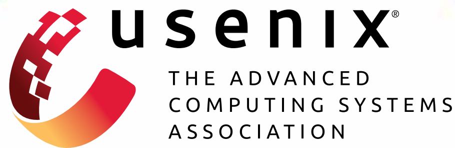


First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage

Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh, *UNIST*

https://www.usenix.org/conference/atc21/presentation/song

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

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

Hyunsub Song Shean Kim J. Hyun Kim Ethan JH Park Sam H. Noh
UNIST (Ulsan National Institute of Science and Technology)

Abstract

Persistent Memory (PM) is a new media with favorable characteristics that can vastly improve storage I/O performance. While new PM based file systems have been developed to exploit PM, most work have not been successful in fully integrating PM media with traditional storage media such as SSDs and HDDs. We present 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. FR brings about multiple benefits. First, we retain the maturity of existing file systems allowing deployment of FR at settings where traditional file systems are deployed. Second, traditional storage devices supported by these file systems can be used allowing easy integration of PM with traditional storage. Finally, FR allows in-order file system semantics at close to PM device latency. 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.

1 Introduction

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 [7, 8, 19, 23, 52, 58]. 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 [11, 30, 42], 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 [52] and DAX [36] 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 [23]. Separately, Ziggurat, proposed by Zheng et al., is a tiered file system that supports a combination of devices [58]. However, both proposals suffer from the first limitation; Strata, at the time of this writing, still is not fully functional [46] 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 [23] 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 [10, 37, 42]. With the advent of PM, such choice seems unnecessary as providing in-order file system semantics should be natural and easy [23, 42, 52]. 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) [23, 58]. Our work encompasses these goals and yet, instead of developing yet another file system, our design allows to retain the modern, mature, constantly evolving

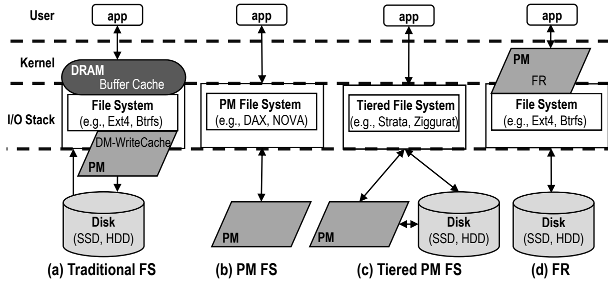


Figure 1: Comparison of PM use scenarios for file systems

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, as shown in Figure 1(d), 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.

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 [3, 43, 48]. 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 [5, 41] 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.

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*

In the remainder of the paper, we first discuss progress on work related to PM and distinguish how FR is different from them. Then, we present the design of FR in two separate sections; Section 3 discusses the overall design and in Section 4, 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, 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 6.

2 Related Work

Persistent Memory (PM) technologies represented by PCM (Phase Change Memory) [38] and STT-MRAM [22] are being considered as high performance storage mediums as they are nonvolatile and yet, provide random byte addressability and latency similar to DRAM. Intel recently commercialized the Optane DC PMM, the only product currently available in the market [15, 17, 53]. 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 [14]. While many recent studies have considered file systems for PM [7,8,19,23,52,58], our work is not about developing a new PM file system, but rather on integrating PM with existing modern file systems developed for traditional storage devices.

The closest set of related work are studies on the buffer (or page) cache. 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 [2, 18, 50]. 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 [8, 21, 52] as well as attempts to revisit this topic [20, 24, 51]. In particular, the DM-WriteCache [20] 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 are different by design.

There are also studies of the buffer cache that exploit the nonvolatile nature of PM [25, 40, 51]. 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 [25]. Tinca is a PM based buffer cache located in the external disk cache below the DRAM-based main memory [51] 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.

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 [13, 35] such as `clwb` and `sfence` as these instructions incur considerable overhead [17], while, at the same time, abiding to the 8-byte failure atomicity restriction to maintain consistency, various data structures for PM have been proposed [12, 27, 31, 55]. FR conforms with these efforts as it carefully designs the use of 8-byte writes with `clwb` and `sfence`.

Other related studies include PM based file systems. Recent studies have proposed many optimizations including user-level approaches to avoid crossing the kernel boundary in an effort to improve performance [7, 19]. While so, only NOVA and DAX are stably supported in Linux. They all, more or less, share the common layout shown in Figure 1(b). Out of these, as mentioned previously, Strata and Ziggurat, are unique in their support of traditional storage media in the form depicted in Figure 1(c). These diagrams contrast the differences between them and FR. Finally, a recent study presents Assise, a new distributed file system based on Strata, where PM is used as a client-local cache layer [4]. 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.

3 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.

3.1 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 [4, 33, 52]. 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 [26, 45, 54]. 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 5.2. 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

Figure 2: Internal organization and components of FR

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 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,

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

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 $S_b =$ 4KB, 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 [47]. 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 [49], I/O requests come at more moderate rates, let us take the Alibaba numbers and say that an $S_b = $ 4KB 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 [14]. 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 static indexing scheme, which we now describe.

Figure 2 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 4.) 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 4). 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 [12,27,31,59].

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 [16, 32], 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.

4 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

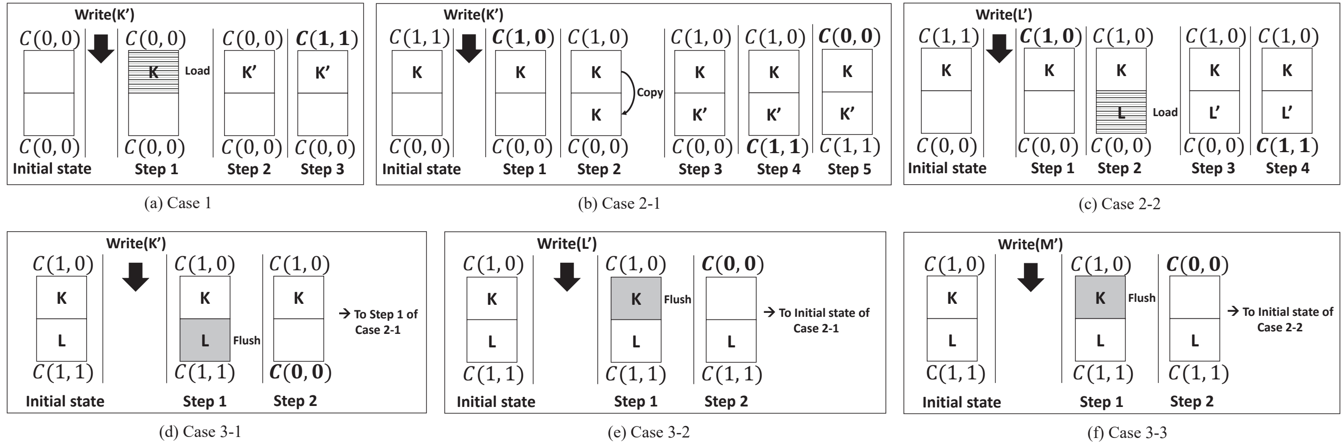


Figure 3: The figure shows the steps taken based on the initial condition of a slot when a write is requested. The C(V,N) values on top and below the slot, that is, the two chunks are the status flags associated with each chunk. The changes to these status flags, denoted by the bold characters, are done atomically.

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.

4.1 The Basic Components

Figure 2 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 4.3. 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.

4.2 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.

The series of figures in Figure 3 show how an incoming write is managed. Upon a chunk write, the slot to place the chunk is determined by (key mod $\lfloor \frac{P}{2} \rfloor$), 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 4.4.) The pair of chunks in the slot indexed can be in one of three states, denoted above and below the chunks in Figure 3:

- $S[C(0,0),C(0,0)]$, 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.

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 3(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 3(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 3(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 3(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 3(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, 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 3(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.

4.3 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 2 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

[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.

(for example, f_0 in Figure 2), then internal fragmentation may occur leading to quicker wrap-around, for example, f_i in Figure 2. 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 [1, 28, 39], while files in database workloads are quite large being in the tens of MB to tens of GB range [29, 44]. 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 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.

4.4 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. When all the chunks have been written to, the not-completed flag is reset, the write requesting application is notified, and the locks for all chunks are released, in this order.

4.5 Failure Recovery

The contents of FR are always recoverable from faults, be it hardware or software faults. The only exception is recovery from PM device failure where PM content is irrecoverable due to media failures or partial or full data corruption. Due to space, we only give the gist of recovery from single chunk write failure, which is that, for all faults occurring at any step in Figures 3(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.

Recovery from multi-chunk write failures is more involved, the key to the solution being the use of the not-complete bit to make sure recovery is done for all. However, we omit the detailed description as well as the experimental validation results, which we conducted, due to space.

5 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 4, with Direct I/O. One limitation of our current implementation of FR is that it does not support `mmap()`. This is because

Table 2: 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

`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.

5.1 Experiment Platform and Benchmarks

Table 2 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 [14]. 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.

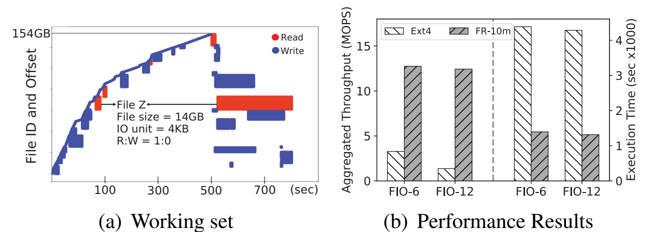
5.2 PM as a Cache

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. 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 [9], which is an I/O testing tool. Using FIO, we generate 21 distinct files using the characteristics shown in Table 3, 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

Table 3: 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 4: (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.

is the request intensity of the Alibaba workload [47]. Note that the request interval of 1 millisecond would be the intensity of the Snowflake workload [49], 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 4(a) shows how the working set changes, where the *y*-axis represents the files accessed (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 4(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

Table 4: 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

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 4(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 hashing function, we found several open sourced ones and out of those used djb2 [57], which showed the best performance. All other code, except for the LRU policy, which was an in-house implementation, were taken from Linux. Table 4 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× 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 Smaller than Cache Size Workloads

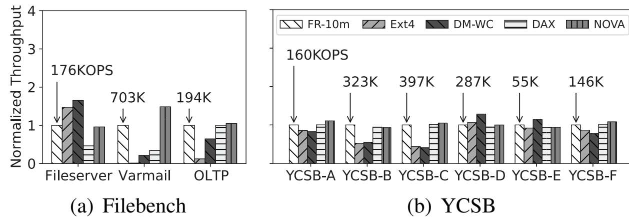
To compare FR performance with state-of-the-art PM file systems such as NOVA or DAX, we need to reduce the dataset size. In this section, we discuss the results for such a setting. Along with baseline Ext4, we also compare with Ext4 with PM DM-WriteCache (denoted DM-WC). However, it must be noted that 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 [4]. 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 5. 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 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 [6], 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 [56] 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

Table 5: 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



(a) Filebench (b) YCSB

Figure 5: Filebench and YCSB throughput normalized to FR, with its absolute performance numbers shown.

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 5(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 [17] and one-third of the workload are reads, and with the added peculiarities of PM [53], Ext4 and DM-WC should do better. In contrast to Fileserver, we see that for the Varmail and OLTP workloads, which are `fsync()`

Table 6: 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

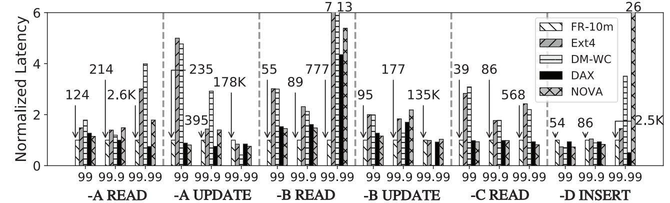


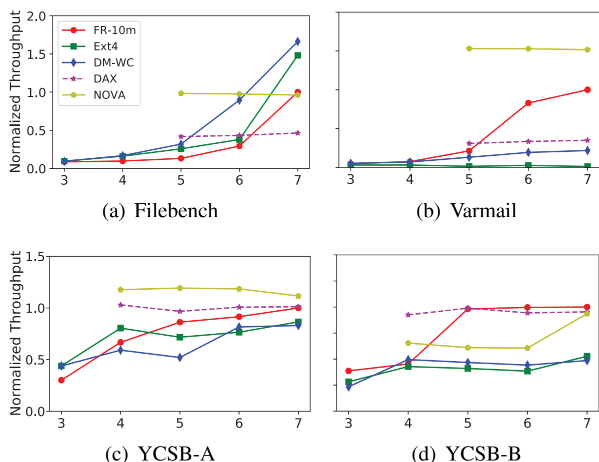
Figure 6: YCSB 99th, 99.9th, and 99.99th percentile tail latency as reported by YCSB normalized to FR. Numbers pointing to FR bars are absolute latency for FR in microseconds.

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 5(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 5(b) and Figures 9(a) and 9(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 6. 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.



(a) Filebench (b) Varmail (c) YCSB-A (d) YCSB-B

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

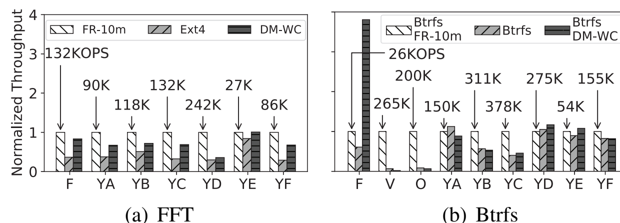
To help understand, we make use of Table 6, which shows how the workloads execute.

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 files as calculated by the stride multiplied by chunk size. These are different from those of Table 5, 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 6 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.

5.4 Other Factors

Effect of PM Size: Figure 7 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.



(a) FFT (b) Btrfs

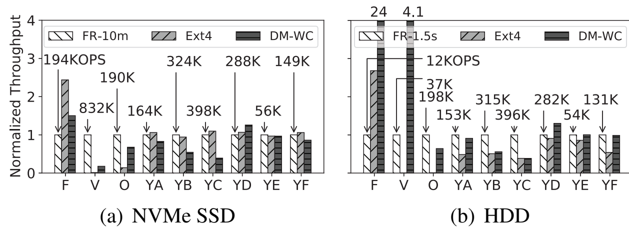
Figure 8: (a) YCSB executed with FFT modules taking up 108GB of DRAM capacity and (b) FR applied to Btrfs.

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 [34] 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 8(a).

Overall, the results show that, while the absolute performance drops compared to those shown in Figure 5, 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 [41], where changes are made to roughly $\sim$30 LOC. As shown in Figure 8(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 5 are that 1) FR-Btrfs does better even for Fileserver compared to Btrfs though in terms of absolute throughput, Btrfs does considerably 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



(a) NVMe SSD (b) HDD

Figure 9: Performance with different storage devices.

lower-end Seagate 1TB Barracuda SATA3/7200/64M HDD. Figure 9(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 5 and 9 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. (We have studied the sensitivity of period setting on performance, but omit them due to space.) 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 5).

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 9(b), with 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 5), the improvements diminish, sometimes even resulting in performance worse than Ext4. Then, with the fastest device (NVMe SSD in Figure 9(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 5, 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.

6 Conclusion

In this paper, 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 $9\times$, 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.

Acknowledgements

We thank our shepherd Professor Sasha Fedorova and the anonymous reviewers for their invaluable comments. We also thank Choulseung Hyun, Sunghwan Kim, Se Kwon Lee, and our colleagues from the NECSST lab for their numerous discussions that helped shape this research. This work was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-IT1402-52.

References

[1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[2] Thomas Alexander and Gershon 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.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2020.

[4] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett 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.

[5] Btrfs. Btrfs. https://github.com/torvalds/linux/tree/master/fs/btrfs.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[7] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-space NVM File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[8] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rahesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2014.

[9] Freecode. fio. http://freecode.com/projects/fio.

[10] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[11] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[12] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2018.

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

[14] 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.

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

[16] Joseph Izraelevitz, Terence Kelly, and Aasheesh 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.

[17] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. Technical report, Computer Science Engineering, University of California, San Diego, 2019.

[18] Song Jiang and Xiaodong 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.

[19] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, and Kolli Aasheesh. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

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

[21] Hyojun Kim and Seongjun 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.

[22] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur 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.

[23] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[24] Eunji Lee, Hyokyung Bahn, and Sam 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.

[25] Eunji Lee, Hyokyung Bahn, and Sam 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.

[26] Eunji Lee, Hyokyung Bahn, Seunghoon Yoo, and Sam 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.

[27] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam 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.

[28] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[29] Bunjamin Memishi, Raja Appuswamy, and Marcus 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.

[30] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay 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.

[31] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[32] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen 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.

[33] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[34] PARSEC. PARSEC. https://parsec.cs.princeton.edu/index.htm.

[35] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceedings of the Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[36] 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.

[37] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[38] Simone 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 Scalable Technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[39] Kai Ren, Qing Zheng, Swapnil Patil, and Garth 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.

[40] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2018.

[41] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 2013.

[42] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, and Andrea 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.

[43] Yizhow Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[44] Juan Sillero and Javier Jiménez. Editorial opinion: public dissemination of raw turbulence data. *Journal of Physics: Conference Series (JPCS)*, April 2016.

[45] Hyunsub Song, Young Je Moon, Se Kwon Lee, and Sam 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.

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

[47] 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/.

[48] Shin-Yeh Tsai, Yizhou Shan, and Yiying 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.

[49] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[50] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[51] Qingsong Wei, Chundong Wang, Cheng Chen, Yechao Yang, Jun Yang, and Mingdi 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.

[52] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[53] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve 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.

[54] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better Than Interrupt. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[55] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai 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.

[56] YCSB. Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

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

[58] Shengan Zheng, Morteza Hoseinzadeh, and Steven 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.

[59] Pengfei Zuo, Yu Hua, and Jie 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.