2017 IEEE 35th International Conference on Computer Design

Adaptive Prefetching for Accelerating Read and Write in NVM-based File Systems

Shengan Zheng, Hong Mei, Linpeng Huang, Yanyan Shen, Yanmin Zhu
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Email: {venero1209, meih, lphuang, shenyy, yzhu}@sjtu.edu.cn

***Abstract*—The byte-addressable Non-Volatile Memory (NVM) offers fast, fine-grained access to persistent storage. While DRAM and NVM have similar read performance, the write operations of existing NVM materials incur longer latency and lower bandwidth than DRAM. This read-write asymmetry nature of NVM causes two bottlenecks for accessing read- and write-intensive file data: expensive data block lookups via file inner structure and high-latency direct writes to data blocks in NVM. However, existing NVM-based file systems fail to address both bottlenecks well.**

This paper presents WARP, an adaptive prefetching module designed for NVM-based file systems, which aims to deal with two bottlenecks effectively. WARP employs two acceleration approaches: 1) mapping data blocks into kernel virtual address space to bypass the indirection of file inner structure for read-intensive file data; and 2) allocating DRAM buffer to absorb frequent writes for write-intensive file data. We design a *WARP benefit model* to identify read- and write-intensive access patterns for file data, and use a successor prediction model to predict future data access based on historical file access traces. With WARP, we are able to prefetch file data according to both file access patterns and traces with consistency guarantee. WARP can be implemented on various NVM-based file systems, and we choose HMVFS for the experiments. The evaluation results show that HMVFS with WARP provides high prefetching accuracy and up to 32%-83% improvement compared with the state-of-the-art NVM-based file systems.

I. Introduction

The emergence of Non-Volatile Memory (NVM) technologies such as PCM and 3D-Xpoint [1] leads to significant changes to the storage hierarchy in personal computers, mobiles, cloud, and high-performance platforms. As a result, various NVM-based file systems such as SCMFS [2], BPFS [3], PMFS [4] and NOVA [5] have been proposed to gain great performance benefits of the byte-addressability and non-volatility properties of NVM [6].

While DRAM and NVM have similar read performance, the write operations of existing NVM technologies incur higher latency and lower bandwidth than DRAM, as illustrated in Table I. On one hand, since the read latency of NVM is close to DRAM, the indirections of file inner structure in locating file data become the bottleneck for file reads [7]. Most NVM-based file systems employ the same hierarchical file inner structure as traditional file systems, which leads to considerable overhead for traversing software routines to search the physical address of each data block. On the other hand, the write latency of NVM is much higher than that of DRAM, and the overhead from the direct write access degrades the performance of file write operations [8]. Unfortunately, direct write to all the file data in NVM is applied to almost all the existing NVM-based file systems, which leads to suboptimal system performance. Hence, accelerating read and write operations has become critical in order to achieve high performance for NVM-based file systems.

Inspired by SIMFS [7] and HiNFS [8], we propose two kinds of optimization approaches to accelerating file reads and writes in NVM-based file systems. For file data being frequently read, we map data pages into kernel virtual address space and users can read them via one *memcpy* operation without using software routines to iteratively search for the addresses of data blocks. For file data being frequently written, a DRAM buffer is allocated to absorb multiple writes to the same NVM block, and then the buffered data is written back to NVM with consistency guarantee. In this way, we are able to yield comparable write performance as DRAM.

However, applying the above two optimization approaches into one file system simultaneously is a nontrivial task. The page table entries in kernel virtual address space should be consistent with the file data either in NVM or DRAM buffer at any time. In addition, each out-of-place file write, including shadow paging in BPFS [3], log-structured write in HMVFS [9] and copy-on-write in PMFS [4] and NOVA [5], will lead to updates in the corresponding page table entries, because the addresses of the valid data blocks on NVM will be changed. Furthermore, allocating and deallocating DRAM buffer will introduce unnecessary time overhead for read-intensive files, since the read latency and bandwidth of NVM are close to those of DRAM. Unfortunately, existing NVM-based file systems are typically blind to file access patterns. If the files can be prefetched using different optimization approaches according to their own access patterns, both optimization approaches can be deployed in one file system more effectively.

The development of an effective prefetching mechanism (for deploying read/write acceleration approaches in NVM-based file systems), however, is challenging. The reason is three-fold. First, the target file data to be prefetched should be selected carefully in order to reach maximum performance benefit with minimized prefetching overhead and the limited DRAM space. Second, the acceleration type (i.e., mapping data blocks into kernel virtual address space for read acceleration, or

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.17

allocating DRAM buffer for write acceleration) should be determined by the previous access pattern of the file data in order to improve prediction accuracy as much as possible. Simply prefetching file data with its recent access pattern may cause frequent switching between different acceleration approaches, which leads to even worse performance. Third, prefetching phase should be removed from the critical path. For instance, if prefetching is performed at the beginning of the actual read/write, its overhead will substantially affect the performance of foreground file I/Os. In particular, the file system has to identify the most beneficial file data, and prefetch it according to its previous access pattern before the actual read/write happens.

In this paper, we propose WARP, an adaptive prefetching module for NVM-based file systems. The prefetching granularity of WARP is **node**, i.e., several continuous data blocks of a file. The key idea of WARP is to: 1) recognize read-intensive and write-intensive access patterns for each node with a *WARP benefit model*; 2) identify the access successor relationships among nodes; and 3) perform prefetching to its successor node in an adaptive way once the predecessor node is accessed. Specifically, we collect file access traces in the file system, and leverage *WARP benefit model* to identify read-intensive and write-intensive access patterns for each node. For read-intensive nodes, we map data blocks into kernel virtual address space in order to bypass the indirection of file inner structure. For write-intensive nodes, we allocate DRAM buffer to absorb writes to them. The stable access patterns are written back to NVM periodically so that the file system can memorize them in order to continuously perform future access prediction after reboot. We implement our adaptive prefetching module in HMVFS [9], and evaluate the performance using micro- and macro-benchmarks.

The contributions of this paper are the following:

• We identify two performance bottlenecks in NVM-based file systems: expensive data block lookups via file inner structure and high-latency direct writes to data blocks in NVM. We propose two optimization approaches to deal with the bottlenecks: mapping data blocks into kernel virtual address space to bypass the indirection of file inner structure for read-intensive file data, and allocating DRAM buffer to absorb writes for write-intensive file data.

• We develop an adaptive prefetching strategy to deploy our optimizations with the objective of maximizing overall I/O performance. We collect and predict file access patterns and file access traces with *WARP benefit model* and *successor prediction model*, and perform prefetching adaptively.

• We implement WARP in HMVFS, a Hybrid Memory Versioning File System [9]. The evaluation results show that HMVFS with WARP outperforms two state-of-the-art NVM-based file systems, PMFS and NOVA, by up to 83% and 32%, respectively.

The remainder of this paper is organized as follows. Section II provides background and our motivations. Section III presents read and write optimization approaches. The design and implementation of WARP are given in Section IV and Section V, respectively. We discuss the evaluation results of WARP in Section VI. Finally, we provide related work in Section VII and conclude our paper in Section VIII.

TABLE I
COMPARISON OF NVM CHARACTERISTICS

Technology	Read Operations		Write Operations	
	Latency	Bandwidth	Latency	Bandwidth
DRAM	15ns	15GB/s	15ns	10GB/s
NAND Flash [10], [11]	25μs	25-400MB/s	200-500μs	10-25MB/s
PCM [12]	50ns	10GB/s	350-1000ns	2GB/s
Memristor [13], [14]	100ns	10GB/s	100ns	5GB/s
STT-RAM [15]	40ns	15GB/s	40ns	10GB/s

II. BACKGROUND AND MOTIVATION

Table I summarizes the characteristics of different NVM technologies and we compare them with traditional memory and storage [16]–[22]. As we can see, different NVM materials have different IO characteristics, but the common points are longer latency and lower bandwidth compared with DRAM [8]. While existing NVM-based file systems take performance degradation caused by read-write asymmetry of NVM into account, none of them provides flexible read/write strategies that can be adaptive to different file access patterns.

We first perform a preliminary experiment to illustrate our motivation. We set the read latency and write latency of NVM to 50ns and 500ns, respectively. In Figure 1, we show the time overhead of the file system performing a read or write operation to 16KB data, which typically includes locating data block and read/write access latency for each 4KB data block.

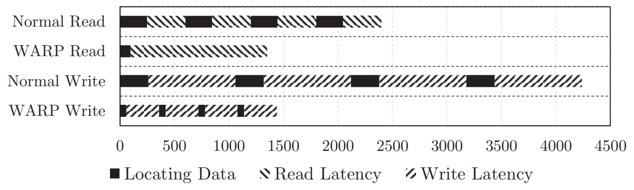
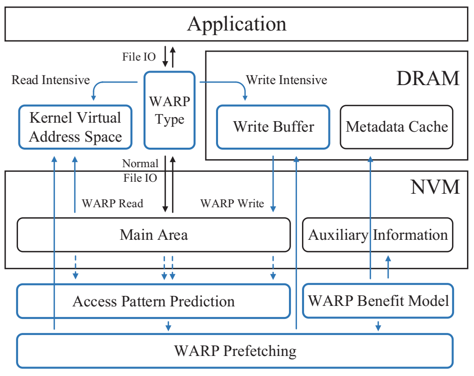
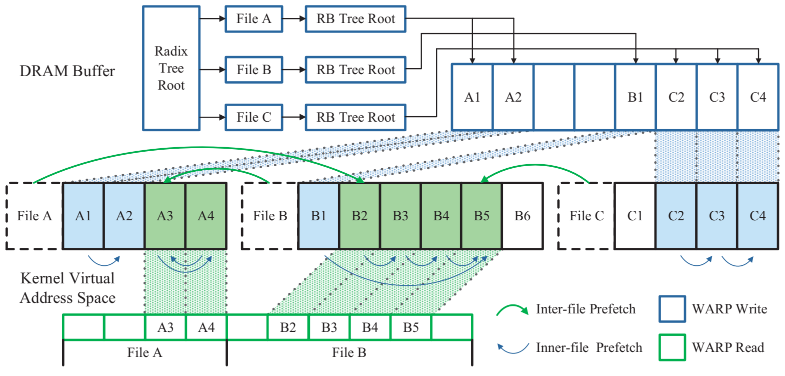


Fig. 1. Time overhead analysis for read and write on 16KB file data (ns)

NVM has similar read latency and bandwidth as DRAM and the proportion of it in read operations is provided in the *normal read* stripe (42%). It is easy to see that **the major bottleneck for read operations is not IO latency but the cost of locating data blocks**. That is, long addressing routine for each data block access limits the throughput of the whole read operation. If these data blocks can be mapped continuously into kernel virtual address space, the addressing procedure will be performed by MMU directly to translate virtual address to physical address of the data block in NVM. Moreover, a single request of accessing multiple continuous data blocks in traditional file systems is broken down to a traversal over individual data blocks separately. With the help of continuous address space, addressing such data blocks can be performed only once and the data can be fetched continuously by one `memcpy` function (the *WARP read* column), which leads to about 81% speed up compared with *normal read*.



(a) Architecture of HMVFS with WARP



(b) Design of WARP

Fig. 2. Architecture and Design of WARP

For write operations, the high write latency of NVM dominates the overhead of NVM writes, as illustrated in the *normal write* stripe (76%). Hence, **the major bottleneck for write operations is mainly caused by I/O latency rather than locating data blocks**. To overcome the high write latency problem of NVM, a natural way is to allocate a DRAM buffer to absorb writes to NVM, and perform write back only when synchronization is needed. As one can see from the *WARP write* stripe, write buffer is helpful to reduce write overhead caused by NVM and perform writes more efficiently. The latency of locating data is also reduced since locating DRAM buffer is faster than original routine. *WARP write* roughly achieves 193% speedup compared with *normal write*.

III. Read and Write Optimizations

Before delving into the details of read and write optimizations, we first discuss the granularity of our prefetching approach. On one hand, we observe from many real workloads that adjacent data blocks often share similar access patterns. Therefore, the granularity of file access pattern prediction should be continuous data blocks to retain spatial locality. As a benefit, the size of metadata can be reduced when using a larger granularity, e.g., it takes about 64 times overhead to implement with cacheline granularity (64B) than block granularity (4KB). On the other hand, the granularity also needs to be a fraction of file size instead of the whole file. Consider an application that constantly accesses a hot area of a large file, in which building write buffer in DRAM for the whole file would squander precious DRAM space. We set the granularity (**node**) to 2MB by default in HMVFS with WARP.

The data consistency of the read/write optimization approach is guaranteed by `checkpoint`, a function for file systems to write back all the data from both CPU cache and DRAM to NVM. It is used to ensure the file system consistency at a certain time point. In HMVFS, `checkpoint` is implemented as an operation to create a snapshot from the last one differentially. In other in-memory file systems, it is equivalent to `sync`. WARP supports both file level checkpoints and file system level checkpoints.

A. Read Optimization Approach

For frequently read nodes or files opened with *READ_ONLY* flag, WARP maps the data into kernel virtual address space with *READ_ONLY* permission, as shown in Figure 2(b). Originally, in order to read file data, file system has to go through the file's inner structure to locate data block for each data block within the range. According to our read optimization, once the node is mapped by the background prefetching thread, named `warp_prefetch`, read accesses to the node will be redirected to kernel virtual address space, which can be accessed directly and continuously through the page table entries via MMU. We provide details as follows.

When the file is ready for prefetching, `warp_prefetch` will allocate a continuous virtual address space that is equal to the current size of the file, and store the start address in the inode cache in DRAM. After that, `warp_prefetch` will map each valid data block of the node in NVM into kernel virtual address space. When the prefetching is completed, the read accesses to the node will be redirected to kernel virtual address space, which ensures consistency. If an out-of-place write emerges after prefetching, WARP will update the mapping address accordingly.

Compared with traditional access routine that repeatedly traverses file inner structure from inode block to data block for each data block with the complexity of $O(\#Datablocks)$, WARP read only requires a simple call to the `copy_to_user` function. The kernel will perform the copy from file address space to user address space with any starting address and length with $O(1)$ cost.

B. Write Optimization Approach

For frequently written nodes or files opened with *WRITE_ONLY* flag, WARP allocates a write buffer in DRAM to absorb writes to NVM. Once the write buffer is active, write requests to the nodes or files will be intercepted by the write buffer in DRAM. WARP will only perform write back to NVM during `checkpoint`. The write buffer will be reclaimed if it has not been accessed for a period of time. Therefore, the size of the DRAM write buffer is only a small fraction of total DRAM space.

The write optimization function will build a radix tree for all frequently written files and a red-black tree for the data blocks of each file. As shown in Figure 2(b), we use radix tree to organize files because radix tree is efficient in locating dirty files from clean ones. The inode numbers of the dirty files are discrete, which makes radix tree the best choice to pop them out one by one during writeback. The dirty blocks of each file are in a different situation, since they tend to be clustered due to the locality of file access. The red-black tree contains pointers to adjacent data blocks, which enables WARP to write back multiple continuous data blocks via one function call. To write back data from DRAM buffer to NVM, we modify `fsync` and `sync` to perform an actual write to NVM. The time interval of write back can be set by the user or periodically performed in the background.

IV. WARP: Adaptive Prefetching

A. Overview

To facilitate the read and write optimizations in the NVM-based file systems, we introduce WARP, an adaptive prefetching module. WARP predicts the access patterns and traces of file data, and utilizes the read and write acceleration approach described in Section III to provide high overall I/O performance. The architecture of WARP-enabled NVM-based file system is shown in Figure 2(a) and the design details of WARP are illustrated in Figure 2(b). At a high level, WARP maps data blocks into kernel virtual address space and allocates DRAM write buffer for read- and write-intensive file data, respectively. When a node is accessed during runtime, WARP records its access trace and pattern. During `checkpoint`, WARP identifies the most beneficial acceleration type based on a benefit model. If the node has a stable access pattern, the acceleration type will become active and be stored in DRAM cache and NVM. The following access to this node or its predecessor node will activate the background prefetching thread `warp_prefetch` to invoke the corresponding read or write acceleration function. After that, the node can be accessed in an accelerated way. Algorithm 1 summarizes the major steps of `warp_prefetch` and `checkpoint`.

The design of WARP consists of several key components: *WARP benefit model* to predict access patterns (i.e. the most beneficial acceleration approaches), *successor prediction model* to predict access traces (i.e. the most probable successor nodes), and *prefetching algorithm* to integrate them with read and write acceleration approaches. We provide details in the following sections.

B. WARP Benefit Model

We first propose a benefit model to identify the most beneficial acceleration type (i.e. read/write-acceleration or none) for each node based on its own access pattern. As we can see in Figure 1, access latency and I/O bandwidth dominate the time overhead of completing I/O requests. Our benefit model dynamically chooses to perform read or write acceleration

Algorithm 1 WARP Prefetch

```
 1: function WARP PreFetch($node$)
 2:     if $node.warpType = READ$ then
 3:         vmap($node.addr$,$file.vmapaddr$)
 4:     else if $node.warpType = WRITE$ then
 5:         rb_tree_insert($node \rightarrow blocks$)
 6:     end if
 7: end function
 8: function WARP Checkpoint
 9:     Write back dirty $nodes$ from DRAM buffer
10:     for each accessed $node$ do
11:         Gather the access information of $node$
12:         Calculate $T$ with $WARP\ Benefit\ Model$
13:         Find the $minimum$ of $T_{normal}$,$T_{read}$, and $T_{write}$
14:         if the access pattern of $node$ is stable then
15:             Update $node.warpType$ according to $T$
16:         end if
17:     end for
18: end function
```

approach (in Section III-A and III-B) for the node with the objective of minimizing *the node's overall I/O time between two consecutive checkpoints.*

Table II shows the time overhead breakdown of the overall I/O of any node, which consists of locating data, access latency and data transmission. Compared with normal data access situation, read optimization approach has lower locating overhead in reads since the data is already mapped. However, it also incurs higher locating overhead in writes since out-of-place writes change the actual addresses of data blocks. Hence, for frequently-read nodes with few writes, the read acceleration approach provides better read performance.

Write optimization approach, on the other hand, has less locating overhead in locating and data transmission since all the data buffer is maintained in DRAM. However, it incurs additional overhead in write back during checkpointing. Dirty data blocks have to be written back to NVM periodically to prevent data loss, which makes this approach vulnerable with respect to frequent writebacks of eager-persistent files to NVM. Otherwise, write optimization approach provides near-DRAM write speed for frequently written nodes thanks to the DRAM buffer.

Our *WARP benefit model* takes both read and write operations into account to offer the best overall I/O performance. We use the following expression to calculate the time overhead of adopting read or write optimization approach between two consecutive checkpoints:

$$T = N_{Read} \times L_{ReadLatency} + S_{Read} \times V_{ReadBandwidth}^{-1} + N_{Write} \times L_{WriteLatency} + S_{Write} \times V_{WriteBandwidth}^{-1}$$

Note that the latency includes locating overhead caused by each data block access, and T contains writeback overhead involved in the write optimization approach to write back to NVM during checkpoint. N and S are the number of access times and total I/O access size, respectively. They are increased for each accessed node during runtime. During checkpoint, we compute T for each of normal, read-optimized and write-optimized modes, and the mode with smallest T is selected to be the best way to access the node for now. If the access pattern remains stable over several checkpoints,

TABLE II
COMPARISON OF I/O OPTIMIZATION METHODS

Optimization	Material	Read Overhead		Write Overhead		Checkpoint Overhead
		Locating	Data Transmission	Locating	Data Transmission	
Normal	NVM	Normal	$N_{Read} \times L_{ReadLatency} + S_{Read} \times V^{-1}_{ReadBandwidth}$	Normal	$N_{Write} \times L_{WriteLatency} + S_{Write} \times V^{-1}_{WriteBandwidth}$	N/A
Read Opt.	NVM	Shortened		Enlarged		N/A
Write Opt.	DRAM	Shortened		Shortened		$T_{WriteBack}$

N: The number of access times L: The access latency of memory S: The sum of size of the I/O access V: The transmission bandwidth of memory

we store the acceleration type for the node in NVM and invoke the corresponding prefetching function to carry out the optimization.

C. Successor Prediction Model

There are two types of successor prediction in WARP, i.e., inner-file and inter-file, which correspond to node and file prefetching, respectively. We design a successor prediction model based on Noah [23], [24] to predict future node and file access with little space and time overhead. Noah maintains a record for each recently accessed file, which forms a chain of successors of files. We modify Noah in several aspects to fully exploit the byte-addressability of NVM to conduct accurate and beneficial prefetch on the file data.

We keep the record of the file access traces in both node and file granularity, since the inner- and inter-file access traces are very likely to be stable. For each accessed node, we store inner-file access prediction result in the metadata of the node, which is the last successor node ID of this node. We also store inter-file access prediction result in the inode block, which includes several tuples with the following format: $\{process\_path\_hash\_value, next\_ino, hint\_node\_id\}$ where $process\_path\_hash\_value$ is the hash value of the current working directory path of the process, $next\_ino$ is the inode number of the successor file, and $hint\_node\_id$ is the node ID of the first accessed node in the successor file.

Intuitively, each tuple indicates the access trace of a recent process which has just accessed the file. When a node of a file is accessed again by the same process with a matching hash value in the existing tuples, `warp_prefetch` will automatically invoke the acceleration function to the inner-file successor node and inter-file hint node based on their acceleration types.

D. Prefetching Method

The prefetching method in WARP is basically calling the read or write acceleration function (described in Section III-A and Section III-B) for a node. The prefetching type is the most beneficial acceleration type for the node determined by *WARP benefit model*. The prefetching victim nodes are the most likely to be immediately accessed according to the historical access traces calculated by our *Successor Prediction Model*. Once the victim nodes and their acceleration types are determined, `warp_prefetch` will call the corresponding optimization functions to prefetch these nodes in the background. When the prefetching is completed, the following read/write accesses to this node will be accelerated by the corresponding acceleration approach.

The access trace information is written back to NVM during `checkpoint`, which takes little overhead thanks to the byte-addressability of NVM. The information enables I/O acceleration right after reboot with little time overhead on recomputing the access patterns and traces. Deploying acceleration approaches in the very beginning allows applications to perform fast I/O once the file system is mounted.

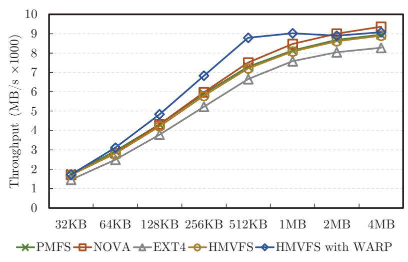
Prefetching a node with the most beneficial acceleration type boosts the node I/O access significantly. Inaccurate prefetching, however, causes the file system to suffer from suboptimal performance or undo the existing acceleration, e.g., unmapping the data blocks or deallocating the DRAM buffer. Files with unstable access patterns may cause frequent switches between different acceleration approaches. Even for files with stable access patterns within a time period, their access patterns may change in the long run.

To avoid inaccurate prefetching as much as possible, we introduce a threshold ΔT and *period of confirmation* (several checkpoints) to decide whether to change the current node access mode or not. Specifically, when the time overhead T of the new mode (with the smallest T) outperforms the current mode by ΔT throughout the *period of confirmation*, the new mode becomes the stable acceleration type for the node. WARP stores stable acceleration type in the node's cache in DRAM and metadata in NVM afterwards. By default, we set ΔT to $20\%T$ and the *period of confirmation* to three checkpoints.

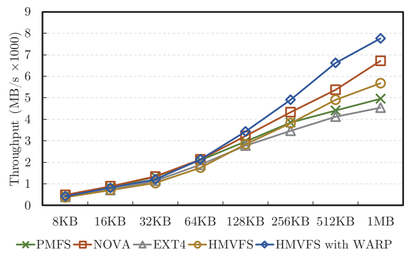
V. IMPLEMENTATION

We implement WARP in HMVFS, an NVM-based versioning file system [9]. Note that WARP can also be implemented in other NVM-based file systems such as PMFS [4] or NOVA [5] to improve read and write performance. The implementation details of WARP involve three aspects as follows.

Granularity. We use node as the granularity of file prefetching, which covers 2MB data of a file in HMVFS. In other file systems, the size of the node can be modified according to the specific file structures and usage patterns. For example, PMFS can be implemented with larger page sizes (up to 1GB), in this case, the data block itself can be used as the granularity of WARP instead of node. Moreover, the granularity can also be adjusted according to the file access patterns. A smaller granularity is beneficial for random

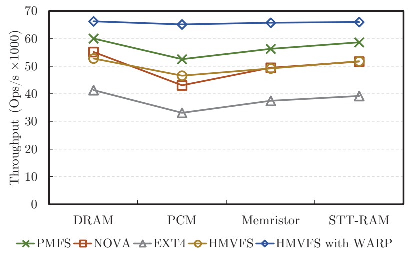


(a) Sequential read

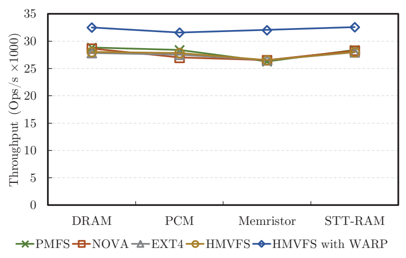


(b) Random write

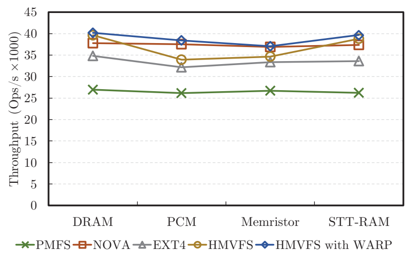
Fig. 3. Sensitivity to the I/O size



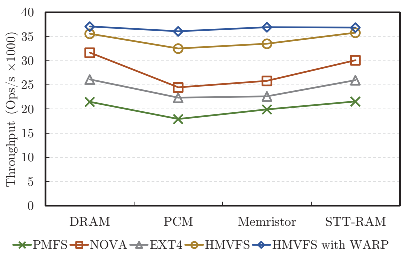
(a) Fileserver



(b) Webserver



(c) Webproxy



(d) Varmail

Fig. 4. Overall performance

accesses to the files, while a larger granularity is better for sequentially accessed files.

Checkpoint. NVM-based file systems have to guarantee consistency during runtime. Therefore, the functionality of checkpoint is available in almost all the existing file systems, by either setting up a strictly consistent file system image or recording each file system update with journaling and logging. Checkpoints are normally implemented in either file system level or file level. The checkpoints of HMVFS are at file-system level, hence a search tree is needed for file system to track accessed file status and perform optimization approach during checkpointing. However, in other file systems such as NOVA [5], there is no need for WARP to store auxiliary information and perform acceleration at file system level. Rather, access information can be gathered, processed, stored and accelerated at file level. When a pattern is recognized, file system can still alert `warp_prefetch` to prefetch data blocks, write back from DRAM buffer, and store auxiliary information of WARP at file level.

Consistency. The auxiliary information of WARP for each node is stored and updated in DRAM at runtime. During checkpointing, the stable access type and successor node ID of each accessed node are written back to NVM. There is no need to maintain strict consistency guarantee when writing back auxiliary information of WARP. This is because incorrect auxiliary information of WARP will not jeopardize data or metadata consistency of the file system, but only cause a longer period to recompute the access patterns of nodes.

VI. Evaluation

In this section, we evaluate the performance of HMVFS with WARP and answer three questions: (1) What is the benefit of read/write acceleration approach? (2) How does WARP perform against existing in-memory file systems in real workloads? (3) What is the accuracy of WARP prefetching with WARP benefit model and the successor prediction model?

We use Filebench micro- and macro-benchmarks to answer the first two questions, and provide detailed analysis of prefetching accuracy to address question (3).

A. Experimental Setup

Since real NVM devices are not available yet, we develop a simple performance emulator based on the NVM emulator used in the Mnemosyne [25] and HiNFS [8] to evaluate the performance of HMVFS with WARP. Our NVM emulator introduces extra latencies for both NVM load and store operations to emulate the slower reads and writes of NVM. Furthermore, regarding the limited bandwidth of NVM, we set a maximum bandwidth for NVM according to Table I. We evaluate the performance of HMVFS with WARP against three existing file systems: PMFS, NOVA, and EXT4(DAX). They are the available open-source NVM-based file systems which access NVM directly.

We conduct experiments on a commodity server with 64 Intel Xeon 2GHz processors and 128GB DRAM, running the Linux 3.11 kernel (We only switch to Linux 4.3 for NOVA). For EXT4, we use ramdisk carved from DRAM and configure 64GB as ramdisk to simulate NVM. For HMVFS, PMFS, and NOVA, we reserve 64GB of memory using the *grub* option *memmap*. User processes and buffer cache use the rest of the free DRAM space. All the results are averaged over five runs.

B. Sensitivity to Different I/O Sizes

I/O size is an important factor in the performance of file systems. We use sequential read and random write benchmarks from Filebench to evaluate the sensitivity to the I/O size. The latencies of reads and writes are set to 50ns and 300ns as the mean latencies of different NVM materials, respectively. Figure 3 evaluates the bandwidth of sequential read and random write. As we can see, HMVFS with WARP performs the best among all file systems. The performance gain is up to 24% in sequential read and 37% in random write compared to HMVFS.

In sequential read benchmark, WARP gains maximum performance benefit with the I/O size of 512KB. This is because smaller I/O size introduces more read requests that can hardly be merged. In the meantime, the throughput of larger I/O sizes such as 2MB or 4MB is approaching the memory bandwidth. PMFS, NOVA, and EXT4 have similar performance because the data is directly accessed in NVM.

For random write, the performance gain provided by WARP is increasing with larger I/O sizes. The granularity of WARP is 2MB for HMVFS which means that random write requests within 2MB can be handled via the accelerated DRAM buffer without further metadata level queries. For other NVM-based file systems, NOVA achieves the best throughput because of its log-structured metadata design. NOVA keeps data updates as logs in the inodes, which can perform faster writes with larger I/O size.

C. *Overall Performance*

We select four Filebench marco-workloads: fileserver, webproxy, webserver and varmail to evaluate the application-level performance of WARP. For each workload, we add additional I/O latency and bandwidth limitations of PCM, Memristor, and STT-RAM according to Table I. We use the default configurations in each workload. Figure 4 shows the Filebench throughputs with respect to different NVM materials.

In the fileserver workload, HMVFS with WARP outperforms other file systems by 23%-96% in terms of the throughput. The read/write ratio of fileserver is 1:2 which contains both reads and writes with 1MB I/O size, and 16KB append size. The performance drops by about 27% for HMVFS without WARP because WARP identifies write-intensive nodes and accelerates them with DRAM write buffer.

Webserver is a read-dominated workload with multiple read threads and one append thread. It does not involve any directory operations. The results show that most file systems perform similarly to each other due to the low read latency of NVM. Meanwhile, WARP identifies read-intensive nodes and maps them into kernel virtual address space, which results in a 15% higher performance on average.

Webproxy is a read-intensive workload. WARP performs similarly to NOVA because the read/write ratio is more balanced and it has operations like file creation and deletion which can not be accelerated by WARP. PMFS performs the worst in the Webproxy workload because of its poor scalability in a directory containing too many files.

Varmail emulates an email server with a lot of append and read operations. Since the I/O size of reads and writes is 1MB and 16KB, respectively, it fits perfectly to the log-structured nature of HMVFS which outperforms other file systems. HMVFS with WARP also contributes to the performance gain by facilitating read acceleration to some files in the workload. On average, HMVFS with WARP outperforms PMFS, NOVA, and EXT4 by 83%, 32% and 52%, respectively.

To summarize, WARP achieves a significant performance gain in almost all cases and successfully alleviates the performance degradation caused by high write latency and limited bandwidth of NVM. The advantages of WARP are maximized for the workloads with stable access patterns.

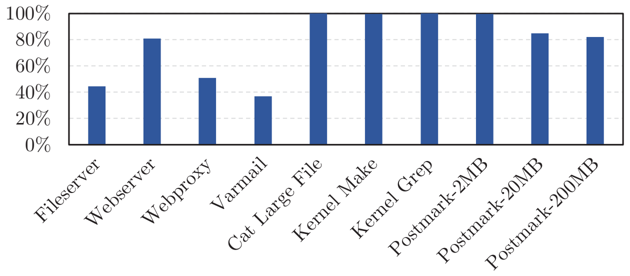


Fig. 5. The prefetching accuracy of WARP

D. *Prefetching Accuracy*

WARP stores the access pattern of each node in DRAM and NVM periodically to boost current and future access prediction after reboot. If the access prediction is correct, the background prefetching thread will provide better overall performance. Therefore, it is essential to evaluate the prefetching accuracy over real workloads. We define *an accurate prefetch* as: not only does the currently being accessed node ID equal to the record in the last accessed node, but also the access type is matched with its own acceleration type record. We label each access between two consecutive checkpoints as an accurate or inaccurate prefetch and the prefetching accuracy is measured by the percentage of accurate prefetches.

We select fileserver, webserver, webproxy and varmail workloads from Filebench, as well as `cat` 1GB file, kernel make, and kernel grep. The prefetching accuracy of Postmark is also evaluated with respect to different file sizes, and the transaction number of Postmark is set to 100,000.

Figure 5 provides prefetching accuracy over all the workloads. We can see that the workloads in Filebench have relatively low accuracy because of multi-threading and the read-write-mixed nature of the workloads. Regarding kernel make, kernel grep and `cat` large file, their accuracies are all above 99%. This is because their access patterns follow a fixed routine during each execution. Postmark achieves the highest accuracy when the file size is 2MB. Larger file size reduces accuracy since the inner file access can also cause inaccuracy.

To summarize, WARP provides high prefetching accuracy to workloads with stable and preferably, read- or write-dominated access patterns. Applications like frequently-read media player or frequently-written journal recorder will receive higher performance gain. We expect Big Data applications such as graph-computing and super-computing may benefit from WARP.

VII. Related Work

Since block-based file systems cannot fully exploit the byte-addressability of NVM, many NVM-aware in-memory file systems have been proposed. Prior NVM-aware file systems such as PMFS [4], SCMFS [2] and BPFS [3] focus on providing byte-addressable I/O functionality and consistency guarantee, while recent ones such as SIMFS [7], HiNFS [8] and NOVA [5] emphasize on achieving higher performance.

Researches have been conducted on how to accelerate file reads and writes separately. On one hand, SIMFS [7] accelerates file reads by mapping the address space of an opened file into the process address space. In SIMFS, the data of a file is organized by a structure called *file page table*, which is similar to a segment of any normal page table. *File page table* is attached to or detached from the process address space when the file is opened or released by the process. This eliminates the overhead of accessing the indirections of the internal structure of a file.

However, NVM has longer access latency and lower bandwidth than DRAM. As a consequence, the delay of memory copy from user buffer to NVM becomes the bottleneck of the whole write operation. To make things worse, SIMFS uses *pseudo-file-write* to ensure write consistency among processes, which introduces additional overhead in updating *file page table* per write.

On the other hand, HiNFS [8] uses an NVM-aware write buffer policy to maintain the lazy-persistent file writes in DRAM, and persists them to NVM lazily in order to hide long write latency of NVM. Given different write latencies of DRAM and NVM, a *Buffer Benefit Model* is proposed to distinguish whether buffering is more efficient than non-buffering for any data block.

Although the *Buffer Benefit Model* accelerates file writes accurately, HiNFS has to keep records of file writes in cacheline granularity, which introduces additional write amplification and indexing overhead. Moreover, file reads in HiNFS have to go through additional layers to check whether the data blocks are buffered in DRAM or not, which degrades the performance of read-intensive applications.

To accelerate overall file access, NOVA [5] gives each file a private metadata log, allowing NOVA to operate on files without contending for any shared resources. The data blocks are organized with metadata logs, which are small to facilitate efficient garbage collection and fast recovery. Compared with the file systems above, WARP enables adaptive prefetching for NVM-based file systems, which accelerates both file reads and writes simultaneously according to different file access traces and patterns.

VIII. Conclusion

This paper proposes WARP, an adaptive prefetching module designed for NVM-based file systems to accelerate file data access according to access traces and patterns. WARP determines the most beneficial acceleration approach with a detailed time-overhead based benefit model, and maps file data into kernel virtual address space or allocates DRAM write buffer accordingly, to improve the overall performance of the file system. Our measurements show that the file system with WARP achieves high prefetching accuracy and up to 83% and 32% improvements compared with the state-of-the-art NVM-based file systems PMFS and NOVA, respectively.

IX. Acknowledgment

We thank the anonymous reviewers for their insightful and helpful comments, which improve the paper. We also thank Linzhu Wu for his assistance. The work described in this paper is supported by the National High-tech R&D Program of China (863 Program) under Grant No. 2015AA015303 and the National Natural Science Foundation of China under Grant No. 61472241. Yanyan Shen is partially suppported by NSFC (No. 61602297).

References

[1] “3d xpoint,” https://en.wikipedia.org/wiki/3D_XPoint.
[2] X. Wu and A. Reddy, “Scmfs: a file system for storage class memory,” in *SC*. ACM, 2011, p. 39.
[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek *et al.*, “Better i/o through byte-addressable, persistent memory,” in *SOSP*, 2009, pp. 133–146.
[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy *et al.*, “System software for persistent memory,” in *EuroSys*, 2014, p. 15.
[5] J. Xu and S. Swanson, “Nova: A log-structured file system for hybrid volatile/non-volatile main memories,” in *FAST*, 2016, pp. 323–338.
[6] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, “An empirical study of file systems on nvm,” in *MSST*. IEEE, 2015, pp. 1–14.
[7] E. H.-M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, “A new design of in-memory file system based on file virtual address framework,” *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 2959–2972, 2016.
[8] J. Ou, J. Shu, and Y. Lu, “A high performance file system for non-volatile main memory,” in *EuroSys*. ACM, 2016, p. 12.
[9] S. Zheng, L. Huang, H. Liu, L. Wu, and J. Zha, “Hmvfs: A hybrid memory versioning file system,” in *MSST*, 2016.
[10] L. M. Grupp, J. D. Davis, and S. Swanson, “The bleak future of nand flash memory,” in *FAST*. USENIX Association, 2012, pp. 2–2.
[11] “Nand flash 101 introduction,” https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2919_nand_101.pdf.
[12] K. Chen, P. Jin, and other, “A novel page replacement algorithm for the hybrid memory architecture involving pcm and dram,” in *NPC*, 2014.
[13] J. J. Yang, D. B. Strukov, and D. R. Stewart, “Memristive devices for computing,” *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.
[14] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie, “Design implications of memristor-based rram cross-point structures,” in *DATE*, 2011.
[15] T. Kawahara, “Scalable spin-transfer torque ram technology for normally-off computing,” *IEEE Design & Test of Computers*, 2010.
[16] J. Chen, Q. Wei, C. Chen, and L. Wu, “Fsmac: A file system metadata accelerator with non-volatile memory,” in *MSST*. IEEE, 2013, pp. 1–11.
[17] F. Chen, M. P. Mesnier, and S. Hahn, “A protected block device for persistent memory,” in *MSST*. IEEE, 2014, pp. 1–12.
[18] M. K. Qureshi and other, “Scalable high performance main memory system using phase-change memory technology,” *ISCA*, 2009.
[19] Y. Zhang and S. Swanson, “A study of application performance with non-volatile main memory,” in *MSST*. IEEE, 2015, pp. 1–10.
[20] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, “Optimizing checkpoints using nvm as virtual memory,” in *IPDPS*. IEEE, 2013.
[21] J. Zhao and Y. Xie, “Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration,” in *ICCAD*. ACM, 2012, pp. 81–87.
[22] K. Suzuki and S. Swanson, “A survey of trends in non-volatile memory technologies: 2000-2014,” in *Memory Workshop (IMW)*, 2015.
[23] A. Amer and D. D. Long, “Noah: Low-cost file access prediction through pairs,” in *IPCCC*. IEEE, 2001, pp. 27–33.
[24] A. Amer, D. D. Long, J.-F. Pâris, and R. C. Burns, “File access prediction with adjustable accuracy,” in *IPCCC*, 2002, pp. 131–140.
[25] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *ASPLOS*, vol. 39. ACM, 2011, pp. 91–104.