creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Doctoral Thesis

Empowering Storage Systems with Lightweight Critical Path via Persistent Memory

Hyunsub Song

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

2021

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Empowering Storage Systems with Lightweight Critical Path via Persistent Memory

Hyunsub Song

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Empowering Storage Systems with Lightweight Critical Path via Persistent Memory

A dissertation submitted to
Ulsan National Institute of Science and Technology
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Hyunsub Song

07.07.2021 of submission

Approved by

Advisor

Sam H. Noh

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Empowering Storage Systems with Lightweight Critical Path via Persistent Memory

Hyunsub Song

This certifies that the dissertation of Hyunsub Song is approved.

07.07.2021 of submission

Signature

Advisor: Sam H. Noh

Signature

Young-ri Choi

Signature

Woongki Baek

Signature

Myeongjae Jeon

Signature

Beomseok Nam

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Abstract

As the recently commercialized next-generation memory technology, Persistent Memory (PM), has been introduced into the computing system environment, it has brought many changes to the critical path of the storage system. PM, which has similar performance and nonvolatility to DRAM, can be applied to the storage systems and has a dramatic performance effect in terms of hardware. However, the performance bottleneck in the critical path has been moved from the hardware part to the software stack because the performance of the media has increased dramatically. Many research groups have presented a lot of new technologies for the critical path of the conventional storage system to ensure the original performance of PM hardware.

In this dissertation, we conduct research to efficiently design and improve critical path from various viewpoints in order to maximize the performance of the PM in the storage system environment.

First, we analyze the empirical findings of traditional storage system's critical path. We have analyzed various popular Linux kernel file systems such as Ext4, Btrfs and F2FS. For comparison, we also perform a breakdown of the I/O flow of DAX, PMFS, and NOVA that are relatively new PM-dedicated file systems.

Second, we propose the Persistent Memory Adaptation Layer (PMAL), a module that can be added to the existing file system to control the I/O flow to PM media. PMAL is lightweight and can easily be integrated into legacy file systems to take advantage of PM storage. Using Ext4, we show that the performance of PMAL integrated Ext4 is in par with PM file systems for the Filebench and key-value store benchmarks.

Third, we present a MeLo@V (Metadata Logging at the VFS layer) technology that can lightly guarantee file system consistency without multi-versioning technologies. The MeLo@V approach is implemented on Linux kernel and its performance measured for contemporary file systems including both legacy and PM-dedicated file systems. Performance improvements for PM-dedicated file systems range between 2 to 9% for the workloads that were used for the experiments.

Finally, we propose First Responder (FR), a means to exploit the beneficial features of PM, while making use of modern and mature file systems such as Ext4 developed for traditional storage devices. Conceptually, FR is much like a buffer cache, but much more is involved such as maintaining consistency under failure and providing featherweight management overhead. With experimental evaluations with the Intel DC PMM, we show that FR, when used in cache form, can outperform Ext4 by more than 9×, while providing durable in-order file system semantics, whereas Ext4 cannot. We also show that when used as part of a typical file system, performance is comparable with NOVA and Ext4-DAX. Additionally, we show performance evaluation about FR applied to Ceph, a distributed storage platform.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

DEDICATION

To my parents, In Seog Song and Kyungok Cho for their unconditional love and support.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Contents

I Introduction 1

II Related Work 3

2.1 Persistent Memory 3

2.2 Making Use of Persistent Memory 3

III PMAL: Enabling Lightweight Adaptation of Legacy File Systems on Persistent Memory Systems 8

3.1 Analysis of File Systems 9

3.2 PMAL Design and Implementation 11

3.3 Experimental Results 17

3.4 Summary 22

IV Improving Performance of Persistent Memory File Systems Through a Lightweight Consistency Mechanism at the VFS Layer 24

4.1 File Systems and the Challenges 25

4.2 MeLo@V: Design and Implementation 27

4.3 Performance Evaluation 30

4.4 Summary 35

V First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage 37

5.1 First Responder: The Design 38

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

5.2 Data Consistency and Recovery Protocol 42

5.3 Performance Evaluation 48

5.4 FR on Distributed Storage Platform (Ceph) 61

5.5 Summary 63

VI Conclusion 64

References 65

Acknowledgements 76

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

List of Figures

1 Comparison of PM use scenarios for file systems 4

2 PMAL integration into legacy file systems 12

3 Traditional page cache flush vs. PMAL Flush 13

4 Execution sequence of PMAL Consistency mechanism that provides journal mode journaling . . 15

5 Overall performance of various benchmarks 18

6 Performance comparison of PMAL and PMFS for the Fileserver workload as the number of files managed is increased 20

7 Performance comparison of PMAL and DAX for different consistency guarantees 20

8 Call flow of `copy_file_range()` in PMAL and DAX where boxed numbers represent the execution steps 21

9 Function `copy_file_range()` performance according to whether or not page cache is used . . 22

10 The figure on the left shows the execution sequence, while the Steps on the right are MeLo@V code inserted in sequence at arrow designated points. 28

11 Performance of Filebench and key-value store benchmarks 32

12 Performance of Ext4, Btrfs, Ext4$^{+M}_{-PC}$, and Btrfs$^{+M}_{-PC}$ for the Fileserver workload as the page cache size is increased 34

13 Performance of function `copy_file_range()` as the data size is varied 35

14 Internal organization and components of FR 40

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

15 The figure shows the steps taken based on the initial condition of a slot when a write is requested. The C(V,N) values on top and below the slot, that is, the two chunks are the status flags associated with each chunk. The changes to these status flags, denoted by the bold characters, are done atomically. 41

16 Algorithm `FR_write()` 43

17 (a) Changing of the working set with time for the synthetic workload, only showing the first 900 of a 1300 second run. (b) Performance comparison between FR and Ext4 for light and heavy synthetic workloads. 49

18 Filebench and YCSB throughput normalized to FR, with its absolute performance numbers shown. 53

19 YCSB 99th, 99.9th, and 99.99th percentile tail latency as reported by YCSB normalized to FR. Numbers pointing to FR bars are absolute latency for FR in microseconds. 54

20 Fxmark test results. Note that the *y*-axis scales are different for all 4 graphs. 55

21 Performance results for PM size of 2^xGB, where x is value of points in x-axis, normalized to the performance of FR when $x = 7$. 56

22 performance when heterogeneous workloads Varmail and YCBS-A are executed together. 57

23 (a) YCSB executed with FFT modules taking up 108GB of DRAM capacity and (b) FR applied to Btrfs. 58

24 Performance with different storage devices. 59

25 Performance when storage is (a) SATA SSD, (b) NVMe and (c) HDD. 60

26 The top depiction is state of FR tags just after reboot upon failure but before recovery, while bottom is state right after recovery. 61

27 (a) Overall performance of FR applied to Ceph platform and (b) sensitivity for period. . . 62

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

List of Tables

1 Characteristics of file systems in Figure 1. In-order: In-order semantics support, Media: Storage media supported, Mature: Mature file system support. (* refers to underlying file system being mature, not that FR is.) 7

2 Description of components and their running time (ns) in `write()` I/O execution flow for various file systems 9

3 System configuration 17

4 Characteristics of workloads 17

5 Running time (ns) of components in `write()` I/O execution flow of PMAL and DAX obtained similarly to Table 2 19

6 Running time (microseconds) for components in `write()` execution flow with MeLo@V deployed 33

7 System configuration 48

8 Characteristics of the 21 files used to generate the synthetic workload. Numbers in brackets are number of files in the particular mix. N: Normal, R: Random. 49

9 Average latency for managing FR for various indexing and management policies. The average latency for FR static indexing is 67ns while for hashed indexing it is 75ns. (* Insert includes mechanism to find empty blocks.) 51

10 Characteristics of benchmarks. Request distribution for YCSB-D (YD) is Latest, while all others are Zipfian. 52

11 Various characteristics of workload execution. (M: million, WA: wrap-arounds) 54

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

I Introduction

A storage system is a computing service that allows users to permanently store data. The storage system provides various types, such as file systems, key-value stores provided by local computer, and distributed storage platforms provided in cloud environment [1–6]. In storage system, a critical path refers to the flow of data from software stack to a persistent medium, it also guarantees that the data become fully durable. Performance in a storage system is highly dependent on how this critical path is designed and implemented. In the past, the hardware part including the storage medium, which is evaluated as relatively slow from the point of view of the memory hierarchy, was a major performance bottleneck. As the recently commercialized next-generation memory technology, Persistent Memory (PM), has been introduced into the storage system environment, it has brought many changes to the critical path of the storage system. PM can be accessed in byte units and has performance similar to DRAM, and at the same time, data is preserved even when the power is turned off. PM can be applied to the storage systems and has a dramatic performance effect in terms of hardware. However, the performance bottleneck in the critical path has been moved from the hardware part to the software stack because the performance of the media has increased dramatically [7–9]. Many research groups have presented a lot of new technologies for the critical path of the conventional storage system to ensure the original performance of PM hardware [10, 11, 11–21].

In this dissertation, we conduct research to efficiently design and improve critical path from various viewpoints in order to maximize the performance of the PM in the storage system environment. We first analyze the elements that must be changed as the medium is changed to PM in the critical path from the point of view of the local file system. Since the existing file system was developed for the block-based storage device, the critical path of the existing file system has many unnecessary elements to drive the PM. Typical examples are a block management layer in a file system and a page cache added to improve performance of a block-based storage device. In order to design a mechanism for efficiently managing data in PM, we conduct a study to remove unnecessary elements from the file system perspective.

In many related studies using PM as a storage device, there are cases in which new PM-only file systems that have eliminated unnecessary elements of existing file systems have been proposed [10–15]. A Direct Access eXciting (DAX) interface that can shorten the critical path of existing file system was developed in Linux kernel [18]. The above related studies deal with a new data management mechanism in an environment where only the page cache and block management layer are removed. We propose technology to efficiently modify not only the page cache and block management layer, but also the file system consistency mechanism, which is chronically considered to have the greatest load in the existing file system [8]. Furthermore, we study a system that does not require a consistency mechanism internally in the file system. What we focus on is how to fundamentally solve the software load of the operating system, when file-related data are managed in PM.

Next, we conduct a study to dramatically improve the critical path by using the buffer cache applied with the PM medium. PM-dedicated file systems enable high-performance I/O to PM, but have the following limitations. First, newly created file systems are not tested for stability for a long period of

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

time, so their maturity is low. Second, since these file systems were developed by excluding storage devices such as HDDs and SSDs, it is not possible or simple to integrate them with existing storage devices. We propose a persistent buffer cache technology that can utilize the performance advantages of PM while simultaneously utilizing various existing storage devices [22]. It can provide high performance while guaranteeing a high level of consistency for I/O operations using the characteristics of PM.

The main contributions of this dissertation are as follows.

- We analyze the empirical findings of traditional storage system's critical path. We have analyzed various popular Linux kernel file systems such as Ext4, Btrfs and F2FS. For comparison, we also perform a breakdown of the I/O flow of DAX, PMFS, and NOVA that are relatively new PM-dedicated file systems.

- We propose the Persistent Memory Adaptation Layer (PMAL), a module that can be applied to the existing file system to control the I/O flow to PM media [8]. We consider making use of legacy file systems for PM-based storage so that we can leverage the maturity ingrained in legacy file systems while, at the same time, reaping the high performance offered by PM. Using Ext4, we show that the performance of the as-is file system with PMAL integrated is in par with other PM file systems such as NOVA, PMFS and Ext4 with DAX extension, for the Filebench and key-value database benchmarks that we considered on the Linux platform. (Chapter III)

- We present a MeLo@V (Metadata Logging at the VFS layer) technology that can lightly guarantee file system consistency without multi-versioning technologies (for example, journaling, logging and checkpointing, etc.). MeLo@V , similar to PMAL, is applied to the existing file system to hide the load of unnecessary elements of the existing file system and to guarantee the original performance of the PM as much as possible. Our experiments on various workloads also show that the performance of Ext4 with DAX extension, PMFS and NOVA can be improved upon by 2 to 9% by incorporating MeLo@V and slight modifications. (Chapter IV)

- We introduce First Responder (FR), the solution that we propose [22]. FR, like a buffer cache, absorbs requests at the topmost layer of the I/O stack in PM, then immediately responds to the requests (hence, the name). For reads, the requests end there, while for writes, the requests are forwarded to the traditional file system, in effect, hiding the entire I/O stack overhead. Our performance evaluations with the Intel DC PMM using an FIO generated synthetic workload show that FR, when used in cache form, can outperform Ext4 by more than 9×, despite providing durable in-order file system semantics, while Ext4 cannot. Using the Filebench and YCSB benchmarks, we also show that when FR is used as part of a typical file system, performance is comparable with the default Ext4, Ext4 with DM-WriteCache, NOVA, and DAX, while, again, providing durable in-order file system semantics. Also, we apply FR to the distributed storage platform Ceph [5], and then conduct performance evaluation with Ext4 and DM-WriteCache. (Chapter V)

This dissertation integrates upon previously published work [8, 22].

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

II Related Work

In this chapter, we first introduce Persistent Memory (PM) and review their characteristics. Then, we review some of the recent studies that have considered PM as part of the system configuration.

2.1 Persistent Memory

Persistent Memory (PM) technologies represented by PRAM or PCM (Phase Change RAM) [23], RRAM (Resistive RAM) [24], and STT-MRAM [25] are being considered as high performance storage mediums as they are nonvolatile and yet, provide random byte addressability and latency similar to DRAM. Latency and durability of STT-MRAM and PCM are expected to be better than those of NAND flash memory. In particular, the performance of STT-MRAM is expected to be in par with DRAM in terms of both latency and durability. This is the environment this study targets where DRAM is entirely replaced with medium that has STT-MRAM-like characteristics. Intel recently commercialized the Optane DC PMM, the only product currently available in the market [26–28]. It can be used in one of various forms, specifically, as storage, as memory with DRAM as its cache, and as an extension of memory [29]. As these PM technologies are nonvolatile, they are expected to open the path for new high-performance storage systems totally different from conventional storage systems [10–21, 30–35].

NVDIMM is another form of PM that is getting traction [36]. NVDIMM is an emulated PM device, which is commercially available, that provides persistence and DRAM performance. This device itself consists of DRAM and NAND flash memory. If a system failure occurs, all data in DRAM is safely transferred to NAND flash memory while a supercapacitor is used to prevent data from disappearing. As these tasks are performed internally within the NVDIMM, they do not affect other layers of the system. Hence, users can view and use NVDIMM as PM.

2.2 Making Use of Persistent Memory

Considerable research have been conducted in integrating PM technology into computer systems. These studies can be categorized largely into three realms: PM as a replacement or supplement of DRAM as main memory, PM as a new form of storage, and PM as simultaneously being main memory and storage. For using PM as main memory, there have been studies where PM is considered as a replacement or supplement of DRAM without considering the nonvolatile aspect of PM. The main motivation for such replacement is due to other characteristics such as energy consumption [37–40]. In contrast, the Whole System Persistence (WSP) study by Narayanan and Hodson [41] and the SOONN and ZEUS prototype that Doh et al. [42] propose exploit the nonvolatile nature of PM as main memory to enhance reliability or energy efficiency. Other studies that consider nonvolatility include work by Volos et al. [33] and Coburn et al. [43] where the authors present programming models so that applications can directly exploit PM nonvolatility. A comprehensive study on memory persistence in regards to PM is presented by Pelley et al. [44].

For studies using PM as storage, close to the device level, PM-based SSDs have been developed [30,

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

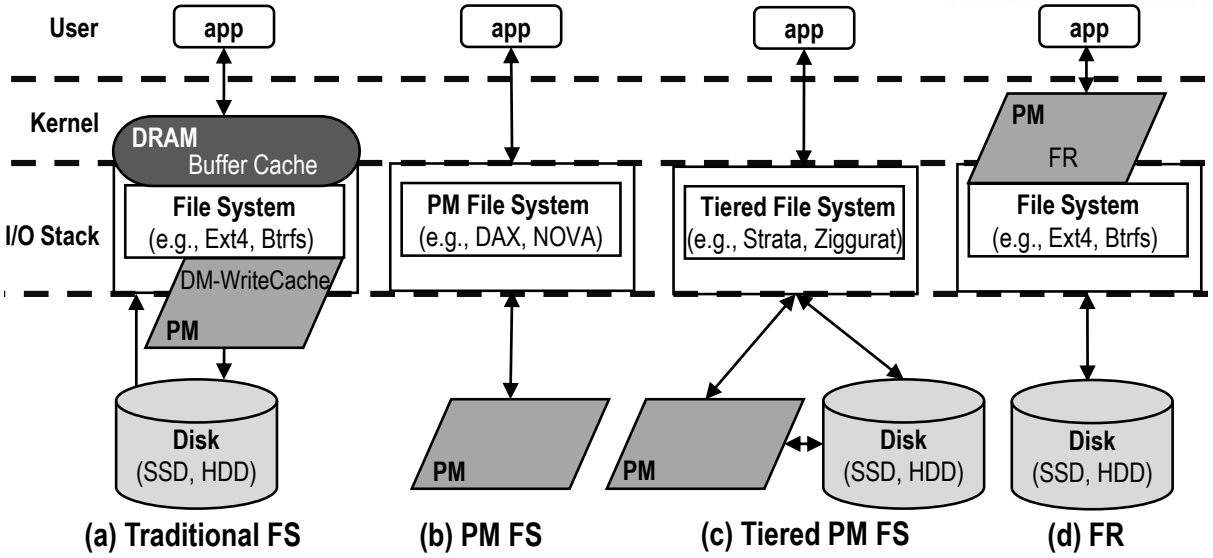


Figure 1: Comparison of PM use scenarios for file systems

45–47]. Storage system performance studies using an actual PM-based SSD and memory bus attached pseudo PM storage have also been conducted. Kim et al. perform an empirical study using PCM-based SSDs configured through tiering and caching of PCM. The results here show that such SSDs can be a promising approach for enterprise storage systems [48]. Lee et al. present the challenges and implications in terms of an existing operating system that have to be considered in a PM storage system through various empirical analyses [49]. Sehgal et al. conduct experiments that compare the effects of various features of legacy file systems in a simulated PM storage environment [50]. Various techniques to adapt the existing storage stack to accommodate PM have also been conducted [30, 45, 51, 52]. New storage management systems targeted for PM-based storage such as HEAPO, Aerie, and Mojim have also been developed [31, 34, 53]. Also, an entire category of work that considers file systems and buffer caches for PM medium exists. As these studies are closely related to our study and need deeper understanding, we discuss them separately in Section 2.2.1 and 2.2.2.

Finally, in terms of simultaneously exploiting PM as main memory and storage, the work by Baek et al. exploits nonvolatility and allows memory entities to be converted into storage entities, and vice versa. They show that this allows avoidance of unnecessary copying overhead resulting in enhanced performance for various applications [54].

2.2.1 File Systems for Persistent Memory

In this section, we discuss file systems for PM shown in the Figure 1(b). BPFS is one of the earliest studies on PM-dedicated file systems. The key difference of BPFS from legacy file systems is that it guarantees file system consistency through a Short Circuit Shadow Paging technique, which short circuits the insurgence of copy-on-writes (COWs) that can be instigated in traditional shadow paging file systems. This file system is based on the assumption that hardware guarantees 8-byte atomic writes, which we assume as well, and that reordering of memory operations by the compiler is prevented through

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

epoch barriers for persistence of data in PM storage.

Another PM-dedicated file system is PMFS, which is open source available [10]. In PMFS, the metadata is managed via a B-tree structure. This work also suggested a new hardware primitive called `pm_wbarrier` to effectively guarantee the durability of data into PM storage. This primitive was to be realized as the `pcommit` instruction, but was recently deprecated as Asynchronous DRAM Refresh (ADR) is planned to be supported in all Intel DIMM platforms [55]. This file system also provides fine-grained logging and COW techniques to ensure file system consistency.

NOVA, recent PM-dedicated file system that is also open source available, extends legacy log-structured file systems to hybrid memory systems (DRAM/PM) [11, 56]. Key features of NOVA are 1) that the metadata of this file system is managed separately for each CPU to obtain high scalability, 2) that it uses a lighter atomic update technique, which stores metadata in a log-structured format (appending), to efficiently guarantee file system consistency than traditional journaling or shadow paging techniques, 3) that it guarantees the sanity of metadata by using a lightweight journaling technique when several metadata are modified simultaneously, and 4) that it ensures the sanity of data by using the COW technique when data is overwritten.

Another approach for making use of PM is DAX, which was introduced for direct data access on PM type memory within Linux [18,57–60]. While XIP for code execution has been available since Ext2, direct access for data was added with DAX in Ext4 and XFS [18]. DAX makes use of the existing `mmap()` system call flow as well as the file I/O flow using the POSIX interface. In short, DAX provides a means to directly access the backing store using the existing interface. That is, even though it is implemented within the framework of existing file systems, DAX detours the page cache using Direct I/O. It also uses journaling to ensure file system consistency.

Recently, to further enhance performance of PM-based file systems, user level approaches such as FLEX, ZoFS, SplitFS and KucoFS have been proposed [12, 13, 35, 61]. However, like all user level approaches, it is not clear what kind of security vulnerabilities these are opening up.

With the advent of various types of storage media, recent studies have considered tiered file systems as shown in Figure 1(c). Strata is a multi-layered storage file system that allows for managing PM, SSD, and HDD in unison [14]. This multiple storage architecture enables Strata to provide the performance, capacity, and strong consistency guarantee all together. The basic design behind Strata is logging, and the notion of a digest is introduced to merge data written in logs. While logs are highly desirable for writes, a limitation of this approach is that search and read can suffer. A recent study presents Assise, a new distributed file system based on Strata, where PM is used as a client-local cache layer [62].

Ziggurat is another tiered file system that leverages the strengths of both PM and disks [15]. It exploits the benefits of PM through judicious data placement during file writes and through data migration. To do so, it analyzes the file write sequences and predicts whether the incoming writes are large and stable or whether updates to the file are likely to be synchronous.

While many recent studies have considered file systems for PM [10–15, 35], our dissertation is not about developing a new PM file system, but rather on integrating PM with existing modern file systems developed for traditional storage devices. PMAL [8] and MeLo@V technologies proposed in this dis-

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

sertation are simply applied to the traditional file system, shown in Figure 1(a), then they guarantee the original performance of PM and provide high reliability.

2.2.2 Buffer Caches for Persistent Memory

Another close working set of our dissertation is the study of buffer (or page) caches. The buffer cache, as depicted in Figure 1(a), is a DRAM layer that attempts to hide the low performance of storage and has been a topic of study for generations, mostly concentrating on the replacement or prefetching policy issues [63–65]. This was important as performance between DRAM and storage was many orders of magnitude different. With the advent of higher performing storage devices, there have been arguments as to the need for the buffer cache [10, 11, 66] as well as attempts to revisit this topic [51, 67, 68]. In particular, the DM-WriteCache [68] is an interesting optimization that introduces a new nonvolatile layer, as PM or SSD, just before the slower storage device, which could be an SSD or HDD, as depicted in the lower part of Figure 1(a). DM-WriteCache, supported from Linux 4.18 and beyond, is a writeback cache that helps to improve performance by caching writes from the page cache to the storage media. However, as writes are first written to the page cache, in-order file system semantics is not supported. The contrast between Figures 1(a) and (d) show how DM-WriteCache and FR [22] we propose are different by design.

There are also studies of the buffer cache that exploit the nonvolatile nature of PM [67, 69, 70]. UBJ, one of the first studies in this realm, makes use of PM based main memory as a buffer cache that unifies the functionalities of journaling and caching [69]. Commit transactions in UBJ do not write the journal logs to the underlying slow storage (HDD or SSD) but, simply convert the cached blocks into journal logs by committing them in-place. This scheme called In-place Commit removes the synchronization overhead caused by a journal commit. Furthermore, it provides almost the same cache effect of the traditional volatile buffer cache because the committed journal logs also serve as cached data before being evicted by checkpointing.

Tinca is a PM based buffer cache located in the external disk cache below the DRAM-based main memory [67] that uses a similar technique proposed in UBJ. FR is different from these studies in that durability is provided from the earliest layer of the I/O stack allowing immediate response. This requires careful design for consistency with the underlying file system, which these earlier studies neglect.

Another study on PM buffers is Three-tier BM (Buffer Management), a database storage engine that supports DRAM, PM, and SSD at once [70]. To make best use of the byte-addressable characteristic of DRAM and PM, Three-tier BM facilitates fine-grained accesses between these two devices by managing them by cache-line granularity and small page size. Unlike FR, the focus of this work is not on durability or consistency of data, but rather on the management of the PM from the perspective of it being another layer between DRAM and SSDs.

Retaining consistency is a key design issue for FR. Similarly, consistency has been the central issue in the design of numerous studies on data structures for PM. With efforts to reduce instructions that control write order [44, 71] such as `clwb` and `sfence` as these instructions incur considerable overhead [27],

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Table 1: Characteristics of file systems in Figure 1. In-order: In-order semantics support, Media: Storage media supported, Mature: Mature file system support. (* refers to underlying file system being mature, not that FR is.)

	(a) Trad. FS	(b) PM FS	(c) Tiered FS	(d) FR
In-order	No	Yes	Yes	Yes
Media	All	PM only	All	All
Mature	Yes	No	No	Yes*

while, at the same time, abiding to the 8-byte failure atomicity restriction to maintain consistency, various data structures for PM have been proposed [72–75]. FR conforms with these efforts as it carefully designs the use of 8-byte writes with `clwb` and `sfence`.

The use of PM cache and its effectiveness is coordinated in a distributed setting through replication and choices between pessimistic and optimistic consistency modes. This is different from our approach where we exploit existing file systems and consistency is always immediate, though locally ensured. Table 1 summarizes how previous local file systems differ from FR.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

III PMAL: Enabling Lightweight Adaptation of Legacy File Systems on Persistent Memory Systems

New memory technologies such as PCM, STT-MRAM and the recently announced 3D XPoint, which we henceforth refer to as Persistent Memory (PM), boasts performance in par with DRAM while providing nonvolatility. The advent of PM is anticipated to bring about considerable changes to the well-established computing framework. We anticipate memory and storage, and even CPU caches, to be completely replaced with PM [41, 42, 76–79], which allows for system state to be retained even upon reboot. This study follows this line of thought and is based on the premise that all of DRAM is replaced with PM, that is, a PM-only system.

Recently, there have been significant effort to develop storage systems targeted specifically for PM-based storage [10–21, 30–35, 43, 53, 56, 80, 81]. These research are based on the premise that legacy file systems are sub-optimal on memory bus attached PM storage. New approaches such as changing the existing system software architecture and removing the I/O stack from the file system have been suggested [10–19, 32, 35, 53, 56, 80, 81]. Such changes may improve performance, but integrating new techniques into current systems are usually quite difficult and cumbersome in various ways.

In this dissertation, we consider making use of legacy file systems for PM-based storage so that we can leverage the maturity ingrained in legacy file systems while, at the same time, reaping the high performance offered by PM. To this end, we first go through a thorough analysis of legacy file systems, even though for this study, we concentrate and report on the functionality and performance of only the Ext4 file system. Performance-wise we identify the modules that incur the most burden when used with PM storage. Also, we find that software overhead that is considered to be negligible in traditional disk-based storage can now have a serious effect on performance as their effects are amplified with fast PM. However, overall, we find that the main ingredient lacking in legacy file systems as a file system for PM storage is a component that can efficiently exploit PM.

Based on these analyses, we propose the Persistent Memory Adaptation Layer (PMAL), a lean software module composed of roughly 180 lines of code that intercepts handling of page cache writes and converts these writes to simultaneously write to PM storage. This has the effect of making page cache handling more efficient and providing journal mode journaling, which is avoided in traditional systems due to their heavy overhead, at virtually no extra cost. PMAL can be easily integrated into legacy file systems by simply altering the I/O flow to and from PMAL at particular points of the I/O stack. PMAL allows us to make use of legacy file systems instead of developing an entirely new file system that would require considerable time and effort to develop and mature. Using Ext4, we show that the performance of the as-is file system with PMAL integrated is in par with other PM file systems such as NOVA, PMFS and Ext4 with DAX extension, for the Filebench and key-value database benchmarks that we considered on the Linux platform.

The remainder of the chapter is organized as follows. In Section 3.1, we give a detailed analysis of the legacy Ext4 file system, breaking down the components of Ext4 in terms of functionality and

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 2: Description of components and their running time (ns) in `write()` I/O execution flow for various file systems

Component		Description	Ext4		DAX	PMFS	NOVA
			Async	Sync			
Syscall	System call gate	Internal system call function	291	276	211	328	330
VFS Layer	VFS objects	Set structure related to VFS	980	973	899	923	845
	I/O type switch	Change type of I/O	3.2K	5.7K	2.2K		
SFS Layer	Page cache	Work related to page cache	17.3K	16.8K			
	Memory I/O	Write data to memory	445	471			
	Page cache flush	Flush dirty page to storage		33.1K			
	FS consistency	Mechanism for FS consistency		101.1K	7.1K		
	DAX|PMFS|NOVA	Write data to storage			13.2K	19.1K	19.3K
Total Elapsed Time			22.2K	158.5K	23.7K	20.3K	20.4K

performance. Then, in Section 3.2, we describe the design and implementation of the Persistent Memory Adaptation Layer (PMAL) module. In the Section 3.3, we present the evaluation platform and discuss the evaluation results. Finally, Section 3.4 concludes the chapter with a summary.

3.1 Analysis of File Systems

Recall that our goal is to retain the key characteristics of legacy file systems to take advantage of the time-tested maturity of these systems, while at the same time, taking advantage of the beneficial performance characteristics of PM without overhauling the file system. To this end, we propose the Persistent Memory Adaptation Layer (PMAL), a module that intercepts and alters the flow of I/O from traditional file systems to meet our goal.

In this section, we analyze the empirical findings such that they form the basis for the design of PMAL. Our discussion focusses on Ext4 as the findings from other file systems are generally similar to those of Ext4. For comparison, we also perform a breakdown of the I/O flow of Ext4-DAX, PMFS, and NOVA that are relatively new PM-dedicated file systems.

Through this analysis we wish to compare the various aspects of file systems and see how they affect the performance of file systems. We first analyze the file systems in function units (e.g., `vfs_write()` and `new_sync_write()` within, for example, the `write()` system call) and then, for ease of analysis and understanding, we classify these functions into components based on their main functionality within the file system as shown in Table 2. This classification is based on analysis of source code of various file systems as provided in the Linux version 4.3.3.

Specifically, the components comprise the three layers of a call to the file system, that is, the system call, virtual file system, and specific file system layers. The components for the system call and virtual file system layers provide common functionality in the I/O flow differing only in their implementations. The components of the specific file system vary in terms of their functionality, the flow of execution

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

depending on specific calls invoking the I/O, and in their implementations.

Based on this classification of functionality, we measure the time consumed by each of these components while executing the `write()` call. As this study concentrates on Ext4, we explain our findings for Ext4 through comparison with other PM-dedicated file systems. The Ext4 file system is mounted on Ramdisk used to emulate a PM-based storage system. (The exact hardware platform in which these measurements were taken is discussed in Section 3.2.1.) The DAX, PMFS, and NOVA file systems are mounted on an emulated PM-based storage by using the *pmem* driver that is provided for PM-dedicated file systems in Linux [82–84]. The `ktime` [85] Linux kernel library is used to make the measurements. In particular, we measure the time spent at each of these components as a write request of a small sized (less than 4KB) data is made. The right hand side of Table 2 shows the results of the measurements, with the numbers being the average of 5 executions of each call. We now analyze the results in more detail.

System Call and VFS Layers: In the system call and virtual file system layers, we find that the main performance difference between Ext4 and other file systems comes from the 'I/O type switch' component. Basically, Ext4 has three I/O modes, namely, asynchronous, synchronous, and direct. On the other hand, PMFS and NOVA have only one I/O mode that directly performs I/O on PM storage. In the case of DAX, as DAX is implemented upon the Direct I/O flow of Ext4, DAX has an 'I/O type switch' component. Additionally, the difference between the asynchronous and synchronous I/O mode in Ext4 in the 'I/O type switch' component comes from the fact that synchronous I/O is simply asynchronous I/O plus some extras to immediately flush the page cache. Hence, synchronous I/O involves an additional I/O flow switch for page cache flush incurring higher overhead.

Specific File System `write()` Call: Let us now observe the write I/O within the specific file system layer for Ext4. First, observe the asynchronous I/O results. Time is spent at only two components, namely, 'Page cache' and 'Memory I/O' as a write completes when data is written to the page cache. We see that the 'Page cache' component, which is mainly the software overhead for handling the page cache, consumes much more time than the 'Memory I/O' component, which is the component that does the actual memory write operation. Traditionally, software overhead for handling the page cache has been considered to be insignificant in terms of total performance in block-based storage systems [49]. However, the results here show that in systems with low latency storage such software overhead can have a large effect on overall storage performance.

We now turn to synchronous I/O. Recall synchronous I/O is simply asynchronous I/O plus some extras to flush the page cache. The 'Page cache flush' component is related to page cache flushing, whose main task is to manage data structures and I/O to backing storage. We see that considerably more time is spent in this component compared to other components in the upper layers. Another component that incurs significant overhead is the 'FS (File System) consistency' component. This component is the part that is performed to ensure consistency of the file system. For this, Ext4 performs journaling with ordered mode as default. The reason why these components incur high overhead is because the implementation of the code is quite complicated. We emphasize again that when low latency storage such as PM is used, code efficiency can have a considerable effect on overall performance. This is unlike slow storage devices where storage is so slow that software efficiency has little effect.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

In the Ext4 implementation (as well as other file systems in Linux), we find code that cleanly separates the 'Page cache flush' and 'FS consistency' components for easy manipulation. We take advantage of this later in our design and implementation given in Section 3.2.

We see that DAX, PMFS, and NOVA have no components related to the page cache as shown on the right end part of Table 2. Each of the PM file systems has a component that writes to PM storage designated by the file system name. We see that this portion is significant for each of these file systems. DAX also has a separate 'FS consistency' component as DAX also uses journaling for file system consistency similarly to Ext4. However, there is high discrepancy between DAX and Ext4 (Sync) as journaling in DAX asynchronously commits metadata to the journal area. For reference, PMFS and NOVA use logging and lightweight atomic update to guarantee file system consistency, and this overhead is included in the respective components of the last row.

Summary of Observations and Analysis: Our analysis shows that there are a few key components in traditional file systems that incur considerable overhead. Specifically, they are the components that comprise the synchronous I/O portion of the `write()` call, namely, the 'Page cache flush' and 'FS consistency' components. To take advantage of PM storage with legacy file systems, these components, which are the biggest factors of performance degradation, need to be slimmed down and made efficient. Unfortunately, they cannot simply be replaced with a more efficient implementation. This is because these components, integrated within the page cache mechanism, are the ones that retain the mature attributes of legacy file systems as management of data and metadata is strongly integrated within them.

Recall that our goal is to retain the key characteristics of legacy file systems enabling us to take advantage of the time-tested maturity of these systems. In so doing, we want to also take advantage of the beneficial performance characteristics of PM. To this end, we devise a method to intercept the flow of I/O from traditional file systems at such a point that allows the traditional features to be retained as much as possible. Then, at this point, we insert a lightweight mechanism that makes use of PM. We implement this whole procedure as a module that we call the Persistent Memory Adaptation Layer (PMAL). We next discuss PMAL in detail.

3.2 PMAL Design and Implementation

In this section, we describe the design and implementation of PMAL. In the next subsection, we discuss the PM setting that is assumed and enables us to accomplish our goal. Subsequently, we discuss the overall architecture of the system integrating PMAL. Then, the two components that comprise PMAL, that is, PMAL Flush and PMAL Consistency, are described in detail.

3.2.1 PM-only Environment

Recall we assume that our system operates under the PM-only environment, where storage as well as main memory both have nonvolatile characteristics. This assumption provides a natural multi-versioning structure as was suggested by Lee et al. [51,52] and Zhao et al. [79], where a modified version is retained in (nonvolatile) page cache and the (nonvolatile) LLC cache, respectively, while the original data is in

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

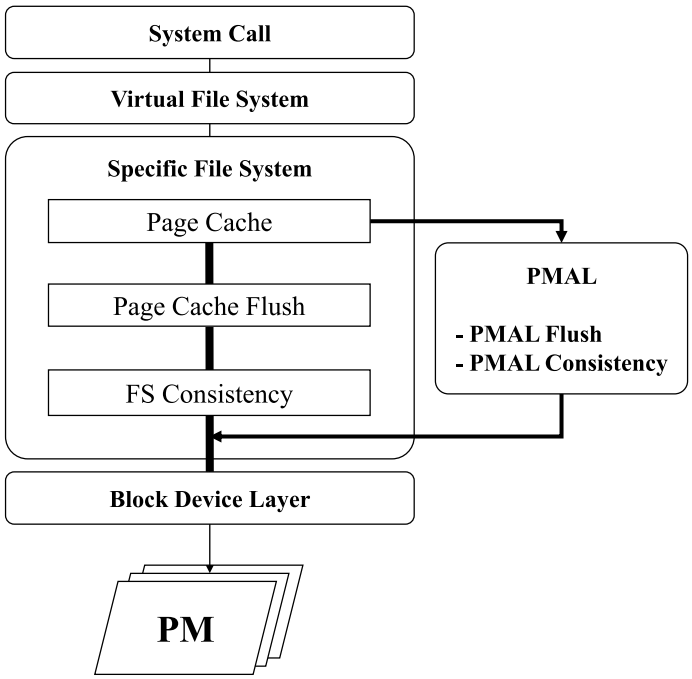


Figure 2: PMAL integration into legacy file systems

another nonvolatile area. In this section, we first discuss how this environment is emulated. Then in subsequent sections, we discuss how we exploit this environment through PMAL.

To implement this system, we set the storage portion of DRAM emulated as PM through Ramdisk and format the storage portion for Ext4. This approach allows for the particular characteristics of the legacy file system to remain intact, while, for our case, having the extra benefit of eliminating the I/O scheduler functionality of the I/O stack. More specifically, with Ramdisk, Linux itself removes the I/O scheduler code and the disk I/O functions of the device driver are changed to memory I/O functions (e.g `memcpy()`), which is how I/O functions for devices should be implemented in PM-dedicated file systems. The remaining DRAM area, which is regarded as PM, is used as main memory. For memory persistency with PM, we adopt the same technique used with memory writes as was done with other PM-dedicated file systems [10, 44, 82, 86]. Specifically, we make use of the `clwb` and `sfence` instructions after calling the memory copy function in the Ramdisk and the page cache code section used by the PMAL module.

Moreover, we consider byte-level I/O in our system. Basically, because legacy file systems are designed for block-based storage, the byte-level I/O user requests are translated into block-level I/O requests at the generic block layer. To preserve byte-level I/O requests, we simply modify the existing write flow; specifically, where byte-level data is converted into block/page units to perform I/O at the Ramdisk driver level. In PMAL, the Ramdisk driver code is modified to use the byte-level size of the write request that is obtained at the beginning of the write flow. This portion is implemented in roughly 30 lines of code (LoC).

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

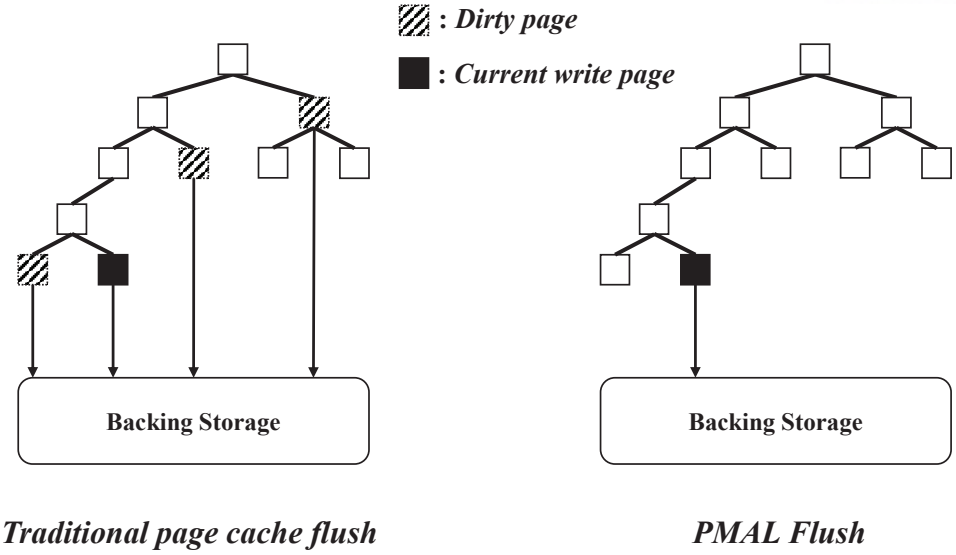


Figure 3: Traditional page cache flush vs. PMAL Flush

3.2.2 PMAL Architecture

Our architecture of PMAL is established based on a key observation of measurement results given in Table 2. That is, the results here show that the components related to the page cache and file system consistency make up the majority of the running time. The key to achieving our goal is, then, to optimize this portion of the file system. A simple method would be to bypass the page cache altogether as PM does not need a page cache as is done with DAX. However, this method is not feasible as the page cache is strongly integrated into the structure of legacy file systems [87]. Hence, pulling the page cache component out while leaving the inherent features of the file system intact is virtually impossible.

Our solution is to leave the page cache mechanism intact, but to insert a software module, which we call PMAL (Persistent Memory Adaptation Layer), that will take advantage of PM at the appropriate point of the I/O flow. PMAL is integrated into legacy file systems so that it intercepts the I/O flow from the page cache as shown in Figure 2. More specifically, after the data structures within the page cache are manipulated and the data is written to the page cache, an acknowledgement is sent to the upper layer. At this point, PMAL, which has two components, namely, PMAL Flush and PMAL Consistency, intercepts the acknowledgement and initiates a write to the file system. PMAL Flush is a component that improves on the traditional flushing mechanism, while PMAL Consistency provides journaling with minimal overhead. We describe each of these components in detail in the following sections. Once this write is done, the intercepted acknowledgement is resumed and sent to the upper layer where it will be accepted as completion of the write. We emphasize that all of PMAL is implemented in roughly 150 LoC. Specifically, $\sim$10 LoC for intercepting the I/O, $\sim$40 LoC for PMAL Flush, and $\sim$100 LoC for PMAL Consistency.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

3.2.3 PMAL Flush Component

The PMAL Flush component that we devise allows us control over how and when to synchronously flush the current write request. In the traditional page cache flush mechanism of legacy file systems, a flush incurs writes of all dirty pages in the page cache to backing storage, as shown on the left hand side of Figure 3. This is a natural consequence of optimizing writes to slow disks. With PM, the story is different as writes are more efficient. Hence, in PMAL, we choose to flush data written to the page cache at our convenience. A positive effect of this is that the flush mechanism is simplified as flush occurs at any write point for only a small amount of data instead of waiting to make bulky writes. This leads to simple management code compared to the relatively heavy code in legacy systems. More importantly, this enhancement allows us to exploit it to provide a higher level of consistency, that is, journal mode journaling, with virtually no overhead, which we describe below.

3.2.4 PMAL Consistency Component

The second part of our solution, which we refer to as PMAL Consistency, is based on the ability to control flushes through PMAL Flush as described above. Recall that we assume that the entire memory is PM. Hence, the page cache is also PM meaning that writes to the page cache is already persistent. Hence, similarly to the observations of Lee et al. [51, 52], the page cache can simultaneously be considered to be the journal. By providing data structures to manage the data in the page cache, that is, the journal, and controlling the flushes to the file system area through PMAL Flush, commits and checkpointing of the journal can be controlled. This is what the PMAL Consistency component does, and in the end, this allows us to provide journal mode journaling, which is a higher level of consistency than the default one provided by current legacy file systems, with virtually no overhead.

In the following, we describe the needed data structures and how we control the write sequence to achieve journal mode journaling. Then, we proceed to discuss the recovery mechanism used to maintain consistency upon system failure in conjunction with the journaling scheme.

Write Sequence for Journal Mode Journaling: Let us now discuss how journal mode journaling is provided in PMAL. For this, we make use of Figure 4, which depicts the step by step sequence for a write updating a file.

Upon a write request, metadata that we refer to as `pjournal`, which has elements `pj_commit`, `*metadata_info`, `*data_info`, `start_offset` and `length` all with initial values of NULL, is created. This `pjournal` is created per file when the file is opened and is removed when the file closes. Note that `pj_commit` will hold the current status of the write sequence, `*metadata_info` will hold the pointer to the new metadata (M) to be written (the address of the `inode` structure in Linux), and `*data_info` will hold the pointer to the kernel structure that manages the page cache where the new data (D) is to be written (in particular, the address of the `address_space` structure in Linux). The `start_offset` and `length` is the position within the file where the new data (D) is written to and the size of the write request, respectively. The only assumption regarding persistency needed for this process to dissertation is that 8-byte writes to change the `pj_commit` value be atomic [10, 44, 86].

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

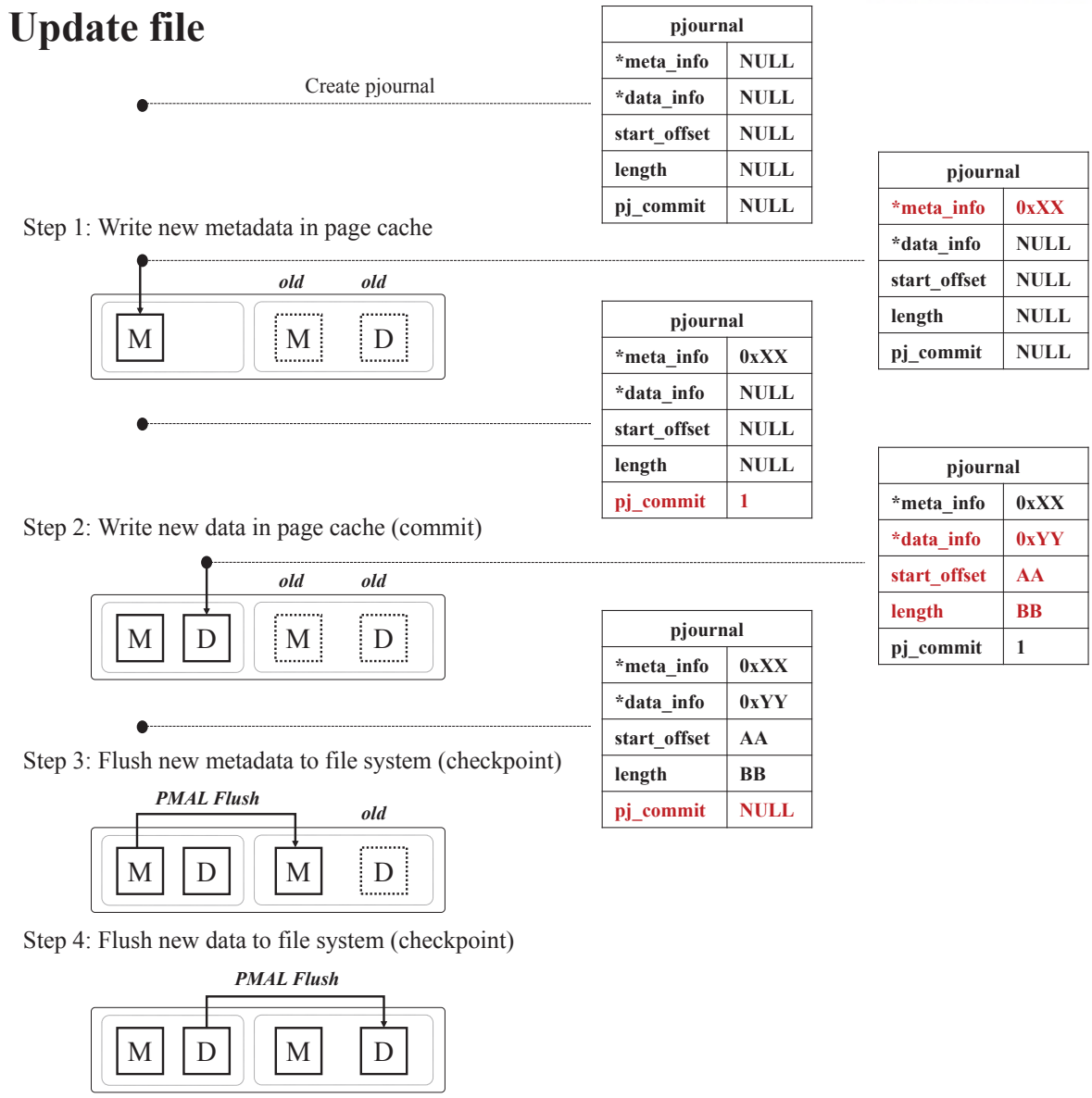


Figure 4: Execution sequence of PMAL Consistency mechanism that provides journal mode journaling

The sequence of changes occurring between Step 1 and Step 2 are as follows. First, before new metadata (M) is written into the journal (page cache), the pointer value to M is written to `*metadata_info`. Then, M is written into the journal. Finally, `pj_commit` is set to 1. Similarly, in between Step 2 and Step 3, first, the pointer value that points to the structure managing the page cache, where the new data (D) is to be written to, is written to `*data_info`. Then, the position and size values of the write request are written to `start_offset` and `length`, respectively. Then, after the new data (D) is written to the journal, `pj_commit` is set to NULL. Note that the order of the sequence is important to ensure recovery upon failure, which we discuss in more detail later.

The setting of `pj_commit` to NULL in Step 2 is equivalent to the journal commit upon which recovery of new data is guaranteed. Thereafter, Steps 3 and 4 are simply the checkpointing process of copying

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

the data in the journal to the file system (backing storage). The sequence of writes in Steps 3 and 4 is controlled through PMAL Flush.

Finally, note that we have described the update process here. However, creating and writing a new file follows exactly the same sequence and the only difference is that the old values, depicted in Figure 4 as single dotted boxes and denoted *old* on top, would not exist. Hence, the PMAL Consistency mechanism does not save any particular field of `pjournal` and there is nothing to recover in case of failure.

Recovery Mechanism: The recovery process upon system failure does not have a special mechanism, but simply relies on the page cache flush daemon that periodically flushes the page cache. Since the page cache is the (nonvolatile) journal, upon reboot all data that existed in the page cache will still remain. However, some of them may not have been committed and hence, be invalid. So for the recovery mechanism, all we need to do is to make sure that all invalid data in the page cache is removed and valid data is kept as-is upon reboot. The page cache flush daemon will do the rest, that is, at its convenience write them to their respective file system areas. Distinguishing invalid data is done using the `pjournal` metadata in PMAL, whose value before failure is retained when the file system is recovered after system failure. Note that as we assume a PM-only system, all data and data structures in main memory before the system failure will remain intact after reboot. Such a state along with `pjournal` is all that is required.

Then, we first need to find the address of `pjournal` before recovery can start. This can be found in the descriptor of the files that were open during abnormal system termination due to failure, which are found in the process descriptor (in Linux, `task_struct`). Hence, we first find the processes that were modifying files in our PM storage before system failure, then search for files that remained open, where we find the `pjournal` address for that file. In particular, in Linux, the member structures of interest in the process descriptor are `files_struct` and `fdtable`.

When `pjournal` is found, the PMAL Consistency mechanism takes recovery action based on the `pj_commit` value:

`pj_commit` **is NULL upon reboot**: This status is divided into two cases. The first case is when `pjournal` was created but never used. Here, removing `pjournal` is all that is needed to do. The second case is when `pjournal` is set to NULL in Step 2. This means that the new write was committed and hence, they are valid data. Then, they are simply left in the page cache. This is all that needs to be checked. The page cache flush daemon will take care of the rest.

`pj_commit` **is 1 upon reboot**: This means that the new write was not able to commit, hence whatever data that had been written should be considered invalid and removed from the page cache. This is performed using the values of `*metadata_info`, `*data_info`, `start_offset` and `length`. NULL values for `*metadata_info` or `*data_info` means the data, whether M or D, was not written to the journal and no action needs to be taken for that data. If it is non-NULL, then using their respective pointer values, the data is evicted from the journal as the data is invalid. Specifically, in Linux, the functions used to perform such actions are `delete_from_page_cache()` and `remove_inode_hash()`.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Table 3: System configuration

	Description
CPU	Intel i7-4820K 3.7GHz (4 cores / 8 threads)
Memory	Samsung DDR3 8GB PC3-12800 × 8 (64GB)
OS	Linux CentOS 6.6 (64bit) kernel v4.3.3
PM storage	Emulated with Ramdisk (32∼64GB)

Table 4: Characteristics of workloads

	R:W	Mean file size	# of files	# of threads
Fileserver	1:2	128K	100K	50
Webserver	10:1	32K	500K	50
Webproxy	5:1	32K	400K	50
Varmail	1:1	16K	800K	50
OLTP	1:1	1.5G	10	W:10 R:200
	R:W	Record selection	Dataset size	# of threads
YCSB-A	1:1	Zipfian	10G	5
ForestDB	2:1	Zipfian	15G	5

3.2.5 Hybrid Memory and `mmap()` Support

In this subsection, we briefly touch on two matters that we did not study in depth, but is relevant to this study, one being the PM-only assumption and the other, supporting the `mmap()` system call.

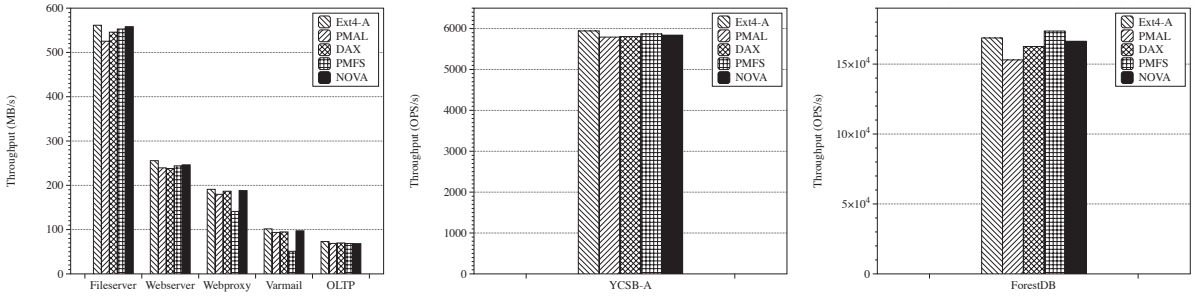
We have so far discussed PMAL under the PM-only assumption. PMAL can also be implemented under the DRAM+PM hybrid memory assumption, in particular, assuming that the page cache is volatile. In such a case, extra multi-versioning area such as the journal would be needed within the PM. The main benefit of this version of PMAL compared to legacy file systems is that the PMAL Flush component can take advantage of small sized flushes to reduce the overhead of page cache flushes. Such changes has performance implications as we discuss in Section 3.3.5.

Regarding the `mmap()` system call, though we do not implement it, `mmap()` can also be supported in PMAL with no additional design effort. This is because the `mmap()` flow is similar to the write/read flow as it uses the page cache, the main difference being that `mmap()` uses `msync()` to flush the data. Thus, `mmap()` can be supported in PMAL simply by calling the PMAL Flush component instead of `msync()`.

3.3 Experimental Results

In this section, we first present the experimental platform and the benchmarks used. We then discuss various aspects of the performance results.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY



(a) Filebench macro benchmarks (b) YCSB-A benchmark (c) ForestDB benchmark

Figure 5: Overall performance of various benchmarks

3.3.1 Experiment Platform and Benchmarks

To evaluate the effectiveness of the PMAL approach, we implement PMAL and integrate it into the Ext4 file system in the Linux kernel version 4.3.3. This version is chosen to compare PMAL against recent PM-dedicated file systems as the only recent Linux version in which both PMFS and NOVA are supported is version 4.3 [83, 84]. Three variations of Ext4 are used, namely, PMAL integrated Ext4 denoted PMAL, Ext4 using DAX denoted DAX, and Ext4 using asynchronous I/O denoted Ext4-A, which is provided as a reference. In particular, in integrating PMAL into Ext4, we remove from Ext4 the code section related to file system consistency. Specifically, within the flushing of data from the page cache to storage, we remove all code that commits data to the journal in the synchronous write flow.

The specifications of the experimental platform on which the experiments are conducted are summarized in Table 3. Specifically, of the 64GB DRAM space, 32GBs are used to emulate PM storage and is set as the capacity of the file system, leaving the rest to be used as memory. This PM is set as Ramdisk as mentioned in Section 3.2.1.

For the experiments, we use the Filebench macro benchmarks [88], which are popular real life-like workload generating benchmarks used to evaluate file systems. In particular, we make use of the Fileserver, Webserver, Webproxy, Varmail and OLTP benchmarks. Each workload represents write intensive, read intensive, strong access locality, sync intensive, and database workloads, respectively. The basic characteristics of these benchmarks are summarized in Table 4 with the footprint that each of the workloads see being set to be between 10 to 15GB.

We also use two key-value store benchmarks for our experiments. YCSB is an open source benchmark program suite provided by Yahoo generally used to evaluate NoSQL database systems [89]. Among these we make use of workload A, denoted YCSB-A, which is a write heavy workload (50/50 reads and writes) that represents applications such as a session store recording recent actions. The other benchmark is ForestDB-Benchmark, denoted ForestDB for brevity, which is a realistic key-value benchmark introduced by Couchbase [90]. Table 4 summarizes the characteristics of these two benchmarks.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 5: Running time (ns) of components in `write()` I/O execution flow of PMAL and DAX obtained similarly to Table 2

	PMAL	DAX	PMAL-O	DAX-S
System call gate	298	211	313	288
VFS objects	815	899	916	957
I/O type switch	3,208	2,223	3,037	2,892
Page cache	15,781		15,600	
Memory I/O	481		490	
PMAL	17,280		13,373	
FS consistency		7,115		98,358
DAX		13,213		14,813
Elapsed Time	37,863	23,661	33,729	117,308

3.3.2 Overall Performance

The overall performance results are given in Figure 5, where the *x*-axis represents the workload used, while the *y*-axis is the throughput. We make a few observations based on the results.

First, we see that Ext4-A indeed shows the best performance. However, we also see that PMAL is not far behind. Also, PMAL performs in par with other PM-dedicated file systems even though PMAL journaling is in journal mode. The PMAL column of Table 5, whose values are obtained in a way similar to Table 2, shows the approximate time spent in each of the components. The main observation is that the PMAL component, which performs journal mode journaling, does not incur extensive overhead compared to the 'Page cache flush' and 'FS consistency' components of Ext4 (Sync) in Table 2.

The second observation is that PMFS does considerably worse than other comparisons for the Webproxy and Varmail workloads. The Webproxy and Varmail workloads create a large number of files and directories, which strongly affect PMFS performance. This is due to the heavy metadata indexing structure construction required for PMFS. This is supported by the results shown in Figure 6, which shows the performance for the Fileserver workload as the number of files is increased, while the mean file size is set so that the total footprint is a constant 20GBs. We see here that as the number of files grows, PMFS performance drops considerably performing far worse than PMAL.

Overall, we see that PMAL is comparable to other PM-dedicated file systems. The performance of PMAL is only slightly below those of the ideal Ext4-A case, where all writes are being asynchronously written within the page cache. This is consistent throughout all workloads. Our results show that the lean implementation of PMAL allows us to attain the goal we set out to achieve.

3.3.3 Journaling Effects on Performance

In this section, we consider a variation of both PMAL and DAX. Recall that PMAL journals in journal mode, which is typically shunned in disk-based storage systems due to their low performance. Here, we also consider the effect of relaxing journaling to ordered mode for PMAL, which we refer to as PMAL-O

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

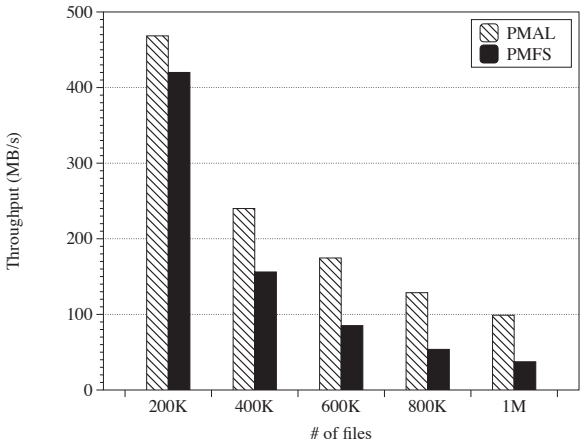
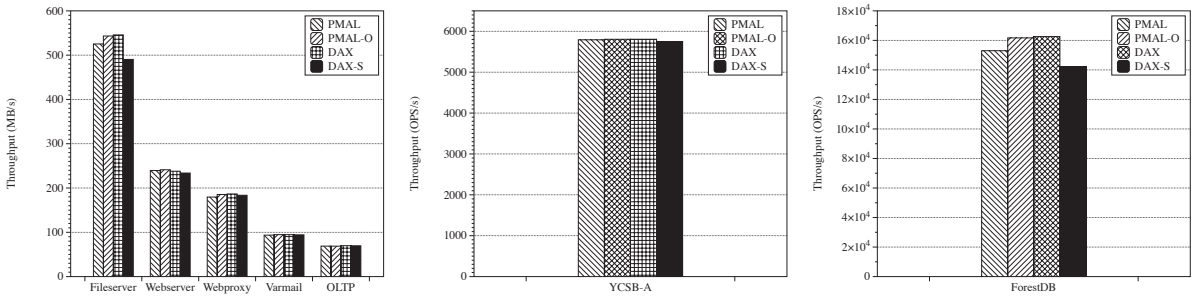


Figure 6: Performance comparison of PMAL and PMFS for the Fileserver workload as the number of files managed is increased



(a) Filebench macro benchmarks (b) YCSB-A benchmark (c) ForestDB benchmark

Figure 7: Performance comparison of PMAL and DAX for different consistency guarantees

(for ordered mode). PMAL-O can be implemented by simply not performing PMAL Flush on the metadata so that the metadata is not written into the metadata portion of the file system until checkpointing time. For a variation of DAX, which does asynchronous metadata commits, we consider synchronously committing the metadata into the journal. This sacrifices performance, but allows for stronger consistency than the current version of DAX, and we refer to this version as DAX-S (for synchronous).

The performance effect of these changes in the I/O flow is shown in the right two PMAL-O and DAX-S columns of Table 5. We see that for PMAL-O, the time consumed at the PMAL component is reduced, while for DAX-S, the time consumed at the FS Consistency component increases considerably. The macro effect of these changes on the performance of the benchmarks are shown in Figure 7 and discussed below.

As expected, the performance of the benchmarks with PMAL-O does better than with PMAL and DAX-S does worse than DAX. Overall, PMAL-O does consistently better than the others, while DAX-S is consistently worse than the others. However, the gains by PMAL-O over PMAL is smaller than the loss by DAX-S over DAX.

The conclusion from these results is that given the same consistency level (PMAL-O and DAX-S), the performance of PMAL is consistency better. Also, PMAL is the only mechanism that can support

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

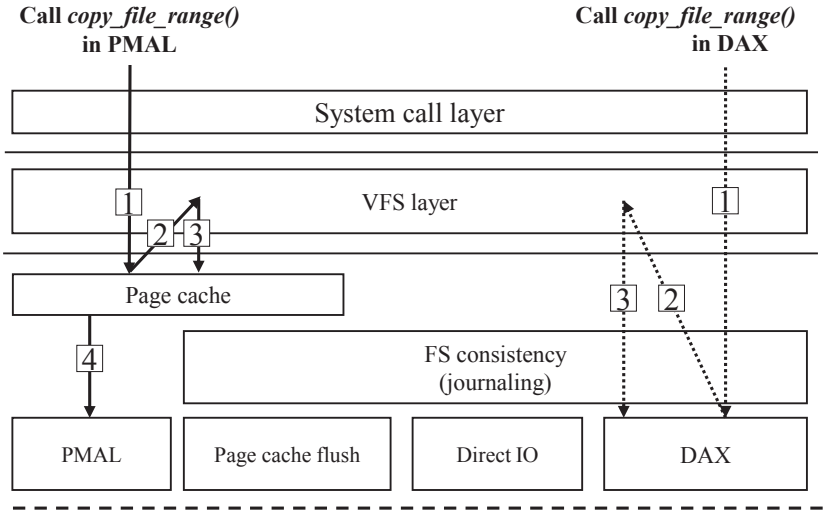


Figure 8: Call flow of `copy_file_range()` in PMAL and DAX where boxed numbers represent the execution steps

journal mode journaling and even with this higher level of consistency guarantee, performance is in par with DAX.

3.3.4 Page Cache as the Focal Point

Legacy file systems retain years of understanding and maturity. In Linux file systems, the page cache, through which key data related activities such as mirroring, compression, and encryption occur, plays a vital role. While these features are constantly being enhanced, new features are also being added. At the center of these activities, the page cache plays a central role. On the other hand, approaches such as DAX, PMFS, or NOVA that avoid the page cache altogether can be regarded as a new PM focused feature rather than an integration into the existing file system even though it is implemented within a particular file system.

In this section, we take a concrete example, in particular, the `copy_file_range()` system call [87, 91–93] and observe the effect of PMAL in comparison to DAX. `copy_file_range()` is a system call that has been added to the Linux kernel since version 4.5, whose role is to optimize the performance of data passing between files through a single user/kernel mode change unlike traditional data passing techniques. This is done by performing data passing through asynchronous page cache reads and writes in the kernel. This process is depicted on the PMAL (left) side of Figure 8.

In contrast to PMAL, DAX must take a different route in servicing `copy_file_range()` as it cannot make use of the page cache. Hence, for DAX, this system call goes through the DAX I/O layers as shown on the DAX (right) side of Figure 8. This incurs considerable extra overhead compared to PMAL, in particular, for Steps 1 and 2 (denoted by the boxed numbers), which are file read requests. Moreover, the fact that there needs to be two different implementation flows within the same Ext4 framework disperses the efforts that can be put into kernel development.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

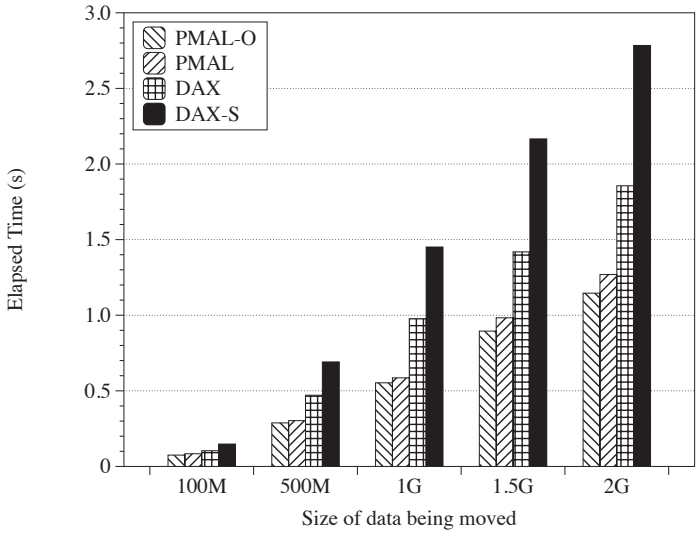


Figure 9: Function `copy_file_range()` performance according to whether or not page cache is used

To see the quantitative effects of the two I/O flows, we measure the elapsed time of the `copy_file_range()` system call for various file sizes. The Linux kernel version 4.7, with PMAL ported, is used for these measurements. The results, depicted in Figure 9 where the *y*-axis is the elapsed time, show that PMAL, which makes use of the page cache, performs considerably better than DAX. While we cannot say that PMAL will bring about these kinds of large performance benefits for all cases, we can say that through the PMAL approach, we should be able to directly and easily transfer current and future traditional storage related kernel development efforts into future PM devices.

3.3.5 PMAL in Hybrid Memory Environment

As mentioned previously, PMAL can be implemented in a hybrid memory environment by maintaining an extra multi-versioning area like the journal in PM. We also implement and conduct experiments with such a scheme. Overall, we find that there is roughly a 10% performance degradation compared to the PM-only version, which is mainly due to the journal commit and checkpoint overhead.

3.4 Summary

New memory technologies such as PCM, STT-MRAM and 3D XPoint are expected to make an impact on all levels of computing. These new memory, which we refer to as Persistent Memory (PM), boasts performance in par with DRAM while providing nonvolatility. Recently, there have been significant efforts to develop storage systems targeted specifically for PM storage [10–21, 30–35, 43, 53, 56, 80, 81]. This is based on the premise that legacy file systems are sub-optimal on memory bus attached PM storage. However, many years of time and effort are ingrained in legacy file systems that are now mature and time-tested, and making use of them seems a logical choice.

In this dissertation, we considered making use of legacy file systems for PM storage so that we can leverage the maturity ingrained in legacy file systems while, at the same time, reaping the high

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

performance offered by PM. Through empirical evaluation, we analyzed the workings of legacy file systems, in particular, the Ext4, and PM-dedicated file systems, DAX, PMFS and NOVA. Based on these analyses, we developed the Persistent Memory Adaptation Layer (PMAL) module that allows us to attain our goal of retaining maturity as well as high performance. PMAL is lightweight comprising roughly 180 lines of code in total and can be easily integrated into legacy file systems. Using Filebench and key-value store benchmarks we showed that performance of PMAL integrated Ext4 performs in par with PM-dedicated file systems. This is in spite of the fact that PMAL integrated Ext4 provides high consistency guarantees.

Finally, and more importantly, while the approach of PM-dedicated file systems separates traditional storage related kernel development efforts from those for PM, the approach that PMAL takes allows kernel developers to concentrate on the outcome that have matured with time and that are available today. Furthermore, future development efforts also need not take a two-track approach, one for traditional storage and another of PM storage, but can be concentrated to one concerted effort.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

IV Improving Performance of Persistent Memory File Systems Through a Lightweight Consistency Mechanism at the VFS Layer

With the advent of Persistent Memory (PM), we are anticipating dramatic changes in computer system organization as well as in the system software that run these systems. File system development has been at the forefront of these changes as the performance characteristics of PM as storage are very different from traditional storage devices. With PM anticipated to be attached to the memory bus through the DIMM interface, file systems that exploit DRAM-like latencies are continually being developed [10–19, 32, 35, 56, 80, 81].

The use of PM, however, is also expected to broaden. Though originally thought to be too slow to replace DRAM main memory outright, recent studies have even considered the use of PM as the CPU cache [76–79]. Also, a recent study by Kim et al. has shown that modern CPUs employ various optimization techniques such that the long write latency typically found with PMs, which was thought to be a critical limitation for deployment of PM as main memory, actually has very little effect, if any, on the performance of actual, meaningful applications [94]. Previous studies have also presented possibilities of replacing all main memory with PM [41, 75, 95–97]. This dissertation also assumes that the entire memory is PM, but unlike Whole-System Persistence [41] where a failure is a suspend/resume event, we assume that only the CPU cache is flushed upon failure. We will refer to this as Cache Flush Persistence (CFP). This allows us to extend the failure model from the transient power failure model to a general transient fault model. This dissertation is presented under this premise.

Overall, the technology to be presented in this dissertation has similar motivation and goal of the PMAL described above. The difference of the proposed technology in this dissertation is that it provides a new concept of lightweight consistency mechanism suitable for PM-based storage by focusing on the file system consistency mechanism. The contribution of this dissertation are twofold. First, we present a simple yet general method, which we call MeLo@V (Metadata Logging at the VFS layer), for providing file system consistency for all VFS-based file systems (not only PM-dedicated file systems, but also the legacy file systems) without overhauling the file systems and incurring overhead. This is significant because modules for providing file system consistency is a large source of overhead in contemporary file systems, as shown in Section 3.1. Hence, a key contribution of MeLo@V is in providing file system consistency with minimal overhead, which can be achieved by inserting a simple mechanism, comprising roughly 150 lines of code, in the VFS layer of the file system. The second contribution is in improving the performance of PM-dedicated file systems by making use of MeLo@V. As MeLo@V itself ensures consistency and CFP ensures persistence, these aspects of PM-dedicated file systems can be removed. As MeLo@V is lightweight, this results in improved performance.

Through experiments conducted on a PM emulating system, we show that the legacy Ext4 and Btrfs can be converted to PM-aware file systems will little performance sacrifice (roughly 2%). Our experiments on various workloads also show that the performance of Ext4 with DAX extension [59], PMFS [10] and NOVA [56] can be improved upon by 2 to 9% by incorporating MeLo@V and slight

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

modifications. From the perspective that development of file systems that better exploit PM must continue, our technique also allows developers to design new file systems for PM without considering the consistency mechanism, as MeLo@V is applied on the VFS layer and does not infringe on the design and implementation of the file system specific layer.

The remainder of the chapter is organized as follows. In Section 4.1, we give a detailed analysis of various file systems, breaking down the components that comprise the file systems in terms of functionality and performance. This stands as the motivation behind the development of MeLo@V. Then, in Section 4.2, we describe the design and the implementation of MeLo@V. Section 4.3 presents the evaluation platform, benchmarks and the experimental results. Finally, Section 4.4 concludes the chapter with a summary and conclusions.

4.1 File Systems and the Challenges

Recall that one of our goals is to provide a lightweight file system consistency mechanism, that is, develop MeLo@V, in particular. In this section, we discuss the empirical analysis that we perform that forms the basis for designing MeLo@V. While we have analyzed essentially all major legacy file systems, in the following, we discuss Ext4 only as the findings from other file systems are generally similar. We also analyze the I/O flow of the DAX, PMFS, and NOVA file systems, which are state-of-the-art PM-based file systems. All file systems are those supported in Linux version 4.3.3.

The objective of this analysis is to compare the components of the file system and see how they affect performance. We first analyze the I/O flow of file systems in function units (for example, `vfs_write()`, `new_sync_write()` and `generic_perform_write()` within the `write()` system call), and then classify them into components based on their main functionality. Based on this classification of functionality, we measure the time consumed by each of these components during execution for writes. The measurements (average of 5 executions) are taken when making a small (less than 4KB) write request for the file systems mounted on the Ramdisk or the `pmem` driver. Note that `pmem` is a device driver that emulates DRAM as a PM-based storage system for PM-dedicated file systems [83, 84]. This driver is implemented by considering persistence issues of the PM medium. The `ktime` [85] Linux kernel library is used to make the measurements. (The exact hardware platform in which these measurements are taken is discussed in Section 4.3.)

4.1.1 Observations

Table 2 shows the measurement results. The observations are as follows.

System Call and VFS Layers: In the system call and virtual file system layers, we observe that the main overhead of Ext4 and DAX comes from the 'I/O type switch' component. Fundamentally, I/O in Ext4 is operated through three modes, namely, asynchronous, synchronous, and direct, while the PM-dedicated file systems are operated through only one I/O mode that directly performs I/O into PM storage. The difference between Async and Sync fields in Ext4 in the 'I/O type switch' component comes from the fact that synchronous I/O is asynchronous I/O plus work associated with the page cache

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

flush. Therefore, synchronous I/O includes extra overhead to switch the I/O flow. As DAX is based on Direct I/O in Ext4, DAX also has the 'I/O type switch' component.

FS Specific Layer: Let us now analyze the file system specific layer for Ext4. First, let us consider Asynchronous mode. As shown in the Async column of Ext4, this flow comprises two components in the FS Specific Layer, namely, 'Page cache' and 'Memory I/O' as a write request completes when data reaches the page cache. We note that the 'Page cache' component, which is purely software overhead for processing the page cache, incurs more overhead than the 'Memory I/O', which is the component that performs the actual memory writes. These numbers show that in PM storage environments software overhead can have a significant effect on overall system performance.

In the Synchronous mode Ext4, there are two other components, namely, 'Page cache flush' and 'FS consistency' as shown in the Sync column. These components' main tasks are to flush data in the page cache into backing storage and to manage the file system consistency mechanism, respectively. We observe that these components incur considerably more overhead than the other components.

We now turn to the DAX, PMFS and NOVA file systems in Table 2. These PM-dedicated file systems have no components related to the page cache since they do not use the page cache. Each of these file systems has a component, which does not exist in Ext4, that writes data to PM storage denoted on the last row of the FS Specific Layer component. Note that this writing of data incurs the most overhead among all components for these file systems. These PM-dedicated file systems also have the 'FS consistency' component. In particular, DAX uses journaling like Ext4, while PMFS and NOVA use logging and lightweight atomic updates to guarantee file system consistency. The overhead here is lower than the Synchronous mode of Ext4, but is still significant.

4.1.2 Analysis and Implications to MeLo@V

Our observations from Section 4.1.1 show that, aside from writing to media that must eventually happen, two components, namely, the 'Page cache flush' and 'FS consistency' components stand out. In legacy file systems, these are mechanisms to persist the data sitting in volatile memory in a consistent manner. This is essential in keeping the data sane. In PM-dedicated file systems, the 'FS consistency' component is essential in guaranteeing file system consistency. However, the results show that these components incur considerable overhead. As we assume Cache Flush Persistence (CFP), where all main memory is nonvolatile and the cache is flushed upon a failure, we need to reconsider the functionality of these components. The key challenge here, then, is to provide functionality of the 'Page cache flush' and 'FS consistency' components with negligible overhead and without making any or minimal changes to the file system.

To relieve of the 'Page cache flush' overhead we make use of the page cache as journal space that keeps a dirty copy of the backing store similarly to UBJ (Union of Buffer cache and Journaling), a buffer cache architecture proposed by Lee et al. [51, 52], the difference being that we do not make any changes to the page cache mechanism. The only change that we make is that all writes are converted to asynchronous writes at the VFS layer. Note that this is because under the CFP assumption, synchronous

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

I/O is no longer necessary. Hence, no page flushes are explicitly incurred, and we simply let the page cache daemon do its work without any changes.

To relieve of the 'FS consistency' overhead, we design a lightweight component that logs VFS layer metadata for recovery upon system failure. We show that this simple design, which we refer to as MeLo@V (Metadata Logging at the VFS layer), is enough to recover the file system into a consistent state upon system failure. This is the core of our contribution, and we discuss MeLo@V in detail in Section 4.2.

4.2 MeLo@V: Design and Implementation

In this section, we first give an overview of the MeLo@V scheme. Then, we discuss in detail the logging and recovery mechanisms using writes. Finally, we discuss modifications due to creating and deleting files.

4.2.1 Overall Architecture

One of our goals is to develop a lightweight consistency mechanism for file systems making use of PM storage under the CFP assumption. For legacy file systems, the key source of overhead is the page cache flush, which does not exist in PM-dedicated file systems, as seen in Section 4.1. The removal of this overhead in legacy file systems is done based on the CFP assumption, where the CPU cache contents can be safely flushed into nonvolatile memory before shutdown of the system. Due to this assumption and that the page cache is nonvolatile, the legacy file system itself supports multi-versioning as a duplicated, dirty copy of clean data in file system space will safely reside separately in the page cache area upon updates. Hence, all synchronous user I/O can safely be converted to asynchronous I/O as the page cache is now nonvolatile. This hides the page cache flush overhead in legacy file systems. No explicit page cache flush is invoked and all page cache flush related operations are delegated to the page cache daemon. The overhead for activities of the page cache daemon is generally hidden from end users as it is run in the background.

Note here that we are in essence solving the same problem as UBJ in maintaining a consistent copy in the nonvolatile page cache [51, 52]. While UBJ does this by freezing and having multiple copies of pages, our consistency mechanism does this by replaying the actions into the page cache as we discuss in detail later. This allows the system to employ the same page caching mechanism provided by the legacy file system. Note also that as the legacy file system applying our consistency mechanism makes use of the page cache, its size will have an effect on performance. We discuss this issue in Section 4.3.2.

Having removed the page cache flush overhead from legacy file systems, maintaining file system consistency without incurring overhead becomes a common target for both legacy and PM-dedicated file systems. For this, we propose MeLo@V, which is based on the key observation that extra care need only be taken for operations that modify existing data, which is done in the page cache space of the legacy file system or the file system space of PM-dedicated file systems. Thus, MeLo@V makes small modifications at the VFS layer to track these changes. As modifications are made at the VFS layer,

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

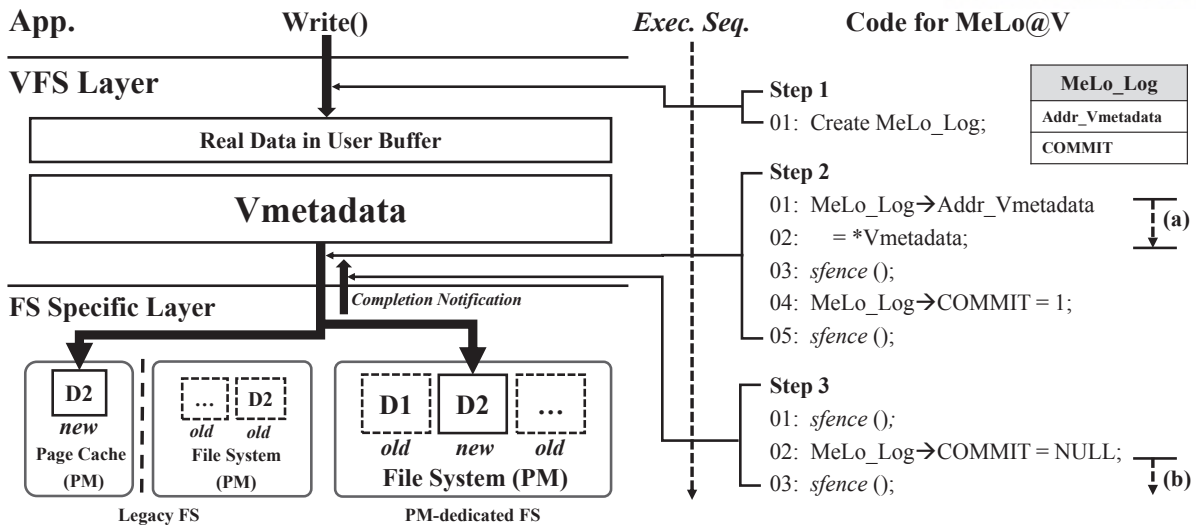


Figure 10: The figure on the left shows the execution sequence, while the Steps on the right are MeLo@V code inserted in sequence at arrow designated points.

MeLo@V is a general solution that can be applied to all file systems that make use of the VFS layer.

Overall, the key role of MeLo@V is logging of information (metadata) in the VFS layer that refers to data that is to be modified (for example, `file`, `inode`, `kiocb` and `iovec` structures in the Linux kernel). Such metadata is created in the VFS layer based on the parameters that are passed through the application system call. While we describe the exact behavior of MeLo@V in the next section, the key idea here is that if a system failure occurs while modifying data into the nonvolatile page cache or file system space, the system, for recovery, can simply replay (redo) the failed modifying activity using the log information MeLo@V had generated. This guarantees file system consistency without the overhead and extra writes typically involved with consistency in file systems.

In the following section, we give a detailed description of the logging mechanism involved with MeLo@V. Then, in Section 4.2.3, we discuss the recovery mechanism. We emphasize that all of MeLo@V is implemented in roughly 150 lines of code.

4.2.2 Logging in MeLo@V

For logging in MeLo@V, consider Figure 10, which depicts the execution sequence for file modification starting from top to bottom. The flow on the left is the sequence in file systems, and the Steps on the right are the specific actions taken with MeLo@V. The arrows pointing to the left indicate the position at which these Steps are inserted and executed within the typical execution flow. We note that file modifications can occur as files are created, updated, and deleted. In the following, the discussion focuses on updating, and issues regarding creating and deleting files are discussed separately in Section 4.2.4.

Upon a file update request, a log data structure, which we refer to as MeLo_Log with elements Addr_Vmetadata and COMMIT both initialized to NULL, is created at the VFS layer. This is Step 1 in Figure 10. This MeLo_Log is created per file when the file is opened and is removed when the file closes. Addr_Vmetadata will hold the pointer to the VFS metadata that the system can fall back to upon failure

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

and the COMMIT marker will hold the current status of the execution sequence. The only persistence assumption that we make is that 8-byte writes to change COMMIT be atomic. Recall that such failure atomicity assumptions are typical [10, 44, 86].

With Step 1 done, the following activities ensue and are repeated for every file update. First, information regarding the data to be written that is passed from the application through the `write()` system call are stored into, what we call, Vmetadata (for VFS metadata). In the Linux kernel implementation, structures `kiocb`, `iovec`, and `file` of the VFS layer fall into this category [98]. Note that other kernels have similar data structures [99,100]. Typically, once Vmetadata is stored, this metadata is passed on to the FS Specific layer by calling a file descriptor operation (which in Linux, is the `file→f_op→write_iter()` operation).

For MeLo@V, though, just before this step, Step 2 is inserted into the sequence where the address of Vmetadata is logged in Addr_Vmetadata. (In Linux, this would correspond to three separate addresses for structures `kiocb`, `iovec`, and `file`.) Then, to mark that MeLo_Log is safely written, the COMMIT marker is set to 1. Note that `sfence` instructions are added before and after setting the COMMIT marker to 1 to prevent reordering of instructions. Typically with PM, to ensure the persistence of the data associated with recovery, the `clwb` instruction would be called to persist the Real Data, Vmetadata, and elements of MeLo_Log that are needed for replay during recover upon system failure. However, since we base our system on CFP, we do not need to call `clwb` whenever the data are changed at runtime.

After Step 2, when the modified data is written into the page cache/file system space, a persistent new version of the data is retained and the I/O flow is complete. To ensure correct recovery in case of a fault, Step 3, where the COMMIT marker is set back to NULL, is performed before I/O completion is notified to the user. At this point, file system consistency is guaranteed. Again, `sfence` instructions are placed to prevent reordering. Note that Step 3 is also performed in the VFS layer right before notification to the application.

4.2.3 Recovery in MeLo@V

We now discuss how file system consistency is maintained upon system failure for updates to a file. In fact, there is no special recovery mechanism and we simply rely on the replay recovery process to recover from the failed write process using the MeLo_Log elements. Since we assume CFP, upon reboot all data and data structures related to I/O execution before the system failure will remain intact. This, with the MeLo_Log, is all that is required for recovery. The details are as follows.

To start the recovery process using MeLo_Log, we first need to find the address of MeLo_Log in the descriptor of the files that were open during abnormal system termination due to failure. As this can be found in the process descriptor, we first find the processes that were modifying files in our PM storage before system failure. We then search the process descriptor (in Linux, `task_struct`) for files that remained open. The descriptor of these files will have the MeLo_Log address for that file. In particular, in Linux, the member structures of interest in the process descriptor are `files_struct` and `fdtable`.

Once MeLo_Log is found we consider the COMMIT marker value and recovery as follows.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

COMMIT is NULL: This means MeLo_Log was created but never used (Step 1) or in either code portions (a) or (b) in Figure 10. In any of these cases, either new data was not written to the page cache/file system space or what was written is valid data. Hence, there is really nothing to do for recovery.

COMMIT is 1: This means that the new modification was not able to commit, that is, no new data was written in the page cache/file system space or the data written to the page cache/file system space is invalid. In this case, MeLo@V recovers simply by replaying the previous failed write using data obtained through MeLo_Log. Specifically, in Linux, `file→f_op→write_iter(kiocb, iovec)` is called.

4.2.4 Creation and Deletion of Files

Let us now see what data is logged and how recovery works for creating and deleting files. When creating a new file, no old values exist. Hence, MeLo@V does not save any particular metadata and there is nothing to recover in case of failure. For the delete case, things are a bit different. For logging, the process is exactly the same as updates, except that the data residing in Vmetadata are the directory entry and the inode. Then, for recovery, in Linux, `inode→i_op→unlink(inode, dentry)` is re-executed to delete the file.

4.3 Performance Evaluation

In this section, we evaluate the effects of MeLo@V on the performance of file systems. Here, we also present results that quantify the second contribution of this dissertation, that is, improving performance of contemporary PM-dedicated file systems through MeLo@V. For the evaluation, we used the experiment platform and benchmarks used in the previous dissertation (Table 3 and 4). In Section 4.3.1, we describe the various file systems and the modification that we make to these file systems in order to quantify the effects of MeLo@V and the modifications made. Finally, in Section 4.3.2, we provide a comprehensive set of experimental results that support our contributions.

4.3.1 File System Variations

For our experiments, we take 5 file systems supported in Linux kernel version 4.3.3, namely, Ext4 and Btrfs, the legacy file systems and PMFS, Ext4 with DAX (denoted Ext4+DAX) and NOVA, the PM-dedicated file systems. Note that for the legacy file systems, they are set to asynchronous I/O mode, but without any persistence mechanism included, hence are the best performance results achievable for legacy file systems, but lack consistency and persistence guarantees. These file systems without any modification are the reference file systems and are referred to as the Original File Systems.

To evaluate the effect of MeLo@V, three variations are considered for this file system set. The first is with only MeLo@V integrated into the VFS layer, without making any changes to the Original File Systems. This version serves to quantify the overhead of MeLo@V on overall performance. These will be denoted FS_Name^{+M}, the superscript $+M$ denoting the fact that MeLo@V has been added. For example, PMFS^{+M} denotes the PMFS file system with MeLo@V deployed. In general, $+M$ will denote that MeLo@V is deployed.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

The next set of file systems are ones with MeLo@V deployed, but the original consistency mechanism disabled. Note this does not sacrifice any consistency as MeLo@V itself guarantees consistency, but has the effect of disabling or removing the extra consistency overhead imposed by the original implementation. How the Original File Systems are modified are as follows.

For Ext4, recall Ext4 uses journaling to ensure file system consistency [1]. Journaling is a multi-versioning mechanism that first writes the new data in the journal area without directly modifying the old data residing in file system space. When the journal area becomes full, the checkpoint operation is invoked to move the data in the journal area to file system space to finalize the write, causing additional copy overhead. To remove the effect of this overhead, we increase the size of the journal area by manipulating the format option of Ext4 so that the entire dataset of the workloads are handled without any checkpointing. This serves our purpose of hiding the overhead for maintaining consistency in the original Ext4. Since Ext4+DAX also uses Ext4-based journaling, we configure Ext4+DAX in the same manner. For Btrfs, which uses COW, we simply make use of the option to remove COW when mounting Btrfs [101].

The PM-dedicated file systems PMFS and NOVA do not provide any means to manipulate the consistency mechanisms as was done with legacy file systems. Hence, we modify their code without invading their own inherent features. For example, for NOVA we leave the parts that make use of log-structured mechanisms untouched as they are inherent characteristics of the file systems. In contrast, we remove other portions of the file system that make use of journaling or COW to retain consistency [56]. We find that in the PMFS and NOVA implementations, these consistency mechanisms are cleanly separated. Hence, making such modifications were relatively straightforward. We denote file systems in this set as FS_Name$^{+M}_{-C}$, with the superscript $+M$ denoting the fact that MeLo@V is deployed and subscript $-C$ to denote that the consistency mechanism has been removed. Again, in general, $-C$ will denote that the original consistency mechanism has been removed.

The final set of file systems add one more modification to the previous set of file systems. That is, we remove the persistence mechanism support, that is, the `clwb` instructions incorporated into the original PM-dedicated file system implementations. As we assume CFP, persistence is obtained through CFP, and so the `clwb` instructions are simply unnecessary extra overhead. These modifications are done manually, and we denote these file systems as FS_Name$^{+M}_{-PC}$ following previous convention, with P referring to persistence. Note that Ext4 and Btrfs do not make use of the `clwb` instruction, hence Ext4$^{+M}_{-PC}$ and Btrfs$^{+M}_{-PC}$ are the same as Ext4$^{+M}_{-C}$ and Btrfs$^{+M}_{-C}$, respectively.

4.3.2 Experimental Results

Overhead of MeLo@V: We first look at the overhead that MeLo@V incurs. Figure 11 shows the performance comparisons of file systems for the Filebench benchmarks and the YCSB-A and ForestDB benchmarks. For each sub-figures, the y-axis is the throughput, where higher is better. The figures show the performance of the original version and all three variant versions for each file system in each subgraph.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

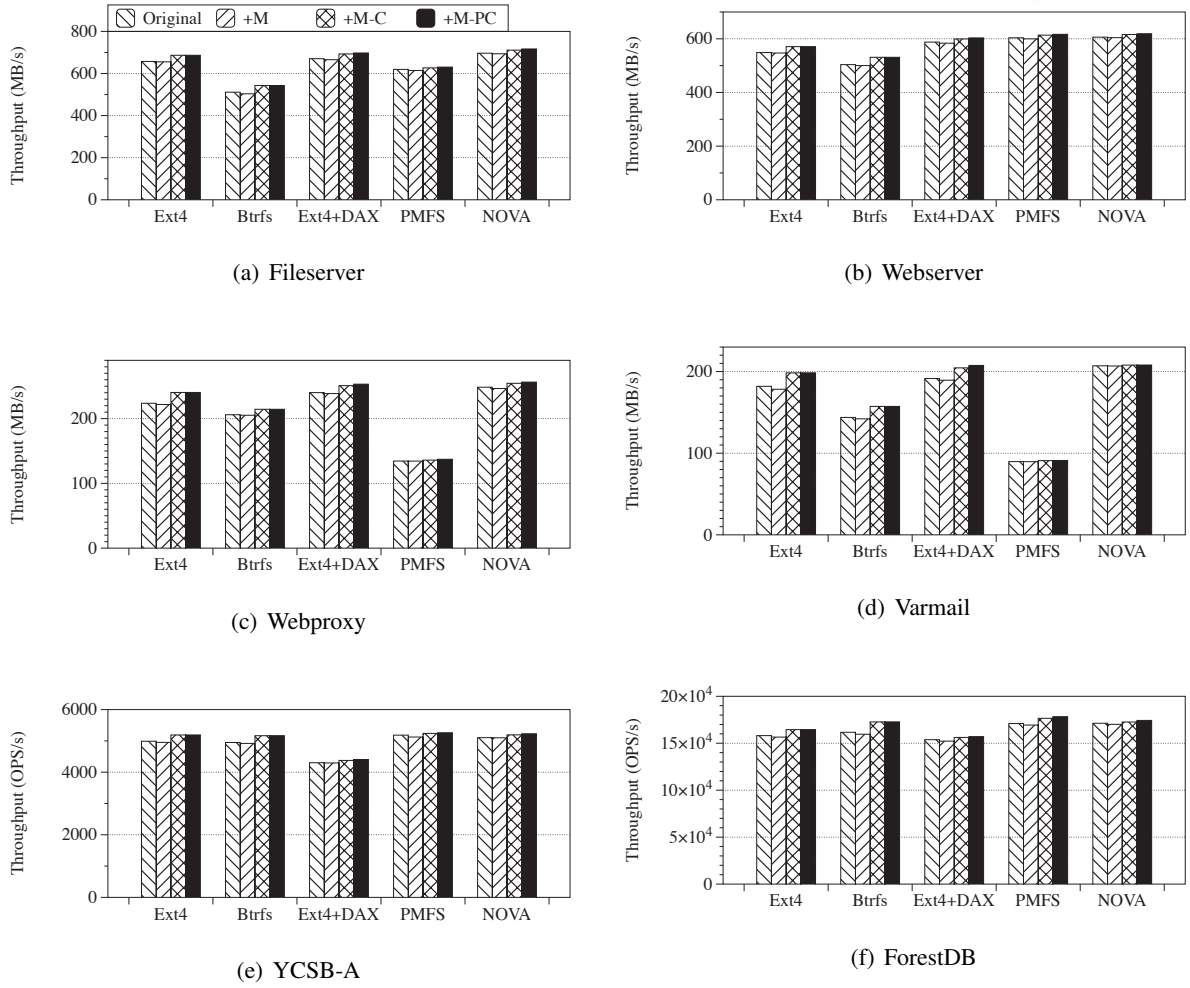


Figure 11: Performance of Filebench and key-value store benchmarks

The difference between the first two bars of each file system, that is, the Original and +M in every sub-graph quantifies the overhead incurred by MeLo@V. Even with MeLo@V deployed performance is only slightly, roughly 2%, below those of the Original File Systems. This is consistent throughout all workloads. Recall that for Ext4 and Btrfs, these are asynchronous versions with the best performance possible, but that do not support consistency and persistency, and that the +M versions are now PM-aware file systems.

Table 6 presents the breakdown of the execution time for a `write()` with MeLo@V deployed, similarly to that obtained for Table **??**. The results show that the overhead of the 'MeLo@V' component is much smaller than the overhead of the 'FS consistency' component of each file system. We also see that the overhead of the 'MeLo@V' component is almost constant.

Overall, we see that the overhead of MeLo@V is light. Thus, employing MeLo@V into legacy file systems and converting them into PM-aware file systems can be done without burden. This allows individual file systems to retain their strengths and yet transparently be used in a PM environment. More is discussed on this issue later.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 6: Running time (microseconds) for components in `write()` execution flow with MeLo@V deployed

Component	Ext4^{+M}	Ext4+DAX^{+M}	*PMFS*$^{+M}$	NOVA^{+M}
System call gate	0.3	0.2	0.3	0.3
VFS objects	1.0	0.9	0.9	0.8
MeLo@V	0.4	0.3	0.4	0.3
I/O type switch	3.2	2.2		
Page cache	17.3			
Memory I/O	0.4			
Page cache flush				
FS consistency		7.1	4.9	3.1
DAX|PMFS|NOVA		13.2	14.2	16.2
Total Elapsed Time	22.6	23.9	20.7	20.7

Overall Performance: We now make some observations from the overall results presented in Figure 11.

First, we see that the removal of the consistency and persistence mechanisms inherent to the file systems as MeLo@V is deployed makes a difference in performance. This is observed by the third and fourth bars, +M-C and +M-PC, in Figure 11, in comparison to the first bar, that is, the original file systems. (Note that for Ext4 and Btrfs, +M-C and +M-PC are the same as mentioned previously.) For Ext4+DAX, PMFS, and NOVA, the performance improvement ranges from 2% to 9%, and this tell us that the use of MeLo@V is lightweight and effective.

Second, we observe that, though depending on the workload and file system, Ext4$^{+M}_{-PC}$ and Btrfs$^{+M}_{-PC}$ perform in par with, sometimes even surpassing, the original PM-dedicated file systems. This tells us that MeLo@V is an effective way of converting legacy file systems into PM-aware file systems without overhauling the entire system.

Finally, we observe that PMFS does particularly worse than other file systems for the Webproxy and Varmail workloads. Our analysis find that this is because PMFS is strongly affected by workloads that create a large number of files and directories, which Webproxy and Varmail are, as it maintains a heavy metadata indexing structure.

Effect of Page Cache Size on MeLo@V: As MeLo@V makes use of the legacy file system page cache, we investigate the effect of the page cache size on MeLo@V performance. As the page cache size is dynamically determined in Linux, we perform experiments as we vary the storage size (which is 56GB for the results reported thus far), hence altering the size of memory, which includes the page cache. These experiments are performed for the original Ext4 and Btrfs file systems and their variants Ext4$^{+M}_{-PC}$ and Btrfs$^{+M}_{-PC}$. The results are depicted in Figure 12.

Our observation is that performance decreases with memory size decrease for all file systems. However, we find that the rate of decrease is much lower for Ext4$^{+M}_{-PC}$ and Btrfs$^{+M}_{-PC}$ compared to the original

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

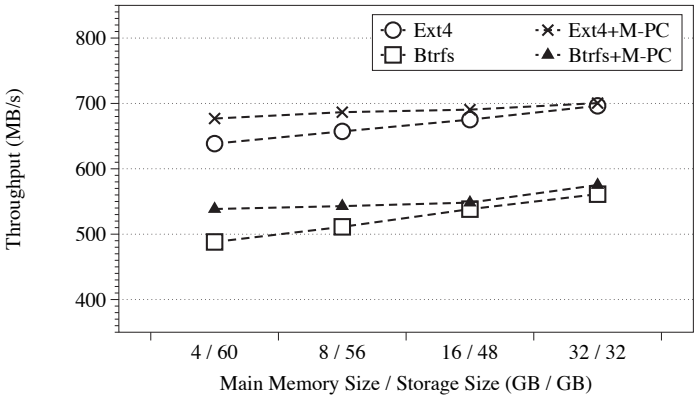


Figure 12: Performance of Ext4, Btrfs, Ext4$^{+M}_{-PC}$, and Btrfs$^{+M}_{-PC}$ for the Fileserver workload as the page cache size is increased

cases. This is because with Ext4 and Btrfs maintaining consistency accompanies flushing, which is sensitive to memory size. Hence, more flushing occurs with less memory. In contrast, Ext4$^{+M}_{-PC}$ and Btrfs$^{+M}_{-PC}$ removes this heavy consistency mechanism and replaces it with a lightweight mechanism provided by MeLo@V. Hence, it is only slightly affected by the change in page cache size.

Retaining Legacy Features with MeLo@V: Existing features are constantly being enhanced and new features are being added to legacy systems. Retaining such features and the years of understanding of legacy file systems is important to users as well as developers. It may take years for PM-dedicated file systems to provide a fully functioning stable version for lay users. As MeLo@V is a simple technique applied to the VFS layer, MeLo@V provides a means to carry old features on and to continue developing new features into widely deployed systems without any interruption.

Here, we take a concrete example of how MeLo@V affects a feature that has been added to Linux since kernel 4.5.x, namely, the `copy_file_range()` system call [87, 91–93]. This call optimizes performance of data passing between files through a single user/kernel mode change by making use of asynchronous page cache reads and writes in the kernel. To support `copy_file_range()` in MeLo@V deployed PM-aware legacy file systems, that is, Ext4$^{+M}_{*}$ and Btrfs$^{+M}_{*}$ nothing needs to be done. In contrast, PM-dedicated file systems would need a totally different implementation as they do not use the page cache. In particular, even though Ext4+DAX is developed under the Ext4 framework, it is implemented in a totally different manner as DAX also cannot make use of the page cache [59, 60, 102]. This is an example of extra effort that must be put in for these new file systems supporting PM; two different implementations for the same functionality.

Not only is extra effort required, it can also have quantitative effects. To show this, we measure the elapsed time of `copy_file_range()` for various file sizes on Linux kernel version 4.7, with MeLo@V ported. (Experiments so far were on Linux 4.3.3 as this is the only version that supports the three PM-dedicated file systems. We use 4.7 here as this supports `copy_file_range()`, while 4.3.3 does not.) The results in Figure 13 show that legacy file systems making use of the page cache through MeLo@V, that is, Ext4$^{+M}_{-PC}$ and Btrfs$^{+M}_{-PC}$, perform considerably better than Ext4+DAX, specifically, Ext4+DAX$^{+M}_{-PC}$.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

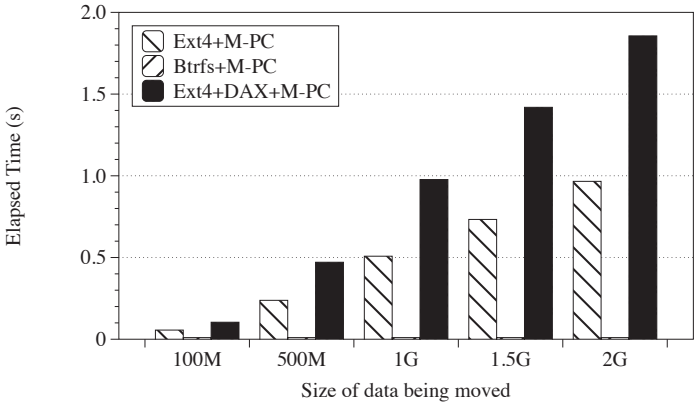


Figure 13: Performance of function `copy_file_range()` as the data size is varied

A particularly noteworthy result here is that of Btrfs$^{+M}_{-PC}$. We observe that Btrfs$^{+M}_{-PC}$ spends essentially no time performing this system call for all data sizes. This is because Btrfs uses `reflink` for data copy [103]. When files are copied using `reflink`, the copy simply shares the physical data of the original file. Only when the original or copied file is modified, is the corresponding data physically duplicated. Note that such peculiarities of legacy file systems are naturally exploited with our MeLo@V approach.

4.4 Summary

New memory technologies such as PCM, STT-MRAM, and 3D XPoint are expected to make an impact on all levels of computing. These new memory, which we refer to as Persistent Memory (PM), boasts performance in par with DRAM while providing nonvolatility. The existing block-based file systems with large overhead for flushing data in the page cache and guaranteeing file system consistency are naturally considered as sub-optimal for PM storage. Recently, numerous PM storage targeted file systems have been developed under the implicit assumption that legacy file systems are not suitable for PM [10–19, 32, 35, 56, 80, 81]. However, PM-dedicated file systems also require mechanisms for file system consistency that has a negative effect on performance.

In this dissertation, under the premise that the entire main memory is nonvolatile and that data in the CPU cache is safely flushed to PM upon a transient failure, we present a simple yet general method called MeLo@V (Metadata Logging at the VFS layer) for ensuring consistency of file systems. As conventional consistency mechanisms used in almost all file systems are heavy, we find that the key issue in making PM-aware file systems is providing file system consistency with minimal overhead. We show that this can be achieved by inserting a simple mechanism in the file system VFS layer. As MeLo@V ensures consistency, we also show that the performance of contemporary PM-dedicated file systems can be further improved by disabling or removing their consistency and persistence mechanisms. We conduct a comprehensive experimental study of our approach using various modern benchmarks on a contemporary system with PM emulated with DRAM. We find that the Ext4 and Btrfs legacy file systems can easily be converted to PM-aware file systems and recent PM-dedicated file systems such

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

as NOVA, PMFS, and Ext4 with DAX extension with MeLo@V show higher performance than their original forms that use the conventional heavy consistency mechanism. Our MeLo@V technique also allows developers to design new file systems for PM without considering the consistency mechanism as it is applied in the VFS layer and does not infringe on the design and implementation of the file system specific layer.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

V First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage

Persistent Memory (PM) is a new media with favorable characteristics such as nonvolatility and close to DRAM performance that can vastly improve storage I/O performance. To exploit these characteristics, new PM based file systems are being developed [10–15, 35]. These file systems, while providing high performance, suffer from the following two limitations. First, as the time to maturity for a file system is long and a file system is a complex beast [16, 17, 20], most of these relatively young and immature file systems will not survive the test of time. For example, among the many file systems proposed, currently, NOVA [11] and DAX [18] are the only ones that are stable and being maintained. Second, how these file systems will integrate with existing storage devices such as SSDs and HDDs is not straightforward. For example, both NOVA and DAX assume that the storage media is PM. In the long run, PM will fill up and some form of migration may be needed to vacate PM, which some file systems are not designed to do. Efforts to resolve this have been proposed. In their seminal paper, Kwon et al. propose a cross-media file system called Strata that can span a set of heterogeneous storage devices such as PM and traditional devices [14]. Separately, Ziggurat, proposed by Zheng et al., is a tiered file system that supports a combination of devices [15]. However, both proposals suffer from the first limitation; Strata, at the time of this writing, still is not fully functional [104] and Ziggurat is not open sourced and thus, its status is unknown.

The goal of this study is to encompass the goals of previous PM file system developments. More specifically, the goals of previously proposed PM based file systems can be summarized as follows. First, file system performance should extract the benefits that PM brings as storage media. This is an obvious goal when deploying PM. Second, it should support the more natural in-order file system semantics [14] brought upon by PM. In-order file system semantics is where all file system operations, including writes, occur in the exact order in which they are executed. Typically, this has been impossible due to the writes, which were too slow to persist in order because of the slow devices. Thus, application developers had to choose between performance and durability. While it is true that some applications do not require durability of writes or are highly optimized to minimize sync's, many applications knowingly choose performance over durability, being aware of the negative consequences of failures [19–21]. With the advent of PM, such choice seems unnecessary as providing in-order file system semantics should be natural and easy [11, 14, 20]. However, to date, supporting such semantics is possible only with a total revamp of the file system. Third, it should support traditional storage media alongside PM (similarly to Strata and Ziggurat) [14, 15]. Our dissertation encompasses these goals and yet, instead of developing yet another file system, our design allows to retain the modern, mature, constantly evolving file systems such as Ext4 developed for traditional storage devices.

In this study, we introduce First Responder (FR), the solution that we propose. FR, like a buffer cache, absorbs requests at the topmost layer of the I/O stack in PM then immediately responds to the requests (hence, the name). For reads, the requests end there, while for writes, the requests are forwarded

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

to the traditional file system, in effect, hiding the entire I/O stack overhead.

At first glance, FR simply appears to be nothing but a buffer cache that is persistent, but in reality, a lot more is involved. In fact, FR is a union of storage and a cache. As it will become more evident later, FR is storage as the contents that reside in FR are consistent and permanent like any that is stored in traditional storage media that have gone through the entire storage stack. Thus, in that sense, flushing to the traditional file system underneath is optional. FR is also a cache as the size is limited (though generally large) and eventually, it is flushed to the traditional storage media through the I/O stack. Naturally, the design of FR is much more involved as maintaining consistency under failure and providing featherweight management overhead become technical issues that need to be dealt with. We elaborate on these issues later.

There are two key benefits to FR. First, writes are immediately durable. Thus, even applications that do not require in-order semantics, but that issue occasional `fsync()`s can benefit. Second, FR can coexist largely independently of the underlying traditional file system. This allows for any mature file system to take advantage of the benefits of PM. This is especially important as efforts to move to disaggregation of resources is gaining traction [105–107]. Thus, much extra effort to integrate PM based file systems to any conventional setting including disaggregated settings is not necessary with FR. While we concentrate on the Ext4 file system in most of our discussions, we also show performance results when using Btrfs [2, 108] as the underlying file system.

Our performance evaluations with the Intel DC PMM using an FIO generated synthetic workload show that FR, when used in cache form, can outperform Ext4 by more than 9×, despite providing durable in-order file system semantics, while Ext4 cannot. Using the Filebench and YCSB benchmarks, we also show that when FR is used as part of a typical file system, performance is comparable with the default Ext4, Ext4 with DM-WriteCache, NOVA, and DAX, while, again, providing durable in-order file system semantics.

We present the design of FR in two separate sections; Section 5.1 discusses the overall design and in Section 5.2, we present a detailed discussion of how FR maintains consistency as this is key in providing a correct, yet efficient system to users. In Section 5.3, we perform a comprehensive evaluation of FR using the Intel DC PMM platform. We compare FR with NOVA and DAX, the two open source PM supporting file systems, and the default Ext4 and DM-WriteCache, which do not provide immediate durable in-order semantics supported by PM file systems. Then, we conclude in Section 5.5.

5.1 First Responder: The Design

In this section, we present the design of First Responder (FR). Before moving on, we set the premise on which our discussion is presented. First, even though FR could be implemented in either the user or kernel layer, we will describe it within the latter layer as we implement it in the VFS layer. Second, as the main dealings of FR are with the `read()` and `write()` calls and as `read()` follows trivially from `write()`, we concentrate on `write()` in the following discussions. We first discuss the assumptions, then the design choices that we make based on the challenges we face.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

5.1.1 Basic Architecture and Design Choices

Like many previous studies that assume memory to be a hybrid of traditional DRAM and PM, FR works under this assumption [11, 32, 62]. PM used by FR is a temporary storing ground, and yet it is also storage. That is, what is stored in PM is durable and consistent, though eventually, we anticipate its contents to be stored in traditional storage media such as SSDs and HDDs.

Traditional systems support two types of write modes, namely, synchronous and asynchronous. In FR, all aligned full chunk and/or new writes are synchronous, that is, they are persisted immediately into PM by FR. Writes of partial chunks already residing in underlying storage must first be read into PM for consistency, but we find them to be only a small portion of writes. FR manages the data and the location of where the data should be placed within PM, guaranteeing consistent and durable writes as PM is nonvolatile. Thus, applications upon receiving the response can continue execution being assured that the write I/O request was serviced successfully. Note that this has the consequence of dramatically reducing the I/O path for such synchronous writes compared to traditional file systems.

In the previous section, we mentioned that FR is like a buffer cache, but that much more is involved. We now discuss the two specific issues that must be resolved, that is, maintaining consistency upon failure and providing featherweight management overhead, which the traditional buffer cache cannot support.

Handling Overhead: Recall that the traditional buffer cache was developed as an optimization to alleviate the burden due to slow storage, which was orders of magnitude slower than DRAM. Thus, the buffer cache maintained separate data structures and elaborate replacement algorithms were developed. Compared to accessing slow disks, maintaining such data structures and running these algorithms were negligible. However, as previously observed, software overhead considered to be negligible in the past is no longer so with faster media [7–9]. This becomes more important as PM is close to DRAM performance, which means that every little overhead counts. Thus, management with such high cost is unacceptable with PM systems.

To alleviate such overhead, FR chooses a simple, static indexing scheme for placement as well as replacement as we explain below. However, we must first argue that this is a valid approach. For this, we perform our own set of experiments to quantify the software effects of managing a cache, which we elaborate on in Section 5.3.2. Briefly, we implement indexing structures such as the Radix tree and the LRU replacement policy and find the overhead to be an order of magnitude higher than the static approach that we propose, concluding that with faster media such as PM, software overhead is profound.

The obvious question here, then, is, won't this static indexing increase the miss ratio? The obvious answer, of course, is yes; but we argue that this is true only under the traditional assumption that the cache size is a very small fraction of storage capacity. This leads to our more important argument that, if the cache is sufficiently large, then, we may be able to avoid most cache misses. For reads, a miss is inevitable on the first read from storage. However, aside from these, it is well known that as the cache grows, the effect of the caching policy diminishes. For writes, a miss can be defined as a forced storage access when the new write collides with a block that is dirty. Then, so long as we keep all blocks

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

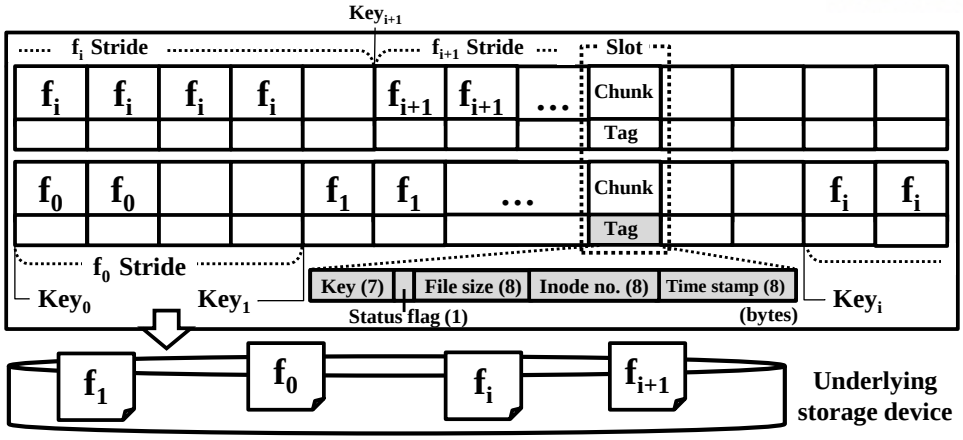


Figure 14: Internal organization and components of FR

clean, that is, synchronized with storage, then all misses can be eliminated. We show below that, under simplifying assumptions, if the cache is large enough all write misses can be avoided.

Imagine the cache to be organized in array form with the index i and S_b denoting the location and block size of the cache, respectively. Let C_s be the cache size and assume that for our system, the time to flush a block to storage and receive an acknowledgement (in order to synchronize) takes at most t_f time. Let us assume that the peak write rate for the cache media is r_{peak}GB/s. Assume that a very large write that is purely sequential arrives consecutively in S_b units and in t_i intervals, being placed from index $i = 0$ of the cache. To avoid all misses, the cache should be large enough such that when the entire cache fills up and wraps around to the beginning of the cache, the cache entries there are all clean. Thus,

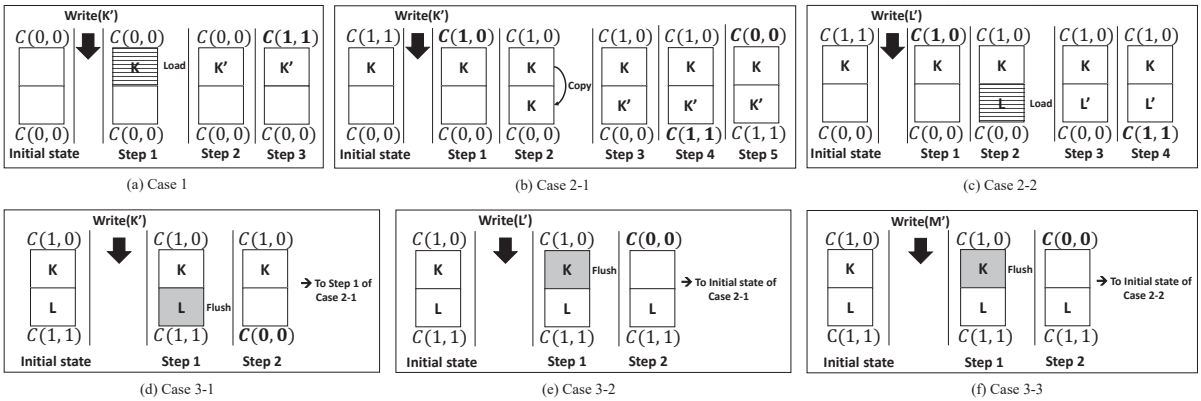
$$t_f \leq \frac{C_s}{r_{peak}} \leq \frac{C_s \times t_i}{S_b} \Rightarrow C_s \geq \frac{t_f}{t_i} \times S_b \tag{1}$$

Then, with C_s satisfying Equation 1, no forced flushing of dirty blocks will occur for sequential writes.

Let us apply Equation 1 on an example. Assuming $S_b = 4$KB, for current PM, we find that the peak write rate is roughly 4GB/s (not including any software overhead) when `clwb` and `sfence` operations are issued per chunk write. Thus, even when writes are coming in at full throttle, simple calculations show that a 128GB PM will need 32 seconds to wrap around the cache. In a recent article, Alibaba reported that it saw a peak order rate of 544,000/s [109]. While we understand that orders do not translate to actual I/O and, as Vuppalapati et al. report for the Snowflake infrastructure running on the Google Cloud Platform and Microsoft Azure [110], I/O requests come at more moderate rates, let us take the Alibaba numbers and say that an $S_b = 4$KB request comes in every 2μs, that is, $t_i = 2\mu$s. Then, to wrap around the 128GB of PM, it takes 64 seconds.

In real life, of course, not all writes are sequential and other factors come into play. Also, keeping a cache in the GB range was not considered the norm though this is changing. One example is the Memory Mode use of PM where all of DRAM (which is in the many GB range) is used as a cache of PM [29]. However, as we will see later, sequential allocation of files is an important strategy in FR. Furthermore, as FR is based on the premise that PM is in the size range used by PM based file systems such as NOVA and DAX, for example, at least 128GB, this sufficiently large cache size allows the accommodation of a

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY



(a) Case 1 (b) Case 2-1 (c) Case 2-2

(d) Case 3-1 (e) Case 3-2 (f) Case 3-3

Figure 15: The figure shows the steps taken based on the initial condition of a slot when a write is requested. The C(V,N) values on top and below the slot, that is, the two chunks are the status flags associated with each chunk. The changes to these status flags, denoted by the bold characters, are done atomically.

static indexing scheme, which we now describe.

Figure 14 shows, at a high level, how FR manages its space. FR uses a simple indexing structure based on a key associated with each file. (The details of what comprises chunks and slots as well as the how the indexing structures are maintained are given in Section 5.2.) Specifically, the index is simply calculated using $key_{c+1} = key_c + stride$ where key_c is the current key (initially 0) and $stride$ is a value dynamically determined for the workload. (Details of terms and their use are discussed in Section 5.2). In short, every file has its destined location within FR with no PM allocator involved, resulting in very little overhead for placement. Similarly, there is no need for replacement victim selection.

Handling Consistency: FR is a management module for data that resides in PM. That is, when a user requests a write, FR is the module that provides the response. The writes are made to PM, and they must be made durable such that they are recoverable upon system failure. For example, we cannot have torn writes; upon recovery from failure, its state should be either before the write or after the write. Also, as FR is the first responder to the underlying file system, eventually all data (and metadata) writes must reach the underlying storage, be it SSD or HDD. For example, consider a case where an overwrite of a block A, which resides in traditional storage happens. Upon a write, the user could receive an acknowledgement immediately after the data is placed in PM. However, this data and block A in storage are not in sync. Ensuring immediate in-order file semantics so that what the user perceives as valid data is indeed the (eventual) actual data, even in case of failures, is not a trivial matter. This may incur considerable performance overhead or require a completely new design of a PM-aware data structure, which is not an easy task as exemplified by previous studies [72–74, 111].

For this consistency issue, we develop a protocol under the PM 8-byte failure atomicity assumption. Like previous methods, the basic idea is to keep a dual copy of the data [112, 113], but the devil is in the details. Thus, a detailed and lengthy discussion is required, which we continue in the next section, along with a discussion on the failure recovery mechanism.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

5.2 Data Consistency and Recovery Protocol

In this study, we say data in a file is consistent if, upon a write request to that file by an application, the data in the file is as before the write request or reflects all of the contents of the issued write. Furthermore, guaranteeing data consistency means that write requests that have been acknowledged as having been written are eventually made durable and found through the underlying file system. In this section, we present the protocol that FR uses to guarantee data consistency.

5.2.1 The Basic Components

Figure 14 also shows the inner components of FR. There are a total of P continuous *chunks*, where a chunk is the load/store unit to storage, and P *tags*, with each tag being associated with one chunk. The chunks are organized in pairs, which we call a $slot$. As will be evident later, both chunks within a slot are used in full, making full use of the entire PM. The tag consists of the *key*, bits to represent the status of the corresponding chunks, the file size, the inode number, and the timestamp. The key is a unique number associated with the file slot assigned to it by FR upon file creation and stored in the inode of the file. This key is simply a global key maintained by FR, initialized to 0 and monotonically increased in $stride$ increments upon every file creation, where the stride is set to reduce collisions, where a $collision$ is defined as an allocation of two (or more) files to the same slot. We discuss how the stride is set in Section 5.2.3. Thus, the number of files that can be created is limited by the number of bits used to represent this value. In our current implementation, we use 56 bits and, for now, we make no effort to remedy this limitation as we think this is enough to represent the number of files for a system lifetime. There are other metadata in the tag, but they are discussed as the need arises.

5.2.2 Data Consistency Protocol

With PM, design for consistency must be done with the 8-byte failure-atomic write supported by the architecture. With traditional systems, providing durability with consistency becomes complex as the critical path to durability can be quite long. However, as we elaborate below, with FR, no allocation algorithm is invoked and only a few bytes are needed to guarantee data consistency of the file system.

Writes: With FR, it is vital that writes to chunks are done in an atomic manner, and thus, immediately durable. We now present the core design of FR that quickly enables durable in-order semantics of the data written to the file system.

For our discussion, we assume that the key and status flag in the tag are written with 8-byte failure atomicity guarantee supported by the hardware, where the status flag contains 2 flags V and N (and other information that we ignore for now). V distinguishes valid versus invalid and N with value 1 denotes the new, more recently written chunk in the slot. We denote the flag states of chunk C as $C(V,N)$, and further, denote the states of the pair of chunks in slot S as $S[C_1,C_2]$. We regard $S[C_1,C_2]$ and $S[C_2,C_1]$ the same as the algorithm works in the same way given the two states.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Algorithm 1 FR_write(key, buffer, size, offset)

```
index ← index of chosen chunk; // invalid chunk is first chosen; otherwise old chunk
t_index ← index of the other chunk in the slot;
FR_lock(slot);
if (tag[index].V) then // Case 3: my chunk is in use
    FR_flush(index); sfence; // flush data to backing storage
    tag[index].V ← 0; tag[index].N ← 0; clwb; sfence;
end if
if (tag[t_index].N) then // Case 2-1, 2-2: other chunk is new
    tag[t_index].N ← 0; clwb; sfence;
end if
if (tag[t_index].key == key && tag[t_index].V) then // Case 2-1: attempt to overwrite chunk
    memcpy(chunk[index], chunk[t_index], CHUNK_SIZE) ; // copy chunk from t_index to index
    clwb;
else // Case 1, 2-2: attempt to write to data in storage
    FR_load(key, index); // load chunk from storage
end if
memcpy(chunk[index] + offset, buffer, size); clwb; sfence; // write buffer to chunk[index]
set_tag(tag[index], key, 1, 1); clwb; sfence; // set tag(key, V, N) thru atomic update
if (tag[t_index].key == key) then // Case 2-1, Step 5: invalidating old chunk
    tag[t_index].V ← 0; clwb; sfence;
end if
FR_unlock(slot);
```

Figure 16: Algorithm `FR_write()`

The series of figures in Figure 15 show how an incoming write is managed. Upon a chunk write, the slot to place the chunk is determined by $(key \bmod \lfloor \frac{P}{2} \rfloor)$, which designates the first chunk of the file determined by the key attribute *key* assigned to each file upon creation, and its displacement from the start of the file. The write that is serviced by FR is denoted `Write`(X), and X is an object that is of any size, though in our discussion, for simplicity, we assume that X is smaller than or equal to the chunk size. (If X is larger, then it simply takes multiple chunks, which we discuss separately in Section 5.2.4.) The pair of chunks in the slot indexed can be in one of three states, denoted above and below the chunks in Figure 15:

- $S[C(0,0), C(0,0)]$, the slot is empty, that is, both chunks of the pair contain no data (empty/invalid)
- $S[C(1,1), C(0,0)]$, only one chunk has valid data
- $S[C(1,1), C(1,0)]$, both chunks have valid data.

We now discuss each of the cases in detail, and the discussion is presented in algorithmic form in Figure 16.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Case 1: This is when both chunks of the slot are invalid, that is, $S[C(0,0),C(0,0)]$. We can choose any chunk of the pair to write to. Upon `Write`(K')), if a chunk of the unmodified K exists in storage, it is first read in to the chunk.[1] Then, K' is written to the chunk and persisted with the `clwb` and `sfence` operations. Then, the tag $C(0,0)$ is set to $C(1,1)$, and only after this setting is the request acknowledged.

Note that `clwb` and `sfence` operations must be performed by default for writes to FR for the write to persist. Hence, we assume that they follow by default unless otherwise mentioned and will be omitted in discussions hereafter. We reiterate that, here and throughout our discussion, these flags are part of the tag and changes to them are done atomically. The state of the chunk pair in FR before and after `Write`(K') is shown in Figure 15(a).

Case 2: Here one chunk holds a valid object K while the other is invalid, that is, $S[C(1,1),C(0,0)]$. There are two cases to consider here; first is when K is being modified and the other is when a new object is being written, which can be distinguished by the uniqueness of the key. When K is updated, denoted by `Write`(K'), $C(1,1)$ is first modified to $C(1,0)$, stating that K is still valid but is no longer the new chunk. Then, K is copied to the chunk with state $C(0,0)$ and the update is reflected in this chunk. Then, $C(0,0)$ is modified to $C(1,1)$, and finally, $C(1,0)$ is changed to $C(0,0)$ to reflect that this chunk is now invalid. Thereafter, `Write`(K') is acknowledged. This case is depicted as Case 2-1 in Figure 15(b).

Note that K' cannot be written to the first chunk directly, but must be overwritten on a copy of the chunk containing K. This is because the write may be an overwrite and failure during this write may result in a torn write. Also note that this case covers consecutive writes to the same chunk of the same file, and that in such cases all writes are being handled within the PM without any interaction with the storage device.

The second case within Case 2 is when a new object L' is written to the same slot as the chunk containing object K as in Figure 15(c). This will happen only when a different file is allocated to the same slot, that is, upon collision. Starting from the same initial state, $C(1,1)$ is first modified to $C(1,0)$ as this chunk will no longer be the new chunk in this slot. If L' is an overwrite of a chunk in storage, then the chunk containing L must first be read into the chunk with state $C(0,0)$, unto which object L' is written. Finally, $C(0,0)$ is modified to $C(1,1)$ as object L' is now valid and new.

Case 3: The final case is when both chunks in the slot are valid, that is, $S[C(1,1),C(1,0)]$, and a different valid chunk needs to be put into this slot. The initial states of such cases are shown in Figures 15(d)$\sim$(f). Let us assume the initial situation in these figures, where chunks containing K and L occupy the slot, with L being the newer of the two chunks. Note that this situation can arise only when a collision has already occurred as K and L must be of different files. Let us also assume for now that both K and L are dirty. There are three cases to consider. The first two are updates of an existing object K or L. In these cases, the chunk containing the object that is not being updated is first flushed. This is to synchronize the dirty chunk with that in storage, which has the effect of making the chunk clean. Figures 15(d) and (e) depict these cases.

For Case 3-1 (and Case 3-2) where the older chunk K (the newer chunk L) is being overwritten,

[1]Note that if the write size is equal to the chunk size, then it need not be read from storage. Here, we are considering the worst case. This holds for subsequent discussions as well.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

the other chunk L (K) is flushed and awaits the acknowledgement that the flush has completed. Once it arrives, $C(1,1)$ ($C(1,0)$) is changed to $C(0,0)$, which results in a state identical to the initial state of Case 2-1. Thus, from here, the steps are the same as Case 2-1. Note that all flushes in FR are done by writing in Direct I/O mode (to be specific, with the `write_iter()` call).

Finally, the third case is when a new object is written to FR. This would be another collision with a new file on this slot. Case 3-3 in Figure 15(f) depicts this situation with `Write`(M'). Similarly to the previous two cases, we must flush a chunk, and for this, we choose to flush the older chunk. Hence, in our example, the chunk containing K is flushed. Once the acknowledgement of the flush is received, $C(1,0)$ is changed to $C(0,0)$. This results in the same initial state as Case 2-2. Thus, we proceed in the same steps as Case 2-2.

Note that, as described, Case 3 always incurs a flush to the storage device, which can be a source of overhead as the lock to the slot must be held until the acknowledgement for the flush is returned. Such flushes occur when a collision occurs and the chunk to be flushed is valid and dirty. However, note that if that chunk is either invalid or clean, the chunk need not be flushed. Fortunately, FR minimizes such flushes by design; the first source, that is, collision, is remedied by dynamically and judiciously assigning the stride value, while the second source, that is, dirty chunks, is remedied by periodically flushing the chunks in the background to make chunks clean, which we discuss in more detail in the next section. But first, we discuss how reads are handled.

Reads: Reads are similar to writes. First, the key is used to find the relevant slot. Then, a read hit is trivial as they are simply serviced out from FR. Upon a miss, we need to bring in data from storage. This case is similar to writes in that the invalid chunk will be used or if there is no invalid chunk, the old valid and dirty chunk will first be flushed out to a clean chunk. Then, the data requested will be brought in overwriting the clean chunk. The process of flushing, atomically changing the tag values, and loading follow similar steps as writes.

5.2.3 Strides, Periodic Flushing, and Metadata

Stride Setting: In FR, as a file is created, it is assigned a unique key, whose value is used to designate a fixed slot location in FR. Ideally, the files will be assigned as shown in Figure 14 with no overlap to avoid collisions. However, this is difficult to achieve as we need prior knowledge of the files that are created, even for those that may be appended to later on. If estimated to be too small, then collisions will occur, possibly incurring forced flushes. If estimated too large (for example, f_0 in Figure 14), then internal fragmentation may occur leading to quicker wrap-around, for example, f_i in Figure 14. A wrap-around forces collision that could lead to forced flushes, leading to degraded performance.

We take an empirical approach in setting the stride. The basis of our approach is that file sizes are, in general, relatively homogeneous depending on the workload. For example, files in scientific workloads are small ranging in the few tens to hundreds of kilobytes [114–116], while files in database workloads are quite large being in the tens of MB to tens of GB range [117,118]. Taking this observation, we create a Stride Table with <file size, file count> pairs where the 'file size' is a set of fixed numbers, and 'file

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

count' is the number of files that are smaller but closest to 'file size'. When a new file is created, we select the next larger 'file size' of the entry with highest 'file count' value as the stride value. The next larger one is selected as a larger value will lead to less possibility of collisions. As actual writes occur, the Stride Table is updated, possibly, decrementing one entry while incrementing another. Maintaining this table take only minimal memory, which in our implementation is 32KB.

Periodic Flush: While stride setting is one aspect of avoiding collision, periodic flushing is an active measure to avoid forced flushing due to collisions. Recall that forced flushing could have a negative effect on performance. To minimize this effect, we choose to periodically flush the dirty valid chunks to make them clean. Note that periodic flushing has no bearing on the consistency of data as FR is simultaneously storage and a cache. Thus, the period, p, can be set to 0 or to any value lower than the time to wrap around.

In selecting p, we take hints from Equation 1. For a sufficiently large C_s value such as 128GB or more, the t_f value may be sufficiently small to satisfy Equation 1 even under severely high request rates. Thus, a reasonable value that will not overload the underlying file system may be chosen. For our study, we choose to use $p = 10$ milliseconds as the default as this is the value that is generally used for the page cache.

Metadata: The discussion so far primarily focused on data and maintaining its consistency. As data is flushed to the underlying file system in periodic intervals, the metadata that is also maintained in the file system will only be synchronized within interval bounds. To remedy this, FR can take two measures. The first is regarding the time-related metadata such as `atime` or `mtime`. For this, FR keeps the actual timestamp for these metadata in the FR tag and flushes them as flushes occur, whether periodically or by forced flushes. The second measure is to modify metadata reading calls such as `stat()` to first flush the relevant chunks, and then, read the metadata. In FR, we have implemented the first measure, but only the `stat()` call for the second measure.

5.2.4 Multi-chunk Writes

While we have described writes for a single chunk, in reality writes may be composed of multiple chunks. We refer to such writes as multi-chunk writes, which we now describe.

Fundamentally, multi-chunk writes are the same as single chunk writes. The key difference is that we provide a means to act upon the multi-chunks in an atomic manner. As accesses to chunks are controlled through individual locks, we acquire locks for each of the chunks in sequence from the first chunk to the last. Writes to chunks happen along with the acquiring of the locks. On writing the first chunk, the *not-complete* flag (another bit in the status flag byte in the tag) of this chunk is set to designate that the multi-chunk write has not yet completed. Then, the subsequent chunks are written to, along with the timestamp tag, where the timestamp of the first chunk is written. This timestamp plays a vital role in identifying all the chunks written from the same `write()` call when recovering from a failure. Once the last chunk lock is acquired and data written, the *complete* flag of this chunk is set to designate

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

that the multi-chunk write has completed, and the application is notified of this. Thereafter, background flushing of the chunks begin. When all the acknowledgements from the underlying file system return, the complete bit is reset and the locks for all chunks are released.

5.2.5 Failure Recovery

In this section, we show that the contents of FR are always recoverable from faults, be it hardware or software faults. The only failure that we cannot recover from is PM device failure where PM content is irrecoverable due to media failures or partial or full data corruption. We now show how FR returns to a consistent state upon a failure.

Recovery from Single Chunk Write Failure: The key thing to note for recovery from single chunk write failure is that if a fault occurs at any step in Figures 15(a)∼(f), the recovered state will either be one of the initial states of the figures (in terms of the tags, specifically, C(V,N) values of the chunks) or have already completed the intended write. For example, if a failure occurs during Step 1 of Case 3-2, that is, while waiting for the synchronous flush of chunk K to complete, when FR recovers, it will simply return to the initial state $S[C(1,0),C(1,1)]$ of Case 3-2. In this manner, we can consider all cases of failure and show that recovery from faults leads to one of the initial states or a completed state, both of which are consistent states.

Note that upon failure, for recovery, there is no extra step to execute for single chunks. Simply coming back on and continuing to access the file system suffices to retain file system consistency as every step in all cases eventually lead to a stable state and hence, a consistent file system.

Recovery from Multi-Chunk Write Failure: For recovery of multi-chunk writes, recall that the not-complete bit of the first chunk of the multi-chunk write is set when initiating the multi-chunk write and then reset after all the chunks have been written. The recovery mechanism works by first searching for chunks with the not-complete bit set upon restart from failure. When one is found, this means there is a multi-chunk write that was unable to complete. Thus, all chunks of the multi-chunk that had been written must be reinstated to its pre-write state. This is done by first checking, starting from the next slot the timestamp of each chunk in the slot, until we meet a slot where the timestamp of both chunks are different from the first chunk. This identifies all the chunks that had been changed during the incomplete multi-chunk write. Then, starting from the last chunk with the same timestamp, we work backwards as follows. As the chunks with the same timestamp should all be invalid, we need to reset the valid bit, making the chunk invalid. However, one can run into a situation as in Step 5 of Figure 15(b) where the previous valid copy, which sits in the counterpart chunk, has been invalidated by making the copy, which can be checked by looking for chunks with both valid and new bits that are reset. Thus, we need to make that copy valid, by simply setting the valid bit, before invalidating the current chunk. We do this up to the chunk that has the not-complete set, and as a final step, reset the not-complete bit. Note that recovery itself is also fault tolerant, as we perform the recovery backwards.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 7: System configuration

	Description
CPU	Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz (16 cores)
DRAM	Samsung 32GB 2666MHz DDR4 RDIMM×4 (128GB)
PM	Intel Optane DC Persistent Memory (512GB)
Storage	Samsung V-NAND SSD 860 EVO (1TB)
OS	Linux Ubuntu 18.04.3 LTS (64bit) kernel v4.18

5.3 Performance Evaluation

In this section, we evaluate the effects of FR on Ext4 performance, which we implement in Linux kernel version 4.18. A total of ∼3000 lines of code were added; ∼2900 for the FR module, ∼70 in the VFS layer, and the rest in the Ext4 file system. In the implementation, we modify five system calls, two of which are major and three are minor changes. In particular, minor changes are made to `creat()`, to assign the key, `fsync()`, to bypass the page cache, and `unlink()`, to clear the tag values once the chunks of the file to be deleted are found using the key. Major changes are made to `read()`, to make use of the key and Direct I/O in reading, and `write()`, to implement the consistency protocol, described in Section 5.2, with Direct I/O. One limitation of our current implementation of FR is that it does not support `mmap()`. This is because `mmap()` is supported upon the page cache, but FR bypasses it. Thus, the semantics of a file simultaneously opened and accessed via `mmap()` and FR does not conform to POSIX. Resolving this issue is left for future work. The chunk size is set to 4KB and the 7-byte key, which contains the unique number assigned to the file upon file creation that is also stored in the inode, and the 1-byte status flag, which contains the V, N, the *dirty*, and not-complete bits, comprise the 8 bytes that are written in failure atomic manner.

5.3.1 Experiment Platform and Benchmarks

Table 7 lists the specifications of the basic experimental platform that we use. Our system is equipped with an Intel Xeon Gold 6242 CPU with 16 physical cores with 128GB DRAM and an underlying SATA SSD with 1TB capacity. The 512GB Intel Optane DC PMM is run on App Direct `fsdax` mode [29]. To format FR to the device and control the PM size used, we use `ioremap()`. Thus, all storage I/O occurs through this `ioremap`ped FR region. For FR, as default, we make use of 128GB of the 512GB as storage, and periodic flushing occurs every 10ms as mentioned previously (denoted FR-10m or simply FR). We make use of a synthetic workload derived with the FIO tool as well as two sets of benchmarks that we describe in detail later.

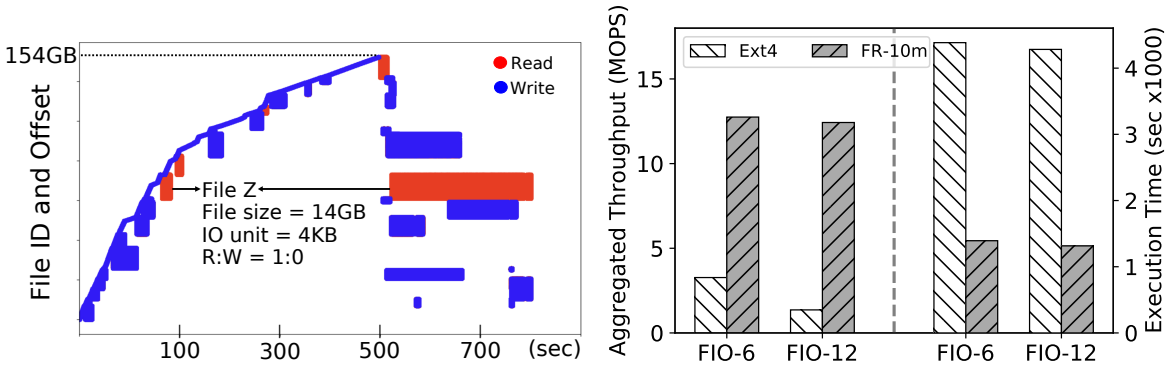
5.3.2 PM as a Cache

In a typical storage setting, the underlying file system manages a large storage device. Also, typically, this storage is not in full use, filled with data only to some capacity, which we refer to as the dataset.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 8: Characteristics of the 21 files used to generate the synthetic workload. Numbers in brackets are number of files in the particular mix. N: Normal, R: Random.

	Description
File size	1∼14GB [whole numbers only: 1 to 2 files of each]
Distribution	Pareto 0.1/0.5 [2/2], Zipf 0.2/1.2 [5/1], N [6], R [5]
IO unit	4KB [14], 8KB [7]
Read:Write	1:0 [4], 19:1 [9], 1:1 [8]
Fsync	0∼5% [whole numbers only: 2 to 5 files of each]
Intensity	Request interval: 1μs∼1ms (randomly distributed)



(a) Working set

(b) Performance results

Figure 17: (a) Changing of the working set with time for the synthetic workload, only showing the first 900 of a 1300 second run. (b) Performance comparison between FR and Ext4 for light and heavy synthetic workloads.

Of this dataset, a fraction of it would be accessed at any given time, which will be the working set. This working set will grow and shrink as time evolves as files being accessed will come and go, and access intensities to them will also vary with time. The role of an effective cache is to hold the working set in the cache.

To evaluate FR as a cache on a general workload, we make use of synthetic workloads generated, in the following manner, using FIO [119], which is an I/O testing tool. Using FIO, we generate 21 distinct files using the characteristics shown in Table 8, whose dataset size is the aggregate of all the file sizes, that is, 154GB. The files themselves all have distinct access characteristics that are randomly selected from the bounds shown in the table. For example, one file will be 2GB in size, the file accessed in 8KB units with Pareto 0.1 distribution, with a read:write ratio of 95:5 of which 2% of the writes are synchronized, with a request interval of 1 microsecond, which is the request intensity of the Alibaba workload [109]. Note that the request interval of 1 millisecond would be the intensity of the Snowflake workload [110], so overall, the intensity is quite high. 21 files, each with a unique set of characteristics are generated. Then, we control the working set by controlling the number of files that are accessed in parallel. Figure 17(a) shows how the working set changes, where the y-axis represents the files accessed

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

(by accessed offsets within the file), accumulated by the file access order from bottom to top, with time on the *x*-axis. The maximum number of files accessed in parallel at any point in time is set to 6, for light workload (FIO-6), and 12, for heavy workload (FIO-12), periodically starting new file accesses to meet this number. The working set of the light workload tops at roughly 50GB, while for the heavy workload, it is roughly 100GB. As all the 21 files finish off its accesses, the same 21 files are once again accessed repeatedly, but in random order starting another round of accesses. The second round of activities are shown to start just before the 500 second mark in the *x*-axis in Figure 17(a), and we observe that the access order is now changing. We also observe that the same files are being accessed much longer in the second round. For example, for the file marked File Z, which is a read-only file, the bar length, which represents time, is much longer on the second round than the first. (The height of the bar represents the file's footprint.) This is because, it turns out, the number of files being accessed together is larger than at the first round, taking it longer to execute. While to the naked eye, the coloring of the bars look the same, when observed more closely, the longer right bar is considerably more sparse than that on the left meaning that it is taking longer to execute the same set of requests. To fully stress the system, we repeat this rounding two more times for a total of four rounds of accesses, randomly selecting the files to execute as time progresses. The total I/O for these experiments was around 575GB, and it is similar for both workloads as they are essentially making requests with the same characteristics. For FR the number of wrap-arounds took place about 19 times. To access individual files in parallel, each sequence of accesses to a file is generated by a separate thread run on a separate physical core. As the number of files accessed for these experiments is small, instead of the stride size to be that based on the most often seen file size, the size of the largest file seen is used. This has the negative effect of increasing the number of wrap-arounds, thereby increasing collisions.

Figure 17(b) compares the results of Ext4 and FR, the left bars showing the aggregated throughput of all the file accesses and the right bars showing how long the experiments took to finish. Note that we use the exact same file access sequence for the two cases, by monitoring the file access sequence when first executing with FR and then using that to execute Ext4. The results show that FR provides more than 9× higher aggregate throughput and ended over 3× faster than Ext4, despite the fact that FR is providing immediately durable in-order semantics. Note that write synchronization is light, with only relatively low write requests of which the sync rate is a maximum of 5%. We find that slight increase to sync rates to be bounded to 10% (which we believe is still light) diminishes Ext4 performance by roughly another 10-fold. Finally, we emphasize again that NOVA and DAX could not execute because the dataset was larger than the PM size.

PM Management Overhead: Another important aspect regarding the cache is its management. One could argue that instead of the rather radical static placement approach that FR uses, a more intelligent scheme based on sophisticated replacement algorithms could benefit FR. We briefly mentioned earlier that this would be too heavy when used with PM. Or, if it is going to use a static scheme, why not just use hashing? To check this, we implement multiple variations for managing PM such as hashing with LRU replacement, a file-based Radix tree just like that used in the Linux page cache along with the LRU replacement policy as well as just hashing for static indexing without a replacement policy. For the

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 9: Average latency for managing FR for various indexing and management policies. The average latency for FR static indexing is 67ns while for hashed indexing it is 75ns. (* Insert includes mechanism to find empty blocks.)

Activity		Radix-tree + LRU	Hash + LRU
Radix-tree / Hash	Hashing	-	78ns
	Search	35ns	162ns
	Insert*	19μs	19μs
	Delete	190ns	43ns
LRU	Touch	161ns	153ns
	Add	73ns	65ns
	Remove	88ns	82ns

hashing function, we found several open sourced ones and out of those used djb2 [120], which showed the best performance. All other code, except for the LRU policy, which was an in-house implementation, were taken from Linux. Table 9 shows the average overhead of the various components for managing the FR layer with the various schemes, when run with the Fileserver benchmark (whose characteristics are presented in the next section). We observe that the overhead can be orders of magnitude higher than our approach.

We conducted experiments using these data structures with the Fileserver workload and found that they were around 13$\times$ slower compared to FR. (The benchmarks and the execution platform are described in the next section.) When simple hashing alone is used for placement, the overhead itself is low at 75ns, but we find that too many FR collisions (not hash collisions) occur incurring too much overhead for it to be a feasible approach.

5.3.3 Smaller than Cache Size Workloads

To compare FR performance with state-of-the-art PM file systems such as NOVA or DAX, we need to reduce the dataset size. In this section, we discuss the results for such a setting. Along with baseline Ext4, we also compare with Ext4 with PM DM-WriteCache (denoted DM-WC). However, it must be noted that DM-WC uses double the resources of the other schemes as it makes use of the entire DRAM as the page cache as well as 128GB of PM for the DM cache. Finally, we also attempted to compare with Assise [62]. Unfortunately, it would only run stably with a single thread and only for the workloads that the authors provided, that is, Fileserver and Varmail. As of this writing, other workloads did not execute properly even for single threaded execution. Multi-threading also was not supported. Thus, we do not include their results.

Benchmarks: As we can make use of traditional benchmarks in this setting, we make use of two benchmarks for our experiments whose characteristics are summarized in Table 10. The first is the Filebench benchmark, but of these we use only the Fileserver (F), Varmail (V), and OLTP (O) benchmarks. While it may be considered inappropriate to select particular workloads out of a benchmark suite, as FR supports

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 10: Characteristics of benchmarks. Request distribution for YCSB-D (YD) is Latest, while all others are Zipfian.

Filebench	R:W	Mean file size	# of files	**Key-value store**	Records Aggregated	R:W
Fileserver	1:2	128KB	200K	YA, YF	4GB	1:1
Varmail	1:1	32KB	800K	YB, YD, YE	4GB	19:1
OLTP	1:1	1.5GB	20	YC	4GB	1:0

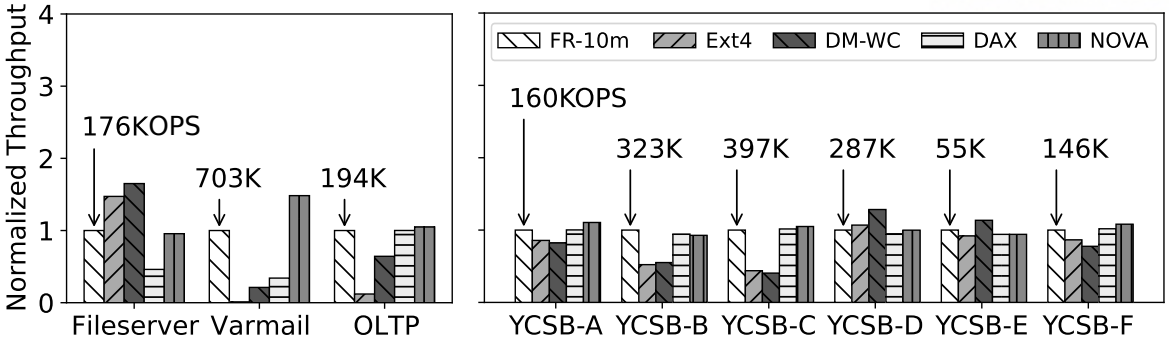
in-order file system semantics these three benchmarks represent the extreme ends of the spectrum of workloads. Varmail and OLTP are workloads that have a considerable number of `fsync()` calls, while Fileserver is one that does not have any such calls.

The second benchmark is YCSB [89], which is generally used to evaluate NoSQL database systems, with the underlying database server being RocksDB. YCSB-A (YA) and YCSB-F (YF) are write heavy workloads (1:1 reads/writes), YCSB-B (YB) and YCSB-D (YD) are read intensive, YCSB-C (YC) is read-only, and YCSB-E (YE) is a range query workload. We follow the execution sequence recommended by the original YCSB authors [121] including the loading phases, but do not report the results for the load phases. All experiments are performed in async mode, thus data loss may occur for Ext4 and DM-WC, while for DAX, NOVA, and FR, which provide in-order semantics, no loss will occur.

For all the workloads, we employ 16 threads as our system has 16 physical cores, except for OLTP, which employs 10 write threads and 20 read threads. For the Filebench workloads, the file size and number of files are set so that the dataset amounts to roughly 32GB. During execution, files are constantly being created and deleted, but overall the dataset does not deviate much from 32GB. For the YCSB workloads, we set the `recordcount` to 4 million, the `fieldcount` to 10, and the `fieldsize` to 100 bytes for records aggregated to 4GB. With these setting, the number of files created is around 450 and the average file size is roughly 80MB resulting in the dataset being in the 32GB vicinity. Our observations while these workloads are running on these datasets show that the maximum use of the page cache (for Ext4) at any time is roughly 10GB. Note that this setting allows the native file system to perform at its peak as the entire workload will fit into the page cache throughout its execution. All results reported are the averages of at least five runs each. We note beforehand that in multiple occasions, we only present partial results for particular discussions, unfortunately, due to space.

Filebench Performance: Figure 18(a) shows the performance results for the Filebench workloads. All results hereafter, unless otherwise mentioned, are shown relative to FR with the absolute values of FR indicated. For Fileserver, the results show that FR performs worse than Ext4 and DM-WC, while compared to DAX and NOVA, it does better. The reason Ext4 and DM-WC does better is that they provide the best environment for Fileserver as nearly all reads and writes are occurring in DRAM. This is because Fileserver does not make any `fsync()` calls (thus lacking durable in-order file system semantics). In contrast, for FR all access are at PM. PM read is roughly 3 times slower [27] and one-third of the workload are reads, and with the added peculiarities of PM [28], Ext4 and DM-WC should do better. In

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY



(a) Filebench (b) YCSB

Figure 18: Filebench and YCSB throughput normalized to FR, with its absolute performance numbers shown.

contrast to Fileserver, we see that for the Varmail and OLTP workloads, which are `fsync()` heavy, performance of FR is, respectively, roughly 94× and 8× better than Ext4 and roughly 4.5× and 1.5× better than DM-WC. Compared to NOVA and DAX, FR performance is slightly lower than NOVA, while DAX suffers for Varmail.

YCSB Performance: Figure 18(b) shows the results for YCSB. We observe that FR, NOVA, and DAX show similar performance, while Ext4 and DM-WC do the worst for the YCSB-A, YCSB-B, and YCSB-C workloads, while for YCSB-D, YCSB-E, and YCSB-F they are comparable or do better. However, we emphasize once more that neither Ext4 nor DM-WC guarantee in-order semantics. Note, in particular, the results for YCSB-C. As this is a read-only workload, one would expect Ext4 to perform similar to or better than FR as the entire dataset should be accessed from the page cache. However, we observe the contrary. This is because we run the workload with RocksDB, which issues around 50 `fsync()`s to manage a small number of files and perform compaction (movement between levels). In contrast, in terms of collisions in FR, YCSB-C is ideal as none occurs. Nuances of the effect of media are also observed when we compare the YCSB-C results from Figure 18(b) and Figures 24(a) and 24(b) that uses faster NVMe SSD and slower HDD media. We observe that the results for FR are the same for all three media, while for Ext4, performance is proportional to media performance, specifically, 436KOPS, 174KOPS, 153KOPS for NVMe SSD, SATA SSD, and HDD, respectively.

As YCSB also reports tail latency numbers we report them in Figure 19. The results show that Ext4 does worst in many cases. NOVA and FR are comparable but overall, FR does better, especially at the 99.99 tail.

Sources of Forced Flushing: We now analyze the sources of the performance differences. As FR works with large PM and the default periodic flushing is 10ms, one should expect FR to be free of any forced flushing. This is not what we observe. To help understand, we make use of Table 11, which shows how the workloads execute.

The ‘Flush/Access’ column shows the total chunk accesses of the workloads of which ‘Flush’ are the forced flush counts, while the next column shows the total number of copies that occurred between chunks within slots. The next two columns show the number of files and average space allocated for the

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Table 11: Various characteristics of workload execution. (M: million, WA: wrap-arounds)

	Flush/ Access	Mem cpys	# of files	Average stride×4KB	Footprint (GB)	# of WA
F	1M/70M	35M	1M	614KB	572	9
V	0.6M/64M	52M	4M	410KB	1536	24
O	0/19M	22M	21	1.6GB	32	0.5
YA	100/25M	25M	450	268MB	112.5	2

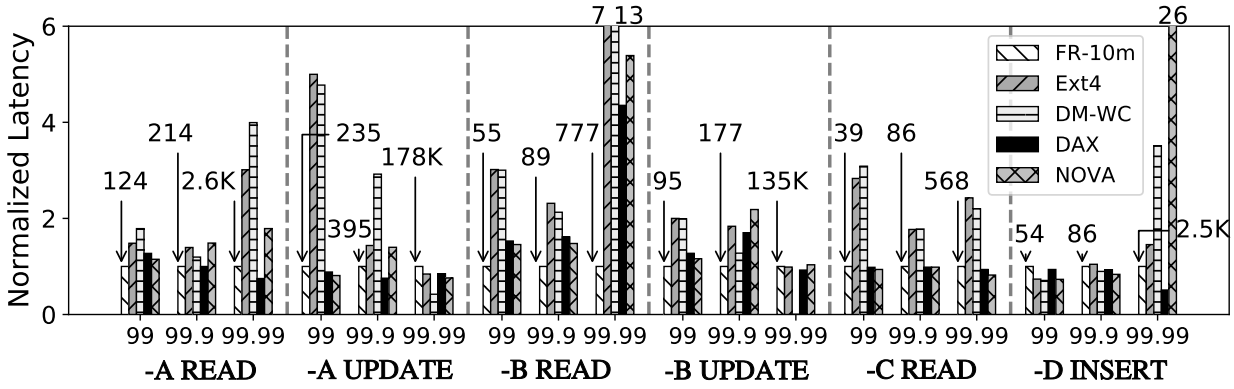


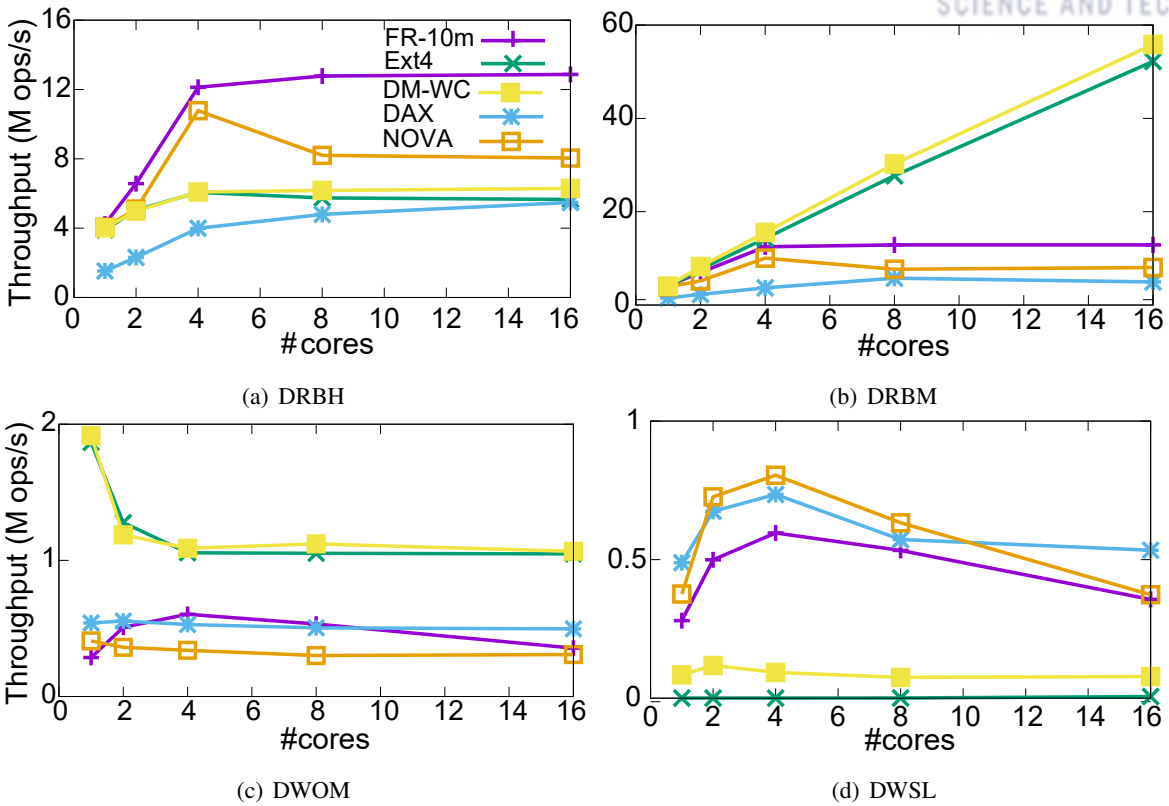
Figure 19: YCSB 99th, 99.9th, and 99.99th percentile tail latency as reported by YCSB normalized to FR. Numbers pointing to FR bars are absolute latency for FR in microseconds.

files as calculated by the stride multiplied by chunk size. These are different from those of Table 10, whose numbers are settings to run the workload, while here, they are those observed during execution, counting all files that were created. The final two columns show the footprint of the files within FR obtained by multiplying the values of the previous two columns and how many times the workload wrapped around, respectively, during the entire execution of the workload.

As observed from the 'Flush/Access' column, forced flushing occurs for around 1% of the accesses. There are two sources of forced flushes. The first is stride mis-estimation. While strides are set dynamically based on request size, it cannot foresee files that will grow. Hence, as files grow, overlap between neighboring files can occur, resulting in forced flushes. Detailed analysis (not shown) show that for the Filebench workloads, these situations are rare, but for YCSB-A, they account for all 100 flushes. The second source is wrap-around, as files are allocated in sequence in FR. We find that most of the forced flushes for Fileserver and Varmail are due to this. We observe from Table 11 that over a million files are being created, and the last two columns show that each slot is being shared by many files. Thus, collisions occur, possibly forcing flushes and resulting in performance loss.

Wear out: Wear out is not a serious issue when PM is used as storage. Compared to memory usage, the frequency of writes to storage should be minuscule. Also, these products provide hardware mechanisms for wear distribution to provide 5-year warranties [122]. We also calculate the potential lifetime of PM

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY



(a) DRBH (b) DRBM

(c) DWOM (d) DWSL

Figure 20: Fxmark test results. Note that the *y*-axis scales are different for all 4 graphs.

based on the endurance numbers available (endurance of 360 Peta Byte Writes [123]) and the intensity of the workloads in Table 11 (e.g., 52M chunk copies in 85 seconds of execution for Varmail). The results show that PM should survive over 4.5 years even under this unrealistically heavy workload.

Scalability: We now consider the scalability of FR. For this we make use of the `FXMARK` benchmark suite [124, 125]. `FXMARK`, originally developed to test the scalability of manycore systems, is divided into data and metadata tests. The results of running `FXMARK` are shown in Figure 20. (Note the different *y*-axis scales.) The letters used for the benchmarks represent the following: D/M: Data/Metadata, R/W: Read/Write, B/O/S/A:Block/Overwrite/Sync/Append, L/M/H: Low/Medium/High. We refer the readers to the study by Min et al. [124] for more details regarding the benchmarks.

For the eleven metadata microbenchmarks, as the measured activities are executed by the underlying file system, FR shows the same effect on metadata scalability as Ext4 and DM-WC. Let us now consider the data microbenchmarks. Two read results are shown in Figures 20(a) and (b), of which the last letters H and M distinguish the sharing intensity. The results show that FR does the best, relatively, for reads when sharing intensity is high (DRBH). We also see that FR does not scale well compared to Ext4 and DM-WC for DRBM. The reason FR does considerably better in case of high sharing intensity and worse for medium intensity is that FR performs global management of PM, while Ext4 manages the page cache in file units. Thus, in Ext4, when sharing is medium or low, high concurrency results in high performance while intense sharing results in low performance due to locking. FR, on the other, performs the same for all cases. Notice that the absolute numbers for FR are similar for both cases meaning FR performs similarly for all sharing intensities.

We now observe writes, whose results are shown in Figures 20(c) and (d). Here, O, and S denotes

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

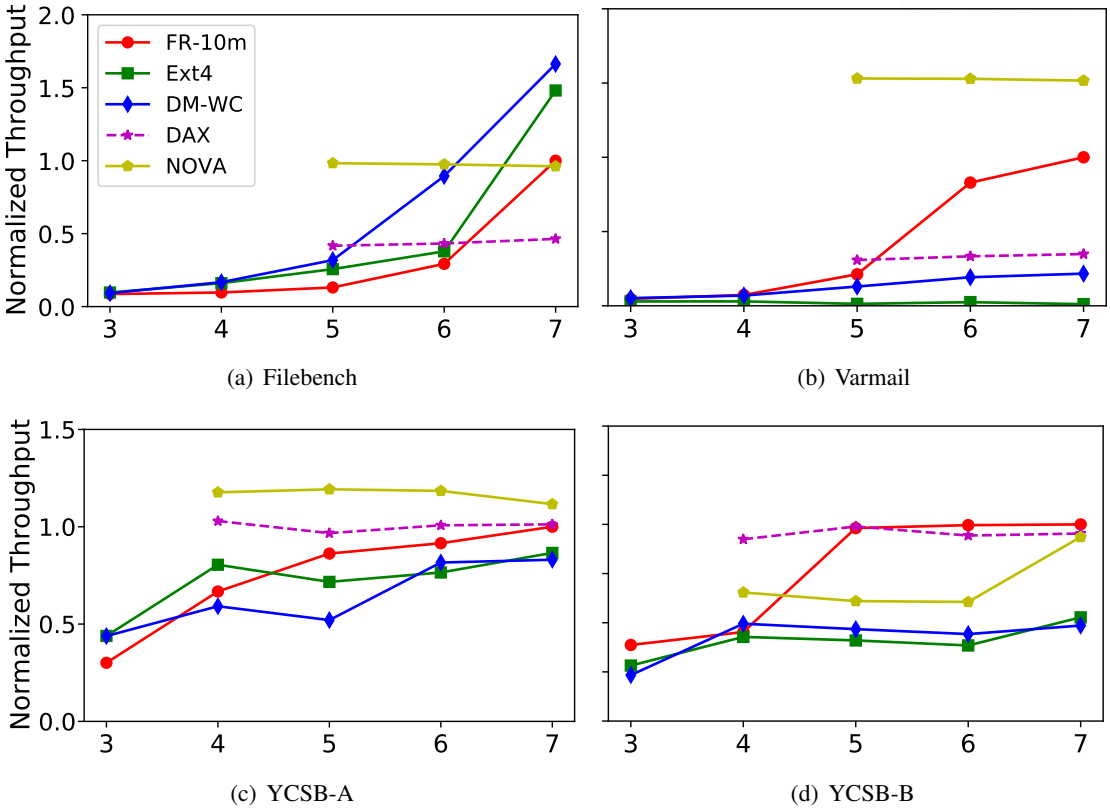


Figure 21: Performance results for PM size of 2^xGB, where x is value of points in x-axis, normalized to the performance of FR when $x = 7$.

overwrite and synchronous writes. Noting that the scales of each of the graphs are different, we find that FR shows similar performance for both types of writes. In contrast, we see that variations are relatively high for the other cases. In particular, we see that Ext4 and DM-WC do considerably well for DWOM as it is making use of the DRAM page cache, while they do worst for DWSL due to synchronization with the storage device.

Effect of PM Size: Figure 21 shows the performance of a select group of workloads (again, for clarity) when we vary the PM size, reducing it by half for each point to the left, for the various schemes. For DM-WC, both the DRAM and PM size is reduced by halves. All performance numbers are normalized to the 128GB FR results. There are two observations that can be made. First, the overall trend for FR, Ext4, and DM-WC are similar, though the drop in performance for FR is slightly greater. FR is performing similarly to the other two caching methods as the cache size gets tight. Second, for NOVA and DAX, there is barely any performance drop with the size drop. However, as PM drops beyond 32GB/16GB, they are not able to execute. This is the same situation as the synthetic workload experiments where the dataset exceeds the PM size at which point they are not able to execute. This, again, shows the limitations of PM based file systems.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

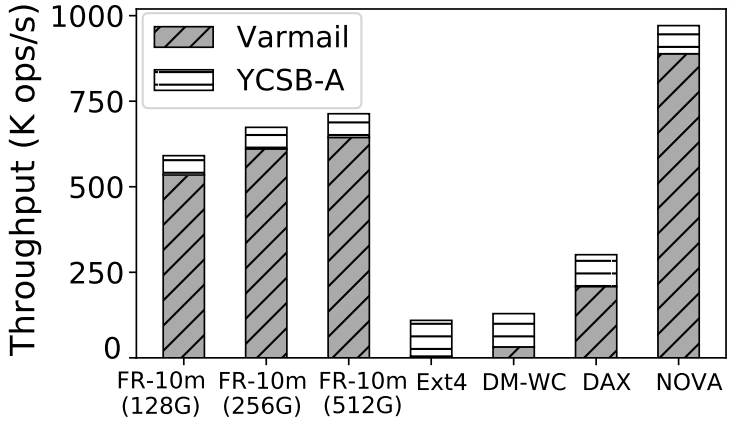


Figure 22: performance when heterogeneous workloads Varmail and YCBS-A are executed together.

5.3.4 Other Factors

Heterogeneous Workloads: The stride setting method that we use is based on the assumption that file sizes are relatively homogeneous depending on workload. However, with cloud services, for example, multiple virtual machines (VM), individually with unique characteristics, could be executing simultaneously. In such situations, each VM will need a unique stride value. We can resolve such situations by maintaining separate Stride Tables for each VM and assigning stride values based on each table.

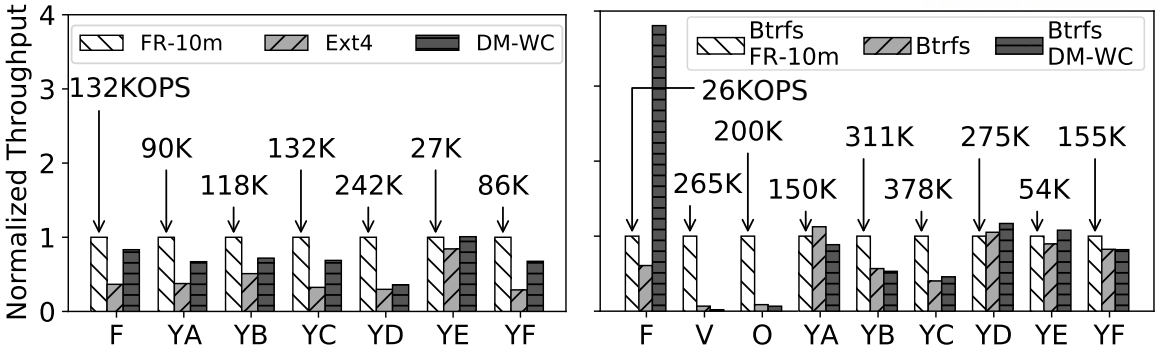
We experiment such a scenario by executing Varmail and YCBS-A together, each maintaining a Stride Table. The results of their execution is shown in Figure 22. We observe that, in the 128GB case, the absolute performance drops for both Varmail and YCBS-A compared to those shown in Figure 18. This is natural as two applications are sharing the same PM capacity. We also observe similar performance degradation with the other schemes. Closer analysis shows that for Varmail, the number of forced flushes remain stable, but see more occurring due to stride conflicts, while for YCBS-A, the number of forced flushes due to wrap-around increases.

Compensating for Extra PM: In our system configuration, PM is extra cost, but FR frees up DRAM used for the page cache. To evaluate this effect, we set up an environment where we have a memory intensive application, the FFT workload of Splash2x in the Parsec version 3.0 benchmark suite [126] set to use 12GB of memory, to run concurrently with the original workloads. Nine of these modules are invoked to take up 108GB of memory. The results are shown in Figure 23(a).

Overall, the results show that, while the absolute performance drops compared to those shown in Figure 18, the performance gap between FR over Ext4 and DM-WC grows. In particular, for Fileserver, we now see FR performing considerably better than both Ext4 and DM-WC. These results show that FR can effectively segregate I/O and memory intensive workloads relieving DRAM capacity for other use.

Btrfs: To show that FR is applicable to other file systems, we present results for FR with Btrfs [2], where changes are made to roughly $\sim$30 LOC. As shown in Figure 23(b), the results are similar to FR-Ext4, with many cases performing better when FR is deployed, despite the fact that Btrfs is not providing durable in-order semantics. Notable differences compared to Figure 18 are that 1) FR-Btrfs does better even for Fileserver compared to Btrfs though in terms of absolute throughput, Btrfs does considerably

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY



(a) FFT (b) Btrfs

Figure 23: (a) YCSB executed with FFT modules taking up 108GB of DRAM capacity and (b) FR applied to Btrfs.

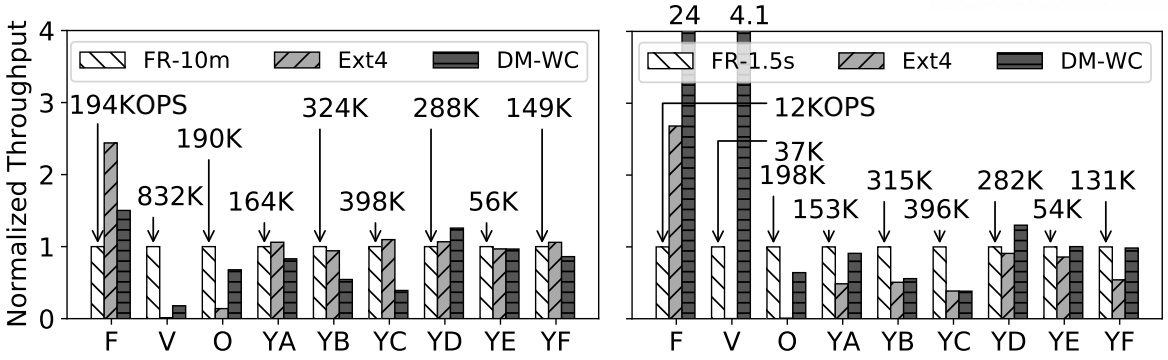
worse than Ext4, 2) performance improvements with DM-WC are large, and 3) for YCSB-A and YCSB-D, Btrfs does better than FR-Btrfs.

Effect of Storage Media: We consider how FR is affected when the underlying storage media is changed to a higher-end Samsung 1TB V-NAND 970 PRO NVMe SSD and a lower-end Seagate 1TB Barracuda SATA3/7200/64M HDD. Figure 24(a) shows that with the NVMe SSD, the absolute throughput of FR improves benefiting more the Filebench workloads. More interestingly, we see that the stock Ext4 benefits considerably especially for the YCSB workloads. This is due to the bandwidth of the media. That is, FR is physically organized using a single PM chip with a throughput of 0.58GB/s, while the NVMe SSD throughput is 2.7GB/s. In some cases, Ext4 is able to exploit this, resulting in performance surpassing FR as exemplified by some of the YCSB results. Note again, however, that FR is providing immediate durability, while Ext4 and DM-WC are not, and that DM-WC uses double the resources. Note also that FR still shows superior performance for applications where synchronization is prevalent as shown by the Varmail and OLTP results. Another implication of these results is that we may not need to move to these more expensive devices if PM is deployed appropriately. The YCSB results shown in Figures 18 and 24 show that they are all similar irrespective of the underlying storage media be it an SSD or an HDD. However, the Filebench results forbid us from generalizing so easily. The reasoning behind these discrepancies are left for future work.

For the experiments with HDD, the periodic flushes for FR is set to 1.5 seconds to accommodate the slow HDDs. If set to the default 10ms, we find that performance drops considerably because the HDD cannot sustain service for such heavy I/O. The results show that 1) even with the large period, for Fileserver and Varmail, the request rate is too high for the HDD to hide and thus, we see drops in throughput from 194KOPS to 12KOPS and 832KOPS to 37KOPS for Fileserver and Varmail, respectively, when the media is changed from NVMe SSD to HDD, and 2) for other workloads, we see performance that is similar to that of FR when using the SATA SSD (Figure 18).

Let us now focus on DM-WC that takes on the best of both worlds by using DRAM as a read cache and PM for the write cache (though, using double the resources). We observe in Figure 24(b), with

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY



(a) NVMe SSD (b) HDD

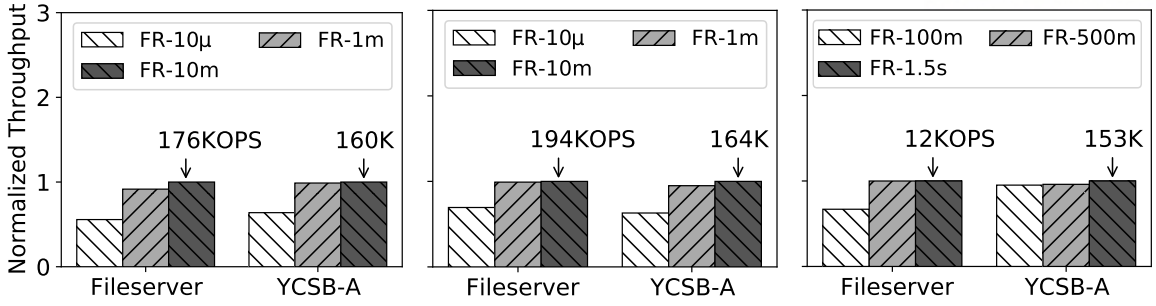
Figure 24: Performance with different storage devices.

HDD, how DM-WC is almost always superior over Ext4 and even sometimes performing better than FR. Interestingly, as we move to a faster device (SATA SSD in Figure 18), the improvements diminish, sometimes even resulting in performance worse than Ext4. Then, with the fastest device (NVMe SSD in Figure 24(a)), we see the effect of DM-WC diminishing further, with the majority performing worse than Ext4. There are two reasons for this degradation. One is device performance. DM-WC was devised to take advantage of superior device performance (PM, in this case), which is evident with the HDD results. However, for SATA SSD (0.52GB/s bandwidth), its performance is similar to that of PM (0.58GB/s). Thus, using PM should bring only a small improvement if any, meaning that, for example, for YCSB-A and YCSB-F, the write intensive ones in Figure 18, we should see comparable or slightly better performance for DM-WC compared to Ext4. However, we observe that DM-WC performs worse than Ext4. This is because of the second reason for performance degradation, that is, management overhead. DM-WC goes through an elaborate sequence of calls within (omitting the details, it goes through eight or so function calls). This incurs considerable overhead of around 35 microseconds of which the majority is overhead for indexing. This is pure overhead and is compounded as more writes are issued. Thus, write intensive workloads YCSB-A and YCSB-F are taking the blow. These results are evidence of the need for light management mechanisms such as that we propose. Finally, with the NVMe SSD, Ext4 performs better because the SSD has higher bandwidth (2.7GB/s vs 0.58GB/s). However, the added capacity does help DM-WC for some workloads.

Effect of Period: From now on, we perform an analysis on the performance of FR according to the period of the Periodic Flush technique. There are two points we want to show through this analysis. The first point is that setting the period larger than the default period has no effect on the performance. The second part we want to show is how the performance trend is formed when a period lower than the default period is set. If the period value is set relatively low, performance deteriorates due to the phenomenon of overlapping flushes. The experiment for analysis was conducted in three environments using SATA SSD, NVMe SSD and HDD as storage device, and Fileserver and YCSB-A workloads were used for this experiment.

Figure 25 shows the performance trend of FR over period. In conclusion, we were able to obtain

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY



(a) SATA (b) NVMe (c) HDD

Figure 25: Performance when storage is (a) SATA SSD, (b) NVMe and (c) HDD.

a result that setting the period larger than the default period in all environments did not affect the performance, so the corresponding results are not shown in the graph. In other words, the performance corresponding to the period value set as the default is the highest. As can be seen from the results, the performance trends of FR according to the period in environments where SATA SSD or NVMe SSD are used as storage device are similar. In both cases, when the period is in microseconds, the performance drops by up to 60%. We can see that the performance when the period is 1ms is almost similar to the performance when the period is 10ms. The graph of the HDD shows different results. As the HDD is very slow, we make use of much larger absolute period values. At 1.5 second, FR has the highest performance. In Fileserver, the performance when the period is 100ms decreased by about 30% compared to the performance when the period is 1.5s. When the period is 500ms, the performance increases significantly up to the performance when the period is 1.5s. There is little change in the trend of the graph in YCSB-A.

In summary, selecting a period that is more than the default period does not affect performance, but choosing a shorter period may affect performance. We chose the period of the point with the same performance as the baseline as the default. As the period does not influence the consistency setting, we can set any value to default.

Recovery: We now show experimental validation of recovery. Since the recovery process for single chunk writes is not necessary, we show only the recovery process for multi-chunk writes. Unfortunately, as the benchmarks that we used do not provide a means to restart from point of failure, we devise our own application that reads and writes files concurrently for failure recovery testing. Within this application, we write code to monitor the state of PM, whose output is as shown in Figure 26. While running this application, we use the Supermicro IPMI (Intelligent Platform Management Interface) utilities [127] to inject failures forcing sudden shutdown.

Figure 26 shows a partial view of the states monitored at right after reboot from failure, but before recovery (top figure) and then, right after recovery is performed (bottom figure), where the shaded parts show the changes that occurred. To elaborate, consider the leftmost, top shaded area ((a) of the top figure) with index 16, 17, and 18, which represent the chunk ID, all with TS (timestamp) value 8. (The chunk pairs (0,16), (1,17), (2,18) and so on form a slot sequence.) The two bits below the timestamp represent the state of the not-complete (left) and valid (right) bits. The bits of chunk 16 show that a

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

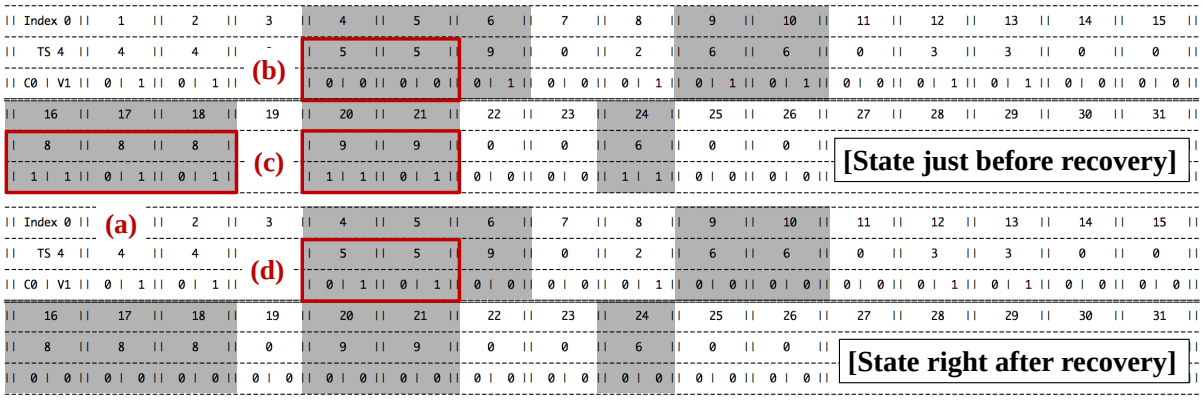


Figure 26: The top depiction is state of FR tags just after reboot upon failure but before recovery, while bottom is state right after recovery.

multi-chunk write was in progress whose timestamp is 8, and chunks 17 and 18 were part of the multi-chunk and written with new data (denoted by valid bit 1). However, this multi-chunk write was not able to complete. Upon recovery (the corresponding bottom figure), we see that all the not-complete and valid bits of chunks 16, 17, and 18 are now clear, meaning that recovery was successful, and FR can continue, which FR did without hiccup.

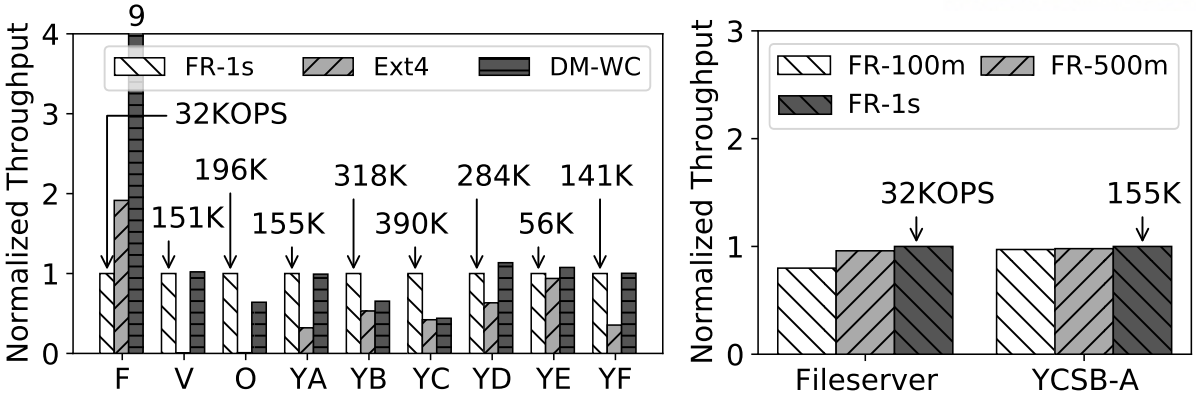
Consider now the other shaded area on the right (with chunks 4, 5, 6, 20, and 21, with timestamps values 5 and 9), which is another case of failure. This shows the recovery process for failures that occurred during an update process. Chunks 4 and 5 are the old values as shown by the timestamp value of 5 and 0's for the valid bits ((b) of top figure), and these chunks had been updated properly with the chunks 20 and 21 ((c) of top figure) as well as chunk 6. However, it was not able to complete. Now in this case, the old value needs to be restored upon recovery, as we described in Section 5.2.5. (d) of the bottom figure shows that this was done properly. We see that the valid bit of chunks 4 and 5 have been set back to 1. Another successful recovery is also seen on the shaded areas on the right.

5.4 FR on Distributed Storage Platform (Ceph)

In this section, we show performance evaluation with FR applied to distributed storage platform. We select Ceph [5], a distributed storage platform used in cloud systems, as the application. Ceph internally manages object storage, providing clients with block, object, and file-based abstractions. We select Ceph RBD to provide a block-based abstraction. Ceph storage cluster can consist of at least one administration server and one or more Object Storage Daemon (OSD)s. The administration server runs the Ceph monitor daemon (MON) and is is also used as a client server, which performs data I/O to a logically provided storage by underlying OSDs. The OSD is the core data node of the Ceph storage cluster that stores, replicates, restores, and rebalances object unit data. We apply FR to the administration server to increase the overall I/O performance of the Ceph storage cluster that provides network-connected OSDs.

The configuration of the experiment for performance evaluation is as follows. We configure one administration server and three storage servers for running the OSDs. All servers consist of the Intel

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY



(a) Overall performance

(b) Network-connected OSDs

Figure 27: (a) Overall performance of FR applied to Ceph platform and (b) sensitivity for period.

Optane DC PMM server used in the previous experiments (Table 7). The administration server and the storage server group are connected by a 10Gbps Ethernet. The storage servers are connected with a 100Gbps InfiniBand. The capacity of FR applied to the administration server is 128GB, the same as in the previous experiment. Each storage server is equipped with Samsung 1TB V-NAND 860 EVO SATA SSD. We conduct the measurement for the Ceph storage backends using SATA SSDs. The total capacity of logical OSDs is 3TB of SATA SSDs. All experiments are conducted on the Linux kernel version 5.8.1 with the latest Ceph version 17. As benchmarks, the programs in Table 10 are used.

Let us see the results. For reference, network-connected OSDs have slightly better performance than local server HDD and far less than local server SATA SSD. Figure 27(a) shows the performance results for the Filebench and YCSB workloads. As shown in this figure, the overall performance scales of comparison are similar to the results in Figure 24(b) when the backing storage is HDD. FR and Ext4 showed varying performance according to the performance of network-connected OSDs. Ext4 has poor performance in Varmail and OLTP workloads where `fsync()` is called frequently. In this figure, we show that DM-WC still has the same performance scale as the results in other media. However, the effect of DM-WC is only occurs when a very slow device is operated as backing storage. As the performance of the media used as the backing storage improves, we can shown that the effect of DM-WC disappears. This is because of management overhead, as mentioned earlier. The overhead of the DM-WC is not prominent in the environment where the penalty due to cache miss in FR and Ext4 is large. But, as the media becomes faster, the penalty decreases, so the overhead of DM-WC is relatively well prominent.

Figure 27(b) shows the performance trend of FR over period. As shown in the results, the performance trend of FR according to the period in environments where network-connected OSDs are used as backing storage are similar to performance trends in other media. As the network-connected OSDs are slow similar to HDD, we make use of a period value in seconds. At 1 second, FR has the highest performance. In Fileserver, the performance when the period is 100ms decreased by about 20% compared to the performance when the period is 1s. There is also little change in the trend of the graph in YCSB-A.

We apply FR to distributed storage platform Ceph. Through this experiment, we proved that FR

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

can provide high performance and high reliability by being applied to various storage platforms. We can guarantee the durability of data and metadata with the performance of the PM, and dramatically improve the performance degradation that can occur due to synchronization for system integrity. FR has high portability, so it can be easily applied to any platform. In addition to the platforms we are currently focusing on, we plan to apply FR to various storage platforms or similar environments using buffer caches to provide high performance and reliability.

5.5 Summary

In this dissertation, we presented First Responder (FR), a means to exploit the beneficial features of PM. While FR shares the goal of all PM based file systems in exploiting the performance benefits of PM as a storage device, its approach is unique in that it allows the use of existing modern file systems. Conceptually, FR is much like a buffer cache, but we showed that much more is involved as consistency had to be maintained under failure while providing light management. Built at the VFS layer, while the rest of the I/O stack, including the specific file system layer, remained largely unchanged, FR was able to provide immediate response to users from this layer. In experimental evaluations with the Intel DC PMM, we showed, with the FIO synthetic workload, FR, when used in cache form, can outperform Ext4 by more than 9×, while providing durable in-order file system semantics. Using the Filebench and YCSB benchmarks, we also showed that FR, when used as part of a typical file system, performs comparably with the default Ext4, Ext4 with DM-WriteCache, NOVA, and DAX, while also providing durable in-order semantics. In addition, we show performance evaluation about FR applied to Ceph, then, we prove that FR can provide high performance and high reliability by being applied to various storage platforms.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

VI Conclusion

In this dissertation, we study research on efficiently redesigning and improving the critical path of the storage system using Persistent Memory (PM). We analyze the empirical findings of traditional storage system's critical path. Based on the analysis, we first propose the Persistent Memory Adaptation Layer (PMAL), a module that can be added to the existing file system to control the I/O flow to PM media. In the experiment for performance evaluation, the performance of the as-is file system with PMAL integrated is in par with other PM file systems such as NOVA, PMFS and Ext4 with DAX extension, for the Filebench and key-value database benchmarks that we considered on the Linux platform. We also present a MeLo@V (Metadata Logging at the VFS layer) technology that can lightly guarantee file system consistency without multi-versioning technologies. Our experiments on various workloads also show that the performance of Ext4 with DAX extension, PMFS and NOVA can be improved upon by 2 to 9% by incorporating MeLo@V and slight modifications. We finally propose First Responder (FR), like a buffer cache, absorbs requests at the topmost layer of the I/O stack in PM, then immediately responds to the requests (hence, the name). With experimental evaluations with the Intel DC PMM, we show that FR, when used in cache form, can outperform Ext4 by more than 9×, while providing durable in-order file system semantics, whereas Ext4 cannot. We also show that when used as part of a typical file system, performance is comparable with NOVA and Ext4-DAX. Additionally, We show performance evaluation about FR applied to Ceph, a distributed storage platform.

To conclude this dissertation, we present various technologies for critical path that can guarantee the original performance of the PM media in the storage systems. Our proposed technologies effectively improved the performance of the conventional storage system by providing software redesigns and techniques optimized for the characteristics of the PM media. These technologies led to high effective utilization of PM, and through these studies, we were able to retain innovative management technology for a new PM hardware structure. Many research groups are rethinking the existing software stack in the face of changes in hardware parts, our studies can provide new perspectives to related research fields, and are expected to play an important role in advancing technology for storage systems.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

References

[1] Wikipedia, "Ext4," https://en.wikipedia.org/wiki/Ext4.

[2] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Transactions on Storage (TOS)*, 2013.

[3] RocksDB, "Rocksdb basic operations," https://github.com/facebook/rocksdb/wiki/Basic-Operations.

[4] LevelDB, "Leveldb document," https://github.com/google/leveldb/blob/master/doc/index.md.

[5] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. E. Long, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[7] E. Lee, H. Bahn, S. Yoo, and S. H. Noh, "Empirical Study of NVM Storage: An Operating System's Perspective and Implications," in *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2014.

[8] H. Song, Y. J. Moon, S. K. Lee, and S. H. Noh, "PMAL: Enabling Lightweight Adaptation of Legacy File Systems on Persistent Memory Systems," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[9] J. Yang, D. B. Minturn, and F. Hady, "When Poll is Better Than Interrupt," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[10] 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 European Conference on Computer Systems (EuroSys)*, 2014.

[11] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. D. Silva, S. Swanson, and A. Rudoff, "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[12] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, and K. Aasheesh, "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[13] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and Protection in the ZoFS User-space NVM File System," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[14] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A Cross Media File System," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[15] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[16] J. Huang, M. K. Qureshi, and K. Schwan, "An Evolutionary Study of Linux Memory Management for Fun and Profit," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[17] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[18] Phoronix, "XFS Will Get DAX Support In The Linux 4.2 Kernel," https://www.phoronix.com/scan.php?page=news_item&px=XFS-Linux-4.2-DAX-And-More.

[19] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel, "TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[20] T. Sankaranarayana Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, and A. C. Arpaci-Dusseau, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[21] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Application crash consistency and performance with CCFS," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[22] H. Song, S. Kim, J. H. Kim, E. J. Park, and S. H. Noh, "First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2021.

[23] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, T. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam, "Phase-Change Random Access Memory: A

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Scalable Technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[24] M. Jung, J. Shalf, and M. Kandemir, "Design of a Large-Scale Storage-Class RRAM System," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS)*, 2013.

[25] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[26] Intel, "Memory Optimized for Data-Centric Workloads," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[27] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," Computer Science Engineering, University of California, San Diego, Tech. Rep., 2019.

[28] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[29] Intel, "Intel Optane DC Persistent Memory Quick Start Guide," https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[30] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing Safe, User Space Access to Fast, Solid State Disks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[31] T. Hwang, J. Jung, and Y. Won, "HEAPO: Heap-Based Persistent Object Store," *ACM Transactions on Storage (TOS)*, vol. 11, no. 1, p. 3, 2014.

[32] J. Ou, J. Shu, and Y. Lu, "A High Performance File System for Non-Volatile Main Memory," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[33] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[34] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A Reliable and Highly-Available Non-Volatile Memory System," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[35] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable Persistent Memory File System with Kernel-Userspace Collaboration," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[36] Viking Technology, "ArxCis-NV: Non-volatile Memory Technology," http://www.vikingtechnology.com/arxcis-nv.

[37] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[38] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures," *IEEE Transactions on Computers (TC)*, vol. 63, no. 9, pp. 2187–2200, 2014.

[39] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[40] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[41] D. Narayanan and O. Hodson, "Whole-System Persistence," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[42] I. H. Doh, Y. J. Kim, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Towards Greener Data Centers with Storage Class Memory," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 1969–1980, 2013.

[43] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[44] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proceedings of the Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[45] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, "Onyx: A Protoype Phase Change Memory Storage Array," in *Proceedings of the 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011.

[46] AnandTech, "Intel's 140GB Optane 3D XPoint PCIe SSD Spotted at IDF," http://www.anandtech.com/show/10604/intels-140gb-optane-3d-xpoint-pcie-ssd-spotted-at-idf.

[47] J. Handy, "Understanding the Intel/Micron 3D XPoint Memory," in *Proceedings of the Storage Developer Conference*, 2015.

[48] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[49] E. Lee, H. Bahn, S. Yoo, and S. H. Noh, "Empirical Study of NVRAM Storage: An Operating System's Perspective and Implications," in *Proceedings of the IEEE 22nd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2014.

[50] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganty, "An Empirical Study of File Systems on NVM," in *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)*, 2015.

[51] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[52] ——, "A Unified Buffer Cache Architecture that Subsumes Journaling Functionality via Non-volatile Memory," *ACM Transactions on Storage (TOS)*, 2014.

[53] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible File-System Interfaces to Storage-Class Memory," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014.

[54] S. Baek, J. Choi, D. Lee, and S. H. Noh, "Energy Efficient and High Performance Software Architecture for Storage Class Memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, pp. 81–102, 2013.

[55] Intel, "Deprecating the PCOMMIT Instruction," https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[56] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2016.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[57] J. Corbet, “Supporting Filesystems in Persistent Memory,” https://lwn.net/Articles/610174/.

[58] SNIA, “pmem.io persistent memory programming,” http://http://pmem.io.

[59] M. Wilcox, “DAX: Page Cache Bypass for Filesystems on Memory Storage,” http://lwn.net/Articles/618064/.

[60] ——, “Support ext4 on NV-DIMMs,” https://lwn.net/Articles/588218/.

[61] J. Xu, J. Kim, A. Memaripour, and S. Swanson, “Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[62] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, “Assise: Performance and Availability via Client-local NVM in a Distributed File System,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[63] S. Jiang and X. Zhang, “LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance,” in *Proceedings of the ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, 2002.

[64] C. A. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, “Cache Modeling and Optimization using Miniature Simulations,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[65] T. Alexander and G. Kedem, “Distributed Prefetch-buffer/cache Design for High Performance Memory Systems,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 1996.

[66] H. Kim and S. Ahn, “BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage,” in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[67] Q. Wei, C. Wang, C. Chen, Y. Yang, J. Yang, and M. Xue, “Transactional NVM Cache with High Performance and Crash Consistency,” in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[68] Kernel Documentation, “Writecache target,” https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/writecache.html.

[69] E. Lee, H. Bahn, and S. H. Noh, “Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory,” in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[70] A. v. Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato, "Managing Non-Volatile Memory in Database Systems," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2018.

[71] Intel, "Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes," https://software.intel.com/en-us/articles/intel-sdm.

[72] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[73] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[74] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-Optimized Dynamic Hashing for Persistent Memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[75] J. Yang, Q. Wei, C. Chen, C. Wang, and K. Leong Yong, "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[76] S. Fujita, K. Nomura, H. Noguchi, S. Takeda, and K. Abe, "Novel Nonvolatile Memory Hierarchies to Realize "Normally-Off Mobile Processors"," in *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[77] M. S. Haque, A. Li, A. Kumar, and Q. Wei, "Accelerating Non-volatile/Hybrid Processor Cache Design Space Exploration for Application Specific Embedded Systems," in *Proceedings of the 20th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015.

[78] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs," in *Proceedings of the 49th Annual Design Automation Conference (DAC)*, 2012.

[79] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[80] 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 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[81] X. Wu and A. Reddy, "SCMFS: A File System for Storage Class Memory," in *Proceedings of the 14th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[82] Persistent Memory Wiki, "Persistent memory," https://nvdimm.wiki.kernel.org/.

[83] GitHub, "Porting pmfs to the latest linux kernel," https://github.com/Andiry/pmfs-3.19.

[84] ——, "Nova: Non-volatile memory accelerated log-structured file system," https://github.com/Andiry/nova.

[85] J. Corbet, "The High-Resolution Timer API," https://lwn.net/Articles/167897/.

[86] A. Rudoff, "In a world with persistent memory," in *6th Annual Non-Volatile Memories Workshop (NVMW)*, 2015.

[87] LWN.net, "Directly mapped persistent memory page cache," http://lwn.net/Articles/644120/.

[88] Filebench, http://filebench.sourceforge.net/wiki/index.php/Main_Page.

[89] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[90] ForestDB-Benchmark, https://github.com/couchbaselabs/ForestDB-Benchmark.

[91] LWN.net, "copy_file_range()," https://lwn.net/Articles/659523/.

[92] Anna Schumaker, "vfs: Add vfs_copy_file_range() support for pagecache copies," http://www.spinics.net/lists/linux-btrfs/msg47738.html.

[93] LWN.net, "VFS:In-kernel copy system call," https://lwn.net/Articles/663746/.

[94] J. H. Kim, Y. J. Moon, and S. H. Noh, "An Experimental Study on the Effect of Asymmetric Memory Latency of New Memory on Application Performance," in *Proceedings of the IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016.

[95] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "NVM Duet: Unified Working Memory and Persistent Store Architecture," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[96] S. Chen and Q. Jin, "Persistent B+-Trees in Non-Volatile Main Memory." in *Proceedings of the 41st VLDB Endowment (PVLDB)*, 2015.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[97] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[98] D. P. Bovet and M. Cesati, *Understanding The Linux Kernel*. O'Reilly Media, Inc., 2005.

[99] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Pearson Education, Inc., 2015.

[100] Wikipedia, "Solaris (operating system)," https://en.wikipedia.org/wiki/Solaris_(operating_system).

[101] ——, "Btrfs," https://btrfs.wiki.kernel.org/index.php/Main_Page#Features.

[102] Linux Cross Reference, "Linux/fs/read_write.c," http://lxr.free-electrons.com/source/fs/read_write.c?v=4.7.

[103] archlinux, "Btrfs," https://wiki.archlinux.org/index.php/Btrfs.

[104] Strata, "Strata: A Cross Media File System," https://github.com/ut-osa/strata.

[105] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[106] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[107] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can Far Memory Improve Job Throughput?" in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2020.

[108] Btrfs, "Btrfs," https://github.com/torvalds/linux/tree/mast-er/fs/btrfs.

[109] TECHPP, "Alibaba Singles' Day 2019 had a Record Peak Order Rate of 544,000 per Second," https://techpp.com/2019/11/19/alibaba-singles-day-2019-record/.

[110] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building An Elastic Query Engine on Disaggregated Storage," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[111] P. Zuo, Y. Hua, and J. Wu, "Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

[112] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[113] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-Atomic Persistent Memory Updates via JUSTDO Logging," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[114] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A Five-Year Study of File-System Metadata," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[115] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[116] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[117] J. Sillero and J. Jiménez, "Editorial opinion: public dissemination of raw turbulence data," *Journal of Physics: Conference Series (JPCS)*, Apr. 2016.

[118] B. Memishi, R. Appuswamy, and M. Paradies, "Cold Storage Data Archives: More Than Just a Bunch of Tapes," in *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN)*, 2019.

[119] Freecode, "fio," http://freecode.com/projects/fio.

[120] York University, "Hash Functions," http://www.cse.yorku.ca/ oz/hash.html.

[121] YCSB, "Core Workloads," https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[122] Intel, "Warranty for Intel® Optane™ Persistent Memory," https://www.intel.com/content/www/us/en/support/artic-les/000033035/memory-and-storage/data-center-persistent-memory.html.

[123] StorageReview, "Intel Optane DC Persistent Memory Module (PMM)," https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm.

[124] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim, "Understanding Manycore Scalability of File Systems," in *Proceedings of the USENIX International Conference on Annual Technical Conference (ATC)*, 2016.

[125] FxMark, "FxMark: Filesystem Multicore Scalability Benchmark," https://github.com/sslab-gatech/fxmark.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

[126] PARSEC, “PARSEC,” https://parsec.cs.princeton.edu/index.htm.

[127] Supermicro, “Supermicro ipmi utilities,” https://www.supermicro.com/en/solutions/management-software/ipmi-utilities.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Acknowledgements

First and foremost, I would like to praise and thank God, the almighty, who has granted countless blessing, knowledge, and opportunity to the writer, so that I have been finally able to accomplish the dissertation. Apart from the efforts of me, the success of this dissertation depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this dissertation. The writer hardly knows where to start expressing the gratitude but for sure the gratitude goes to all those who have assisted him in the process of completing this dissertation. It would be impossible to list all names, but several people deserve him sincere and special thanks.

I would like first to thank my advisor, Professor Sam H. Noh who has encouraged me in all the time and guided me on the right direction. It was a great honor for me to receive an education from him. His mentorship and guidance have been essential in helping me become a researcher. I was influenced by him for personal as well as professional development. Once again, thank you for his dedicated education, and I sincerely admire him.

I also would like to thank my dissertation committees, Professor Young-ri Choi, Professor Woongki Baek, Professor Myeongjae Jeon, and Professor Beomseok Nam, for their help and valuable comments.

I am grateful to my fellow labmates, Hyeonho Song, Jay H. Park, Eunjae Lee, and Byungseok Kim for the numerous discussions, and for all the fun we have had in the last ten years. I also thank Young Je Moon, J. Hyun Kim, Se Kwon Lee, Shean Kim and my colleagues from the NECSST and CISSR groups for spending their lots of study time with me.

I am grateful to my close friends from the Monday Kids, GyuHyun Lee, Wookhee Kim, Deukyeon Hwang, and Wonbae Kim, who have made me bearable in graduate school by chatting, playing, or whatever, deserve all the credit for making me keep working. I also would like to thank my long-time mates from the Hongik University, ByungJoon Chun, Gunwoo Ki, Jang Hyun Park who have always supported my Ph.D. student course from afar. Also, I am grateful to my spiritual leaders and friends Millak church and UlsanJeil church who have always prayed for me.

I would like to thank my family, In Seog Song, Kyungok Cho, and Yeongsub Song. They always supported and encouraged me. They have made all of this happen. I respect and love both my father and my mother. I am grateful to my dear younger brother and friend, Yeongsub Song for your sincere support for me. I also respect and love him so much.

Finally, last but by no means least, I am truly grateful to my beloved wife and eternal soulmate, Yeree Lee, for her deep love and passionate support for me. I love and support you forever. I also share this joy with Harang Song, our blessing that God has given us. I would like to thank my family united by marriage, Seong Kyu Lee, Eun Mee Kim, Dong Jun Lee and Yu Mi Park, for their prayers and support.

May God bless us.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY