In order to persistently record the updates of a file in the user space, we propose to set a backup space, *U-Log*, located in SCM for each file in UMFS. A U-Log is composed of two parts: the attributes of the related file and the information of the current write operation, including the backups and the flags for consistency. Similar to the file data, the U-Log of a file is mapped into the process virtual address space when the file is opened, as shown in Fig. 3.

The writes that only update the existing file data is performed as follows: First, UMFS copies the affected data and metadata of the file to the related U-Log; then, UMFS updates the original file with the new data; finally, UMFS updates the attributes of the file in the U-Log. For the append writes, UMFS first updates the information in the U-Log. Then, UMFS allocates new pages for the file in the kernel and updates the related file page table. Finally, the file is directly updated in the user space. In case of system crash or power failure, the file system can be recovered to a consistent state taking advantages of the U-Log of each file in UMFS.

The files in UMFS can be concurrently accessed by multiple processes. For the file systems in kernel space, the concurrency of files are protected by the VFS layer. The concurrency in UMFS, however, is protected in the user space, just like *mmap*. Once a file is exposed to the user space, the file becomes a critical section. UMFS can protect the mapped files via the existing synchronization mechanisms, such as semaphores.

Surviving the life cycle of processes. A process may be blocked or terminated in its life cycle. The files of UMFS opened by a process may be affected by the state transition of the process as the file page tables of the files are embedded in the page table of the process. Specifically, there are two cases.

First, when a process is blocked, the kernel may reclaim the physical pages occupied by the process and migrate the related data to the swap areas. The data of the files in UMFS may be swapped and lost since the files use the process virtual address space reserved for memory mapping. Therefore, we inform the kernel that the virtual address space occupied by UMFS should be retained when the process is blocked. This is achieved by setting the flags in the structure of *virtual memory area*.

Second, when a process is terminated, the page table and the physical pages of the process are reclaimed by the system. The file page table of an opened file will also be reclaimed as the file page table is embedded in the page table of the process. Thus, the corresponding file data will get lost. To deal with this issue, when a file is closed by a process, the virtual address space occupied by the file is released and the corresponding entry in the page table of the process is cleared.

IV. Implementation

A. Layout of UMFS

The layout of UMFS is shown in Fig 4. UMFS consists of three sections, including the superblock, inode zone, and data zone. The superblock and inode zone have fixed beginning addresses. The size of each inode is fixed. The data zone contains physical pages that constitute file page table, file data, and U-Log. Generally, the inode of a file system is composed of two parts, i.e., the attributes of the file and the index of the file data. Different from the inode of the kernel-level file systems, the inode of UMFS has a unique component, i.e., U-log. The physical pages of the U-log are organized in the same structure as the file page table. Similar to the file data, the U-logs are stored in the data zone.

A U-log contains two types of information related to the file. The first information is the attributes of the related file, such as file descriptor. The second information is the status of the updating operation in runtime, such as the offset, the flags indicating the progress of the current operation, and the backup of the affected file data. U-logs can be used to recover the file system after a system crash since U-logs can survive the system reboot. When a file is opened, the process accesses the file data in the user space and modify the attributes of the file in the related U-log. When a file is closed, the attributes in the inode are updated with the corresponding attributes stored in the U-log. Furthermore, each process maintains a temporary data structure, called *U-Info*, for the opened files in the user space. *U-Info* maintains the exclusive information of the files related to the process, such as the beginning virtual addresses of the files in the user space.

B. File System Operations

Now we discuss basic file system operations of UMFS.

Create operation. In *create()*, UMFS first allocates an idle inode for the file. Then, UMFS constructs a three-level file page table for the file, where the last two levels each has one page. The beginning physical address of the top-level file page table is reserved in the inode. Subsequently, UMFS builds a U-log for the file. Finally, a new directory entry is added to the corresponding directory file.

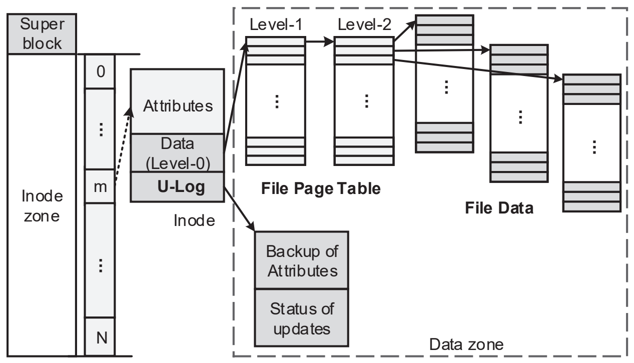


Fig. 4. The layout of UMFS.

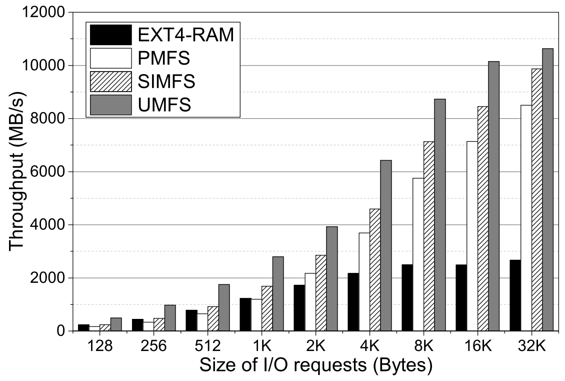
Open operation. First, UMFS applies a segment of virtual address space from the user process for the file data and the related U-log. Then, UMFS inserts the pointers to the top-level file page table and the U-log into the corresponding entries of the process page table, respectively. Finally, UMFS updates the attribute in the U-log with the metadata in the inode. Now, the file data can be efficiently accessed via the contiguous virtual addresses of the user process and the hardware MMU.

Read operation. The read operation in UMFS can be performed in two ways. The first method for read-only files passes the beginning virtual address of the file to the user. The user can directly access the file using the virtual address without copy. The second method for writable files utilizes the existing read interfaces. UMFS copies the requested data from the file to the user buffer using the virtual address of the file in user space. The physical locations of data are efficiently located by the hardware MMU using the contiguous virtual address space of the file.

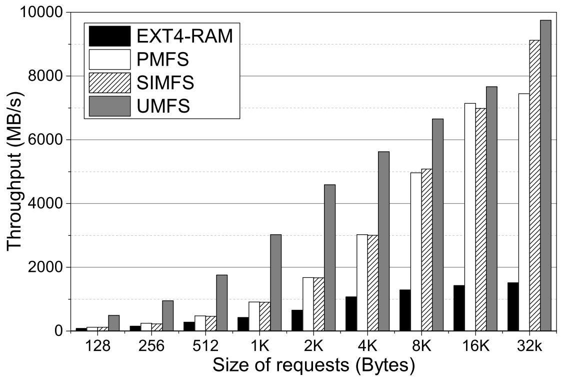
Write operation. As mentioned in Section III, the in-place writes are totally performed in user space and the appends are partly performed in kernel mode for allocating new physical pages. Similar to reads, data are transferred without searching the physical locations by software routines. UMFS utilizes the U-log of the file to ensure data consistency. Furthermore, considering the protection of the file data, the user is not allowed to directly write the files in UMFS using the related virtual addresses of files.

Close operation. When a process closes a file, UMFS releases the process virtual address space occupied by the file gives it back to the vma of the process. It is noteworthy that the file page table must be detached from the page table of the process, as discussed in Section III. The attributes cached in the U-log will be written back to the metadata of the file.

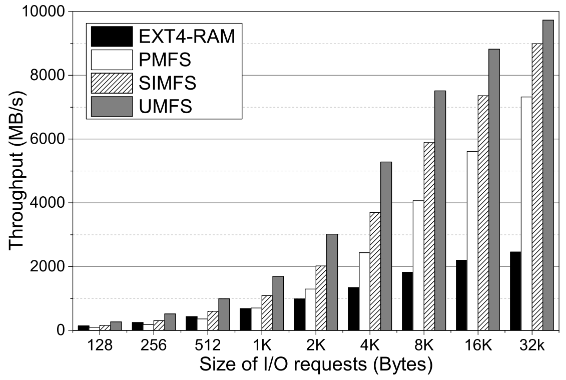
Delete operation. UMFS uses a counter in the inode for the links. When the counter is larger than zero, *delete()* operation reduces the counter by one and remove the entry linked to the file. If the counter is zero, UMFS reclaims the inode, U-log, file page table, and data pages of the file.



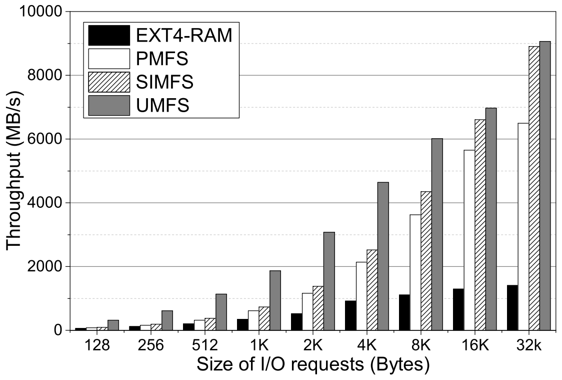
(a) Sequential read.



(b) Sequential write.



(c) Random read.



(d) Random write.

Fig. 5. Comparing throughput for in-memory file systems using single thread with FIO.

V. Experiments

In this section, we compare the performance of UMFS with the existing in-memory file systems, including EXT4 [13] on Ramdisk, PMFS [4], and SIMFS [5]. UMFS is implemented in Linux 3.11.8. The experiments are conducted on a PC system equipped with the 3.3GHz Intel Core i3-3220 and 32GB DRAM using the standard tool Flexible I/O (FIO) [12]. We preserve 24GB DRAM for the file systems and leave the rest 8GB to the system. The experimental results show that UMFS outperforms any of the other file systems.

A. Throughput with Single Thread

Now, we compare UMFS with SIMFS, PMFS, and EXT4 on Ramdisk by single thread. The experimental results are shown in Fig. 5. The “Size of requests” in the figures means the size of data requested in each file access operation issued by FIO.

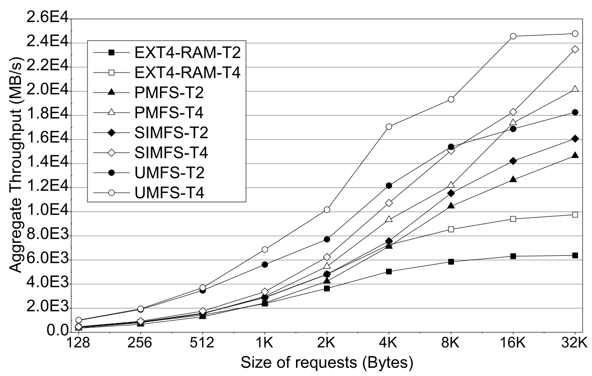
The throughput of UMFS exceeds SIMFS, PMFS, and EXT4-Ram in all cases, as shown in Fig. 5. For sequential reads, the throughput of UMFS is 2.9 times, 2.1 times, and 1.6 times that of EXT4-Ram, PMFS, and SIMFS on average, respectively. For sequential writes, the throughput of UMFS even achieves 7.1 times, 4.1 times, and 4.1 times that of EXT4-Ram, PMFS, and SIMFS, respectively. Because UMFS avoids the overhead for traversing the software stacks in the kernel.

Compared with EXT4-Ram [13], the performance of UMFS is 1.9-4.1 times and 5.2-7.1 times higher for reads and writes, respectively. EXT4 still goes through the VFS layer and the software routines designed for block devices, even though EXT4 is deployed in the memory via Ramdisk. Furthermore, the file accesses of EXT4 search the physical locations of data pages by traversing the metadata structures of the file. Different from EXT4, UMFS not only avoids the software layers of the kernel, but also efficiently locates the data pages by the corresponding contiguous virtual address space of the user process and the hardware MMU.

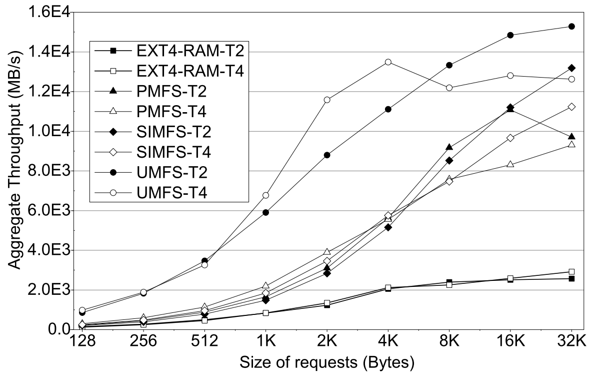
The performance of UMFS is 1.3-3.0 times and 1.1-4.1 times higher than PMFS [4] for reads and writes, respectively. PMFS organizes the data pages of a file by a B-tree. The file accesses of PMFS search the physical locations of data pages by traversing the B-tree of the file. Hence, UMFS surpasses PMFS by avoiding the overhead for traversing the VFS layer and searching the metadata structures of the file.

The performance of UMFS is 1.1-2.1 times and 1.01-4.2 times higher than SIMFS [5] for reads and writes, respectively. Similar to UMFS, SIMFS accesses file data via the related virtual address space of the kernel and the hardware MMU. Hence, UMFS outperforms SIMFS by avoiding the overhead for traversing the layers in the kernel.

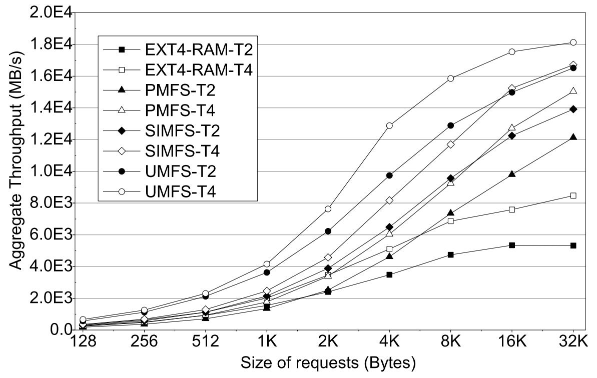
The performance improvement of UMFS over SIMFS and PMFS is decreased when the size of requests is increased, as shown in Fig. 5. The cost for traversing the layers in the kernel is a constant. The cost is obscured by the cost of data transfer when the size of requests gets larger. On the contrary, the



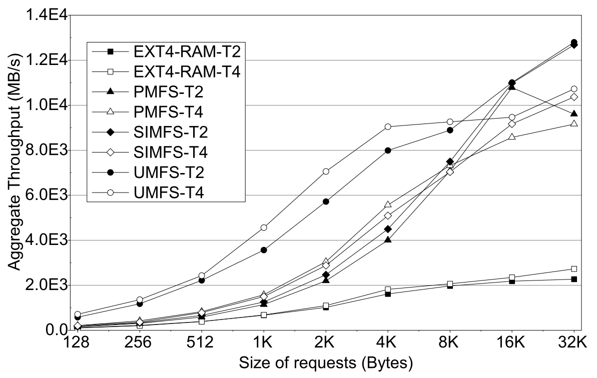
(a) Sequential read.



(b) Sequential write.



(c) Random read.



(d) Random write.

Fig. 6. Comparing throughput for in-memory file systems using multiple thread with FIO.

performance improvement of UMFS over EXT4 on Ramdisk is increased when the size of requests is increased. It is because that the cost for traversing the traditional I/O stacks gets even larger when the size of requests is increased.

B. Throughput with Multiple Thread

Next, we compare UMFS with EXT4-Ram, PMFS, and SIMFS by 2 threads and 4 threads, respectively. Fig. 6 shows the aggregated throughput of multiple threads. UMFS exceeds the other three file systems in all cases. UMFS can be 7.1 times faster than EXT4 on Ram with two running threads. The throughput of UMFS is 4.1 times, 2.4 times, and 2.1 times that of EXT4-Ram, PMFS, and SIMFS on average by two running threads, respectively. In the cases of four threads, UMFS is 4.2 times, 2.2 times, and 2.1 times faster than EXT4-Ram, PMFS, and SIMFS on average, respectively. UMFS still shows the advantages of using the virtual address space of user process by multiple threads.

VI. Conclusion

In this paper, we presented the design of a genuine user space file system. Based on our design, the user-space file system not only offers user-space file accesses with data consistency, but also exposes file to the user space with cost. We implemented a functional user-space file system, UMFS, in Linux that is compatible with the POSIX interfaces and can survive the life cycle of processes. The experimental results show that the performance of UMFS excels that of the state-of-the-art file systems.

References

[1] 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. SOSP*, 2009, pp. 133–146.

[2] E. H.-M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "Designing an efficient persistent in-memory file system," in *Proc. IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2015.

[3] X. Wu, S. Qiu, and A. L. Narasimha Reddy, "Scmfs: A file system for storage class memory and its extensions," *ACM Transactions on Storage*, vol. 9, no. 3, pp. 1 – 11, 2013.

[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. EuroSys*, 2014.

[5] E. H.-M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "A new design of in-memory file system based on file virtual address framework," *IEEE Transactions on Computers*, 2016.

[6] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. FAST*, 2016, pp. 323–338.

[7] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proc. EuroSys*, 2016.

[8] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, pp. 439–447, 2008.

[9] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. EuroSys*, 2014.

[10] M. Szeredi, "Filesystem in userspace," http://fuse.sourceforge.net/., 2005.

[11] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra incognita: on the practicality of user-space file systems," in *Proc. HotStorage*, 2015.

[12] "Fio: flexible i/o tester," http://freecode.com/projects/fio.

[13] 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.

[14] X. Chen, E. H.-M. Sha, Q. Zhuge, C. J. Xue, W. Jiang, and Y. Wang, "Efficient data placement for improving data access performance on domain-wall memory," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2016.

[15] S. Gu, Q. Zhuge, J. Yi, J. Hu, and E. H.-M. Sha, "Optimizing task and data assignment on multi-core systems with multi-port spms," *IEEE Transactions on Parallel & Distributed Systems*, vol. 26, no. 9, pp. 2549–2560, 2015.

[16] X. Chen, E. H.-M. Sha, Q. Zhuge, W. Jiang, J. Chen, and J. Chen, "A unified framework for designing high performance in-memory and hybrid memory file systems," *Journal of Systems Architecture*, 2016.

[17] Y. Son, N. Y. Song, H. Han, H. Eom, and H. Y. Yeom, "Design and evaluation of a user-level file system for fast storage devices," *Cluster Computing*, vol. 18, no. 3, pp. 1075–1086, 2015.

[18] P. Dai, Q. Zhuge, X. Chen, W. Jiang, and E. H.-M. Sha, "Effective file data-block placement for different types of page cache on hybrid main memory architectures," *Design Automation for Embedded Systems*, vol. 17, no. 3-4, pp. 485–506, 2014.