AFPT: Accelerating Read Performance of In-Memory File System Through Adaptive File Page Table

Bingde Cui, and Huansheng Zhang

***Abstract*—Emerging non-volatile memory (NVM) technologies are expected to revolutionize storage systems by providing cheap, persistent and fast data accesses through memory bus interface. In oder to fully exploit NVM, many in-memory file systems are proposed to achieve excellent performance and strong consistency. Besides, to mitigate the read-write asymmetric problem of NVM, many optimization strategies are designed to hide the long write latency to NVM in critical path of file operations, such as path resolution. However, we find that the index structure of state-of-the-art in-memory file systems cannot provide fast read performance in various use scenarios.**

In this paper, we propose *Adaptive File Page Table (AFPT)*, a novel index scheme that combines software search and MMU mapping to provide excellent read performance for different workloads. For small requests, software search routines are used to locate data pages by traversing the file index structure. For large requests, we allocate a continuous address space and build file page table to utilize hardware MMU for address translation. A *Cost Model* is proposed to determine when to build page table for a file. This model is 1.38-competitive against optimal solution. We implement *AFPT* in PMFS and NOVA and evaluate the performance with micro-benchmarks and application workloads. The experimental results show that *AFPT* improves file system performance by up to 55.62% and 41.78% for NOVA and PMFS, respectively.

***Index Terms*—In-Memory File System, File Page Table, Virtual Address Space, Performance Optimization.**

I. Introduction

Emerging byte-addressable, fast non-volatile memory (NVM) technologies, such as PCM [1]–[3], ReRAM [4]–[6], memristor [7]–[9] are promised to revolutionize storage system by providing persistent and instant accesses to storage-class memory systems. Since NVMs can be directly attached to memory bus and accessed by CPU through load/store instructions, many approaches are proposed to integrate NVMs into existing software stack [10]–[13].

In order to fully exploit NVM potential, state-of-the-art in-memory file systems, such as PMFS [14], SIMFS [15], NOVA [16] and BPFS [17], are designed and optimized to achieve excellent performance and provide consistency guarantees. Besides, the asymmetric read-write performance of NVM calls for optimization strategies in critical path of file operations, such as write and path name resolution [18]. Previous research efforts are based on the fact the NVM has longer write latency and has similar read latency compared with DRAM. Thus, many optimization techniques are proposed to hide the long latency of NVM write [18], [19], and few efforts are devoted to optimize read performance of in-memory file systems.

Unfortunately, we find that searching file index structure by software routines leads to suboptimal read performance. For in-memory file systems, software overhead of traversing the file index to locate data pages is identified as the dominating bottleneck for performance improvement. Even though SIMFS [15] and SCMFS [20] demonstrates the advantages of using MMU along with file page table and virtual address space to speedup address translation, file page table is not efficiently utilized by state-of-the-art in-memory file systems. On one hand, SIMFS builds file page table for all files, not considering file size and access pattern. Thus, SIMFS exhibits considerable space and runtime overhead for small files [21]. On the other hand, NOVA and PMFS does not utilize file page table concept for large files, which leads to slower read performance than SIMFS [15].

The problem lies in the fact that fixed index structure cannot meet the demands of various workloads. To address this problem, we propose *Adaptive File Page Table (AFPT)*, a novel index scheme for in-memory file systems. The goal of *AFPT* is to combine software search and file page table to provide fast read speed in various use cases, regardless of access pattern. Specifically, *AFPT* keeps track of file access pattern using history information and utilizes a pre-defined threshold to determine whether to build file page table. For small requests, software search routine is used to locate data pages. When the access size exceeds the threshold, a continuous address space is allocated and file page table is used to serve incoming requests. We build a cost model to determine the appropriate threshold according to system state. We implement *AFPT* in PMFS and NOVA, extensive experiments show that *AFPT* can improve performance by up to 55.62%.

In summary, this paper makes the following contributions:

- We reveal the inefficiency problem of file index structure of state-of-the-art in-memory file systems. Based on our experimental results, we find that software search routines are effective for small requests and MMU is efficient for large data accesses.
- We propose a method to identify access pattern of files. By consulting access history, we get the total amount of accessed data and whether a file is one-shot access or continuous access.
- We propose *Adaptive Switching Algorithm* to dynamically switching between software search and file page table according to file access pattern.
- We build a *Cost Model* to efficiently determine threshold used in *Adaptive Switching Algorithm*. This model can

Manuscript received June 22, 2018; revised January 29, 2019.

Bingde Cui and Huansheng Zhang are with the Department of Computer Science, Hebei University of Water Resources and Electric Engineering, 061001, China. e-mail: cuibd1975@163.com.

Huansheng Zhang is the corresponding author(plsyyq_8358@163.com).

achieve 1.38-competitiveness against optimal solution.

The remainder of this paper is organized as follows. Section II presents background and discusses challenges of index structure of in-memory file systems. Section III presents our detailed design of *AFPT*. Section IV presents the evaluation results of *AFPT*. We discuss related work in Section V and this paper is concluded in Section VI.

II. Background & Motivation

In this section, we first present the system architecture of in-memory file system. Then we discuss the file index structure commonly used to organize file data blocks. Finally, we describe the challenges of current file index structure design.

A. System Architecture for In-Memory File System

Most in-memory file systems are designed to take the byte-addressability of NVM. NVM can be directly attached to memory bus and CPU can access NVM through load/store instructions. Due to the limited write endurance of NVM, the system can deploy DRAM alongside with NVM to form a unified address space.

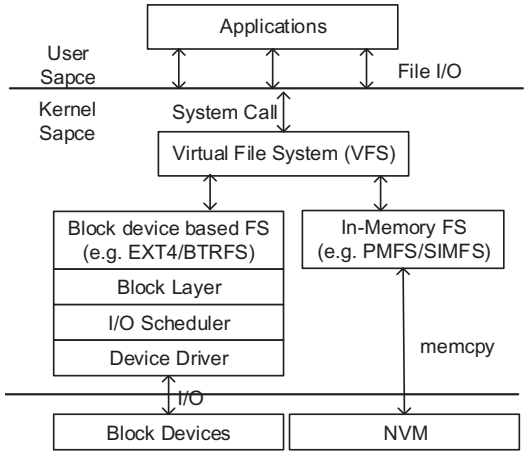


Fig. 1: Architecture overview of block device based file system and in-memory file system.

Figure 1 shows the architectural overview of in-memory file system and traditional block device based file systems. Applications issue file I/O operations to Virtual File System VFS) through system call interfaces. The VFS propagates the I/O requests to the underlying file systems. For block device based file system, the I/O requests traverse block layer, I/O scheduler and device driver before reaching block devices (disks). On the other hand, for in-memory file system, the software stack is significantly shortened. In-memory file system can directly copy contents between user buffer and NVM.

Compared with traditional block based file system, the software stack of in-memory file system may impose more overhead to the system performance. Because the access latency of block device is three orders of magnitude slower than that of NVM. Thus, many optimization strategies are proposed to hide the latency in the critical path of in-memory file systems [18], [19].

B. File Index Structures of In-Memory File System

File index structure is of great importance for in-memory file system. Since each I/O request needs to consult the index structure to copy contents to/from NVM. Accessing NVM and transferring data is fast. The efficiency of index structure can significantly affect file system performance exposed to users. Most of in-memory file systems adopt tree-like structures for file index. For example, PMFS [14], NOVA [16] and HiNFS [19] use B-Tree. SIMFS [15] uses page table, which is radix tree by nature. PRAMFS [22], a small in-memory file system for embedded system, utilizes two-dimensional arrays for index structure.

These index structures differ significantly in how the indexes are traversed to get the contents of file data. We category the file index structure into two types: **software search** and **MMU mapping**.

Software Search. In this approach, the data blocks of a file are accessed one by one. For example, to access data blocks stored in a typical B-Tree. The root node of the tree stores many pointers to data blocks. Software routines must get the first pointer and copy the contents to user buffer and then fetch the next pointer and copy the contents. When accessing large file, the overhead of software search can be the bottleneck for performance improvement.

MMU Mapping. MMU is designed to accelerate address translation in operating system. The virtual address space of process is associated with page table to enable virtual memory and isolation. MMU is involved in each address access to get the physical location of virtual address. If file index mimics the structure of page table. MMU can be exploited to fast traverse page table and locate data blocks. In order to use MMU, a file virtual address space is required. SCMFS [20] and SIMFS [15] take advantages of virtual address and file page table to achieve excellent performance.

C. Challenges for File Index Structure Design

Both SCMFS [20] and SIMFS [15] demonstrate the advantages of using MMU mapping. The essential step of using MMU is to allocate virtual address space and build file page table for the associated virtual address. However, building file page table comes with certain cost, especially for small files. In this section, we analyze the pros and cons of building file page table and discuss how to optimize the index structure to achieve stable performance in all use cases.

Due to lack of source code of SIMFS, we implement file page table and virtual address space concept in PMFS. The B-Tree structure in PMFS uses 4 KB blocks, which can also be used to construct page table entries for a specific virtual memory area.

We first present the overhead breakdown of read operations when different approaches (software search and file page table) are used for data blocks lookup. For software search, the overhead consists of three parts: *system call, software search* and *memcpy*. For file page table, the overhead consists of *system call, building page table* and *memcpy*.

Based on the results of figure 2 and figure 3, we can get the following conclusions:

1) For both approaches, the time for reading/memcpy increases as the request size grows.

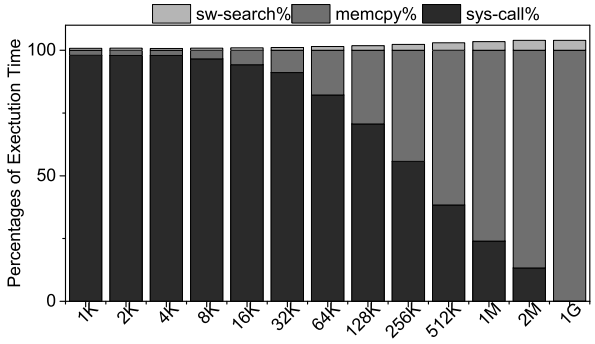


Fig. 2: Execution time breakdown of software search when reading different amount of data.

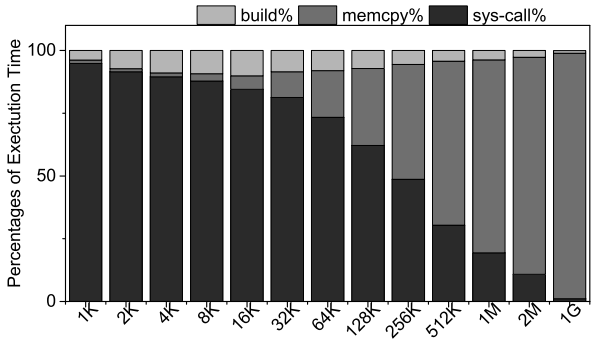


Fig. 3: Execution time breakdown of file page table when reading different amount of data.

2) The system call imposes considerable overhead for small requests (less than 16KB), which can reach up to more than 90% in both cases.
3) For small request sizes, the time for building page table is larger than that of software search. The overhead of building page table decreases as the request size grows.

Based on the above analysis, we argue that it is necessary to design an efficient policy to take full advantages of file page table and software search. The policy should take into account the file size and request size.

III. AFPT Design

We design an adaptive switching scheme that aims to combine software search and file page table to provide fast file accesses in various use cases. The proposed strategy is called *Adaptive File Page Table (AFPT)*. In this section, we first present design overview of AFPT. Then we discuss in detail the major concepts of AFPT.

A. Overview

The design of the proposed *Adaptive File Page Table* scheme is based on three observations:

1) Software search is efficient for small requests, while file page table is efficient for large requests.
2) Efficiency is closely related to file access pattern. Specifically, continuous access to the same file benefits most with file page table and virtual address space than one-shot access.
3) How and when to build file page table should be determined by system state. It is necessary to build a cost model to make intelligent decision.

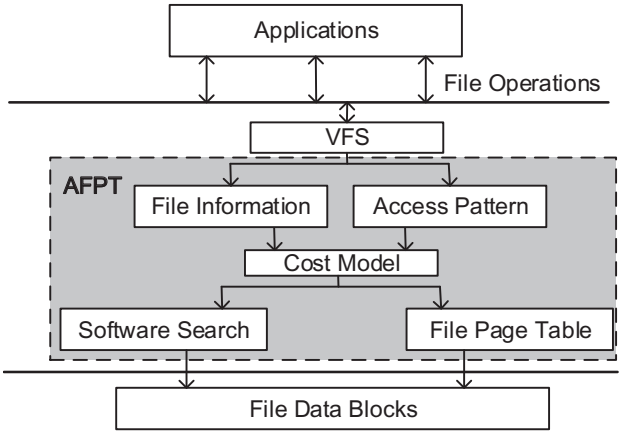


Fig. 4: Overview of Adaptive File Page Table (AFPT).

Figure 4 shows the overview of AFPT. We make the following design decisions:

Keep track of access pattern for every opened file. The benefit for building file page table varies with file size and file access patterns. On one hand, the file size can provide immediate hint on whether it is worthwhile to build file page table. For example, if a file is small, the overhead of building page table can dominate the performance overhead. On the other hand, file access frequency and total request size are more accurate to model the access pattern. For example, if a big file is opened but only the first page is read and closed, i.e., one-shot access. There is no need to build file page table. In a typical map-reduce workload, the input data is divided into 64 MB or 128MB blocks by HDFS and stored in local file systems. For such workloads, the whole file is continuously accessed and it can benefit from MMU mapping.

The access pattern of a file is updated in DRAM and it is written back to NVM when the file is closed. We assume the access pattern of a file is stable once it's identified. The access pattern does not require strong consistency, if it is lost in case of system crash, it can be rebuilt by history access information.

Combine software search and file page table. As discussed in Section II, software search and file page table both bear pros and cons. Neither approaches are efficient for all use scenarios. Thus, we use software search as default scheme and build file page table when continuous pattern is detected.

Build a cost model to determine when to build file page table. One challenge of combining software search and file page table is to determine which files and when to build file page table. If a file is accessed only a few pages, the overhead of building file page table may offset the benefit of using MMU and virtual address space. Thus, we build a cost model to determine the condition for building file page table.

B. Access Pattern Identification

In this section, we propose a scheme to identify file access pattern based on request history window. The essential concept is to record consecutive requests information, such as *operation type*, *file offset* and *request size*. If a *read/write* request is followed immediately by a *close* request, this file

is characterized as *One-Shot Access*. Otherwise, if many data pages are accessed before *close*, this file is characterized as *Continuous Access*.

We then discuss in detail how access pattern is identified. We propose an algorithm, called *Access Pattern Identification (API) Algorithm* to determine it. A file request is represented by *type*, *offset* and *size* pair, it is denoted by $R(op, offset, size)$. The *op* of a request can be *read, write* and *close*.

Algorithm III.1 API Algorithm

Input: R: an incoming request; T: threshold of data pages to determine whether a file has continuous access pattern; N_{total}: the total number of pages a file has been accessed before close.
Output: Access pattern of a file.

```
1: Pattern ← Unkown.
2: if R.op == close then
3:     if N_total < T then
4:         Pattern ← One-Shot Access.
5:     else
6:         Pattern ← Continuous Access.
7:     end if
8: end if
9: if R.op == read OR R.op == write then
10:    N_total ← N_total + R.size / PAGE_SIZE
11:    if N_total > T then
12:        Pattern ← Continuous Access.
13:    end if
14: end if
15: return Pattern.
```

Algorithm III.1 works by checking the total number of accessed pages (N_{total}) before *close* operation. If N_{total} is less than T, the file has *One-Shot Access* pattern. Otherwise, the file has *Continuous Access* pattern. In line 10, we calculate the total number of pages of request R by dividing request size. Note that we use total number of pages rather than total amount of accessed data as threshold, because each individual page needs to be searched by software routine from file index. We discuss how the threshold T is determined in Section III-D.

To improve the accuracy of access pattern identification and prediction, some intelligent algorithms can be utilized, such as neural networks and genetic algorithm [23]–[26], we plan to further improve the algorithm in our future work.

C. Adaptive Switching Between Software Search and File Page Table

In this section, we first present the structure change in inode in order to support two approaches for data accesses. Then, we discuss the algorithm for switching between software search and file page table.

To support adaptive switching between software search and MMU mapping, we add another pointer in inode to indicate whether a virtual address space is assigned to a file. The resulting structure of inode is depicted in figure 5.

In figure 5, the inode contains two pointers for file accesses. The default one is the pointer to the root node of file index tree. The file contents can be accessed by conventional software search routines. That is, to traverse the tree and copy contents from data blocks one by one. There is a new pointer, *VA*, which points to file page table. The page table entries are constructed with file data blocks. In this way, subsequent accesses to the file virtual address can be instantly served by MMU.

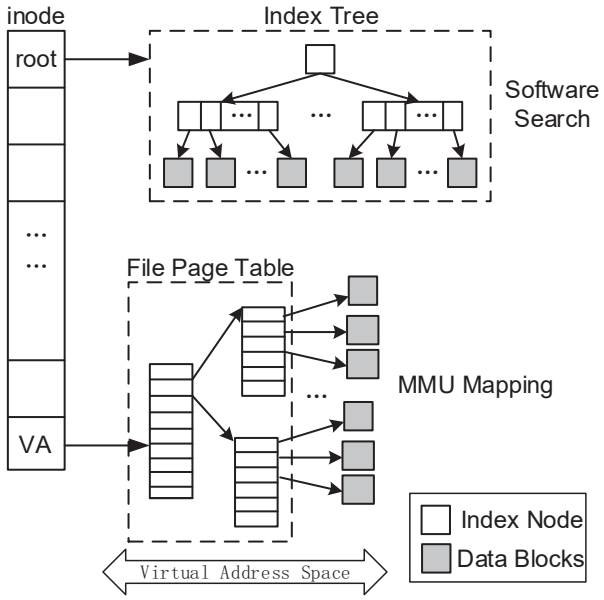


Fig. 5: Structure of inode when applying adaptive file page table scheme.

File page table can provide fast read/write performance. There is no page fault when reading file contents, since all the pages are pre-fault-in. Note that the data blocks are same in both the file index tree and file page table. There is no overhead of duplicating file data blocks. The physical address of data blocks are used to build page table. That is, the last level page table (PTE) contains pointers to data blocks. The same blocks appear in both index tree and file page table. The virtual address space and file page table just provides a new route for data accesses.

To support file growth, the virtual address space is allocated two times larger than the actual file size. If write requests are issued to the file and new data blocks are allocated. The new blocks are inserted to both the file index tree and file page table. When the file is closed, the virtual address space and file page table is released so that other processes can reuse the virtual address space.

The virtual address is not stored in NVM for two reasons. First, it occupies extra NVM space to store the file page table. Second, the virtual address space is dynamically allocated from operating system, the same address is not guaranteed to be valid after system reboot. Thus, we always allocate new virtual address space when it is determined that the file is about to build file page table.

We propose a strategy called *Adaptive Switching (AS)* to build file page table when required. *Adaptive Switching Strategy* tolerates limited number of software search operations to reduce the cost of building file page table.

Algorithm III.2 shows the major steps of *Adaptive Switching Strategy*. For each opened file, we keep a counter to indicate the total number of pages that are accessed by software search. If the counter is smaller than a pre-defined threshold T, file I/O requests are served by software search. On the other hand, once the counter exceeds the threshold, we allocate a virtual address space and build file page table for that file. The subsequent I/O requests are served by MMU mapping.

Algorithm III.2 *Adaptive Switching* Algorithm

Input: F: the accessed file; N: the total number of pages that are accessed by software search; T: the threshold for building file page table. R: request to file F.
Output: The desired access approach for file F.

1: **if** N is smaller than threshold T **then**
2: $N \leftarrow N + 1$.
3: Serve request R by software search routines.
4: **else**
5: Allocate virtual address space and build file page table for file F.
6: Serve request R by virtual address space.
7: **end if**

Note that the two schemes can work seamlessly to serve I/O requests. Because once virtual address space is built, the target blocks can be calculated by simply adding the request offset to the beginning virtual address.

As discussed in Section III-A, file page table is not the default method for file operations. Instead, we allocate virtual address space and build file page table only when certain conditions are met, such as the file has continuous access pattern and request size is large so that it is worthwhile to build file page table. It is straightforward to check the file size to decide whether it is necessary to build file page table. However, the decision of when and whether to build file page table by access pattern is particularly challenging, as it is unknown whether a file is to be accessed again after building file page table with considerable cost. To address this challenge, we propose a cost model to decide when to build file page table.

D. Cost Model

We propose a (1+ϵ)-competitive cost model to determine the threshold T used in Algorithm III.1 and Algorithm III.2. The cost model considers the cost for building file page table and the cost for software searching. ϵ is a small constant. We prove that in a given system, the cost model can achieve near optimal result. The goal of the cost model is to gain benefit from file operations that are served by software search at most T times.

We first present some notations used in the cost model. The cost for build file page table is denoted by C_{bt}, which includes the costs for allocating free blocks, allocating virtual address space and inserting pages to page table entries. The cost for software search is denoted by C_{sw}. The cost for MMU mapping is denoted by C_{mmu}.

We then give the detailed steps for building the cost model.

First, to gain benefit by directly serving I/O requests using software search, the accumulated cost of software searching T pages should satisfy

$$T \times C_{sw} \leq C_{bt}. \tag{1}$$

Second, the accumulated cost should not exceed the cost of building file page table. Thus,

$$C_{bt} \leq (T+1) \times C_{sw}. \tag{2}$$

Third, we analyze the impact of threshold T on the proposed *Adaptive Switching Strategy*. Since requests are initially served by software search, the cost of total file accesses can be represented as follows:

$$C_{total} = \begin{cases} C_{sw} \times i, & i \leq T \\ C_{sw} \times T + C_{bt} + C_{mmu} \times (i-T), & i > T. \end{cases} \tag{3}$$

where i is the total number of pages that the file has been accessed.

Now let us analyze the ideal solution for the two cases in equation 3. On one hand, if eventually $i \leq T$, the optimal approach to serve I/O requests is to use software search to find the total i file pages. On the other hand, if eventually $i > T$, the optimal approach should build file page table for the first I/O request. Thus, we get the cost of the optimal approach:

$$C_{optimal} = \begin{cases} C_{sw} \times i, & i \leq T \\ C_{bt} + C_{mmu} \times i, & i > T. \end{cases} \tag{4}$$

According to equations (3) and (4), the proposed *Adaptive Switching Strategy* can achieve optimal cost when $i \leq T$. When $i > T$, the ratio R of the solution generated by adaptive switching strategy against the optimal strategy is

$$\begin{aligned} R &= \frac{C_{sw} \times T + C_{bt} + C_{mmu} \times (i-T)}{C_{bt} + C_{mmu} \times i} \\ &= 1 + \frac{C_{sw} - C_{mmu}}{C_{bt} + C_{mmu} \times i} \times T. \end{aligned} \tag{5}$$

Thus, competitiveness ratio R is a monotone decreasing function of i. The minimum value of i in equation (5) is $T+1$, since $i > T$ and i must be integer. Therefore, the worst case is a request accesses $T+1$ blocks of a file. The cost of the worst case is $C_{sw} \times T + C_{bt} + C_{mmu}$. Hence,

$$\epsilon = \frac{C_{sw} - C_{mmu}}{C_{bt} + C_{mmu} \times (T+1)} \times T. \tag{6}$$

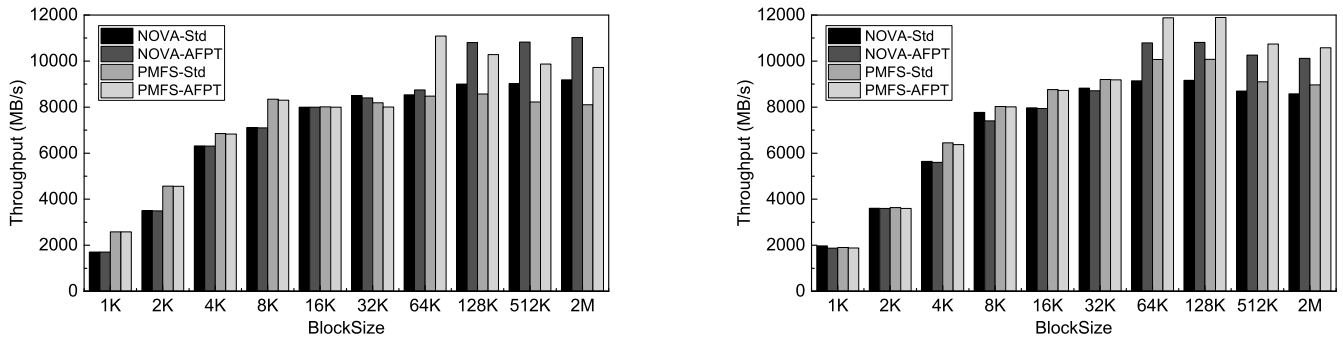
In a given storage system, ϵ is uniquely determined by C_{sw}, C_{mmu} and C_{bt}, which means ϵ is constant. For different CPU architecture and hardware platform, these values may vary. For example, in our experiment, we estimated that the cost of building a page table entry is 21600 ns, and reading a page by MMU takes 800 ns while software search takes 1800 ns. i.e., $C_{bt} = 21600, C_{sw} = 1800$ and $C_{mmu} = 800$. According to equation (1), the threshold T is set to 12. Using equation (6), we can get the ϵ is 0.38. Thus, the adaptive switching strategy can is 1.38-competitive against the optimal approach.

IV. Evaluation

In this section, we evaluate the performance of *Adaptive File Page Table* (AFPT) and answer the following two questions.

- How does AFPT perform against state-of-the-art index structure of in-memory file systems.
- How does AFPT behave in real-world applications and enterprise workloads.

We first describe the experimental setup and then evaluate AFPT with micro-benchmarks and application workloads.



(a) Read (b) Random Read

Fig. 6: Performance comparison of PMFS and NOVA when using standard file index structure and AFPT strategy.

A. Experimental Setup

We have implemented AFPT in PMFS [14] and NOVA [16] in linux kernel 4.4.4. The index structure (B-Tree) of PMFS and NOVA is extended with file page table. We add a new pointer in *inode* to indicate whether file page table and virtual address space is allocated for a file. The access pattern information is tracked and updated in each file operation.

The experiments are conducted on a 4-Core PC with Intel Core i5-7500@3.40GHZ CPU. The platform is equipped with 64GB DDR4 memory. The reported memory bandwidth by STREAM [27] benchmark is 13.5GB/s. We use *Persistent Memory Development Kit* [28] to emulate a persistent memory device and mount PMFS and NOVA on it. The threshold for building file page table is set to 12 as discussed in Section III-D. That is, once more than 48KB blocks of a file have been continuously accessed, we build a file page table for that file.

In the experiments, the original PMFS and NOVA is denoted by PMFS-Std and NOVA-Std. The file systems with AFPT strategy applied is called PMFS/NOVA-AFPT.

B. Micro-Benchmarks

We use FIO [29] to evaluate read and random read performance. The file size is 4GB. The block size varies from 1KB to 2MB. To avoid build file page table for small requests, we modify FIO so that a file is closed after each read operation. Thus, for block size which is less than 48KB, conventional software search routine is used.

Figure 6 shows the throughput of sequential and random read performance of NOVA and PMFS when using the default B-Tree index and AFPT strategy. We can get the following conclusions.

First, AFPT has similar performance with the standard B-Tree index for small requests. This is AFPT falls back to the conventional software routines for file operation when the request size is less than 48KB. For both PMFS and NOVA, the performance degradation caused by access pattern identification is less than 2% for small block sizes. This is because only limited information is recorded in each file operation, as discussed in Section III-B. In other words, small requests can reflect the overhead of AFPT and the result shows that it is negligible.

Second, AFPT can accelerate read and random read performance for large data accesses. The improvement for PMFS and NOVA can reach up to 21.15% and 19.34%, respectively. When request size exceeds 48KB, file page table is built and the read operations are served by continuous virtual address space, which is efficient for both read and random read operations. For random read, the address of the desired file data page can be calculated by simply adding the offset to the file virtual address space. In this way, no software search is involved and software overhead can be totally avoided.

C. Macro-Benchmarks

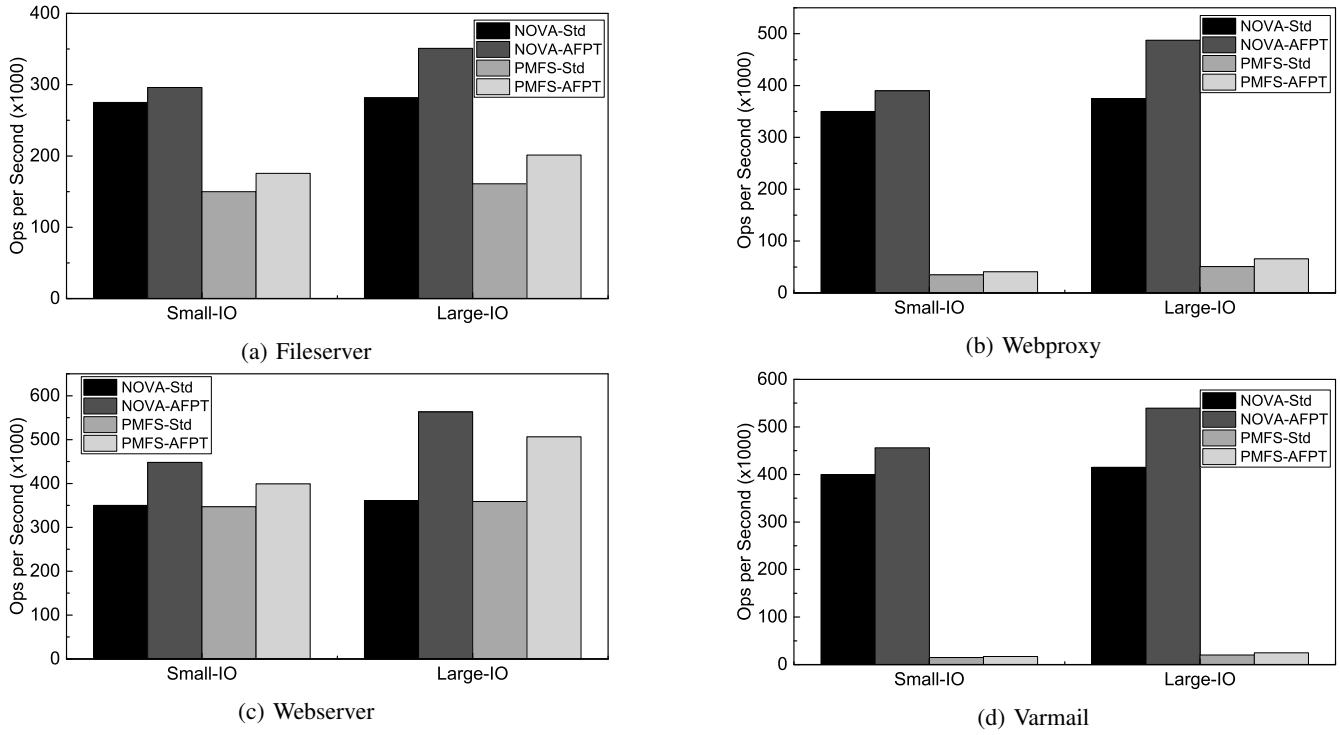
We use four Filebench [30] workloads, including fileserver, webproxy, webserver and varmail, to evaluate how AFPT behave when running real-world applications. Table I lists the characteristics of the selected benchmarks. They have different read/write ratio and access pattern. We test two different configuration of workloads by using small and large I/O sizes. For small I/O size, the file page table is not built until the a file is accessed many times to reach the threshold. For large I/O size, the file page table is built when the file is accessed for the first time. We run each workload configuration for 5 times and report the average throughput results.

TABLE I: Workload characteristics.

Workload	Average file size	I/O Size (Small/Large)	# of Threads	R/W Ratio
Fileserver	512KB	16KB/64KB	50	1:2
Webproxy	64KB	16KB/1MB	50	5:1
Webserver	256KB	8KB/1MB	50	10:1
Varmail	128KB	16KB/1MB	50	1:1

Figure 7 shows the throughput of Filebench workloads, when issuing small and large I/O to PMFS and NOVA. We can ge the following conclusions. First, AFPT strategy outperforms the standard file index for both file systems and AFPT is especially efficient for read-intensive workloads, such as webproxy and webserver. Second, AFPT is more efficient for large I/O requests than small I/O requests. This is because some small I/O requests have to be served by software search routines before using file page table. While large I/O can trigger file page table when it is issued for the first time. The performance improvement of AFPT against the B-Tree file index can be up to 55.62% for NOVA and 41.78% for PMFS.

Fileserver and varmail are write-intensive workloads. AFPT strategy can improve performance by up to 25.12% for NOVA. The improvement for PMFS in varmail workload is only 3.24%, this is because varmail scatters many files in large directory, PMFS suffers from poor directory operations.



(a) Fileserver

(b) Webproxy

(c) Webserver

(d) Varmail

Fig. 7: Filebench throughput with different I/O size when PMFS and NOVA using default file index and AFPT.

The major bottleneck for PMFS in varmail workload is metadata operation.

Webserver and webproxy are read-intensive workloads. AFPT demonstrates significant performance improvement. Since webserver involves little directory operations, PMFS can achieve up to 41.78% speedup. However, for webproxy workload, PMFS suffers from directory lookup inefficiency and has limited improvement.

D. Enterprise Workloads

We select several workload traces from Microsoft data center [21], [31] and evaluate how the proposed *AFPT* scheme can improve the performance of typical enterprise workloads. Table II lists the description of the workloads.

TABLE II: Enterprise workloads description.

Workload	R/W Ratio	Total Request Size	Description
hm	4:1	10GB	Hardware monitor.
mds	1:4	20GB	Metadata server.
prn	5:1	20GB	Printer server.
rsrch	1:6	15GB	Research server.
web	8:1	30GB	Webserver.

We replay the workload traces by issuing I/O requests according to the request sequence. Operations per second (Ops) is used to evaluate the performance when using different file systems.

Table III and Figure 8 show the performance of enterprise workloads when using different file system index schemes. The results show that *AFPT* scheme can better improve read-intensive workloads, such as *hm, prn* and *web*. The improvement over the standard software search scheme can be up to 37.45% for NOVA and 44.93% for PMFS. For write-intensive workload *rsrch*, the improvement is around 4% for both file systems. This is because *rsrch* has multiple small write requests, which limits the potential of file page table.

TABLE III: Performance (Ops) of enterprise workloads when using different index schemes.

Workload	**Strategy**		Imprv.	**Strategy**		Imprv.
	NOVA -Std	NOVA -AFPT		PMFS -Std	PMFS -AFPT	
hm	89430	107463	20.16%	78547	89197	13.56%
mds	76341	88951	16.52%	84276	95907	13.80%
prn	82127	99215	20.81%	67239	89021	32.39%
rsrch	43087	44816	4.01%	30982	32186	3.89%
web	109821	150945	37.45%	98351	142543	44.93%

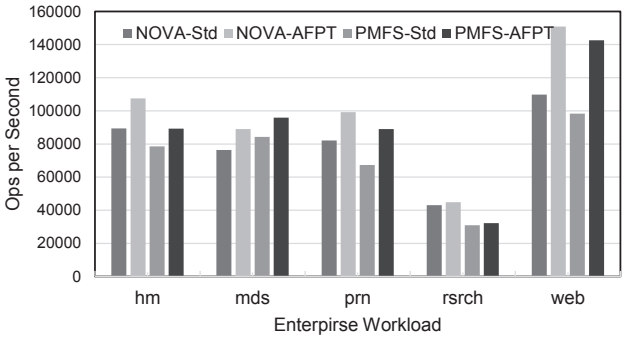


Fig. 8: Performance comparison of enterprise workloads when PMFS and NOVA use the standard index scheme and the proposed *AFPT* scheme.

Since enterprise workloads are mixed with read and write requests, the proposed *Adaptive File Page Table* scheme can be utilized to improve the overall performance.

E. Application Workloads

We select some application workloads to test how the proposed *AFPT* scheme behave when running real-world applications, such as MapReuce, Key-Value Storage and Relational Database. Table IV shows the characteristics of the applications.

TABLE IV: Application workloads characteristics.

Application	Total Data Size	Description
WordCount	25GB	Count the words of wiki pages using mapreduce library.
TeraSort	30GB	Sort randomly-generated numbers using mapreduce library.
RocksDB	30GB	Put and Get value to a KV database.
SQLite	20GB	Insert and query to a relational database.
Kernel-Make	10GB	Compile the Linux kernel source files.

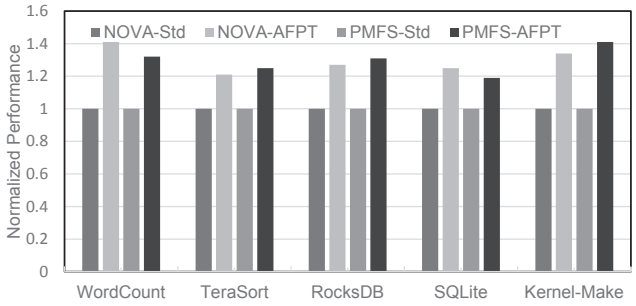


Fig. 9: Performance comparison of application workloads when PMFS and NOVA use the standard index scheme and the proposed *AFPT* scheme.

Figure 9 shows the normalized performance of PMFS and NOVA when using different index schemes. We can get the following conclusions.

For MapReduce workloads (WordCount and TeraSort), the *AFPT* scheme can improve performance by up to 40%. This is because mapreduce workloads needs to read large data blocks continuously. The sequential access pattern can benefit most from file page table index structure.

For Key-Value and relational database workloads, we carry out large number of queries after inserting data into the database. The *AFPT* scheme can improve the query performance by up to 25%.

For Kernel-Make workload, it involves reading source files from file system and writing object files to the file system. There are more than 60000 source files in the linux kernel source tree, both input and output can benefit from the fast *AFPT* index scheme.

F. Sensitivity to File Size

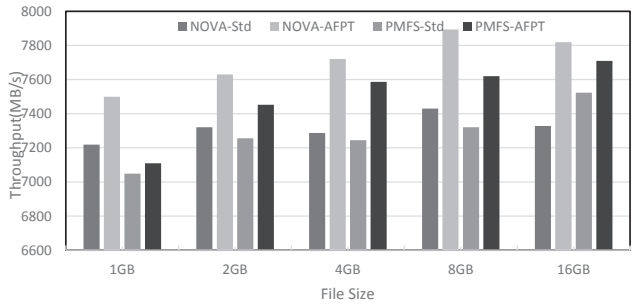


Fig. 10: Sensitivity to file size when PMFS and NOVA accessing file with standard index scheme and the proposed *AFPT* scheme.

Figure 10 shows the performance comparison when PMFS and NOVA accessing different size of files with the standard index scheme and the proposed *AFPT* index scheme. The performance is measured by re-reading file contents multiple times to ensure the file page table is setup and stored in file inode. Both PMFS and NOVA benefit from file page table and virtual address space. The maximal throughput of NOVA and PMFS is 7874MB/s and 7709MB/s respectively. Since the file page table is stored in inode after the file is accessed for the first time. Re-reading operations can be instantly served by the virtual address space. The *AFPT* scheme is especially useful for large file.

V. Related Work

In this section, we discuss some research work that is closely related to ours.

In-Memory File System Design. There are a number of in-memory file systems aiming at exploiting byte-addressability of NVM, such as BPFS [17], SCMFS [20], SIMFS [15], PMFS [14], HiNFS [19], NOVA [16], PRAMFS [22] and EXT4-DAX [32], [33]. These file systems are designed to address the following two concerns in NVM-based system. First, to avoid traditional software stack overhead by directly copying contents to and from file system without involving page cache. Second, to provide a strong consistency mechanism in order to protect file system from corruption caused by power failure or system crash.

BPFS utilizes shadow paging and atomic updates to provide consistency for update operations. PMFS and NOVA provide new mechanisms for metadata and data consistency, such as light-weight journal and log-structured design. D-WARM [21] aims to optimize wear-leveling for in-memory file systems. SCMFS and SIMFS utilizes virtual address space and MMU for file operations. But SIMFS fails to consider the overhead of building file page table for small files may lead to performance degradation in some cases.

In conclusion, state-of-the-art in-memory file systems mainly focus on providing a new architecture for emerging NVM and a set of accompanying consistency mechanisms to achieve consistency. They fail to consider the performance optimization for specific workloads. Our proposed *Adaptive File Page Table* strategy can be seamlessly applied to any in-memory file system with moderate modification to the file system source code, since most of the in-memory file systems use 4KB blocks as file index. It is straightforward to build file page table using the same pages in the existing index structure.

Optimization Strategies for File Systems. Since NVM has large write performance gap compared with DRAM [34], many optimization strategies have been proposed to mitigate the asymmetric read-write performance of NVM [35]–[42]. The essential idea is to hide long write latency to NVM in the critical path, in order to avoid system performance degradation.

PTree [18] utilizes page table to provide fast path name resolution. PTree organizes the metadata of file systems with page table to exploit the fast random access of NVM, it

significantly increases the efficiency and scalability of name resolution. HiNFS [19] proposes an NVM-ware write buffer policy to cache lazy-persistent file writes in DRAM to eliminate the long write latency for NVM accesses. SwapX [43] proposes to map address space of hosts directly to NVM pool so that it can improve energy efficiency and performance of swap operations.

Most of the existing optimization techniques aim to improve write performance of in-memory file systems. And page table and virtual address space concept have been widely adopted in optimizing swap and metadata operations. Our proposed AFPT scheme can be integrated with other strategies to achieve excellent performance for in-memory file systems.

VI. Conclusion

The development of NVM-aware file systems results in many novel architecture innovation and new approaches for consistency. However, few effort is devoted to optimize read performance of in-memory file systems. In this paper, we reveal the challenges of state-of-the-art file index structure, that is, file page table and virtual address space concept is not efficiently utilized by in-memory file systems. This finding motivates the design of a new index scheme, called *Adaptive File Page Table (AFPT)*, that achieves excellent read performance for variety of access patterns. *AFPT* dynamically determines whether to use software search or MMU for file accesses. We implement AFPT scheme in PMFS and NOVA, the experimental results show that AFPT can improve performance by up to 55.62%.

References

[1] H. Jeong, "High density pcm(phase change memory) technology," in *2016 International SoC Design Conference (ISOCC)*, Oct 2016, pp. 187–188.

[2] G. W. Burr, M. J. Brightsky, A. Sebastian, H. Y. Cheng, J. Y. Wu, S. Kim, N. E. Sosa, N. Papandreou, H. L. Lung, H. Pozidis, E. Eleftheriou, and C. H. Lam, "Recent progress in phase-change memory technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 2, pp. 146–162, June 2016.

[3] R. Annunziata, P. Zuliani, M. Borghi, G. D. Sandre, L. Scotti, C. Prelini, M. Tosi, I. Tortorelli, and F. Pellizzer, "Phase change memory technology for embedded non volatile memory applications for 90nm and beyond," in *2009 IEEE International Electron Devices Meeting (IEDM)*, Dec 2009, pp. 1–4.

[4] W. H. Chen, W. J. Lin, L. Y. Lai, S. Li, C. H. Hsu, H. T. Lin, H. Y. Lee, J. W. Su, Y. Xie, S. S. Sheu, and M. F. Chang, "A 16mb dual-mode reram macro with sub-14ns computing-in-memory and memory functions enabled by self-write termination scheme," in *2017 IEEE International Electron Devices Meeting (IEDM)*, Dec 2017, pp. 28.2.1–28.2.4.

[5] S. G. Kim, J. C. Lee, T. J. Ha, J. H. Lee, J. Y. Lee, Y. T. Park, K. W. Kim, W. K. Ju, Y. S. Ko, H. M. Hwang, B. M. Lee, J. Y. Moon, W. Y. Park, B. G. Gyun, B. K. Lee, D. Yim, and S. J. Hong, "Breakthrough of selector technology for cross-point 25-nm reram," in *2017 IEEE International Electron Devices Meeting (IEDM)*, Dec 2017, pp. 2.1.1–2.1.4.

[6] I. Messaris, A. Serb, S. Stathopoulos, A. Khiat, S. Nikolaidis, and T. Prodromakis, "A data-driven verilog-a reram model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.

[7] H. Hossam, M. El-Dessouky, and H. Mostafa, "Time-based read circuit for multi-bit memristor memories," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, May 2018, pp. 1–4.

[8] M. Stork, "Properties of one type memristor emulator," in *2018 28th International Conference Radioelektronika (RADIOELEKTRONIKA)*, April 2018, pp. 1–6.

[9] Z. I. Mannan, C. Yang, and H. Kim, "Oscillation with 4-lobe chua corsage memristor," *IEEE Circuits and Systems Magazine*, vol. 18, no. 2, pp. 14–27, Secondquarter 2018.

[10] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 707–722.

[11] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 385–395.

[12] C. Wu, G. Zhang, and K. Li, "Rethinking computer architectures and software systems for phase-change memory," *J. Emerg. Technol. Comput. Syst.*, vol. 12, no. 4, pp. 33:1–33:40, May 2016.

[13] C. Costa and M. Y. Santos, "Big data: State-of-the-art concepts, techniques, technologies, modeling approaches and research challenges," *IAENG International Journal of Computer Science*, vol. 44, no. 3, pp. 285–301, 2017.

[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 15:1–15:15.

[15] E. H. M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "A new design of in-memory file system based on file virtual address framework," *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 2959–2972, Oct 2016.

[16] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 323–338.

[17] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146.

[18] J. Zeng, N. Xiao, F. Liu, L. Zhu, Y. Li, Y. Xing, and S. Li, "Ptree: Direct lookup with page table tree for nvm file systems," in *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, Nov 2017, pp. 1160–1167.

[19] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:16.

[20] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–11.

[21] L. Wu, Q. Zhuge, E. H.-M. Sha, X. Chen, and L. Cheng, "Dwarm: A wear-aware memory management scheme for in-memory file systems," *Future Generation Computer Systems*, vol. 88, pp. 1 – 15, 2018.

[22] "Protected non-volatile ram filesystem." [Online]. Available: https://pramfs.sourceforge.net/tech.html.

[23] S. H. A. Antoni Wibowo and N. Z. Abidin, "Combined multiple neural networks and genetic algorithm with missing data treatment: Case study of water level forecasting in dungun river - malaysia," *IAENG International Journal of Computer Science*, vol. 45, no. 2, pp. 246–254, 2018.

[24] X.-X. Ma and J.-S. Wang, "Function optimization and parameter performance analysis based on krill herd algorithm," *IAENG International Journal of Computer Science*, vol. 45, no. 2, pp. 294–303, 2018.

[25] Q. Z. Qinghe Zheng, Mingqiang Yang and J. Yang, "A bilinear multi-scale convolutional neural network for fine-grained object classification," *IAENG International Journal of Computer Science*, vol. 45, no. 2, pp. 340–352, 2018.

[26] A. S. Wei Cao and J. Hu, "Stacked residual recurrent neural network with word weight for text classification," *IAENG International Journal of Computer Science*, vol. 44, no. 3, pp. 277–284, 2017.

[27] "Introduction to STREAM benchmark," http://www.cs.virginia.edu/stream/ref.html.

[28] "Persistent Memory Development Kit," https://github.com/pmem/pmdk.

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

[30] "Filebench benchmark." [Online]. Available: https://github.com/filebench/filebench/wiki.

[31] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage*, vol. 4, no. 3, pp. 10:1–10:23, Nov. 2008.

[32] "Supporting filesystems in persistent memory." [Online]. Available: https://lwn.net/Articles/610174/.

[33] "Support ext4 on nv-dimms." [Online]. Available: http://lwn.net/Articles/588218/.

[34] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 91–104, Mar. 2011.

[35] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, ser. CIDR '11, 2011, pp. 21–31.

[36] C. Wu, G. Zhang, and K. Li, "Rethinking computer architectures and software systems for phase-change memory," *J. Emerg. Technol. Comput. Syst.*, vol. 12, no. 4, pp. 33:1–33:40, May 2016.

[37] P. Chi, W. C. Lee, and Y. Xie, "Adapting B^+ -tree for emerging nonvolatile memory-based main memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 9, pp. 1461–1474, Sept 2016.

[38] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 389–400, Dec. 2014.

[39] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, May 2015, pp. 1–10.

[40] D. Niu, Q. He, T. Cai, B. Chen, Y. Zhan, and J. Liang, "Xpmfs: A new nvm file system for vehicle big data," *IEEE Access*, pp. 1–1, 2018.

[41] Z. Zhang, D. Feng, Z. Tan, J. Chen, W. Zhou, J. Zhang, and L. T. Yang, "An approach of spatial usage optimization for nvm-based storage system," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, Dec 2017, pp. 229–236.

[42] Y. Xu, L. Yang, Z. Hou, Q. Huo, and K. Qiu, "Energy-efficient cache management for nvm-based iot systems," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, Dec 2017, pp. 491–493.

[43] G. Zhu, K. Lu, X. Wang, Y. Zhang, P. Zhang, and S. Mittal, "Swapx: An nvm-based hierarchical swapping framework," *IEEE Access*, vol. 5, pp. 16 383–16 392, 2017.