SCMFS PERFORMANCE ENHANCEMENT AND IMPLEMENTATION ON MOBILE PLATFORM

A Thesis

by

QIAN CAO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2012

Major Subject: Computer Engineering

SCMFS Performance Enhancement and Implementation on Mobile Platform

SCMFS PERFORMANCE ENHANCEMENT AND IMPLEMENTATION ON MOBILE PLATFORM

A Thesis

by

QIAN CAO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	A.L.Narasimha Reddy
Committee Members,	Paul V. Gratz
	James Caverlee
Head of Department,	Costas N. Georghiades

August 2012

Major Subject: Computer Engineering

ABSTRACT

SCMFS Performance Enhancement

and Implementation on Mobile Platform. (August 2012)

Qian Cao, B.S., Huazhong University of Science and Technology;

M.S., Huazhong University of Science and Technology

Chair of Advisory Committee: Dr. A.L.Narasimha Reddy

This thesis presents a method for enhancing performance of Storage Class Memory File System (SCMFS) and an implementation of SCMFS on Android platform. It focuses on analyzing performance influencing factors of memory file systems and the differences in implementation of SCMFS on Android and Linux kernels.

SCMFS allocates memory pages as file blocks and employs virtual memory addresses as file block addresses. SCMFS utilizes processor's memory management unit and TLB (Translation Lookaside Buffer) during file accesses. TLB is an expensive resource and has a limited number of entries to cache virtual to physical address translations. TLB miss results in expensive page walks through memory page table. Thus TLB misses play an important role in determining SCMFS performance. In this thesis, SCMFS is designed to support both 4KB and 2MB page sizes in order to reduce TLB misses and to avoid significant internal fragmentation. By comparing SCMFS with YAFFS2 and EXT4 using popular benchmarks, both advantages and disadvantages of

SCMFS huge-page version and small-page version are revealed. In the second part of this thesis, an implementation of SCMFS on Android platform is presented. At the time of working on this research project, Android kernel was not merged into Linux kernel yet. Two main changes of SCMFS kernel code: memory zoning and inode functions, are made to be compatible with Android kernel. AndroSH, a file system benchmark for SCMFS on Android, is developed based on shell script. Evaluations are made from three perspectives to compare SCMFS with YAFFS2 and EXT4: I/O throughput, user data access latency, and application execution latency. SCMFS shows a performance advantage because of its small instruction footprint and its pre-allocation mechanism. However, the singly linked list used by SCMFS to store subdirectories is less efficient than HTree index used by EXT4. The future work can improve lookup efficiency of SCMFS.

DEDICATION

This thesis is dedicated to my parents, whose love and support helped me through life's challenges, motivated me to embrace opportunities and strive for the best that life has to offer.

ACKNOWLEDGEMENTS

Dr. Reddy is an ideal research supervisor. I am grateful for his insightful guidance and patient encouragement throughout the research project and the writing of this thesis. I would also like to thank Dr. Xiaojian Wu whose work on SCMFS file system stimulated my research interest in file systems and enabled the problems explored in this thesis. My committee members Dr. Gratz and Dr. Caverlee took their precious time to review my thesis and gave me valuable feedback. Several research students in the ECE department gave me useful suggestions and helped me discuss ideas before my final defense. I appreciate their time and help sincerely. Last but not least, I acknowledge the consistent support from my family members who always encourage me to be strong and optimistic about life and future.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

1. INTRODUCTION

This section introduces the motivation of this research project, and also presents a main problem enabled in a previous research work.

1.1 Motivation

Phase Change Memory (PCM) is a typical Storage Class Memory which is non-volatile, byte-addressable, and bit-flippable. Its write life endurance is 3 orders of magnitude of NAND flash, and its read/write accesses are 2~7 times faster than NAND flash. Several firms are working on PCM and PCM is expected to be in wide production in a few years.

Android is one of the most popular mobile operating systems nowadays. Many organizations or individuals developed file systems for Android platform. However, there is no available Android file system designed for PCM characteristics. PCM is suited to mobile platforms because of its low power consumption and a suitable file system will enable its adoption.

In a previous work from our research group, a Storage Class Memory File System (SCMFS) [1] for PCM was developed on Linux Operating System. SCMFS was shown to provide higher performance in many ways than other file systems, e.g.

This thesis follows the style of *IEEE Transactions on Computers*.

RAMFS, TMPFS, EXT2, and EXT3. Therefore, this motivates the implementation of SCMFS on Android Operating System to make a high performance file system ready for next generation Android mobile products based on PCM.

1.2 PCM vs NAND

PCM is very different from NAND flash although they are both non-volatile. PCM works by changing the state of an alloy, chalcogenide glass. To write data, heat is generated by the passing electrical current which switches the alloy between its crystalline and its amorphous state. To read data, a smaller current is applied to determine which state the alloy is in. The state of the alloy indicates if a bit 0 or 1 is stored in the cell.

Compared to NAND flash, PCM has several significant advantages. First of all, it inherently has longer write life cycle than NAND on account of its average endurance of 100,000,000 write cycles as compared to 100,000 of NAND. PCM is much more efficient than NAND at accessing smaller chunks of data because it is byte-addressable but NAND is accessed per chunk. PCM can be switched quickly and can be in-place flipped, which avoids the laborious block erase-and-write cycle required by NAND [2].

1.3 Existing Android File Systems

To use PCM as either main memory or an internal backup device, both YAFFS2 and EXT4 can support it, but none of them considers PCM's characteristics into design.

YAFFS2 is a commonly used mobile file system for internal flash storage. It is designed specifically for NAND flash device to prolong the device life cycle and also enhance performance. Because NAND flash is block-erasable, and every block has several chunks (flash pages), rewriting a chunk causes erasing a whole block. The log-structure of YAFFS2 keeps writes always going to sequentially higher-number chunks to avoid rewriting to previous chunks. This mechanism can protect chunks of NAND flash from being rewritten too often, but when it comes to PCM, it becomes unnecessary and causes overhead because PCM can be updated in place at a smaller granularity.

To resolve the performance overhead caused by garbage collecting unused old chunks, a Short-Op Cache [3] can improve performance. The main purpose of the internal cache is to reduce NAND access for badly behaved applications which perform many small reads and writes. It is only involved for operations which are not chunk aligned. It collaborates with Linux-like kernel VFS (virtual file system) page cache. VFS page cache serves read operations and the Short-Op Cache addresses non-chunk aligned writes. YAFFS2 has to rely on a caching mechanism, but this mechanism causes overhead to PCM as the fact that PCM is a memory device, writing to cache and then writing to memory disk doubles work.

EXT4 is considered a good replacement of YAFFS2 because of its several features of performance enhancement and also because it provides better multithreading support.

To use PCM as a persistent storage device, one way is to create a RAMDISK from PCM, and then mount it with EXT4. However, storage devices are assumed by

EXT4 as I/O-bus attached block devices therefore EXT4 accesses the storage devices through generic block layer and block I/O operations are emulated. When it comes to PCM which is directly attached to memory bus and can be accessed through memory operations, generic block layer and emulation become unnecessary. In the traditional storage hierarchy, since hard disk access has a much higher latency than a memory access, the additional overhead can be ignored. On the contrary, these extra cost become substantially impactful when the storage device is attached directly to the memory bus and can be accessed at memory speed, thus they should be avoided.

1.4 SCMFS

The idea of SCMFS came from the characteristics of PCM. Many discussions were made before implementation of this file system. Memory resident file systems such as RAMFS and TMPFS can also be used with PCM, but they don't support persistent data.

SCMFS is designed specifically for PCM characteristics. Since PCM can be attached directly to the memory bus, SCMFS accesses PCM through memory management layer instead of generic block layer [1]. As Figure 1 shows, it is built on virtual memory space and utilizes memory management unit (MMU) to map the file system addresses to physical addresses on PCM.

Traditional file systems, such as EXT3/EXT2, use indirect block mapping once file is read from or written to disk blocks, because files are on disk blocks instead of memory and blocks of a file are usually non-contiguous, as shown in Figure 2.

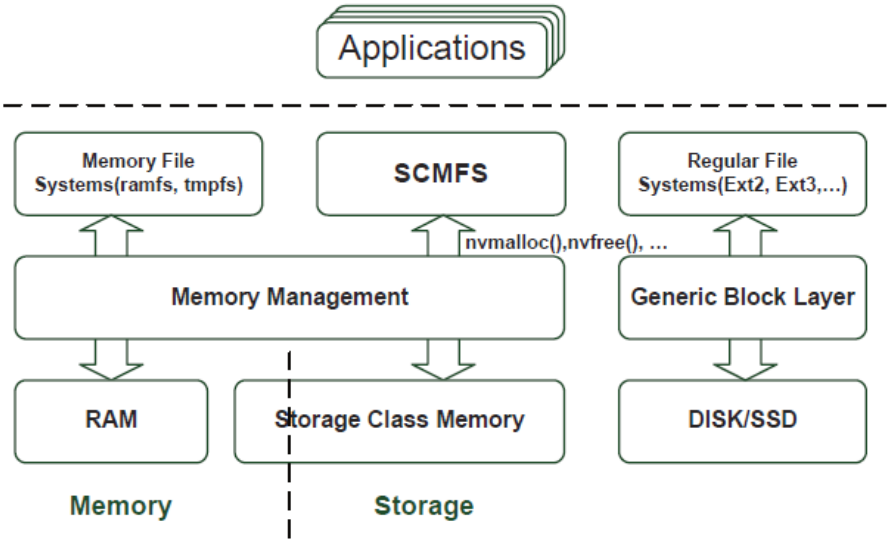


Figure 1: [1] SCMFS is built upon memory management module.

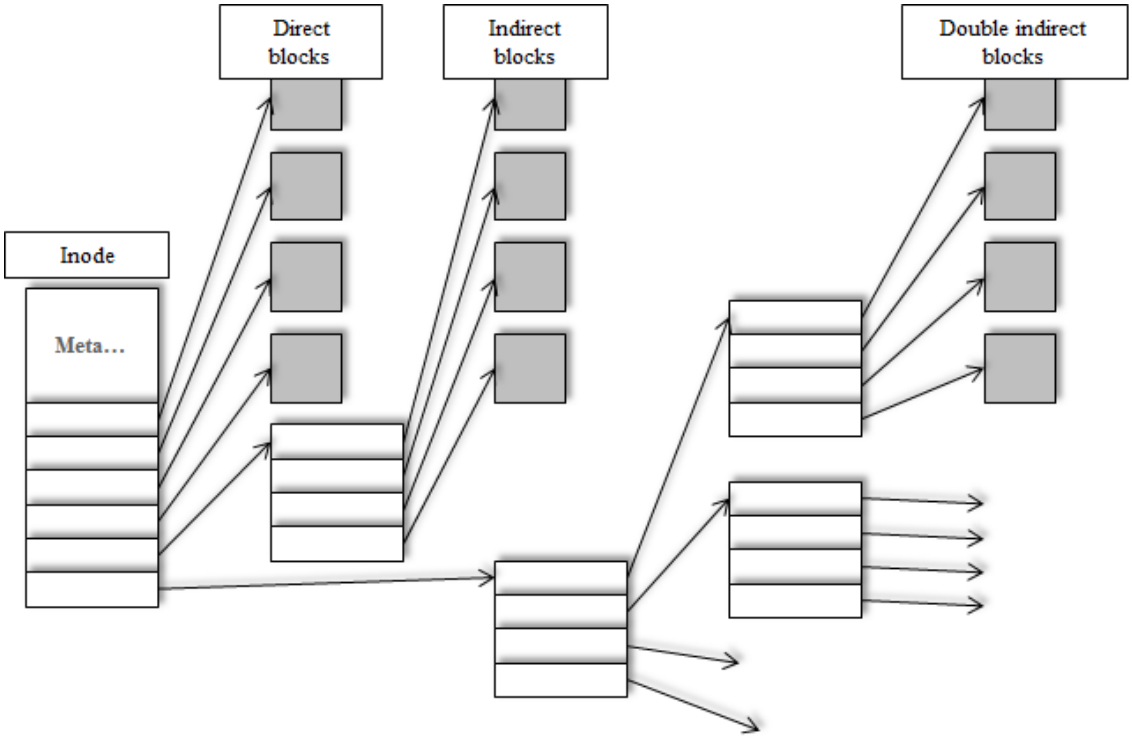


Figure 2: [4] Indirect mapping of blocks in EXT2/EXT3.

SCMFS addresses files on physical memory and it runs on virtual address space, as shown in Figure 3. SCMFS allocates blocks of one file contiguous in virtual address space, thus the process of handling read/write requests is simplified [1].

Figure 3: [4] Contiguous virtual address space for a file in SCMFS.

During reboot, as Figure 4 shows, data on PCM can still be persistent because file system metadata and mapping table are stored on fixed physical addresses on PCM [1]. Metadata stores superblock and inode table information. Mapping table stores address translation from virtual address to physical address of each block.

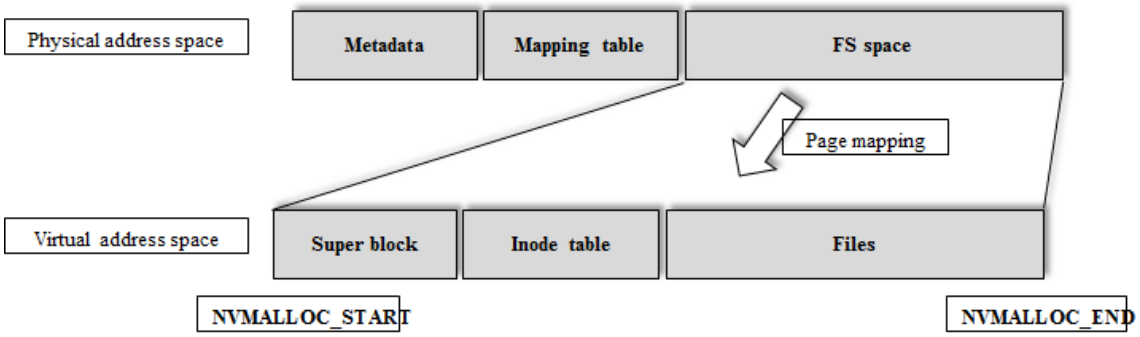


Figure 4: [4] SCMFS file system layout.

With pre-allocation mechanism, SCMFS always maintains certain number of null files within the file system, as shown in Figure 5 [4]. Although these null files have neither name nor data, they have already been allocated some physical space. When a new file is created, SCMFS always looks for a null file first. When a file shrinks or is deleted, the shrunk or deleted space is not de-allocated, instead, it is marked as a null file and kept for future use. Through the space pre-allocation mechanism, the number of allocation and de-allocation operations is reduced significantly. As a result, with accumulative savings from smaller instruction footprint, the file system performance is enhanced significantly.

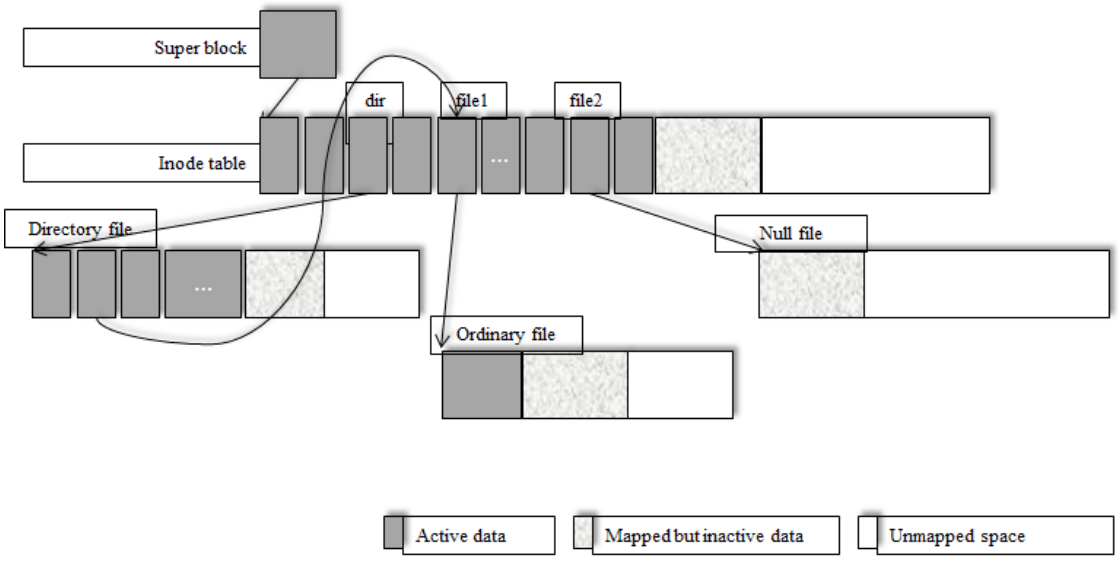


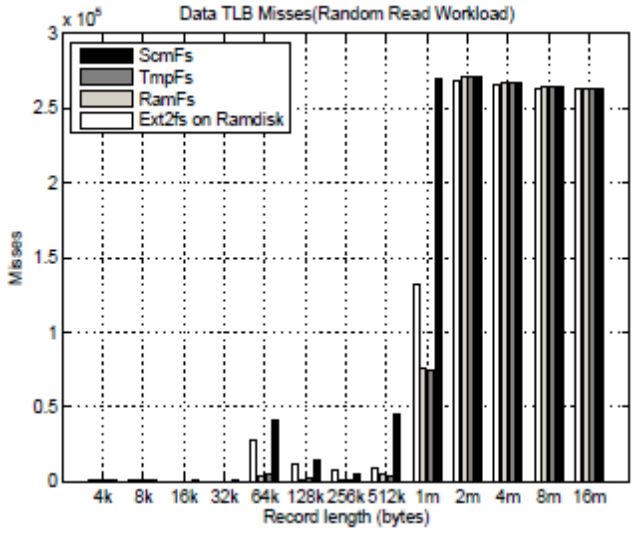
Figure 5: [4] SCMFS pre-allocation using null files.

To recycle the reserved but unused space pre-allocated by the above mechanism, garbage collection is implemented to control the waste [1].

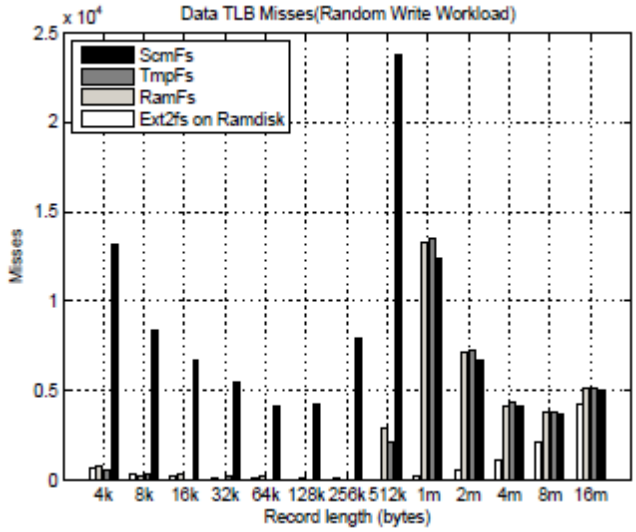
The previous work modified Linux kernel 2.6.33 to add SCMFS. The development was done on Linux x86_64 platform. In x86_64 Linux, three memory zones exist in physical address space: ZONE_DMA, ZONE_DMA32, ZONE_NORMAL [5]. A new memory zone "ZONE_STORAGE" is added for PCM, which is only accessed by SCMFS. Memory allocation/deallocation functions: nvmalloc()/nvfree() gets/releases memory from ZONE_STORAGE. Nvmalloc() derives from vmalloc(), and allocates memory which is contiguous in kernel virtual memory space, while not necessarily contiguous in physical memory space.

With contiguous file virtual address space, the design of SCMFS is largely simplified; on the other hand, this design causes high Data TLB misses compared to

other file systems [1]. Results are shown in Figure 6, from an IOZONE benchmark experiment.



(a)



(b)

Figure 6: [1] SCMF suffers from Data TLB misses.

The main reason is that SCMFS allocates space using small pages (4KB), but RAMDISK works within direct mapping space allocating large pages (2MB) [4].

Next section will discuss a method to reduce TLB misses, and will also analyze if SCMFS has a performance advantage against YAFFS2 and EXT4.

2. SCMFS PERFORMANCE ENHANCEMENT

In this section, first, huge-page allocation mechanism is explained; and then, analysis of further reduction of Data TLB misses is addressed. Next, SCMFS small-page version and huge-page version are compared with YAFFS2 and EXT4 using micro-benchmarks and macro-benchmarks to evaluate the pros and cons of using SCMFS on a mobile platform.

2.1 TLB and Paging in Linux

TLB [6] is included in x86 processors to speed up linear address translation. At the first time that a linear address is accessed, the corresponding physical address is computed through Page Tables in RAM which is a slow and expensive process. The physical address is then loaded and cached in a TLB entry so that further references to the same linear address can be quickly translated. In a multiprocessor system, every processor has its own local TLB.

To load new page table into TLB, CR3 control register of a processor is modified and local TLB are automatically invalidated.

On x86_64 architecture with Linux kernel 2.6, three paging levels and four paging levels are both supported [6]. Linear address bit splitting used by x86_64 platform divides page table address into 5 parts, shown in Figure 7: PGD (Page Global Directory), PUD (Page Upper Directory), PMD (Page Middle Directory), PTE (Page Table Entry), and OFFSET. PGD includes the addresses of several PUDs, which in turn

include the addresses of several PMDs, which in turn include the addresses of several PTEs. Each PTE points to a page frame.

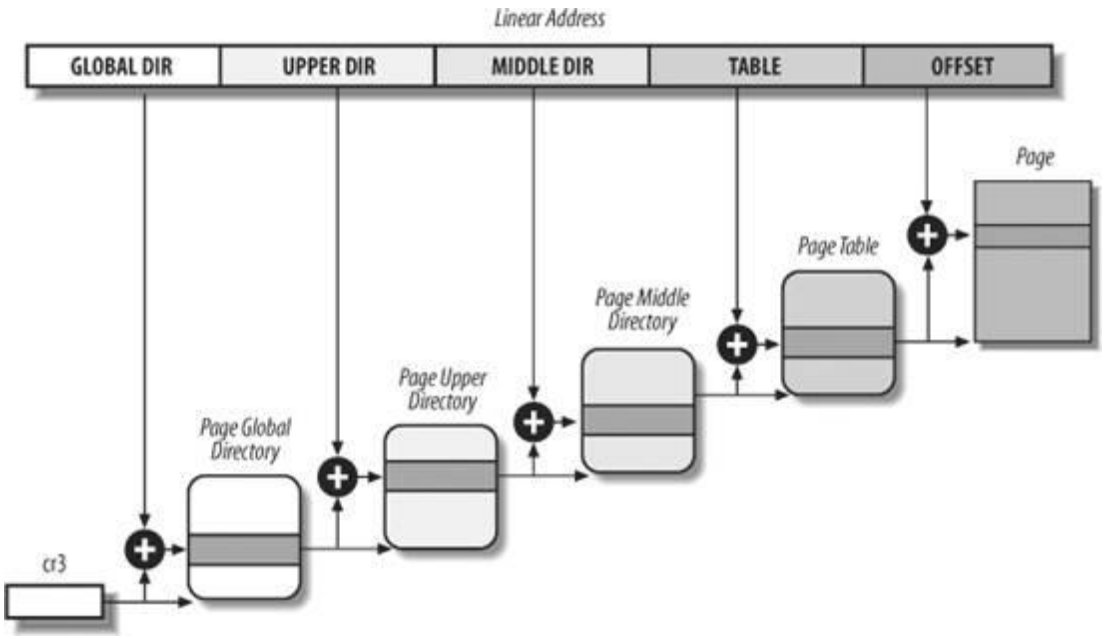


Figure 7: [6] Page table linear address bit splitting.

A number of macros are provided in Linux kernel for each page table level to break up the linear address into parts. Those macros are: a SHIFT, a SIZE and a MASK [5]. PAGE_SHIFT is hash defined as 12 in kernel code and it is used to calculate PAGE_SIZE and PAGE_MASK. PAGE_SHIFT is the length in bits of the offset part of the linear address space, which is 12 bits on the x86. The size of a page is calculated as 2^{PAGE_SHIFT}, which is 4KB.

X86 CPU supports 4KB, 2MB and 1GB page sizes. To enlarge page size to 2MB, more linear addresses are covered by one page, thus OFFSET shifts to left to

expand address coverage. In HugeTLB kernel code, a macro HPAGE_SHIFT is hash defined as PAGE_SHIFT << 9 for 2MB page size. By enabling huge-page support, kernel page table is changed from 4 paging levels to 3 paging levels since PTE is eliminated by OFFSET left shift. Vmap_pmd_range_hugepage() function is implemented to convert PMD to PTE and pte_mkhuge() function is called to set a page table entry for huge-page.

2.2 Employing Huge-pages in SCMFS to Reduce TLB Misses

In the block allocation code of SCMFS, nvmalloc() uses buddy allocator to get physical memory pages as its file blocks, and sets the address of each page in kernel page table which is partially cached in TLB. The capacity of the address coverage of a 2MB page TLB entry is 512 times of the capacity of 4KB page TLB entry. Therefore huge-page can substantially reduce TLB misses.

By default, buddy allocator allocates pages of 4KB. HugeTLB kernel code is leveraged to allocate pages of 2MB for SCMFS to resolve the problem of high Data TLB misses. Given that only using pages of 2MB for all kinds of data may lead to inefficient use of blocks, since file system metadata is usually small and many text files may not exceed 2MB, both 4KB and 2MB page sizes are supported in SCMFS to avoid too much internal fragmentation.

PCM physical address space is divided into two contiguous parts during system boot: one for small-pages, another for huge-pages. When kernel boots, huge-pages are reserved and put into a huge-page queue, as shown in Figure 8. The reason for reserving

huge-pages at the very beginning is that sufficient number of contiguous small pages to form huge pages are not guaranteed after system has run for a while. When writing a file, the file is allocated pages of 4KB first until file size reaches a certain threshold. At the threshold size, one huge page is allocated to it and the file is copied from previous space to the new space, and then all previously allocated small pages are freed. From then onwards, only pages of 2MB are allocated to it.

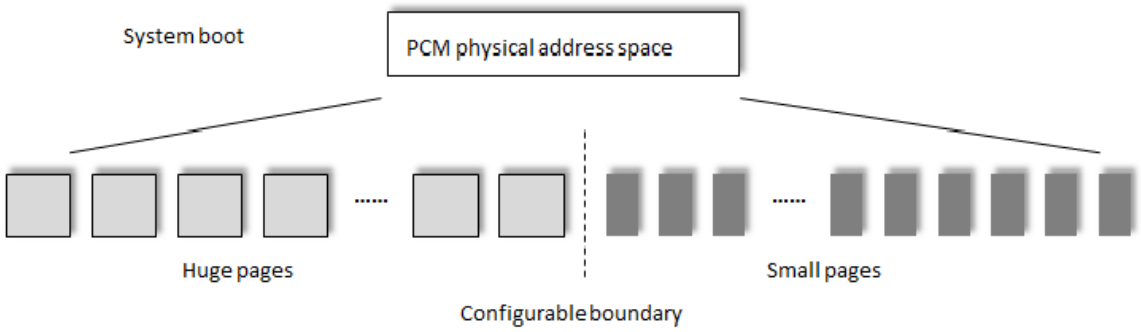


Figure 8: Huge-page reservation during system boot.

As block allocation changes from allocating pages of 4KB to pages of 2MB, the addresses of file blocks start being pointed by huge-page TLB entries instead of small-page TLB entries. In modern x86 processors, such as Intel Core i5, separate TLBs are designed for small-pages and huge-pages. Usually TLBs used by huge-pages provides caching capability between 1 to 2 orders of magnitude of TLBs used by small-pages.

2.3 Further Reducing TLB Misses

Before migrating to a huge-page, TLB entries taken by a file will be equal to $\frac{file\ size}{4K}$. This value reaches a maximum when file size grows to a threshold size, equal to $\frac{threshold}{4K}$. After migrating to huge-pages, the number of TLB entries needed by the file will be $\frac{file\ size}{2M}$. The accumulative TLB entries are the sum of TLB entries before and after migrating. TLB misses are proportional to accumulative TLB entries, therefore it can be presented in the formula below:

$$TLB\ misses\ \propto \frac{threshold}{4K} + \frac{file\ size}{2M}$$

Intuitively, the earlier the file is migrated into huge pages the fewer the TLB misses. However, earlier migration into 2MB huge pages may cause excessive internal fragmentation.

The function updated from previous design is scm_alloc_blocks(). In this function, the algorithm to determine allocating small block or huge block is described as below:

1) When file size is smaller than the threshold size and there is no block allocated for the file, nvmalloc() is called to allocate a certain size of space with small contiguous blocks in it.

2) If the file size is still smaller than the threshold size and the file already has blocks, nvmalloc_extend() is called to allocate more small blocks.

3) If the file size is bigger than the threshold size and there is no block allocated for it, nvmalloc_huge-page() is called to allocate a certain range of space with huge contiguous blocks in it.

4) If the file size is bigger than the threshold size and there are already blocks allocated for it, and if the previous blocks are small blocks, scm_migrate_hugepage() is called to allocate a huge block and migrate file onto it, and then free previously allocated space with small-pages; if the previous blocks are already huge blocks, scm_alloc_blocks_hugepage() is called to expand the space by allocating a larger space with more huge blocks and migrating the file to the new space.

Allocating a huge page and then migrating file from small-page space to huge-page space involves memory copy and memory release which unavoidably increases latency to the file system. If pre-allocating a large space for the file, then the number of migrations can be reduced to improve performance.

The extra mapped space by pre-allocation is not a concern. Given that SCMFS has a garbage collection mechanism which runs a thread in the background to handle mapped but unused spaces [1], when the unmapped space on the PCM is lower than a threshold, this background thread will try to free the extra space which is mapped but not used. The extra space generated by pre-allocation can be recycled instead of being wasted. To be more specific about how it works: during garbage collecting, number of null files is checked first; if the number exceeds a predefined threshold, extra null files are freed. The garbage collection thread considers cold files based on least recently used records. The cold/hot files can be easily classified through the last modified time [1].

2.4 TLB Miss Evaluation

In this test, Data TLB misses of four huge page mechanisms are measured: 1) no huge-pages, 2) migration to huge-pages at 128KB threshold, 3) migration at 512KB and 4) migration at 2MB. Instruction TLB misses account for very little performance loss because SCMFS has a small instruction footprint thus it has low instruction TLB misses discussed in previous research work [1], so this issue is not considered here [7].

Test machine uses Intel Core i5 processor which has separate TLBs: L1 Data TLB0 for small-pages has 64 entries, L1 Data TLB1 for huge-pages has 32 entries; separate L1 Instruction TLB0 and 1 are also present; L2 TLB is shared by data and instructions, and all the TLBs have 64bytes prefetching ability.

The purpose of this test is to verify if huge-page mechanism can reduce Data TLB misses and earlier migration to huge pages can further reduce TLB misses. With IOZONE benchmark, 100MB is chosen as the test file size. IOZONE generates sequential write/read and random write/read workloads. All the operations are normal I/O. To eliminate the randomness of exceptions, data in results are average of 3 runs.

2.4.1 Sequential Write/Read

In sequential write/read tests, the records will be written to a new file and read from it, through sequentially increasing offsets. Each test runs with one record length, varied from 4KB~1MB. Record length means the size of the I/O request. Given a test file of 100MB, 1MB record length means writing to the file 100 times or reading from it 100 times.

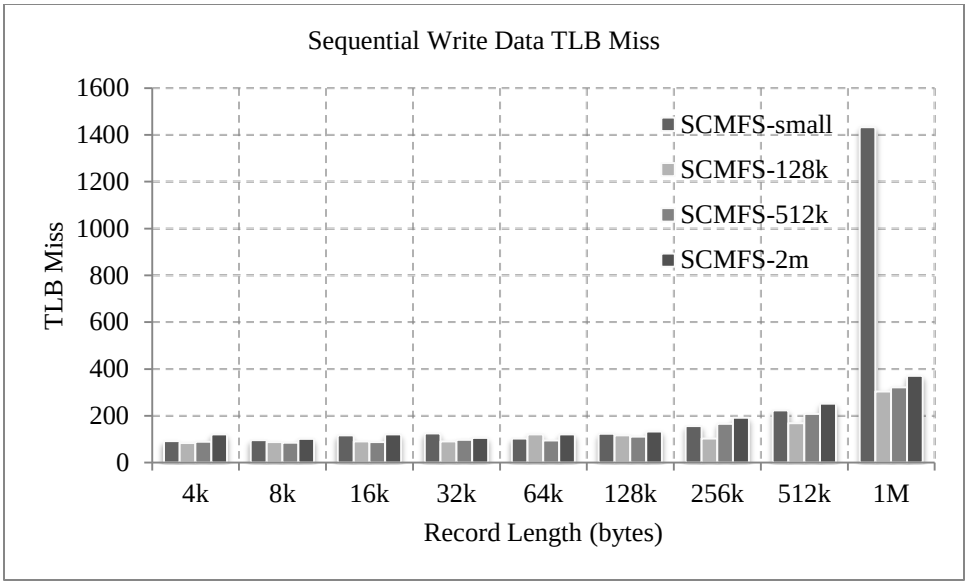


Figure 9: IOZONE sequential write Data TLB misses.

Figure 9 shows the Data TLB misses of sequential write test. At 1MB record length, all huge-page versions have far fewer Data TLB misses than the small-page version. The reason can be explained by showing TLB missing and loading steps:

To begin with, L1 Data TLB0 for small-page has 64 entries, but with small pages, 1MB I/O needs 256 entries, so the I/O size certainly exceeds the capacity of L1 Data TLB0. For each miss, the missing address is loaded from memory to TLB. TLB prefetching mechanism loads more sequentially larger addresses into TLB entries for further TLB access use. When there is enough room in TLB, missing a few entries results in loading more addresses, thus next several accesses can have TLB hits. When there is not enough room in TLB, or when TLB prefetching is not fast enough to fulfill the needs, TLB misses are caused. If this happens in L1 Data TLB, then processor goes to L2 shared TLB to query cached addresses. However data and instructions compete

using L2 shared TLB, so it is possible that not enough TLB entries are guaranteed for data addresses.

On the other hand, after huge-page SCMFS starts migrating to huge-pages, 1MB I/O only needs 1 entry in L1 Data TLB1 used by huge-pages. With prefetching, after loading a new huge-page address into TLB, some more addresses are also loaded, and several next accesses will have TLB hits, so huge-page versions have fewer TLB misses at 1MB record length.

Because of above reason, SCMFS small-page version suffers significant Data TLB misses at 1MB record length, but three huge-page versions maintain fairly lower level Data TLB misses.

In Figure 9, Data TLB misses of all versions are low at small record lengths because L1 Data TLB still has room, and TLB prefetching can fulfill the requests after smaller number of misses.

Figure 10 indicates that all huge-page versions benefit from huge-page mechanism thus their Data TLB misses at 1MB record length are obviously fewer than small-page version.

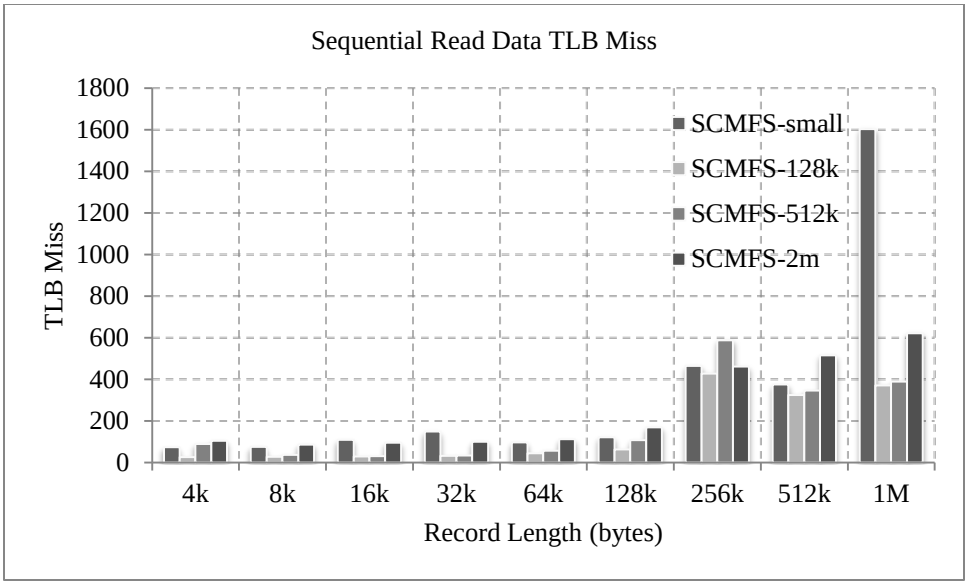


Figure 10: IOZONE sequential read Data TLB misses.

The total TLB misses are affected by TLB prefetching mechanism and replacement algorithm, so there is no one or two simple reasons to explain all the data. TLB miss behavior is also closely related to TLB size. Thus the result may vary across different test machines.

In both Figure 9 and 10, in most cases, huge-page versions have lower Data TLB misses than small-page version in both sequential write and read tests, thus this proves the validity of using huge-page to reduce Data TLB misses. SCMFS-128KB shows fewer TLB misses than other two huge-page versions in all cases shown in the two figures which suggest that earlier migration to huge-page further reduces TLB misses in sequential workloads.

2.4.2 Random Write/Read

In random tests of IOZONE, records will be written and read at random offsets in a file of size 100MB. All operations are normal I/O. Results are shown in Figure 11 and 12.

As Figure 11 and 12 show, in random write and read tests, SCMFS-128k and SCMFS-512k have more reductions of Data TLB misses than small-page version at all record lengths, and SCMFS-128k has the least number of TLB misses among all the huge-page versions which is consistent with our analysis in 2.3 that early migration to huge-page can further reduce TLB misses.

However, migration too early may cause significant internal fragmentation. 128KB is a reasonable threshold which can avoid wasting space. Our experiments here, however, were not constrained by available space and hence fragmentation did not turn out to be an issue.

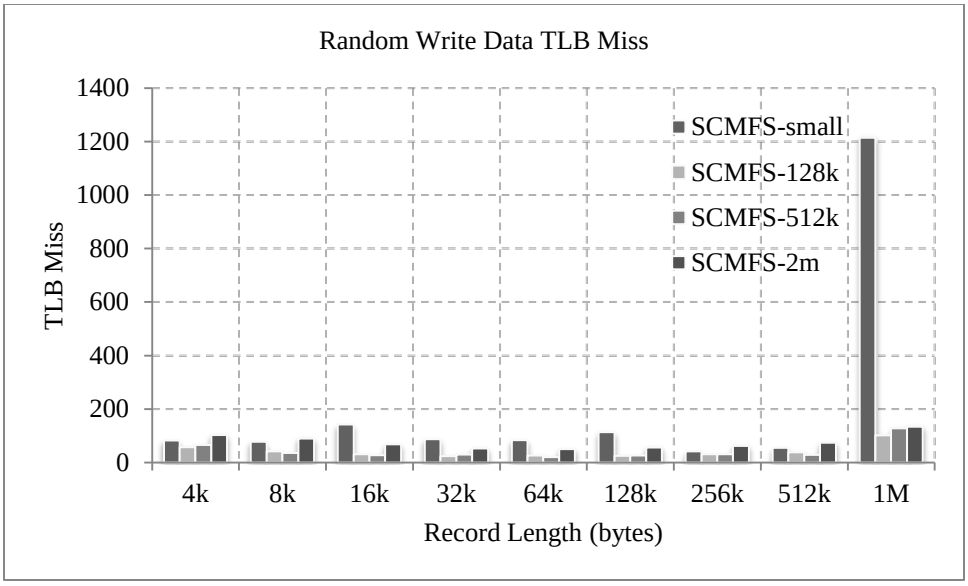


Figure 11: IOZONE random write Data TLB misses.

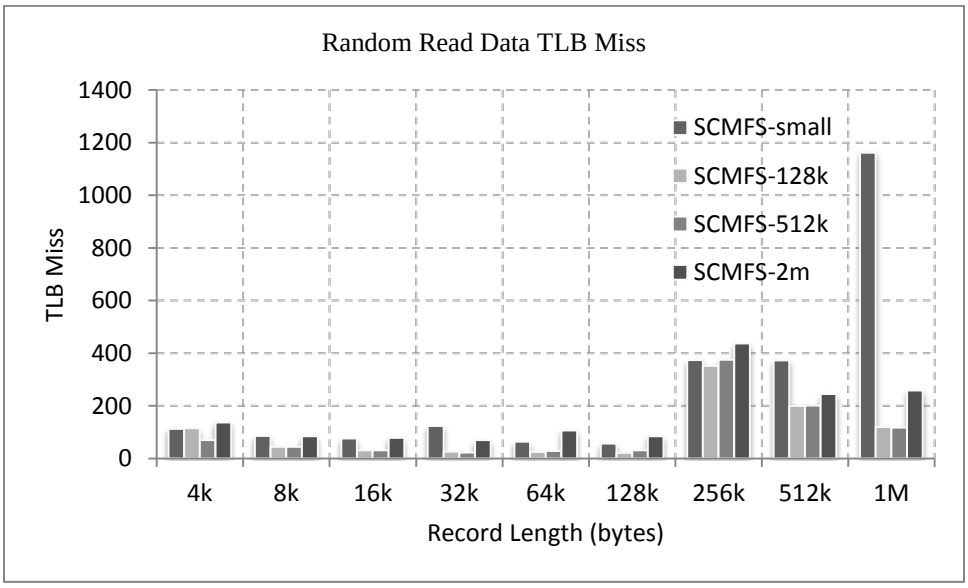


Figure 12: IOZONE random read Data TLB misses.

2.4.3 Enhancement of Throughput

As SCMFS is memory resident file system and PCM is fast enough, TLB misses can become a significant influencing factor in its performance. Next, IOZONE micro-benchmark is still used to measure the throughput of SCMFS small-page version and three huge-page versions, to evaluate the impact of TLB misses on the file system performance. Test configuration is the same as the previous one. Test file is 100MB in size, and sequential and random write/read workloads are executed.

Sequential write is a test to measure the performance of writing a new file. Not only does the data need to be stored but also the metadata information for keeping track of where the data is located on the storage media while writing to the new file. Metadata consists of the directory information, the space allocation and any other data associated with a file that is not part of the data contained in the file [8].

Figure 13 is the result of sequential write throughput test. All huge-page versions have much higher throughput than small-page version because of two main reasons: 1) huge-page versions have less Data TLB misses in this test, 2) allocating a huge-page has an effect of pre-allocating a larger space thus number of space allocation operations of huge-page versions are less than the small-page version, so the total latency of every huge-page version goes down from this point of view. SCMFS-128k benefits from earlier migration to huge-page size, thus it has the best throughput performance compared to all the other versions.

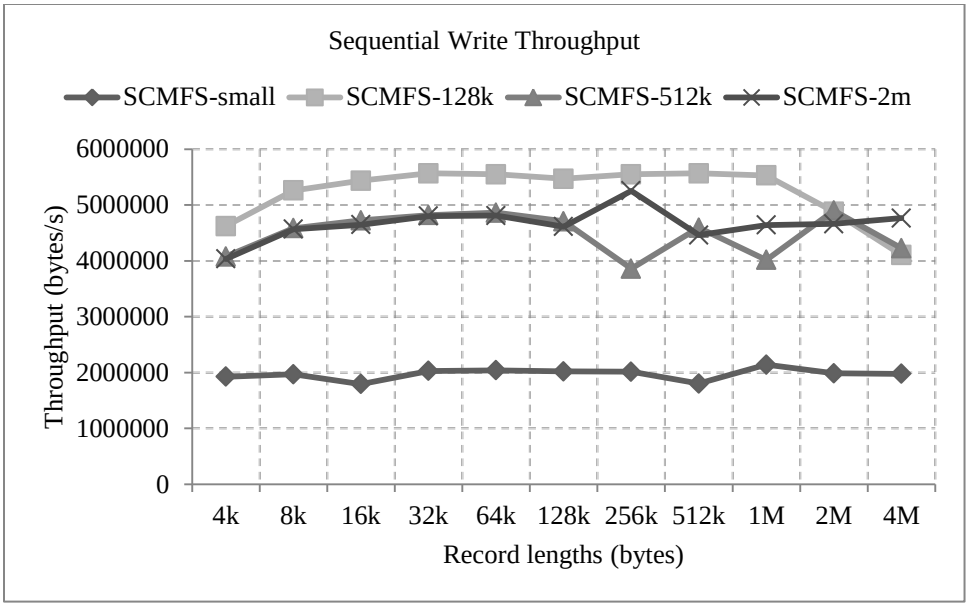


Figure 13: IOZONE sequential write throughput.

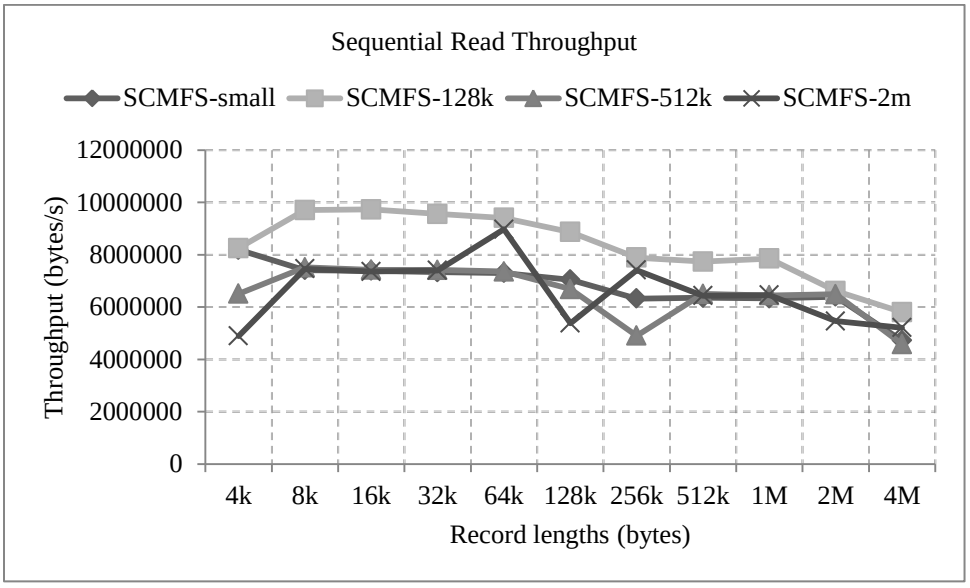


Figure 14: IOZONE sequential read throughput.

Sequential read test measures the performance of reading an existing file. Figure 14 shows the results. SCMFS-128k outperforms other versions at all record lengths.

In single large file tests, sequential write/read throughput results are consistent to TLB miss results. When TLB misses are low, throughput is high; otherwise, throughput decreases. One simple explanation is low latency from low TLB misses. In most cases, SCMFS-128k is better than the other three versions until record length above 2MB. Figure 14 shows that SCMF-small has similar throughput value to SCMFS-512k and SCMFS-2M in sequential read. It appears that TLB misses are not significant enough to affect throughput performance since it is only one of many factors affecting file system performance.

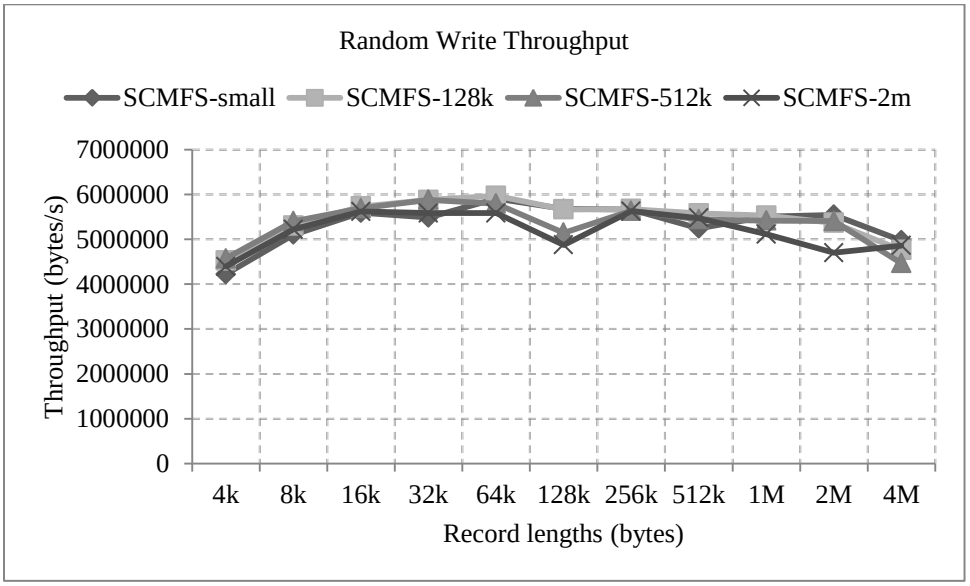


Figure 15: IOZONE random write throughput.

Random write test measures the performance of writing a file with accesses being made to random locations within the file.

Comparing Figure 15 with Figure 13, at all record lengths, random write is better than sequential write in throughput. Since all four versions of SCMFS have fairly low Data TLB misses in random write shown in Figure 11, it is reasonable that Data TLB miss is not a barrier to their performance. Thus four versions of SCMFS reach high throughput.

Random read test measures the performance of reading a file with accesses being made to random locations within the file. Figure 16 shows that SCMFS-128k keeps its outstanding performance at most record lengths.

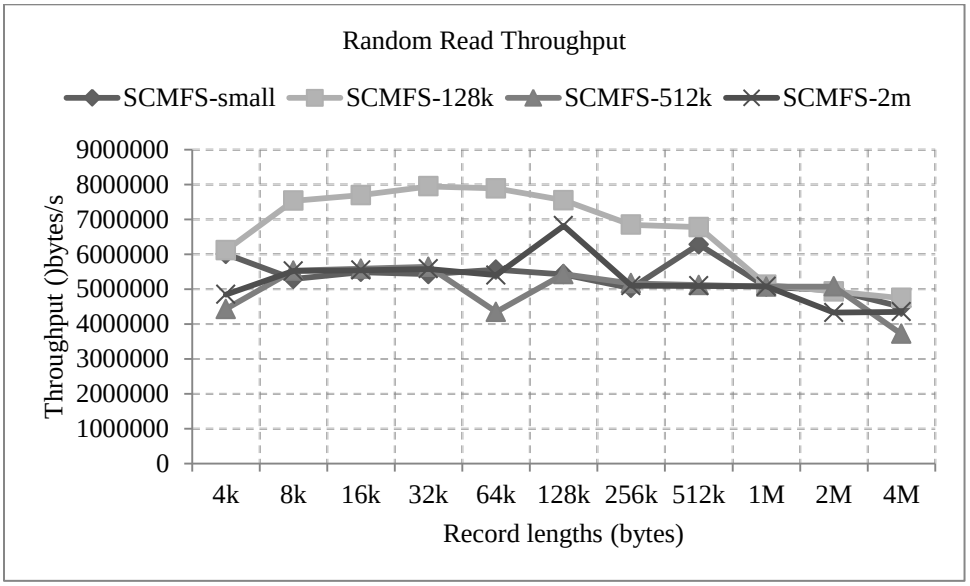


Figure 16: IOZONE random read throughput.

As a conclusion, employing huge-page sizes can significantly increase throughput performance in sequential write operations. Early migration to huge-page can further increase throughput, but the choice of threshold determines the improvement. All the performance results show the advantage of migrating to huge-page at 128KB. Although SCMFS-128k performs the best in most cases in the experiments, this does not mean adopting huge-page at a very early stage of a growing file is better because internal fragmentation can be significant if allocating huge-page when file is small and stays small. Processor TLB capacity plays an important role in determining SCMFS' performance. When a server processor has sufficient TLB entries, the gap between small-page version and huge-page version will be largely reduced.

2.5 Performance Comparison and Analysis

Many vendors in mobile industry are showing interest in SCMFS, therefore implementing SCMFS to Android can make it ready for next generation mobile products using PCM. Section 3 will discuss the methodology of implementing SCMFS to Android. Before that, comparing performance of SCMFS with a widely used Android file system YAFFS2, and its replacement EXT4 can help to evaluate if SCMFS has a performance advantage.

SCMFS small-page version and huge-page version are evaluated and compared with YAFFS2 and EXT4 using widely used file system micro-benchmarks and macro-benchmarks. Micro-benchmark is to isolate specific overheads of a few operations within the system. Macro-benchmark is to run a particular workload that is meant to

represent some real-world workload [9]. Using both types of benchmarks can sufficiently measure I/O level and application level capacities of a file system.

First of all, the features of YAFFS2 and EXT4 are introduced here.

2.5.1 Features of YAFFS2 and EXT4

2.5.1.1 YAFFS2

YAFFS has been designed specifically for NAND flash according to the features of this particular device to maximize its performance. To enhance robustness of NAND flash, YAFFS uses journaling, error correction and verification techniques based on typical NAND failure modes [10].

YAFFS now is in generation 2 to support newer NAND flash chips which have larger pages, 2048 bytes. Each page within an erase block must be written to in sequential order, and each page must be written only once before it is erased again [11]. Log-structure is used to track mapping between logical chunk IDs of a file during operations and real physical chunks (flash page) and blocks. Write request will do operation to a new physical chunk sequentially following the previous chunk, but logically this operation still goes to a reused chunk ID [12]. Because of this feature, YAFFS2 always writes to a new chunk thus it always does sequential writes.

As discussed in Section 1, YAFFS2 has an internal Short-Op Cache to align small writes to chunk in order to keep number of writes low and prolong the life cycle of a flash device; it also collaborates with VFS page cache to cache the path of data to serve small reads [3]. Therefore, it does not support direct I/O. Page cache can enhance

read/write performance of YAFFS2, but when using memory to simulate NAND flash, page cache may cause overhead as explained in Section 1.

2.5.1.2 EXT4

One important performance enhancement feature of EXT4 is “extents” [13]. The traditional Unix-derived file systems such as EXT3 and EXT2 use an indirect block mapping scheme to keep track of each block used for the blocks corresponding to the data of a file. The mapping keeps an entry for every single block. When it comes to a big file, many blocks are mapped. It is inefficient, especially on large file delete and truncate operations, because large latency is caused. With “extents” in EXT4, an extent is basically a bunch of contiguous physical blocks, so indirect mapping maps several blocks to one entry. Since an extent encourages continuous layouts on the disk, this mechanism improves the performance and also helps to reduce the fragmentation.

“Multi-block Allocation” [13] also enhances performance by allocating many blocks in a single call instead of a single block per call in EXT4. SCMFS also allocates multiple blocks per each allocation operation.

“Persistent Pre-allocation” is a feature which allows applications to pre-allocate disk space, but there is no data on it until the application really needs to write the data in the future [13]. This feature also improves fragmentation since the blocks will be allocated at one time as contiguously as possible. This pre-allocation is similar to the corresponding feature of SCMFS. System call of pre-allocation in EXT4 differentiates it from SCMFS.

HTree data structure is used by EXT4 to index all the flat subdirectories. HTree is similar to BTree and has constant depth and large fanout. The superior features of HTree highly enhance the lookup efficiency of the file system [14].

EXT4 enables “Barriers” by default to improve the integrity of the file system at the cost of longer latency. The file system must explicitly instruct the disk to get all of the journal data onto the media before writing the commit record; if the commit record gets written first, the journal may be corrupted. A barrier forbids the writing of any blocks after the barrier until all blocks written before the barrier are committed to the media. By using barriers, the file system can make sure that their on-disk structures remain consistent at all times [15].

From the results of performance enhancement section, SCMFS-128k is concluded as the best huge-page migration mechanism in our test environment, so it represents SCMFS with huge-page ability to compare with SCMFS-small, YAFFS2 and EXT4 in the following tests.

2.5.2 NAND Flash Simulator and RAMDISK Configuration

Section 1 discussed when storage device is a memory device, Data TLB misses can become a significant influencing factor in its performance. In the past, it was not an issue because hard disk is much slower compared to the TLB. In our tests, all storage devices are simulated from DRAM, as YAFFS2 uses NAND flash simulator [16] and EXT4 uses RAMDISK as well as SCMFS uses simulated PCM, thus TLB miss will significantly affect performance results. Since TLB miss is determined by page size, to

make a fair comparison, it is very important to make sure the page sizes of these devices do not vary too much, and all features which cause extra cost of file systems should be disabled.

YAFFS2 is not one of Linux default file systems. It must be ported [17] to Linux before implementing performance comparison. To simulate NAND flash device, MTD (Memory Technology Device) and NAND simulator driver are enabled in kernel configuration. The default NAND simulator is 128M with 512 bytes page size. To enlarge NAND simulator capacity, dummy NAND device can be created by modifying nand_ids table in *nand_ids.c*.

Both NAND simulator and RAMDISK allocate pages from LOWMEM. As YAFFS kernel code defines, when NAND simulator uses page size of 512 bytes, YAFFS1 is automatically selected; if page size is 2048 bytes, YAFFS2 is loaded. A dummy NAND flash device which has page size 2048 bytes, chip size 16KB, device size 1GB, is created in the following tests. According to Android default configuration, directly mapped LOWMEM space has PSE (Page Size Extension) disabled, so RAMDISK page size is 4KB. Before implementing SCMFS on Android, comparing SCMFS with EXT4 with Android setting on Linux can help to predict if SCMFS has an advantage over EXT4 on Android. Therefore, PSE is also disabled on Linux before running benchmarks.

Throughout all tests on Linux, direct I/O of EXT4 is enabled because SCMFS only supports direct I/O, and barrier of EXT4 is disabled to avoid overhead caused by

synchronous writing. Direct I/O cannot be enabled on YAFFS2, because it only supports page cache.

2.5.3 Micro-benchmark Evaluation

POSTMARK is a file system micro-benchmark, which creates a large pool of continually changing files and measures the transaction rates for a workload approximating a large Internet electronic mail server [18]. An initial bunch of random text files ranging in sizes from configurable minimum size to maximum size are generated. A specified number of transactions occur; once the pool has been created. Each transaction consists of a pair of smaller transactions: Create file or Delete file, Read file or Append file. To minimize caching effect, the incidence of each transaction type and its affected files are chosen randomly [18], thus file operations tend to go through target storage device instead of only accessing cache or buffer.

In the next test, to emulate real world random workloads on mobile platform, 200 files ranging from 4KB to 2MB are generated and processed by 4000 transactions to simulate processing files on a mobile platform. In the rest of the performance tests in this section, SCMFS-128k is used to represent huge-page versions because it outperforms other huge-page versions in the previous tests.

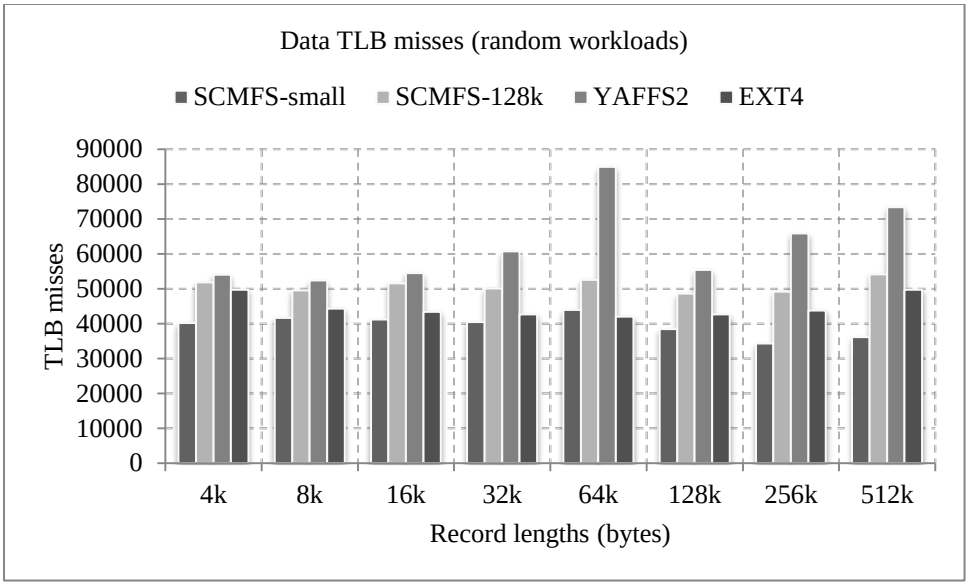


Figure 17: POSTMARK random workloads Data TLB misses.

Figure 17 presents the TLB misses of four file systems. SCMFS-small has fewer Data TLB misses than SCMFS-128k in random workloads which is different from the TLB miss results from earlier IOZONE test. This result can be explained by analyzing the data set. In IOZONE, only a single file is accessed. In POSTMARK test, 200 files of size range 4KB ~ 2MB are processed. Since most files are just a little larger than 128KB, SCMFS-128k allocates extra huge-pages for them, but SCMFS-small won't do that, thus the extra huge-pages cause extra TLB misses. When doing migration to huge-page in SCMFS-128k, the migration operation also causes TLB misses. Therefore, the total TLB misses of huge-page version exceed the total TLB misses of small-page version. It shows that huge-page mechanism does not bring benefit to SCMFS when small files become majority in data set.

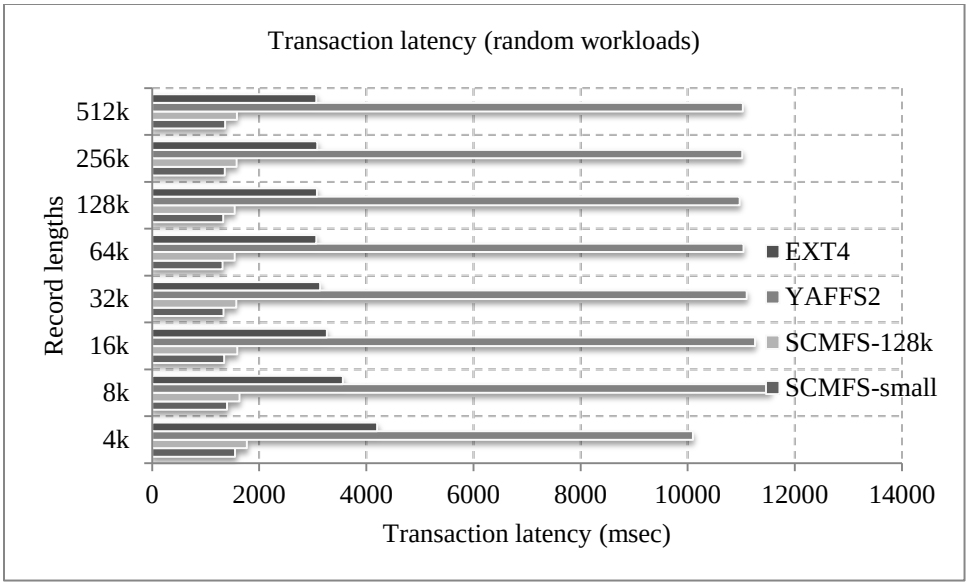


Figure 18: POSTMARK random workloads transaction latency.

Although EXT4 has fewer TLB misses than SCMFS-128k in Figure 17, it still has longer latency in Figure 18 as the fact that it always accesses generic block layer which causes extra operations. SCMFS wins from its simple design and small instruction footprint, so it is faster than EXT4, thus the write and read throughput of SCMFS exceed the throughput of EXT4.

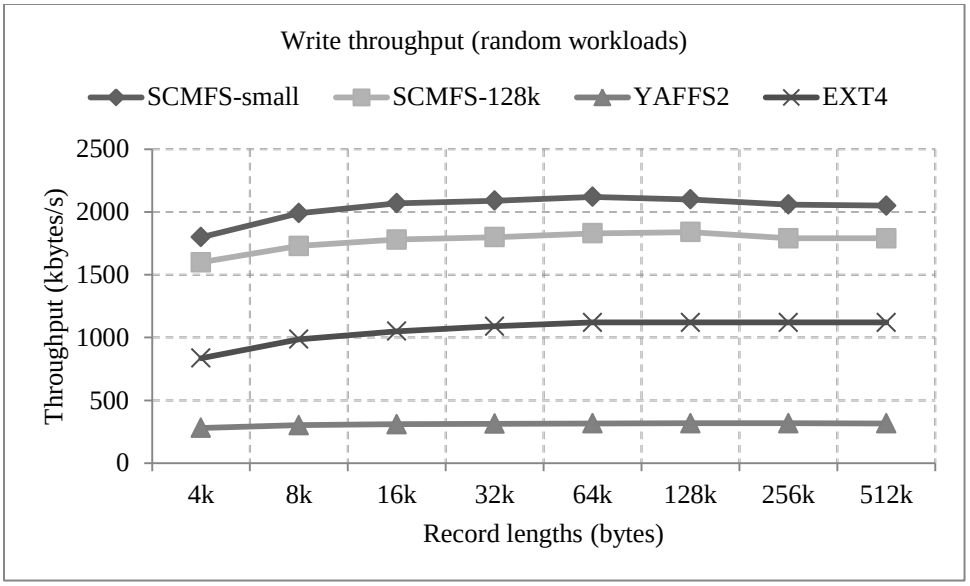


Figure 19: POSTMARK random workloads write throughput.

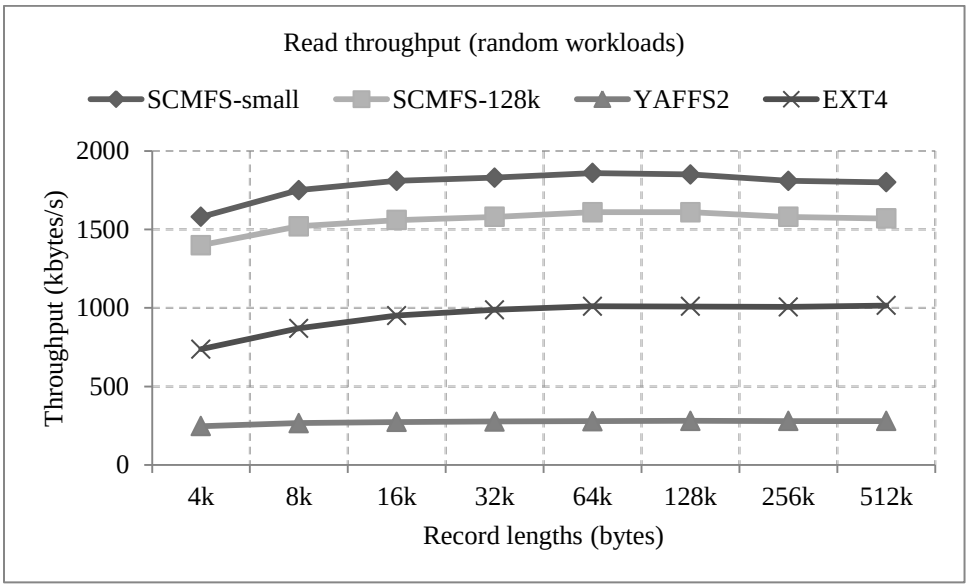


Figure 20: POSTMARK random workloads read throughput.

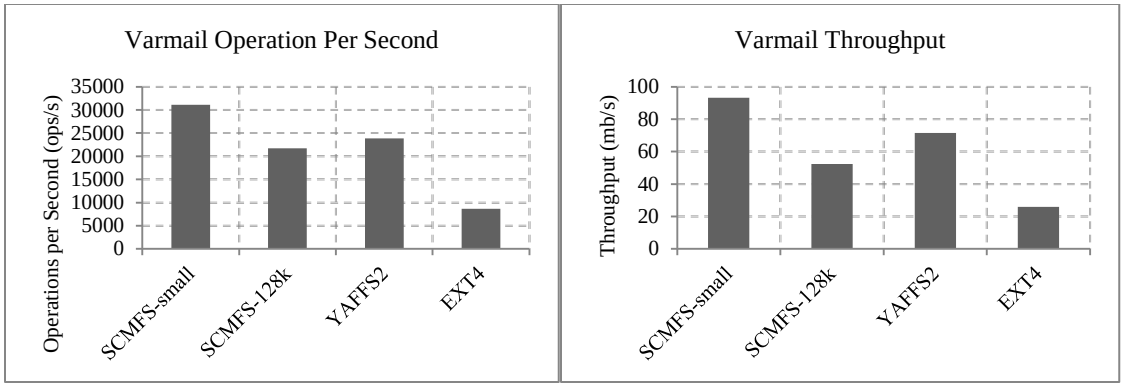
In throughput results in Figure 19 and 20, SCMFS-small is the fastest in all the cases. This result verifies that huge-page cannot help to improve performance when there is large amount of small random data.

Figure 17, 18, 19 and 20 show that YAFFS2 has the highest number of TLB misses, longest latency and lowest throughput. It is because of two main reasons: 1) YAFFS2's page size is 2KB which is half of other file systems' size, thus its TLB misses are higher, 2) it tends to buffer read/write operations until they are chunk aligned thus the latency is much higher than other file systems.

2.5.4 Macro-benchmark Evaluation

To evaluate and compare these four file systems in a real-world environment, FILEBENCH macro-benchmarks are used. FILEBENCH provides some configurations to simulate real-world data workload, e.g. mail server, file server, web server. In all three test cases, 1000 random files are generated with a mean file size of 16KB, and the maximum file size is 1MB.

Varmail is a multi-threaded mail server workload [19]. FILEBENCH performs a sequence of operations to imitate reading mails, composing and deleting mails. Unlike the file server and web proxy workloads introduced later, the mail server workload puts all the files in one directory. It exercises large directory support and fast lookups.

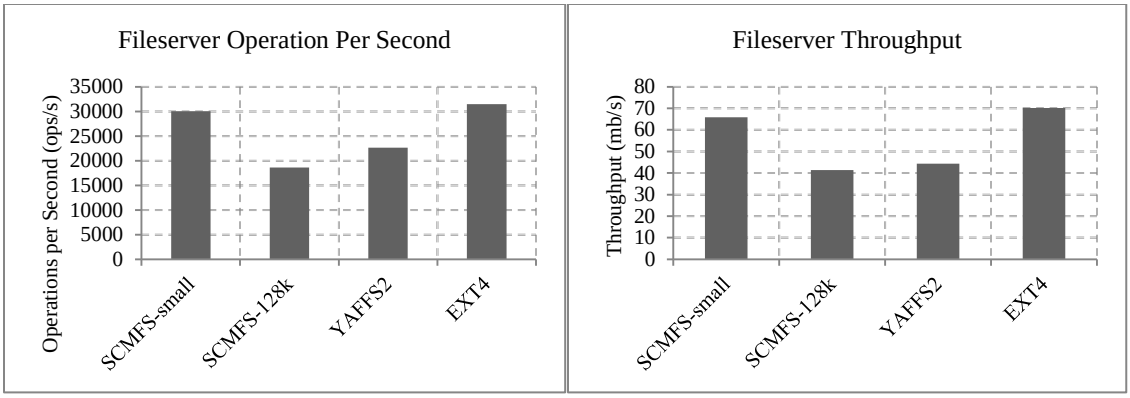


(a) (b)

Figure 21: Mail server simulation. (a) operations per second, (b) throughout.

Figure 21 provides performance results of operations per second and throughput of mail server simulation test. Operations per second is an indicator of I/O efficiency of the file system. It is proportional to throughput. The results show that SCMFS-small is the fastest file system compared to the other three. SCMFS-128k is not as good as SCMFS-small because huge-page mechanism has no advantage in the case of large number of random small files.

Fileserver is a file server workload which emulates a server that hosts home directories of multiple users [19]. Users only access files and directories belonging only to their respective home directories. Each user thread performs a sequence of create, delete, append, read, write, and stat operations, exercising both the meta-data and data paths of the file system.



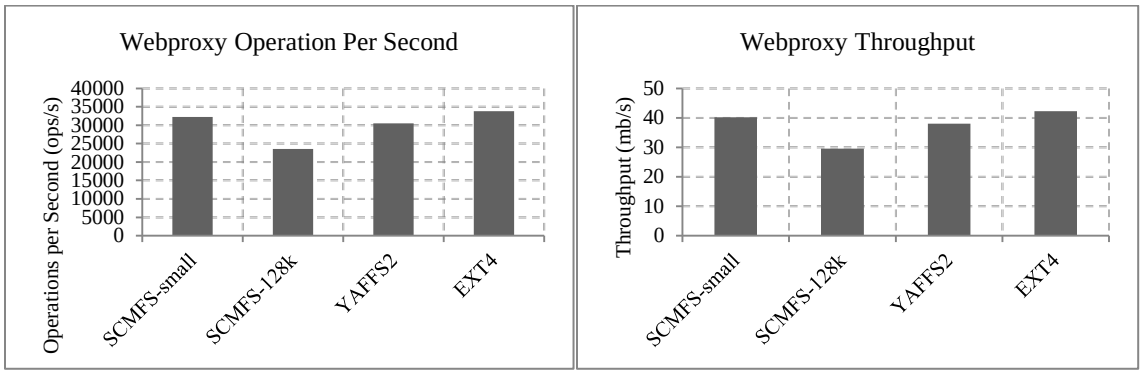
(a) (b)

Figure 22: File server simulation. (a) operations per second, (b) throughput.

In Figure 22 of file server simulation results, EXT4 shows slightly higher performance than SCMFS-small in both operations per second and throughput metrics. One important difference between EXT4 and SCMFS is the data structure used by the file system to index subdirectories. EXT4 uses HTree structure to index all the flat subdirectories and its lookup time complexity is constant. However, because of simple design, SCMFS uses linked list to store all the subdirectories and files. Therefore, lookup time complexity of SCMFS is O(n), which is not as good as HTree. In file server test, multiple subdirectories each indicating a home directory of a user are created. Inside every home directory, multiple subdirectories are created by workload script. It is possible that HTree index is more efficient than singly linked list when handling this particular workload, so that EXT4 has higher performance than SCMFS.

Webproxy workload emulates a proxy server that receives a large set of files and continuously receiving new data of those set of files, and then clients reading from those

files constantly. Each thread performs delete, create, append, close, and repeating open, read and close.



(a) (b)

Figure 23: Web proxy simulation. (a) operations per second, (b) throughput.

Figure 23 shows results of web proxy simulation. The similarity of this workload with the file server workload exists in the multiple subdirectories generated by test script. In this particular test, EXT4 has higher operations per second and throughput value than SCMFS. One possible reason is also that HTree helps EXT4 execute path lookup faster than SCMFS.

In a sum, these three workloads all have a large number of write and read operations of random files. The difference between varmail and the other two is that varmail puts all the files under one flat directory, but the other two put files under multiple subdirectories. The results show that SCMFS-small always achieves higher operations per second and throughput than SCMFS-128k because allocating huge-pages and migrating to huge pages bring extra TLB misses and longer latency. This shows that

in real-world situations, because of large number of small files, the advantage of huge-page mechanism can be canceled out. In the mail server test, EXT4 is not as good as SCMFS, but better than SCMFS in the other two test cases, because file server and web proxy have multiple subdirectories and EXT4 uses more efficient data structure HTree to index all the subdirectories, thus the lookup efficiency is high. SCMFS' simple linked-list design is not as good as HTree when handling large number of subdirectories in above two test scenarios. YAFFS2 is better than SCMFS-128k in all the three test cases. Since all the three cases have a lot of random read operations, it is likely the case that page cache enhances read efficiency of YAFFS2. However page cache is not supported by SCMFS. In addition, SCMFS-128k has overhead caused by huge-page allocation and migration, so its overall performance is worse than YAFFS2 in these three test cases with the particular test dataset.

2.6 Conclusion

Based on the performance analysis in this section, employing huge-page to SCMFS helps to reduce Data TLB misses when handling large files. Huge-page version has higher throughput than small-page version especially in sequential write workloads because of fewer Data TLB misses and larger pre-allocation. Earlier migration to huge-page can further reduce Data TLB misses and improve throughput in large file case. However in the case of large number of random files including small and medium files, SCMFS huge-page version has extra cost because of huge-page allocation and migration.

SCMFS small-page version is better for Android because random small files are usually the case on mobile platforms.

3. IMPLEMENTATION OF SCMFS ON ANDROID

In this section, an overview of the Android platform is provided, followed by a description of the implementation of SCMFS in the Android kernel. The two main differences of this implementation from the previous one in Linux kernel are the reservation of a memory zone for PCM and the concurrency control of inode operations.

3.1 Android Architecture

3.1.1 Android Kernel Differences to Linux Kernel

Compared to the broad coverage of target architectures by Linux kernel, Android only fully supports two at this time: x86 and ARM. Their instruction design philosophy is fundamentally different. The x86 family is primarily a CISC (Complicated Instruction Set Computer) design whereas ARM is a RISC (Reduced Instruction Set Computer) architecture. Therefore simpler instructions are used by ARM processors when compared to an equivalent set of x86 operations. [20].

Android operating system is developed by Google based on the Linux kernel 2.6, but does not use a standard Linux kernel. The kernel enhancements of Android as shown in Figure 24 include alarm driver, ashmem (Android shared memory driver), binder driver (Inter-Process Communication Interface), power management, low memory killer, kernel debugger and logger [20].

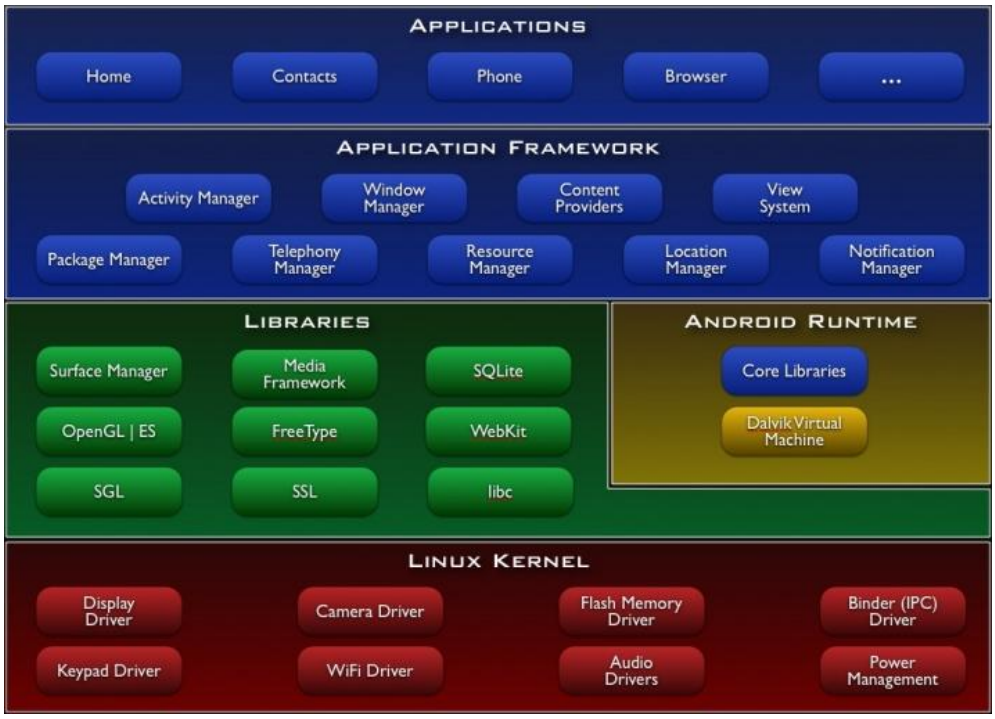


Figure 24: [20] Android architecture.

Compared to Linux kernel, Android kernel is smaller, more preemptive, and more able to support enhanced drivers of mobile devices. For instance, its power management module manages power and saves power more aggressively; its low memory killer driver works for memory constrained mobile devices which set thresholds in user space and kills processes that exceed thresholds.

3.1.2 Android Storage System

Typical Android mobile device has three level storage hierarchies: RAM, internal NAND flash, and external SD card, from the fastest to the slowest. The internal flash storage contains all the important system partitions due to its faster access than external

SD card and bigger capacity than main memory. The external storage is primarily used for storing user content such as media files such as songs, movies, and photographs, documents, and backup images [21].

Figure 25: [21] Android storage system.

As Figure 25 shows, the most important default partitions include [21]: 1) */system*: contains the entire operating system, except the kernel and the RAMDISK. Android user interfaces and pre-installed system applications are located in this partition. 2) */data*: stores user data and installed applications. 3) */cache*: Android stores frequently accessed data and application components in this partition. 4) */sdcard*: external SD card partition to store media, documents, backup files, etc.

In this research work, */system* and */data* partitions must be granted full access permission in order to manipulate system commands, mount target file systems, and change cache of applications. Since Android does not support multiple users login and the default user is not root, user cannot manipulate directories and files in root partition, and also */system* and */data* partitions. The most popular approach to grant user superior permission is to root Android [22]. However since Android x86 full source code is available, the approach we used is to change access mode of some important directories

in *init.rc*, such as */data/data* which contains profile, cache, and database of all applications, */system/xbin* and */system/bin* which contain all root commands and user commands respectively, then those directories are automatically configured as fully accessible during boot.

3.1.3 Other Uniqueness of Android

Android developers viewed the GNU C library as being too resource-consuming for memory constrained embedded system. Moreover, GNU C library is licensed under the GNU Lesser Public License (LGPL) and restricts licensing of derivative works. Due to these concerns, Android developers create their own C library: "Bionic" [20].

In contrast to J2ME (Java 2 Micro Edition) built by some top cell phone manufactures to optimize Java virtual machine, Android has its own Dalvik Virtual Machine [20]. The Dalvik VM is a fast register-based VM providing a small memory footprint. Every application has its own instance of the Dalvik VM [21].

SQLite [21] is the primary means used by Android to store structured data. It is a lightweight transactional database engine which has small memory and disk footprint. Every application has its own SQLite database to store data.

3.2 Implement SCMFS on Android

3.2.1 Choice of Target U-ARCH

Android x86 platform is chosen for reasons stated below:

1) Android x86 [23] is an open source project which ported Android to 32bit x86, so the entire source tree is fully accessible.

2) SCMFS is designed for x86 and it needs to modify x86 memory detection code to reserve memory zone for PCM. ARM does not have memory detection mechanism as much as x86 has, because ARM cores are used in embedded systems thus it is unnecessary for it to be compatible to broad architectures [24].

3) ARM mobile phone has limited DRAM which is not enough to simulate PCM. Android x86 can run in a virtual machine on PC, so PC hardware resources can be available in Android testing, thus ensuring enough memory resources.

3.2.2 Kernel Modification

The main difference between SCMFS on Android x86 and Linux x86_64 is the kernel virtual address space that SCMFS can use. In the following sections, the virtual address space that SCMFS uses on i386 platform and the capacity that SCMFS can reach will be discussed. Android kernel also provides VFS inode APIs which have better concurrency control for inode operations, so changes of inode functions will also be discussed in this section.

3.2.2.1 Memory Zone Modification

On a 32bit architecture, Linux kernel can directly address the first 1GB of physical memory (896MB when considering the reserved range) [25]. Within this 1GB, ZONE_NORMAL is a directly mapped zone that kernel can directly access physical

addresses. In virtual address space, this part is usually called LOWMEM. Above ZONE_NORMAL is ZONE_HIGHMEM. When physical memory is larger than 1GB, HIGHMEM should be enabled in kernel configuration to make the physical memory above 1GB accessible by kernel. ZONE_HIGHMEM is not directly mapped, but mapped into LOWMEM per page when there is an access. On the other hand, in the 64bit architecture, ZONE_NORMAL extends all the way to 64GB or even larger. There is no ZONE_HIGHMEM anymore.

As some papers argue, PCM still cannot replace DRAM to be main memory on account of its smaller write life cycle and lower performance than DRAM. In real usage model, PCM can play the role of a fast secondary storage in mobile devices to store the most frequently accessed data or OS image for fast boot purposes. It is highly likely that PCM will stay in the physical address space above 1GB. Therefore, in the implementation of SCMFS on Android, ZONE_STORAGE for PCM is reserved by shrinking ZONE_HIGHMEM during system boots.

To simulate PCM, as Figure 26 shows, ZONE_STORAGE is reserved between the end of ZONE_HIGHMEM and max_pfn -- the maximum page frame number detected by e820 code. The default end of highmem is max_pfn if HIGHMEM is enabled in the kernel configuration. To reserve a zone, highmem_end is shrunk to a certain extent based on the needs.

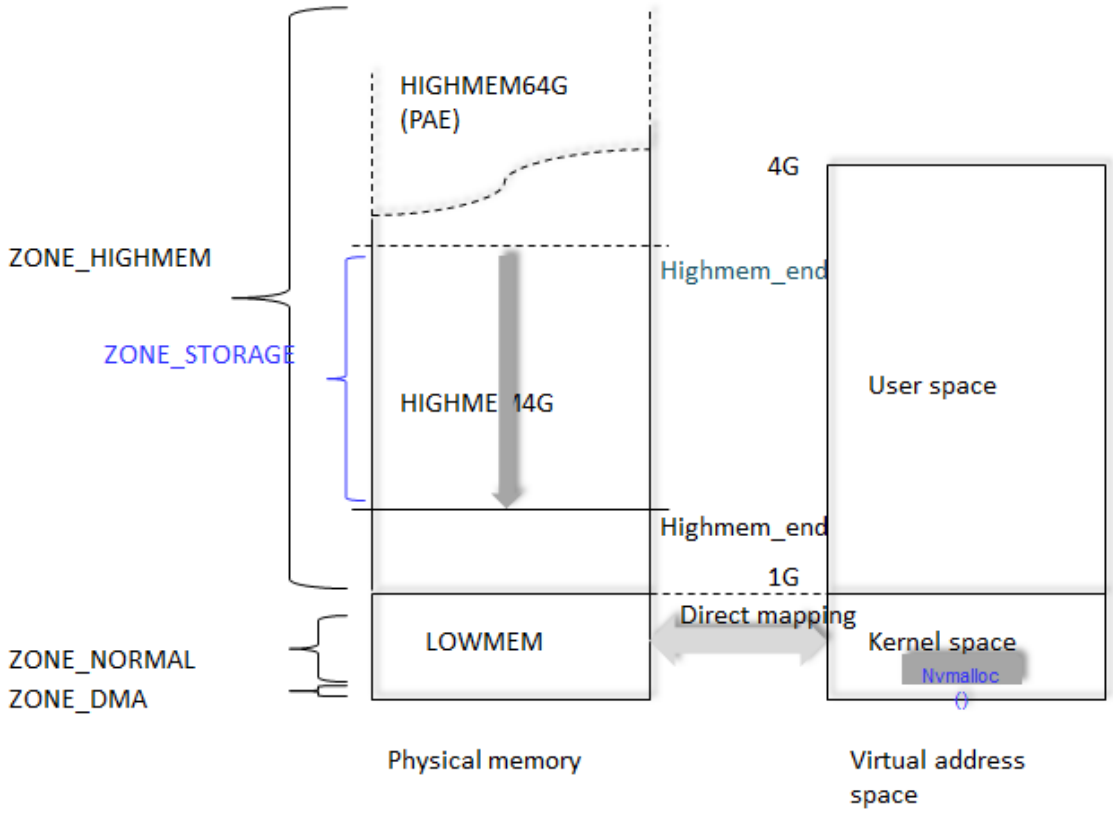


Figure 26: Android x86 memory zones layout.

To allocate/de-allocate blocks from ZONE_STORAGE, nvmalloc()/nvfree() functions which derive from vmalloc()/vfree() are used. There are three built-in memory allocators in kernel: vmalloc(), kmalloc() and _get_free_pages(). Vmalloc() allocates from ZONE_HIGHMEM by default and falls back to ZONE_NORMAL if there is no HIGHMEM available. It gets non-contiguous physical pages using buddy allocator and stores all allocated pages into a page array, then sets up page mapping in the kernel page table, thus it forms a larger contiguous virtual space from many non-contiguous physical pages. The space it provides is not directly mapped. Vmalloc() uses flags

GFP_HIGHMEM|GFP_KERNEL to allocate from HIGHMEM or LOWMEM. As a derivation of vmalloc(), nvmalloc() takes the same approach to form contiguous large virtual space and uses GFP_NV flag to ensure accessing ZONE_STORAGE. Kmalloc() and _get_free_pages() allocate from directly mapped address space, but they can only return contiguous physical pages, thus they cannot provide large space needed by SCMFS. They both use flag GFP_KERNEL to allocate from LOWMEM. Vmalloc(), kmalloc() and _get_free_pages() cannot access ZONE_STORAGE with their own flags.

3.2.2.2 SCMFS Capacity

Table 1: X86_64 virtual address space layout.

0000000000000000 - 00007fffffffffff	(=47 bits) user space
ffff800000000000 - ffff80ffffffffff	(=40 bits) guard hole
ffff880000000000 - ffffc7ffffffffff	(=64 TB) direct mapping of all physical memory
ffffc80000000000 - ffffc8ffffffffff	(=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff	(=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff	(=40 bits) hole
ffffea0000000000 - ffffeaffffffffff	(=40 bits) virtual memory map (1TB)
ffffffff80000000 - ffffffffa0000000	(=512 MB) kernel text mapping
ffffffffa0000000 - fffffffffff00000	(=1536 MB) module mapping space

SCMFS uses nvmalloc() to allocate blocks. The size of the address space accessible by nvmalloc() determines the capacity of SCMFS.

In a 64bit system memory map [26], there are address "holes" in between some contiguous memory mapped spaces, and the holes are large. In the original design on Linux, nvmalloc() uses addresses within an unused hole highlighted in Table 1. This space is up to 2^{45} bytes, so the space accessible by this allocator is large enough. However, in 32bit system memory map, address resources are much more constrained. There are no address "holes" that SCMFS can use, so nvmalloc() reuses vmalloc() space.

Table 2: Android x86 virtual address space layout.

fixmap	0xfff16000 - 0xfffff000	(932 KB)
pkmap	0xff800000 - 0xffc00000	(4096 KB)
vmalloc	0xf7ffe000 - 0xff7fe000	(120 MB)
lowmem	0xc0000000 - 0xf77fe000	(887 MB)
.init	0xc14a2000 - 0xc1503000	(388 kB)
.data	0xc1328bb2 - 0xc14a1140	(1505 kB)
.text	0xc1000000 - 0xc1328bb2	(3234 kB)

On a 64bit machine, VMALLOC_START and VMALLOC_END are both defined as fixed addresses according to the memory map. On a 32bit machine, as Table 2 shows, the memory space is more constrained. The space between VMALLOC_START and VMALLOC_END is determined by VMALLOC_RESERVED, which has a default

value 128MB. Except for an 8MB offset, vmalloc range is 120MB by default, which is not big enough for SCMFS.

Expanding VMALLOC_RESERVED is one way to enlarge SCMFS, but only to a limited extent. For example, if kernel space is 1GB, VMALLOC_RESERVED is 800MB which can be used for SCMFS. This size is still valid as the capacity of a fast secondary storage which stores only the most frequently used data or the Android OS image.

Table 3: SCMFS capacity.

VMsplit	SCMFS Capacity	Max File Size	Number of Files
3G/1G	800M	600M+	2M: 300+
			6M: 100+
			25M: 20+
			100M: 7
2G/2G	1.8G	1G+	2M: 500+
			6M: 180+
			25M: 50+
			100M: 10+
1G/3G	2.8G	2G+	2M: 1000+
			6M: 400+
			25M: 100+
			100M: 20+

On another side, since vmalloc space is constrained by kernel space, enlarging the whole kernel space can fundamentally resolve the problem of capacity limitation. Android kernel supports 3 types of user/kernel VMsplit (virtual memory split): 3G/1G, 2G/2G and 1G/3G. The default one is 3G/1G. To make 2G/2G VMsplit workable, kernel starting address must be changed from 0xC0000000 to 0x80000000, and then PRELINK must be enabled in all default system libraries and application libraries in Android build source code.

Table 3 shows the guaranteed number of files that SCMFS can support under different ways of VMsplit.

3.2.2.3 Better Concurrency Control in Inode Functions

Embedded devices require preemptive kernel since real-time and responsiveness are important characteristics of such devices. To ensure file system data consistency, better concurrency control should be added to protect critical sections.

Android supports multi-tasking [21]. The kernel uses a fine-grained locking mechanism to protect individual data structures. Various locking options are provided and each is optimized for different kernel data usage patterns.

Spinlocks [27] are designed for the shot-term protection of critical sections from being accessed by other processors. Processor repeatedly checks whether it can acquire the lock without going to sleep while kernel is waiting for a spinlock to be released. Spin_lock_irqsave [27] not only acquires the spinlock but also disables the interrupts on the local CPU.

Read/write locks [27] have two types of access to data structures. Concurrent read can be performed by any number of processors to read from a data structure, but write access is restricted to a single processor. Thus read lock is shared lock but write lock is exclusive lock.

Android 2.2 kernel implements a set of inode operation functions with concurrency control by using read/write lock and spinlock. Existing functions in the original design of SCMFS are replaced with these new functions and affected kernel code sections are updated accordingly.

Table 4: Inode function changes.

Functionality	Linux 2.6.33	Android 2.2
Allocate inode	vfs_dq_alloc_inode()	dquot_alloc_inode()
Free inode	vfs_dq_free_inode()	dquot_free_inode()
Drop inode	vfs_dq_drop()	dquot_drop()
Clear inode	clear_inode()	end_writeback()
Delete inode	delete_inode()	evict_inode()

In Table 4, in the first three functionalities, read/write lock is used to protect allocating an inode, freeing an inode or droping an inode. Spinklock is used to protect an inode status change.

In Linux 2.6.33 kernel, clear_inode() checks some flags of inode, after checking and making sure the inode can be cleared, it sets I_CLEAR flag to inode state. In Android 2.2 kernel, end_writeback() checks the same flags of inode with spin_lock_irq which is to disable interrupt while checking, and after checking and making sure the inode can be cleared, it writes I_CLEAR flag in a synchronous way to guarantee the flag is set correctly. Thus end_writeback() also provides better multithreading capability to SCMFS.

Delete_inode() in Linux kernel 2.6.33 truncates inode pages first and then clears inode. Evict_inode() in Android 2.2 kernel uses spinlocks to protect inode state checking, and deleting and freeing inode from the inode LRU list.

3.3 Summary

Change of memory zoning and enhancement of concurrency control of inode functions are two major changes of SCMFS on Android. The next section will address performance evaluation on Android to compare SCMFS with YAFFS2 and EXT4.

4. FILE SYSTEM PERFORMANCE EVALUATION ON ANDROID

This chapter addresses the performance evaluation on Android to compare SCMFS with YAFFS2 and EXT4. The benchmarks used for performance evaluation are introduced in the first section, followed by micro-benchmark tests, user data workload simulation tests and finally application performance evaluation.

4.1 AndroSH

Existing file system benchmarks such as IOZONE and POSTMARK are not available on Android x86. Instead of porting existing benchmarks to Android x86, AndroSH benchmark is developed in this project to evaluate I/O level and application level performance on Android. AndroSH is a script based benchmark which makes use of Android BusyBox command line utilities such as dd, cp, mkdir, rm, tar, scp to create several test cases to exercise file system from many angles. In all the tests, a NAND flash with 2KB page size for YAFFS2 and a RAMDISK with 4KB page size for EXT4 are simulated. PSE is disabled by default in Android.

4.2 Micro-benchmark Tests

In sequential write/read tests, AndroSH exercises target file systems by writing to and reading from a test file with sequentially increasing offsets. As usual, the test file is

100MB large. Due to the limitation of shell script, direct I/O of EXT4 is not enabled, so page cache effect should be taken into account.

Figure 27 shows the sequential write throughput results. SCMFS has higher throughput than YAFFS2 and EXT4 in sequential write tests because of three main reasons: 1) SCMFS is smaller and faster than the other two file systems; 2) SCMFS's pre-allocation reduces number of physical space allocation operations; 3) page cache causes overhead to YAFFS2 and EXT4.

The result of sequential read in Figure 28 shows that SCMFS has higher or equal throughput to the other two file systems across all record lengths. There is no obvious difference of throughput at some record lengths because all three file systems are using memory devices in the test. Three file systems don't have as much gap as the gap in sequential write as the fact that page cache brings benefit to YAFFS2 and EXT4. Results of sequential write sequential read operations are not compared here because different number of iterations are used by them to ensure enough duration of execution.

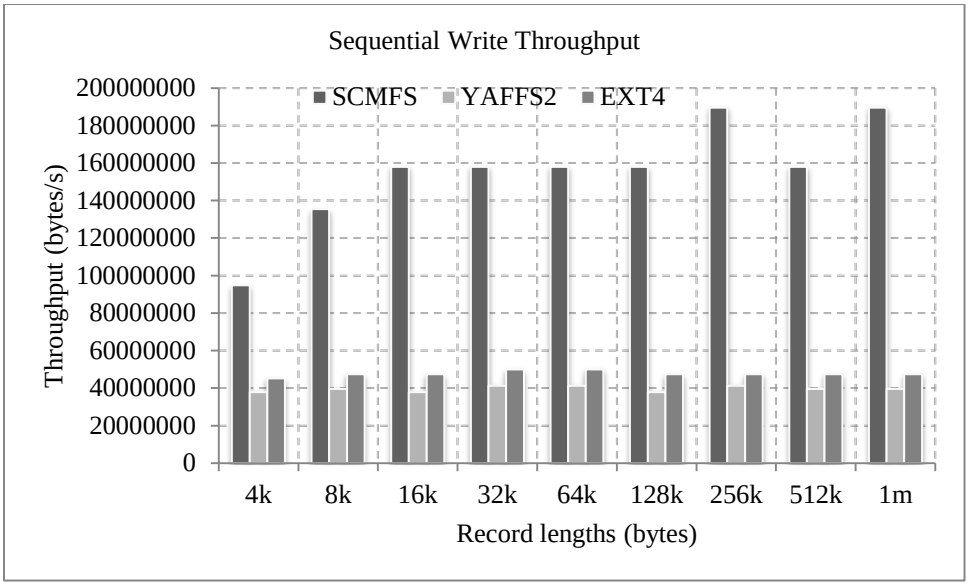


Figure 27: Sequential write throughput on Android.

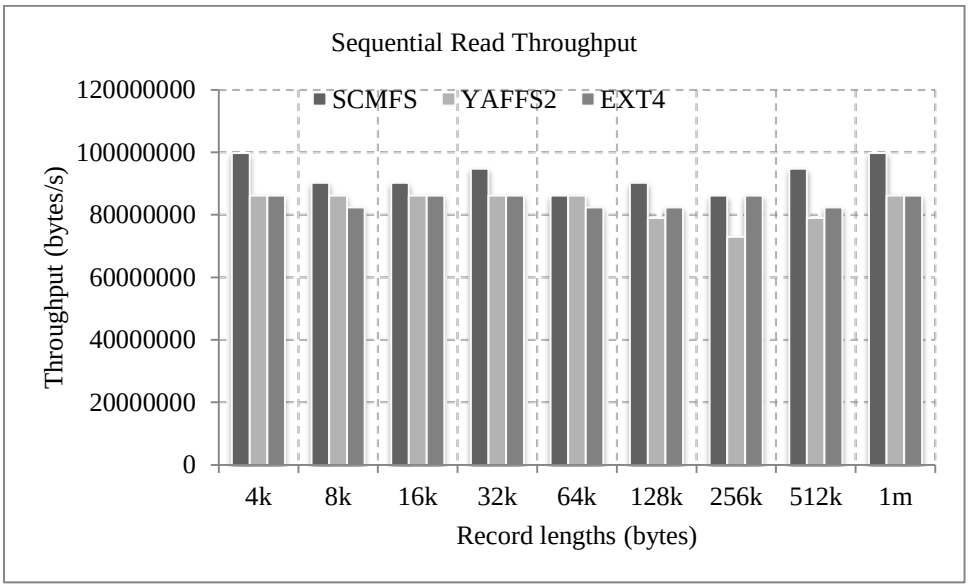


Figure 28: Sequential read throughput on Android.

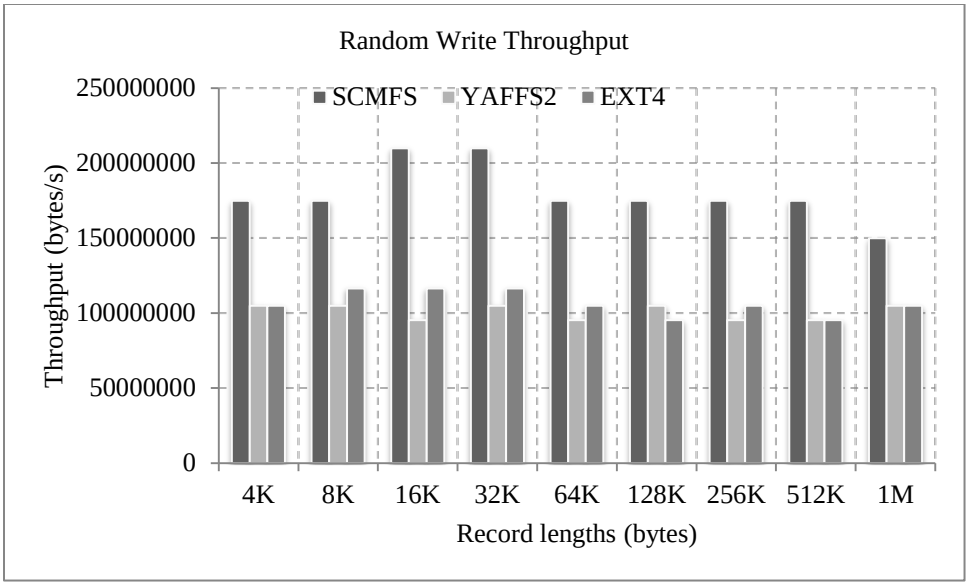


Figure 29: Random write throughput on Android.

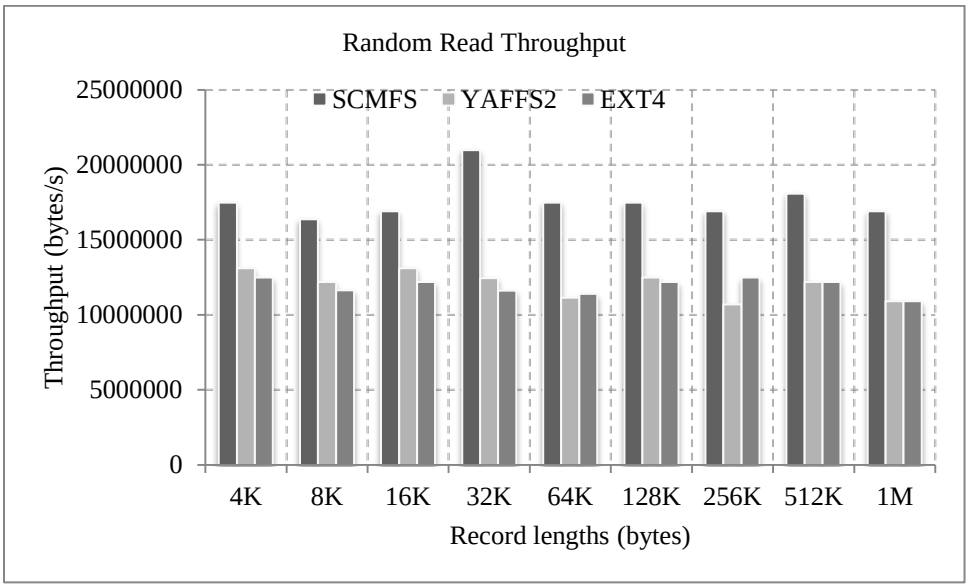


Figure 30: Random read throughput on Android.

And then, in random write/read operations, AndroSH writes to and reads from the test file at random offsets. SCMFS shows higher throughput performance in both random write and random read tests shown in Figure 29 and 30.

As discussed before, SCMFS has a simple design, it stores files and subdirectories in a singly linked list, thus the lookup time complexity is O(n). EXT4 uses a HTree data structure to index all the subdirectories within one directory. HTree has higher lookup efficiency. Directory efficiency test of AndroSH measures directory lookup efficiency of target file systems from three configurations: 1) 50 files are inside the root directory and no subdirectory is created; 2) 50 subdirectories are created under the root directory and each contains one file; 3) 3 levels of subdirectories are created under the root directory in a hierarchical structure: the root directory contains 2 subdirectories each of which has 5 subdirectories with 5 more subdirectories inside each, and the 50 leaf subdirectories have one file inside. AndroSH reads from target file systems with these three configurations. Only read workload is used to exercise lookup efficiency because write workload has other factors affecting its performance.

In the read result shown in Figure 31, without subdirectory, SCMFS is slightly better which is consistent to the sequential read result. With subdirectories, all the file systems have higher throughput than the case of no subdirectory, because directory paths are cached in dentry [28] cache in VFS which enhances lookup efficiency. With HTree index, EXT4 is faster than SCMFS and YAFFS2 in the case of flat directory. However, with hierarchical subdirectory structure, EXT4 is not outstanding because the numbers of subdirectories of each level are not large enough to make the HTree effect obvious.

Since HTree structure shows advantage of enhancing lookup speed, it can be considered to replace the linked list in SCMFS design in the future work.

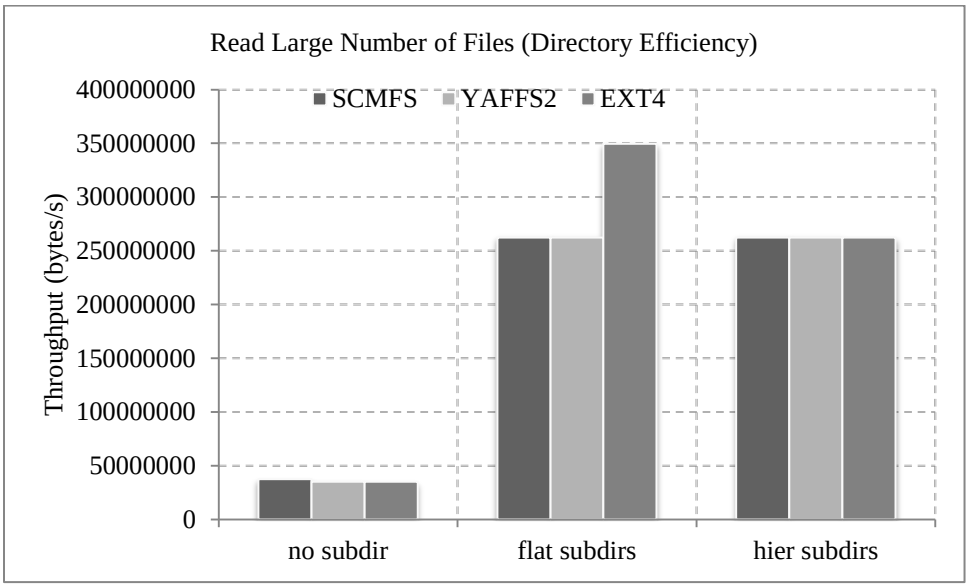


Figure 31: Directory lookup efficiency test.

4.3 User Data Workload Simulation

Other than files of applications, users tend to store four types of data on mobile devices: photos, music, audio podcasts, video podcasts. Compared to application data, the four types have more specific average sizes: 2MB (photo), 6MB (song), 25MB (audio podcast), 100MB (video podcast). With AndroSH, workloads simulating real usage models with the simulation of these four types of data are executed to measure data throughput and latency of file systems.

In real world usage, those user data are copied to mobile devices by downloading from network or PC. Some data transmission software is available on Android to download or upload data, for example, “Browser” and “Software Data Cable”. Android also has some data compressing software such as AndroZip which compresses or decompresses user data files. Due to the fact that these applications by design access */data* and */sdcard* partitions, and these two partitions only support block device attached to I/O bus by design, several test cases are developed in AndroSH to mimic the workloads generated by these Android applications for test purposes.

Downloading/synchronizing data are simulated in four test cases: 1) copy large amount of same type of data to the same directory, 2) copy random data into hierarchical directories, 3) copy one large file and compress it, 4) copy random data into the same directory and compress all into one file.

1) Copy same type of data into the same directory

This script measures the throughput of each file system by copying simulated user data files to the root directory of the file system. It simulates the situation that users usually synchronize many pieces of photos from computer to one directory in an internal partition on the mobile device, or download music/video from network to one folder in an internal partition on the mobile device.

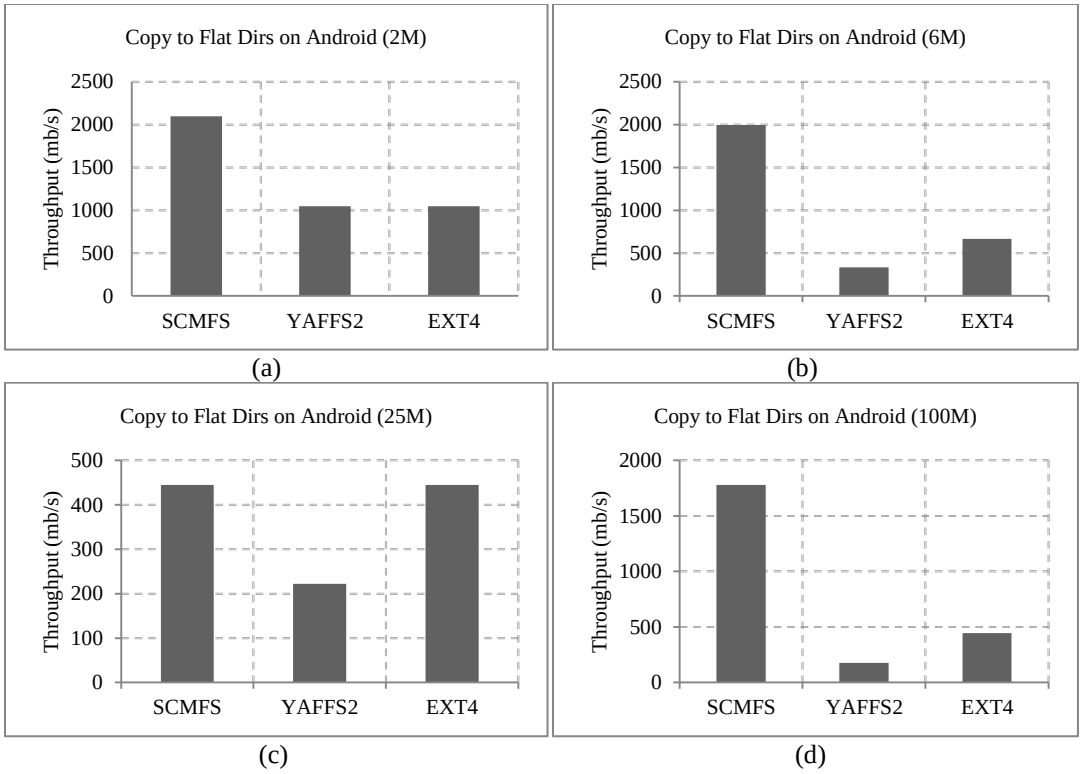


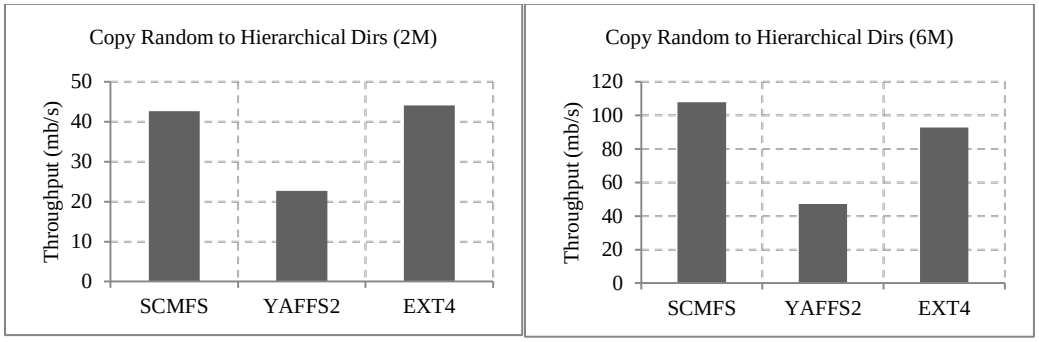
Figure 32: Copy to flat directory on Android. (a) photos (b) music files (c) audio podcasts (d) video podcasts.

Memory file systems generally run fast. To get certain time duration of operation, the number of iterations is changed when testing files of different sizes. For instance, testing larger file takes less number of iterations. Since the purpose is to find out the difference between file systems, and not between different data types, performance results of different data sets are not compared. For instance, in Figure 32 (a) and (b), the copy operation runs more iterations than (c) and (d). As shown in the results, SCMFS has higher throughput than the other two file systems because of its smaller instruction footprint and pre-allocation in most cases.

2) Copy random data into hierarchical directories

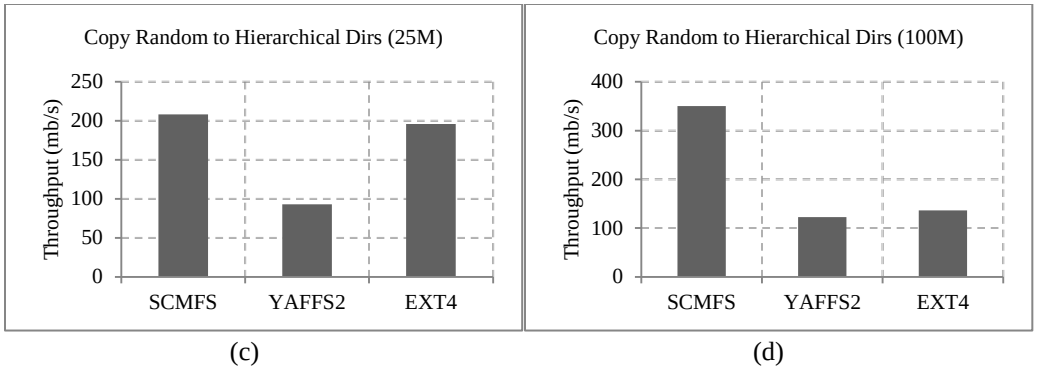
This script simulates some file manager software on Android. It exercises throughput and path lookup ability of file system. Every test case generates file sizes 0~maxium (for e.g. 2MB, 6MB, 25MB or 100MB).

Except in Figure 33(d), EXT4 performs much better than the previous test and shows only slightly lower throughput than SCMFS. This test exercises path lookup efficiency while measuring random file writing ability. EXT4 has efficient HTree data structure to store subdirectories, thus its path lookup efficiency is high. When it comes to test files as large as 100MB, EXT4 appears to be much worse than SCMFS because page cache buffers writes and enlarges latency, and also because SCMFS has better pre-allocation mechanism.



(a) (b)

Figure 33: Copy random files to hierarchical directories on Android. (a) photos (b) music files (c) audio podcasts (d) video podcasts.

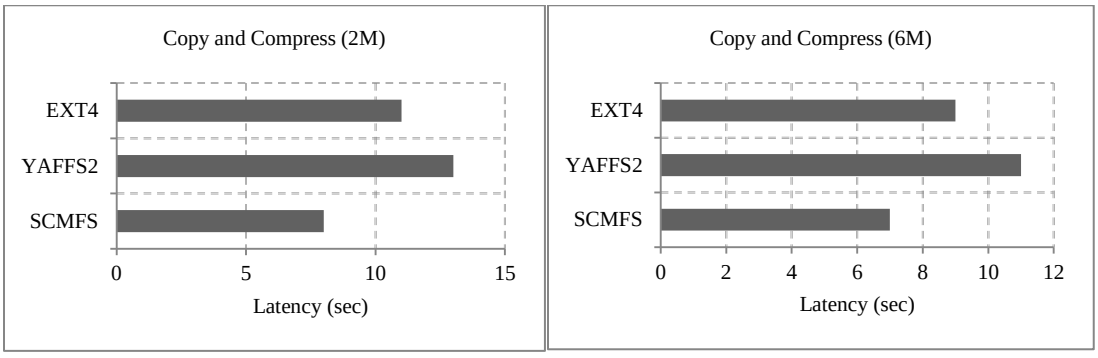


(c) (d)

Figure 33: Continued.

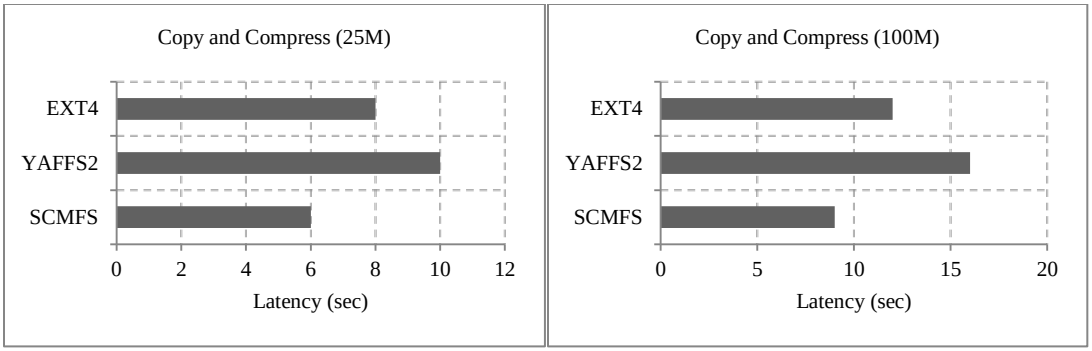
3) Copy and compress

This script copies one large file to the file system and compresses it into a tar.gz file, and then repeats these steps several times to ensure sufficient time duration for accurate measurements.



(a) (b)

Figure 34: Copy and compress file. (a) photos (b) music files (c) audio podcasts (d) video podcasts.



(c) (d)

Figure 34: Continued.

Figure 34 shows with all four types of simulated user data, SCMFS can finish copy and compress operations the fastest.

4) Copy random files and compress them

This script simulates file-compressing software on Android, such as AndroZip. It copies 50 random size files from 4k to 6M into one directory and compresses them into one tar.gz file.

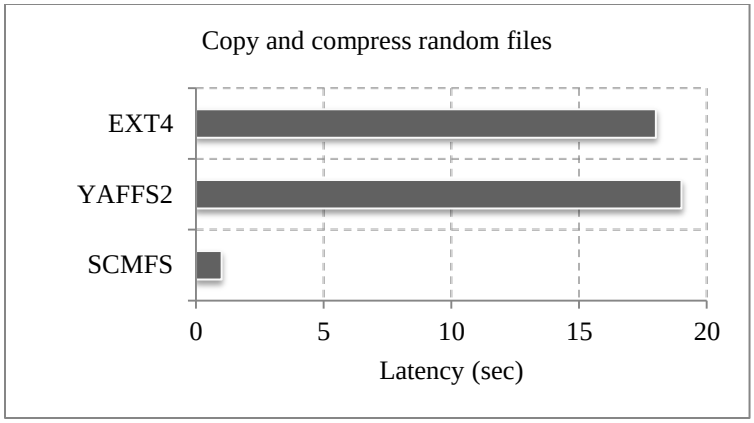


Figure 35: Copy and compress random files.

Figure 35 shows that the performance of SCMFS is significantly better compared to YAFFS2 and EXT4. In Android, SCMFS and EXT4 use 4KB page size, and YAFFS2 uses 2KB page size, so SCMFS does not have disadvantage of TLB misses. When executing the compressing process, the target file system competes in using TLB with the user process thus TLB misses can have a significant impact on latency. As one important reason, since YAFFS2 has a smaller page size, its TLB miss is inevitably higher, and its execution latency is also higher. There are some other possible reasons leading to the longer latency of YAFFS2, such as buffering small writes until they are chunk aligned, or garbage collecting old chunks for rewriting. EXT4 has the overhead of accessing generic block layer hence its latency is higher than SCMFS.

4.4 Application Performance Evaluation

4.4.1 Methodology

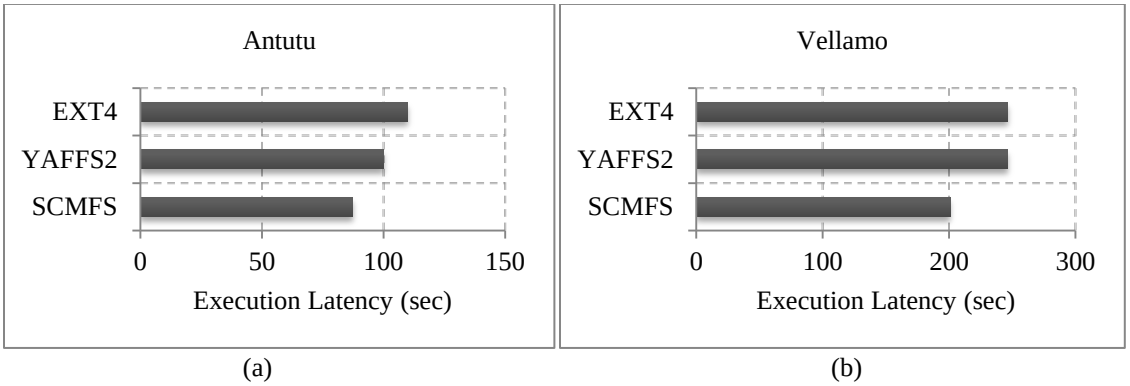
In order to evaluate the performance of the three file systems in real application scenarios, two popular Android benchmarks such as Antutu and Vellamo are used to measure file system's latency.

All applications store data in their corresponding directories under */data/data/*. Most applications have cache and databases directories under their own folder. Cache stores all the temporary data of the application. Databases store SQLite data per application. Moving cache or databases to different target file systems can change the performance of the applications. Page cache is not considered in SCMFS's design because PCM is fast enough and hence page cache is not necessary. Consequently SCMFS cannot handle SQLite transactions because the transactions require a buffer. Owning to this particular reason, only cache of each benchmark application is moved to target file systems in the test.

The methodology is to mount partition with target file system, and create a cache directory for application such as */fs_scmfs/cache* and then create its symbolic link */data/data/<applicationname>/cache*. While running the application, temporary data is generated into the cache in the target file system. The faster the target file system, the lower the execution latency of the application.

4.4.2 Application Execution Performance

Antutu stresses CPU, memory, SQLite database and the file system. In the file system tests, it stresses */data* and */sdcard* partitions. These two partitions cannot use SCMFS because they only support either generic block disk or NAND flash disk, so only RAMDISK or NAND simulator works for it. Vellamo is a web browser benchmark. They both generate certain amount of temporary data while executing.



(a) (b)

Figure 36: Application execution latency. (a) Antutu, (b) Vellamo.

The result depends on how many accesses the applications generate to their cache directories. A benchmark frequently write to and read from cache will certainly have different results from a benchmark rarely doing so. As the results show in Figure 36, SCMFS serves the two applications as a faster cache than the other two file systems.

4.5 Conclusion

All the above tests on Android are run several times, so the throughput and latency data tend to be stable. In the I/O capability micro-benchmark tests, SCMFS demonstrates higher throughput in sequential write/read and random write/read tests than YAFFS2 and EXT4. In the user data usage model simulation tests, SCMFS also demonstrates higher throughput and lower latency than the other two file systems. When used to store cache data of two popular Android benchmarks, SCMFS reduces the execution time more than the other two competitors. In conclusion, SCMFS shows obvious performance advantage because of its simple design and pre-allocation mechanism, therefore it is a good option for next generation mobile products based on PCM.

5. RELATED WORK

5.1 Non-volatile Byte-addressable Memory File System

BPFS [29] is proposed as a file system designed for non-volatile byte-addressable memory, which uses shadow paging techniques to provide fast and consistent updates. This file system also requires architectural enhancements to provide new interfaces for enforcing a flexible level of write ordering. SCMFS aims to simplify the design and eliminate the unnecessary overheads to improve performance.

DFS [30] incorporates the functionality of block management in the device driver and firmware to simplify the file system, and also keeps the files contiguous in a huge address space. It is designed for a PCIe based SSD device by FusionIo, and relies on specific features in the hardware.

5.2 Exploiting Huge-pages to Reduce TLB Misses

As [31] analyzes, standard x86_64 pages are 4KB, so there is a single 12-bit page offset. The remainder of the 48-bit virtual address is divided into four 9-bit indices, which are used to select entries from the four levels of the page table. In Linux kernel, the four levels of the x86_64 page table are named L4(PGD), L3(PUD), L2(PMD), L1(PTE) [5]. To support 2MB page size, page offset is enlarged to 21-bit, so the first level (PTE) is eliminated from kernel page table in order to present offset with more bits. In our research work, macros of two page sizes are defined accordingly. Our test machine has Intel Core i5 processor which uses separate TLBs for small page tables and

large page tables [32]. 4K pages and 2M pages cannot co-exist in the same page table, but their corresponding page tables can be cached in different TLBs. In our implementation, small pages are allocated to a file first until the file reaches a certain size, then huge-page is allocated to it and from then onwards only huge-pages are allocated to the file.

5.3 Android Storage Benchmark

Design of Androbench, a good file system benchmark of Android, is introduced by Je-Min Kim et.al [33]. This paper also explains the idea to stress internal */data* and external */sdcard* partitions. In our approach, instead of using the existing benchmarks, we developed a script based benchmark to exercise file systems with micro-benchmark workloads and user data usage model simulation workloads.

Hyojun Kim et.al [21] provides great details of using Android adb to transmit data to mobile device, and using reverse tethering to stress mobile device by network traffic. They mount RAMDISK, internal flash and external SD card to */data* and */sdcard*, the two most frequently used partitions by user applications, Dalvik VM and SQLite to compare performance among different storage devices. SCMFS cannot be mounted on */data* and */sdcard*, because these two partitions only support generic block device or NAND flash block device by design. Instead of trying to mount SCMFS on */data* and */sdcard* partitions, we change permission of */data* partition to fully accessible, create mount point under */data*, and mount SCMFS to it. Most applications have a cache directory under their application directories under */data/data*. Our methodology is to

create symbolic link of a cache folder created in target file system partition and put it in */data/data/<applicationname>/* to replace the cache of the application.

Two typical types of file system performance benchmarks: micro-benchmark and macro-benchmark are introduced and analyzed by Avishay Traeger et.al [9]. Based on 9 years of study of file system and storage benchmarking, this paper provides sufficient analysis of the usage of various benchmarks, and reveals their pros and cons. We choose benchmark tools by studying the analysis of this paper. IOZONE and POSTMARK are good micro-benchmarks for individual storage file systems. FILEBENCH supports both micro-benchmarking and macro-benchmarking. Some script based benchmarks, e.g. LFS-SH[9], SPEC SDM[34], give us idea of how to evaluate Android file system performance with command line utilities in Android Busybox.

6. CONCLUSION AND FUTURE WORK

Page size is an important influencing factor on memory file system's performance. Employing huge-pages in SCMFS can reduce TLB misses, and reasonably early migration to huge-pages can further reduce TLB misses and enhance throughput, and also control efficiency of file system utilization. Small-page version of SCMFS supports operations of large number of random files better than huge-page version, thus it is more suitable for Android. SCMFS outperforms other mainstream Android internal flash file systems because of its small instruction set and pre-allocation mechanism.

In the future, to further enhance SCMFS, more efficient data structures for managing files and directories can be considered into design to reduce lookup latency. A user-friendly Java-based Android file system benchmark which is compatible with storage class memory file systems can be developed. It is also possible that SCMFS will be applied to a distributed computing environment on account of the fact that some Storage Attached Network (SAN) products are built on DRAM and NAND flash nowadays and they will probably be replaced by SCM in the near future, so expanding SCMFS to support distributed environments is also valuable to future usage models.

REFERENCES

[1] X. Wu and A.L.N. Reddy, "SCMFS: A file system for Storage Class Memory," in *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Anaglsis*, 2011, pp. 1498-1503.

[2] M. Feldman, "Researchers Challange NAND Flash with Phase Change Memory," Available on Jun 2011 from http://www.hpcwire.com/hpcwire/2011-06-06/researchers_challenge_nand_flash_with_phase_change_memory.html, 2011.

[3] C.Manning, "How YAFFS Works," Available on July 2010 from http://www.dubeiko.com/development/FileSystems/YAFFS/HowYaffsWorks.pdf, 2010.

[4] X. Wu, "Storage Systems for Non-volatile Memory Devices," Ph.D. dissertation, Texas A&M University, College Station, TX, 2011.

[5] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ: Pearson Education, Inc., 2004, pp.15-34.

[6] D. P. Bovet and M. Cesati. "Memory Addressing" in *Understanding the Linux Kernel*, 3rd ed.. D.Kelly, Ed. Sebastopol, CA: O'Reilly Media, Inc., 2005, pp.69-70.

[7] R. H. Savvedra and A. J. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes," *IEEE Transactions on Computers*, vol.

44, no. 10, pp. 1223-1235, 1995.

[8] "IOzone File System Benchmark," Available on July 2011 from http://www.iozone.org, 2011.

[9] A. Traeger, E. Zadok, N. Joukov and C. P. Wright, "A Nine Year Study of File System and Storage Benchmarking," *ACM Transactions on Storage*, vol. 4, no. 2, article 5, May 2008.

[10] Aleph One, Ltd., "YAFFS 2 Specification and Development Notes," Available from http://www.yaffs.net/yaffs-2-specification-and-development-notes.

[11] C.Lee and S.Lim, "Caching and Deferred Write of Metadata for Yaffs2 Flash File System," presented at Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference, Melbourne, Australia, 2011.

[12] Wookey, "YAFFS - A NAND Flash Filesystem," presented at Embedded Linux Conference, Linz, Austria, 2007.

[13] M.Cao, S.Bhattacharya and T.Tso, "Ext4: The Next Generation of Ext2/3 Filesystem," presented at Linux Storage and Filesystem Workshop, San Jose, CA, 2007.

[14] M.Cao, "Directory Indexing," Available on July 2005 from http://ext2.sourceforge.net/2005-ols/paper-html/node3.html, 2005.

[15] N. Kaiser, "EXT4 features," Available on May 2011 from http://kernelnewbies.org/Ext4, 2011.

[16] Semihalf, "NAND Flash Framework Introduction," Available from

http://wiki.freebsd.org/NAND#NAND_chip_simulator_.28NANDsim.29.

[17] N.C. Bane, "How to Incorporate YAFFS to Linux," Available from http://www.yaffs.net/howto-incorporate-yaffs-linux.

[18] J.Katcher, "PostMark: A New File System Benchmark," Available on October 1997 from http://unix4.com/p/postmark-a-new-file-system-benchmark-w23.html, 1997.

[19] P. Sehgal, V. Tarasov and E. Zadok, "Evaluating Performance and Energy in File System Server Workloads," in *Proc. of the 8th USENIX conference on File and storage technologies*, 2010, pp.253-266.

[20] F. Maker and Y.-h. Chan, "A Survey on Android vs Linux," Available on Feb 2011 from http://handycodeworks.com/wp-content/uploads/2011/02/linux_versus_android.pdf, 2011.

[21] H. Kim, N. Agrawal and C. Ungureanu, "Revisiting Storage for Smartphones," in *Proc. of 10th USENIX Conference on File and Storage Technologies*, 2012, pp.209-226.

[22] "Rooting (Android OS)," Available on Feb 2012 from http://en.wikipedia.org/wiki/Rooting_(Android_OS), 2012.

[23] C.W. Huang, "Android x86 - Porting Android to x86," Available on Dec 2011 from http://www.android-x86.org/getsourcecode, 2011.

[24] "ARM Overview," Availale on Apr 2012 from http://wiki.osdev.org/ARM_Overview, 2012.

[25] A. Shah, "Feature: High Memory In The Linux Kernel," Available on Feb 2004 from http://kerneltrap.org/node/2450, 2004.

[26] H.F. Lee, "X86_64 Virtual Memory Map," Available on March 2010 from http://lkml.indiana.edu/hypermail/linux/kernel/1003.3/00582.html, 2010.

[27] W. Mauerer, "Locking and Interprocess Communication" in *Professional Linux Kernel Architecture*, 1st ed.. C.Long, Ed. Indianapolis, IN: Wiley Publising, Inc., 2008, pp.351.

[28] D. P. Bovet and M. Cesati, "The Virtual Filesystem" in *Understanding the Linux Kernel*, 3rd ed., D.Kelly, Ed. Sebastopol, CA: O'Reilly Media, Inc., 2005, pp.457.

[29] J. Condit, E. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger and D. Coetzee, "Better I/O through Byte-addressable, Persistent Memory," in *Proc. of the Symposium on Operating Systems Principles*, 2009, pp. 133–146.

[30] W. K. Josephson, L. A. Bongo, D. Flynn and K. Li, "DFS: A File System for Virtualized Flash Storage," in *Proc. of the USENIX Conference on File and Storage Technologies*, vol. 6, 2010, pp. 85–100.

[31] T. W. Barr, "Exploiting Address Space Contiguity to Accelerate TLB Miss Handling," M.S. thesis, Rice University, Houston, TX, 2010.

[32] Cpu world, "Intel Core i5 Processor," Available on May 2012 from http://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5-750%20BV80605001911AP%20(BX80605I5750%20-

%20BXC80605I5750).html, 2012.

[33] J.-M. Kim and J.-S. Kim, "AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices," in *Proc. of Advances in Intelligent and Soft Computing 2012*, vol. 133/2012, 2012, pp. 667-674.

[34] "SPEC SDM Suite," Available on June 2004 from www.spec.org/osg/sdm91/, 2004.