Better I/O Through Byte-Addressable, Persistent Memory

Jeremy Condit Edmund B. Nightingale Christopher Frost[†]
Engin Ipek Benjamin Lee Doug Burger Derrick Coetzee

Microsoft Research [†]UCLA

ABSTRACT

Modern computer systems have been built around the assumption that persistent storage is accessed via a slow, block-based interface. However, new byte-addressable, persistent memory technologies such as phase change memory (PCM) offer fast, fine-grained access to persistent storage.

In this paper, we present a file system and a hardware architecture that are designed around the properties of persistent, byte-addressable memory. Our file system, BPFS, uses a new technique called *short-circuit shadow paging* to provide atomic, fine-grained updates to persistent storage. As a result, BPFS provides strong reliability guarantees *and* offers better performance than traditional file systems, even when both are run on top of byte-addressable, persistent memory. Our hardware architecture enforces atomicity and ordering guarantees required by BPFS while still providing the performance benefits of the L1 and L2 caches.

Since these memory technologies are not yet widely available, we evaluate BPFS on DRAM against NTFS on both a RAM disk and a traditional disk. Then, we use microarchitectural simulations to estimate the performance of BPFS on PCM. Despite providing strong safety and consistency guarantees, BPFS on DRAM is typically twice as fast as NTFS on a RAM disk and 4–10 times faster than NTFS on disk. We also show that BPFS on PCM should be significantly faster than a traditional disk-based file system.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management; D.4.5 [**Operating Systems**]: Reliability; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Performance, Reliability

Keywords

File systems, performance, phase change memory

[†]Work completed during a Microsoft Research internship.

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

1. INTRODUCTION

For decades, computer systems have been faced with a trade-off between volatile and non-volatile storage. All persistent data must eventually be stored on non-volatile media such as disk or flash, but since these devices support only slow, bulk data transfers, persistent data must be temporarily buffered in fast, byte-addressable DRAM. Unfortunately, data that resides only in volatile memory can be lost during a crash or a power failure, which means that existing storage systems often sacrifice durability, consistency, or performance in balancing their use of these two types of storage media.

However, new *byte-addressable* persistent memory technologies (BPRAM) eliminate many of the traditional differences between volatile and non-volatile storage. In particular, technologies such as phase change memory and memristors are byte-addressable like DRAM, persistent like disk and flash, and up to four orders of magnitude faster than disk or flash for typical file system I/O. BPRAM can be placed side-by-side with DRAM on the memory bus, available to ordinary loads and stores by a CPU.

This paper examines the benefits of BPRAM by focusing on one of the primary abstractions for storage: file systems. We have implemented a new file system for BPRAM, called BPFS, which performs up to five times faster than existing file systems designed for traditional, block-based storage devices (e.g., disk or flash), even when those file systems are run on a RAM disk. In addition, BPFS provides strong safety and consistency guarantees compared to existing systems; specifically, it guarantees that file system writes will become durable on persistent storage in the time it takes to flush the cache *(safety)* and that each file system operation is performed atomically and in program order *(consistency)*.

BPFS provides these guarantees by using a new technique called *short-circuit shadow paging*. In traditional shadow-paging file systems, such as ZFS [23] and WAFL [7], updates to the file system trigger a cascade of copy-on-write operations from the modified location up to the root of the file system tree; when the root of the file system is updated, the change has been committed. Short-circuit shadow paging allows BPFS to use copy-on-write at fine granularity, atomically committing small changes at any level of the file system tree. Indeed, BPFS can often avoid copies altogether, writing updates in place without sacrificing reliability.

Short-circuit shadow paging is made possible by two simple hardware primitives proposed in this paper: atomic 8-byte writes and epoch barriers. Atomic writes allow BPFS to commit changes by writing a single value to BPRAM such that power failures and crashes cannot create a corrupted file system image. Epoch barriers allow BPFS to declare ordering constraints among BPRAM writes while still using the L1 and L2 caches to improve performance.

BPFS's approach to storage differs from traditional file systems in several important ways. First, BPFS does not use a DRAM buffer

cache for file system data, which frees DRAM for other purposes. Although accessing BPRAM directly is slower than accessing a DRAM buffer cache, CPU prefetching and caching hide much of this cost. Second, BPFS is optimized for small, random writes instead of bulk data transfer. Where it was once advantageous to amortize the cost of storage transfers over a large amount of data, performing large block-based writes to BPRAM can *hinder* performance by sending unneeded traffic over the memory bus; thus, BPFS writes only a few bytes of data in places where a traditional disk-based file system would write kilobytes. Finally, BPFS dramatically reduces the window of vulnerability for data that has not yet been made durable. Whereas previous file systems typically buffer data for 5–30 seconds before flushing it to disk, data written to BPFS can be made durable in the time it takes to flush the CPU's data cache. In a sense, using BPRAM for file storage allows us to substitute the CPU's data cache for the DRAM buffer cache.

For our evaluation, we focused on the most promising BPRAM technology, called *phase change memory* (PCM). Because DDR-compatible PCM is not yet available, we evaluated BPFS on DRAM, comparing it to NTFS on disk and to NTFS on a RAM disk. We ran microarchitectural simulations to validate our proposed hardware features with simulated PCM, and we used the results to predict the performance of BPFS when running on PCM. Even with conservative estimates, BPFS outperforms NTFS on a RAM disk while simultaneously providing strong reliability guarantees.

In the next section, we will discuss the high-level design principles that we followed during this work. Then, we will present the details of BPFS and of the hardware we have designed to support it. Finally, we will evaluate the performance of these systems on both DRAM and PCM.

2. DESIGN PRINCIPLES

Our work has two primary goals. First, we want to design architectural support for BPRAM that allows operating systems and applications to easily exploit the benefits of fast, byte-addressable, non-volatile memory. Second, we want to design a file system that provides improvements in performance and reliability by taking advantage of the unique properties of BPRAM.

In this section, we discuss in detail three design principles that guided this work:

- BPRAM should be exposed directly to the CPU and not hidden behind an I/O controller.
- Hardware should provide ordering and atomicity primitives to support software reliability guarantees.
- Short-circuit shadow paging should be used to provide fast and consistent updates to BPRAM.

2.1 Expose BPRAM Directly to the CPU

Persistent storage has traditionally resided behind both a bus controller and a storage controller. Since the latency of a read or a write is dominated by the access to the device, the overhead of this architecture does not materially affect performance. Even the fastest NAND flash SSDs have latencies in the tens of microseconds, which dwarf the cost of bus accesses.

In contrast, technologies such as phase change memory have access latencies in the hundreds of nanoseconds [1, 9], which is only 2–5 times slower than DRAM; thus, keeping BPRAM storage technologies behind an I/O bus would waste the performance benefits of the storage medium. Further, I/O buses prevent us from using byte-addressability by forcing block-based accesses. Even the PCI Express bus is primarily designed for bulk data transfer as opposed to high-bandwidth random-access I/O.

Thus, we propose that BPRAM be placed directly on the memory bus, side-by-side with DRAM. The 64-bit physical address space will be divided between volatile and non-volatile memory, so the CPU can directly address BPRAM with common loads and stores. This architecture keeps access latency low and allows us to take advantage of BPRAM's byte-addressability, which would not be possible if BPRAM were placed on an I/O bus or treated as another level of the memory hierarchy behind DRAM. In addition, making BPRAM addressable permits us to use the cache hierarchy to improve the performance of writes to persistent memory.

There are three disadvantages to placing BPRAM on the memory bus. First, there is the possibility that traffic to BPRAM will interfere with volatile memory accesses and harm overall system performance; however, our microarchitectural simulation shows that this is not an issue. Second, the amount of BPRAM available in a system is limited by BPRAM densities and the number of free DIMM slots in a machine. However, since DRAM and PCM have similar capacities at the same technology node [1, 9], we expect to have 32 GB PCM DIMMs at the 45 nm node, which is comparable to the size of first-generation SSDs. Third, placing persistent storage on the memory bus may make it more vulnerable to stray writes. However, previous work on the Rio file cache demonstrated that even without memory protection, corruption via stray writes is rare; about 1.5% of crashes caused corruption with Rio, as opposed to 1.1% with disk [4].

Note that we do not propose completely replacing DRAM with BPRAM. Since BPRAM is still slower than DRAM by a factor of 2–5, and since phase change memory cells wear out after about 10^8 writes, it is still better to use DRAM for volatile and frequently-accessed data such as the stack and the heap.

2.2 Enforce Ordering and Atomicity in Hardware

To provide safety and consistency, file systems must reason about when and in what order writes are made durable. However, existing cache hierarchies and memory controllers that were designed for volatile memory may reorder writes to improve performance, and most existing architectures (including x86) provide no mechanism to prevent this reordering. Although operations such as `mfence` ensure that each CPU has a consistent global view of memory, they do not impose any constraints on the order of writebacks to main memory. One could enforce ordering constraints by treating BPRAM as uncached memory or by explicitly flushing appropriate cache lines, but these approaches are costly in terms of performance.

Instead, we propose a mechanism for software to declare ordering constraints to hardware. In our proposal, software can issue special write barriers that delimit a set of writes called an *epoch*, and hardware will guarantee that each epoch is written back to main memory in order, even if individual writes are reordered within an epoch. This approach decouples ordering from durability; whereas previous approaches enforced ordering by simply flushing dirty buffers, our approach allows us to enforce ordering while still leaving dirty data in the cache. Our proposal requires relatively simple hardware modifications and provides a powerful primitive with which we can build efficient, reliable software.

In addition to constraints on ordering, file systems have generally had to contend with the lack of a simple but elusive primitive: failure atomicity, or atomicity of writes to persistent storage with respect to power failures. As with the problem of ordering, existing systems are designed for volatile memory only; there are plenty of mechanisms for enforcing atomicity with respect to other threads or cores, but none for enforcing atomicity with respect to power failures. Thus, if a write to persistent memory is interrupted by a power

failure, the memory could be left in an intermediate state, violating consistency. Some journaling file systems use checksums on transaction records to achieve atomicity [17]; however, with BPRAM, we can provide a simple atomic write primitive directly in hardware. As we will discuss later, implementing failure atomicity requires having as little as 300 *nanojoules* of energy available in a capacitor [9]. Note that unless specified otherwise, references to atomicity in this paper will refer specifically to failure atomicity.

In our experience, this approach to atomicity and ordering is a useful division of labor between software and hardware. In the case of atomicity, the hardware implementation is extremely simple, and it dramatically simplifies the task of enforcing consistency in BPFS. For ordering, epoch barriers allow software to declare ordering constraints at a natural level of abstraction, and this information is sufficient for the hardware to cache writes to persistent data. Indeed, we believe that these primitives will find uses in many more software applications beyond BPFS.

2.3 Use Short-Circuit Shadow Paging

Most storage systems ensure reliability by using one of two techniques: write-ahead logging or shadow paging [13]. With write-ahead logging (or journaling) [6], the storage system writes the operations it intends to perform to a separate location (often as a sequential file) before updating the primary storage location. Thus, many writes are completed twice: once to the log, and once to the final location. The benefit of this approach is that the first write to the log is completed quickly, without overwriting old data. However, the cost is that many writes must be performed twice. In fact, the cost of using this technique for all file system data is so large that most file systems journal only metadata by default.

In contrast, shadow paging [7, 23] uses copy-on-write to perform all updates, so that the original data is untouched while the updated data is written to persistent storage. Data is typically stored in a tree, and when new data is written via copy-on-write, parent blocks must be updated via copy-on-write as well. When updates have propagated to the top of the tree, a single write to the root of the tree commits all updates to "live" storage. Unfortunately, the "bubbling-up" of data to the root of the tree incurs significant copying overhead; therefore, updates are often committed infrequently and in batches in order to amortize the cost of copies.

In summary, many reliable storage systems have used one of two techniques: quick updates to a log, with the caveat that many writes are completed twice, or copy-on-write updates that must be batched together in order to amortize their cost. Disk-based file systems have typically favored logging over shadow paging, since the costs of shadow paging's copies outweigh the costs of logging.

In the case of BPRAM, though, byte-addressability and fast, random writes make shadow paging an attractive approach for file system design. In fact, BPFS goes beyond traditional shadow paging systems by implementing a new technique that we call *short-circuit shadow paging* (SCSP). SCSP allows BPFS to commit updates at *any* location in the file system tree, avoiding the overhead of propagating copies to the root of the file system. BPFS can often perform small updates in place, without performing any copies at all, and even when copies are necessary for larger writes, they can be restricted to a small subtree of the file system, copying only those portions of the old data that will not be changed by the update. SCSP is made possible by the availability of atomic writes in hardware, and it is made fast by exploiting our epoch-aware CPU caches.

3. BPFS DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of BPFS, a new file system for BPRAM. BPFS is designed to take advantage of the unique features of BPRAM in order to achieve both high performance and strong safety and consistency guarantees. Specifically, BPFS guarantees that all system calls are reflected to BPRAM atomically and in program order. It also guarantees consistency of the file system image in BPRAM, and it allows data to be made durable as soon as the cache's contents are flushed to persistent storage.

In BPFS, all file system data and metadata is stored in a tree structure in persistent memory. Consistency is enforced using short-circuit shadow paging, which means that updates are committed either in-place or using a localized copy-on-write. In either case, updates are committed to the file system by performing an atomic write at an appropriate point in the tree. We also use the ordering primitives provided by our hardware by marking epoch boundaries before and after each atomic "commit" of file system state, thus ensuring that the committing operation will be written to BPRAM only after the write operations upon which it depends have been made persistent.

3.1 File System Layout

BPFS's persistent data structures are organized into a simple tree of fixed-size blocks. Although it is possible to store more complex data structures in BPRAM (e.g., variable block sizes or multiple pointers to a given piece of data), this approach has two important advantages. First, because there is only one path from the root to any given node, we can update an arbitrary portion of the tree (even multiple files or directories) with a single atomic pointer write; this mechanism is the key to enforcing strong consistency guarantees in BPFS. Second, because all blocks in this tree are of the same size, allocation and deallocation are simple.

BPFS's data structures, which are inspired by WAFL [7], consist of three kinds of files, each of which is represented by the same tree data structure. The *inode file* is a single file containing an array of fixed-size inodes, each uniquely representing a file or directory in the file system. The root of the inode file represents the root of the file system as a whole, and this root pointer is stored in a well-known location in persistent memory. Inodes contain file metadata including the root pointer and size of the associated file. An entry in the inode file is only considered valid if it is referred to by a valid directory entry. *Directory files* contain an array of directory entries that consist of an inumber (i.e., the index of an inode in the inode file) and the name of the corresponding file. Directory entries are only considered valid if they contain a non-zero inumber. *Data files* contain user data only.

The overall structure of the file system is shown in Figure 1. The top half of the file system is the inode file, and the dashed box shows the "data" for this file, which consists of an array of inodes. Each inode points to a directory file or a data file; Figure 1 shows three such files, whose data is also stored in a tree structure. Other files are omitted from this figure for clarity.

Each of our three kinds of files (i.e., data files, directory files, and the inode file) is represented by the same basic data structure: a tree consisting entirely of 4 KB blocks. The leaves of the tree represent the file's data (i.e., user data, directory entries, or inodes), and the interior nodes of each tree contain 512 64-bit pointers to the next level of the tree. In Figure 1, the leaves of each file are shown in a dashed box; taken in sequence, the blocks in this dashed box represent the file's contents. (This figure shows only two pointers per block for simplicity.) Each file has a root pointer and a file size stored in an inode or, in the case of the inode file, in a well-known location. Since this data structure is the same for all kinds of files, the remainder of this section will discuss the features of this data structure in general, for any kind of file.

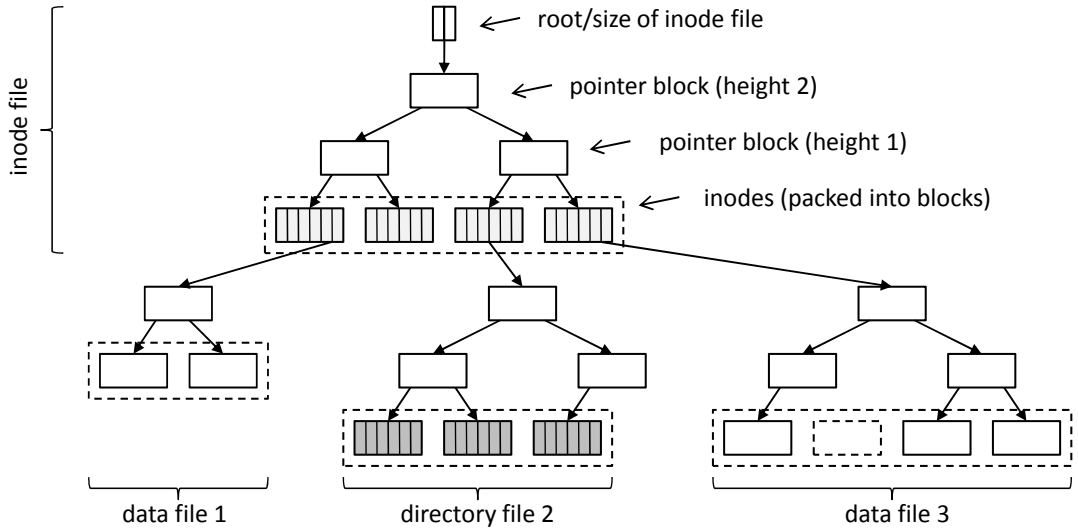


Figure 1: Sample BPFS file system. The root of the file system is an inode file, which contains inodes that point to directory files and data files. Pointer blocks are shown with two pointers but in reality contain 512.

The height of each tree data structure is indicated by the low-order bits of the tree's root pointer, which allows BPFS to determine whether a given block is an interior (pointer) block or a leaf (data) block by remembering the number of hops taken from the root pointer. For example, with a tree of height 0, the root pointer points directly to a data block, which can contain up to 4 KB of file data. With a tree of height 1, the root pointer points to an interior block of 512 pointers, each of which points to a 4 KB data block, for a total of 2 MB. A tree of height 3 can store 1 GB of data, and a tree of height 5 can store 256 TB of data. Note that any given tree is of uniform height: if a tree has height 3, then *all* file data will be found three hops down from the root pointer; no file data is stored at interior nodes. Also, because the root pointer and its height are stored in one 64-bit value, they can be updated atomically.

In order to simplify the task of writing data to the middle of a file, we use a null pointer at any level of the tree to represent zero data for the entire range of the file spanned by that pointer. For example, if a file's root pointer is a null pointer with height 5, then it represents an empty (i.e., zeroed) 256 TB file. Null pointers can also appear at interior nodes, so a write to the end of this 256 TB file will not cause us to write 256 TB of zeros; rather, it will result in a chain of 5 pointers down to a single data block, with null pointers in the remainder of the interior nodes. Thus, this file representation can achieve compact representations of large, sparse files.

Figure 1 shows several examples. First, we have trees of varying height: data file 1 has height 1, and the other files have height 2. Second, all data blocks are at the same level of each tree; for example, in directory file 2, the third data block is still located 3 hops from the directory file's root, even though its parent only has one pointer. Finally, data file 3 is missing its second block due to a null pointer in the parent; this block is assumed to be entirely zero.

We also store the size of each file along with each root pointer. For the inode file, this file size is stored in a well-known location; for all other files, it is stored in the file's inode next to the file's root pointer. The file size can be either larger or smaller than the amount of data represented by the tree itself. If it is larger, then the tail of the file is assumed to be zero, and if it is smaller, then any data in the tree beyond the end of the file is ignored and may contain garbage. For example, if we have a tree of height 1 (with a maximum of 2 MB) and a file size of 1 MB, then the first 256 pointers of the interior node point to valid data, and the last 256 pointers are ignored and may contain arbitrary bits. This design allows us to change the file size without updating the tree itself, and it allows in-place appends, as discussed in the following section.

3.2 Persistent Data Updates

Short-circuit shadow paging consists of three distinct methods for updating persistent data: in-place updates, in-place appends, and partial copy-on-write. All three techniques use BPRAM's byte-addressability as well as our atomic write primitive to provide fast updates to file system data while still preserving consistency. This approach represents a significant departure from existing disk-based file systems, which do not have the luxury of performing fine-grained writes to persistent storage.

Our three approaches are illustrated in Figure 2, which depicts a single file with four data blocks. The root of the file and the file size are stored in an inode block shown at the top of each figure, which is assumed to be part of a larger file system. The dashed vertical line indicates the end of the file according to the file size variable.

In-place updates are the most efficient approach available. For data files, in-place updates can only be performed for writes of 64 bits or less, since our hardware guarantees that these updates are atomic. For example, in Figure 2(a), we have updated 64 bits of file data by writing directly to the data block itself. For metadata files (i.e., directory files or the inode file), we can often use file system invariants to do in-place updates even when more than 64 bits need to be updated. For example, when adding an entry to a directory file, we find an unoccupied (i.e., zeroed) directory entry, write the name of the new entry, and then write its inumber. Since an entry is only considered valid when it contains a non-zero inumber, this final write commits the change to the file system. Similarly, inodes that do not have a directory entry pointing to them are not considered valid inodes, and thus they can be modified in-place.

In-place appends take advantage of the file size variable that accompanies the root pointer for each file. Since all data beyond the file size is ignored, we can safely write to these locations in-place, and once all of the data has been written, we can atomically update the file size to extend the valid data range. Figure 2(b) shows an

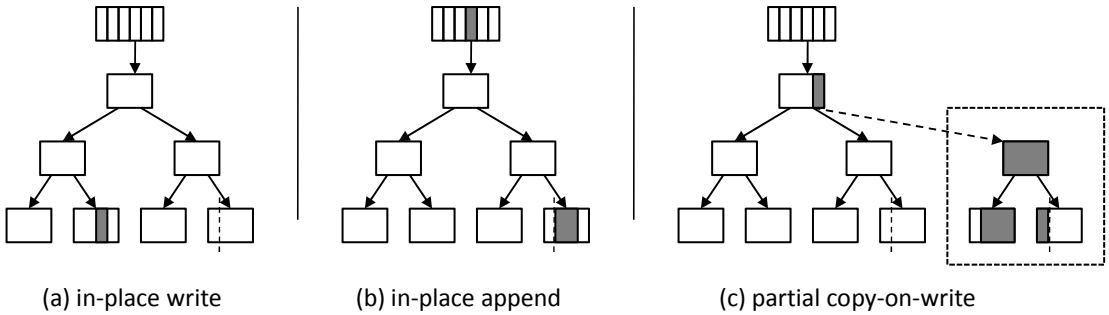


Figure 2: Three approaches to updating a BPFS file. Gray boxes indicate the portions of the data structures that have been updated.

in-place append: we first write data beyond the end of the file, and then we update the file size. If a crash occurs before the file size is updated, any incomplete appends will be ignored.

Partial copy-on-write is the most general technique for updating persistent data, allowing us to perform an atomic update to an arbitrarily large portion of the file system. In this approach, we perform a copy-on-write on all portions of the tree that will be affected by this operation, up to the lowest point at which a change can be committed with a single write. For example, in Figure 2(c), the user wants to write data that spans both the third and fourth data blocks of the file. To do so, we allocate new space in BPRAM, copy any existing data that we do not plan to overwrite (e.g., the beginning of the third data block), and then update these new blocks as appropriate. We also copy and update any pointer blocks that cannot be updated atomically. Only when the updates are complete do we commit this change by performing an atomic update of the pointer to this portion of the file tree.

In practice, these copy-on-write operations are quite efficient. One reason is that we copy only those portions of the data that will not be overwritten; for example, in Figure 2(c), we need not perform any copies on the new version of the fourth data block, since the beginning of the block will be overwritten with new data and the end of the block is beyond the end of the file. Also, we do not copy the *entire* tree below the commit point; rather, we copy only the blocks on each *path* from the commit point down to the modified blocks. It is perfectly valid to have new pointer blocks point to old blocks that are unaffected by the change.

This copy-on-write technique allows us to make atomic modifications to any range of data and is therefore extremely general. In our implementation, it is most often used for large writes to data files or for complex metadata operations. Copy-on-write operations can even be propagated from data or directory files all the way up through the inode file; for example, in a cross-directory move operation, we can use this technique to atomically update both the source and target directories. Nevertheless, most operations tend to be committed locally, within a single inode; currently, cross-directory moves are the *only* operations that can propagate as high as the root pointer of the file system. In our experiments (described in detail in Section 5), most tests only resulted in one or two updates to the file system root out of hundreds of thousands of operations; the one test that used move operations heavily (Patch) updated the file system root on only 10% of file system updates.

With all of these operations, we must issue epoch barriers before and after the atomic write that commits the operation. These barriers ensure that all previous writes will be flushed to BPRAM before the commit occurs and that any subsequent file system operations will take place after the commit.

3.3 Volatile Data Structures

Our file system layout allows efficient and reliable updates to persistent state, but it does not allow us to store complex data structures such as hash tables in persistent memory. Since efficient data structures can improve performance significantly, we maintain some derived data structures in volatile memory. In general, we found this approach to be quite useful: we store simple, non-redundant data structures in persistent memory, and then we cache this data in more efficient volatile data structures where necessary for performance. In this section, we will discuss a number of these volatile data structures.

First, we store in DRAM a list of free BPRAM blocks as well as a list of freed and allocated inumbers. These data structures are initialized from file system metadata at every boot; however, because this scan is performed on BPRAM and not disk, it can be done in a fraction of a second, even on a moderately full file system. We avoid storing these data structures in BPRAM because it would be difficult to commit small changes to the file system while atomically updating these global lists. Note that this initialization procedure differs from a traditional file system checker such as `fsck`, since the purpose of this procedure is to load metadata, not to check and repair file system consistency.

The second volatile data structure is a list of freed and allocated blocks from an in-flight copy-on-write operation. For example, while performing a write, we will keep track of any newly-allocated blocks as well as any blocks that will need to be freed if the operation succeeds. When the operation is complete, we iterate over either the freed list or the allocated list (depending on the success of the operation) and add these blocks to the global free list. Because commits are atomic, this data never needs to be stored in persistent memory or reconstructed.

The third data structure kept in DRAM stores a cache of directory entries from each directory opened by the user. (In Windows, this task is the responsibility of each individual file system.) Each directory entry in the cache is stored simultaneously in a list and in a hash table so that we can support quick, ordered directory listings as well as quick individual name lookups. Any updates to directories are immediately reflected to persistent memory as well.

Since these data structures are only found in volatile memory, we need not use atomic writes to update them; rather, they are synchronized with respect to file system updates using only conventional locks. Conversely, note that BPFS's atomic operations do not obviate the need for conventional locking; for example, without locks, if thread A starts a large copy-on-write operation and then thread B performs an 8-byte write to one of the old pages, that write may be lost when thread A commits, even if the ranges of the two writes do not overlap.

3.4 File System Operations

Next, we discuss the low-level details of our file system implementation. We start with a general framework for applying changes to our tree data structures, and then we discuss specific file system operations.

Since all BPFS file types use the BPFS tree data structure, our implementation has a core set of routines, called the *crawler*, that can traverse these trees and perform reads and writes to any of the three kinds of files (i.e., data files, directory files, and the inode file). To implement a file system operation, the crawler is given a root pointer (for any kind of file), the height of the tree, a range of file offsets, and a callback function. Because we can easily compute the file offsets spanned by each pointer, the crawler needs to visit only the pointers included in the specified range of offsets. Once the crawler gets to the leaf nodes, it will invoke the callback with the appropriate addresses.

The crawler is responsible for updating the tree height and any internal pointers. To update the tree height, the crawler looks to see if the requested file offsets are beyond the offsets spanned by the current file tree. If so, it increases the height of the tree by an appropriate amount. An increase in the height of the tree is a simple operation: the crawler allocates a new pointer block, sets the first pointer in this block to the old tree, and then sets the root pointer of this tree to point to this new block (along with the new height, encoded as low-order bits), repeating as necessary. These updates can all be performed atomically, independent of the write operation that is about to be performed.

At leaf nodes, the crawler invokes a callback, and if the callback wishes to perform a copy-on-write operation, it will allocate a new block, perform any necessary updates, and return the pointer to that new block. The crawler must then update any internal nodes (i.e., pointer blocks) as appropriate. If no modifications are made by the callbacks, the crawler returns the existing pointer block untouched. If only one pointer is modified by the callbacks, then the crawler commits that operation in-place. If more than one pointer is modified, the crawler makes a complete copy of that pointer block, deferring the commit to a higher level in the tree.

Sometimes only copy-on-write is allowed. For example, when a write operation proceeds down two branches of the tree, neither branch is allowed to commit in-place, since any commits need to happen at a common ancestor. This case also arises when the user performs a write that will update existing data *and* extend the end of the file. Since we need to update both the file size and the root pointer of the file atomically, we need to perform a copy-on-write on the inode itself, and we need to disallow in-place commits during the file write.

Since BPFS has two levels of tree data structures (i.e., the inode file and everything below it), many operations invoke the crawler twice: once to find an inode in the inode file, and a second time to perform some operation on that inode. The callback for the top-level crawl invokes the crawler a second time for the bottom-level file. Copy-on-write operations can be propagated upward through both invocations of the crawler.

Now we will discuss individual file system operations. BPFS is implemented in the Windows Driver Model, but here we present a simplified view of these operations.

Open. When a file is opened, BPFS parses the path and uses the directory entry cache to look up the target file or directory. Because the directory entry cache stores complete directory information in volatile memory, this operation only needs to read the full directory if it is being opened for the first time.

If the file does not exist and a create is requested, we claim a new inumber from the free list and then write a new inode to the inode file at the appropriate offset. Because inodes are invalid unless referenced by a directory entry, these updates can be performed in-place. Once the inode is ready, we write a new directory entry into the directory containing the file. Once again, this update can be done in-place, because the directory entry is not valid until a non-zero inumber is written to the appropriate field. Finally, we update the directory entry cache in volatile memory.

Note that this entire operation can effectively be performed with in-place updates to metadata; thus, file creation is consistent, synchronous, and extremely fast. (A few extra writes are required when the inode file or directory file must be extended, but appends are also cheap.)

Read. When a file is read, BPFS invokes the crawler on the appropriate range of the file. The read callback copies data from the data block into a user-supplied buffer, and then the access time is updated with an in-place atomic write. Note that data is never buffered in DRAM; it is copied directly from BPRAM into the user's address space.

When a directory is read, BPFS first loads the directory into the directory entry cache (if it is not already cached) by invoking the crawler on the entire range of the directory's data. Then it searches the cache for the requested name, looks up all relevant inodes from persistent memory, and completes the request.

Write. When a file is written, we may need to perform a copy-on-write of the inode itself, so this operation requires a two-level crawl. The top level crawl operates on the inode file and locates the target file's inode. Then we invoke the write crawler on the appropriate range of this file. The write callback determines whether an in-place write is possible, and if so, it performs that write. If not, it makes a copy of the block, updates the copy, and returns it to the crawler. The crawler then updates the internal nodes of the file tree using the logic described above.

We atomically update either the file's size or the file's root pointer within the inode as necessary. If both must be updated, then we perform a copy-on-write on the inode block itself and return the new version to the inode file crawler to be committed higher up in the tree. For efficiency, we update the file modification time separately; if we required atomicity here, it would force a copy-on-write on every write operation.

Close. When a file or directory is closed, BPFS checks to see whether the file or directory has been marked for deletion (via a separate call), and if so, we delete it by crawling the directory file to the location of the directory entry and writing a zero to the inumber field in-place. Because a zero inumber indicates an invalid directory entry, this atomic write instantly invalidates the directory entry and the target inode. Finally, we update our volatile data structures, including the free block list and the free inumber list.

This implementation exhibits many of the benefits of short-circuit shadow paging. Through careful use of byte-level accesses, we can perform in-place updates for a large number of operations, and through use of an atomic write, we can provide strong consistency and safety guarantees for arbitrarily large changes to the file system. On a disk-based file system, which requires block-based updates, we would not be able to achieve the same combination of high performance and strong guarantees.

3.5 Multiprocessor Operation

BPFS guarantees that updates are committed to BPRAM in program order. On a uniprocessor system, epoch barriers enforce this guarantee by flushing epochs out of the cache subsystem in the order in which they were created. However, on a multiprocessor system, we must consider cases where multiple CPUs contain uncommitted epochs.

Normally, our hardware modifications ensure that if two epochs that share state are issued on two different CPUs, then the epochs will be serialized. However, since BPFS was designed to support independent, in-place updates in the file system tree, if a process or thread updates two *different* pieces of state while executing on two different CPUs, then the updates could be written back to PCM in any order. There are three cases that must be considered to implement these types of updates correctly.

First, a thread could be scheduled on multiple CPUs during a single file system operation. If a thread is preempted within a system call, BPFS must ensure that it is rescheduled on the same CPU, which guarantees that all epochs generated within a system call are committed in the order in which they were generated.

Second, a thread could be switched to a new CPU between two different file system operations. To provide program-order consistency guarantees, these operations must be committed to PCM in the order they were issued. To do so, BPFS tracks the last BPRAM location written by each thread (i.e., the last commit point) in a volatile data structure. When a thread executes a system call that mutates file system state, BPFS reads from the saved BPRAM location. This read creates a dependency that causes the old CPU's data to be flushed, thus guaranteeing that the updates will be committed in program order.

Third, two processes may update two different locations in the file system from two different CPUs. Traditional journaling or shadow paging file systems guarantee that such operations are committed in temporal order by creating a single, total ordering among operations. However, BPFS does not create a total ordering of updates; instead, it allows concurrent, in-place updates to different portions of the file system tree. As a consequence, if two processes execute sequentially on two different CPUs, then their updates may be committed to PCM in any order, since there is no dependency within the file system that forces serialization. If a total ordering is required, then an explicit `sync` must be used to flush dirty data from the CPU's cache.

The current implementation of BPFS does not yet enforce the first two constraints. However, the overhead amounts to a single 8-byte read on each file system update, which we account for in our analytical evaluation. In addition, the correctness of our DRAM-based evaluation is unaffected, since these constraints are only relevant during a power failure.

Another subtlety of multiprocessor systems involves the use of volatile data structures to cache data. If persistent data is cached in volatile memory, then two threads accessing the volatile cache might not generate accesses to common locations in BPRAM; as a result, the hardware will be unaware of the dependency. Thus, when volatile data structures are used, BPFS must "touch" a single word of the corresponding data structure in BPRAM to ensure that all ordering constraints are tracked in hardware. For most volatile data structures, BPFS already reads or writes a corresponding persistent location; however, for the directory entry cache, we add an extra write specifically for this purpose.

3.6 Limitations

One limitation of BPFS is that write times are not updated atomically with respect to the write itself, because our technique would require all write operations to be propagated up to the inode itself using copy-on-write. Although splitting these operations into two separate atomic operations (one to update the file data and one to update the write time) is less than ideal, we consider it a reasonable trade-off in the name of performance. If this trade-off is deemed unacceptable, then this problem could be addressed by implementing a wider atomic write primitive or by squeezing the modification time and the file's root pointer into a single 64-bit value. Also, note that conventional locking still ensures atomicity with respect to other threads in the system; we only sacrifice atomicity with respect to power failures.

Another limitation with respect to journaling is that atomic operations that span a large portion of the tree can require a significant amount of extra copies compared to the journaled equivalent. The primary example of this limitation is the move operation, which can span a large portion of the tree in order to update two directory entries. Fortunately, BPFS handles the most common file system operations with relatively little copying.

Our current prototype does not yet support memory-mapped files. However, we believe that it would be straightforward to support this feature by either mapping data into DRAM and occasionally flushing it out to PCM (as is done by current disk-based file systems), or by mapping PCM pages directly into an application's address space. In the latter case, atomicity and ordering guarantees would not be provided when writing to the memory-mapped file, but the file's data could be accessed without a trap to the kernel. Wear leveling will be required to ensure that malicious programs cannot burn out the PCM device; we will discuss these issues further in the next section.

A final limitation is the overall interface to BPRAM. Rather than implementing a new file system, we could offer the programmer a fully persistent heap. However, this approach has the disadvantage of requiring significant changes to applications, whereas BPFS allows existing programs to reap the benefits of BPRAM immediately. In addition, the file system interface provides a well-known abstraction for separating persistent data from non-persistent data, and it allows the file system to enforce consistency in a straightforward manner. Although our current design provides a balance between performance, reliability, and backward-compatibility, we nevertheless believe that persistence within the user-level heap will be a fruitful area of future research.

4. HARDWARE SUPPORT

In this section, we will discuss the hardware support required to provide non-volatile storage along with atomic 8-byte writes and epoch barriers. First, we will discuss the details of phase change memory, which is currently the most promising form of BPRAM. Second, we will discuss wear leveling and write failures for PCM. Third, we will show how we enforce atomicity, and finally, we will show how we can modify the cache controller and the memory controller to enforce ordering constraints.

4.1 Phase Change Memory

Phase change memory, or PCM, is a new memory technology that is both non-volatile and byte-addressable; in addition, PCM provides these features at speeds within an order of magnitude of DRAM [1, 9]. Unlike DRAM, however, PCM stores data by using resistivity instead of electrical charge. It is made from a chalcogenide glass, a material that can be switched between two "phases", crystalline and amorphous, by heating it to 650°C and then cooling it either slowly or rapidly. These phases have different resistivity, which is used to represent a 0 or a 1.

PCM cells can be organized into an array structure much like that of DRAM [3]. Thus, it is possible to manufacture a PCM DIMM that operates in much the same way as an existing DRAM DIMM, albeit with different timing parameters and access scheduling constraints [9]. At a minimum, memory controllers could support PCM DIMMs by modifying the timing parameters of an existing DDR interface; however, the additional modifications proposed in this paper will allow us to build reliable software on top of non-volatile

system memory while also making use of the CPU's caches. For this paper, we assume that the PCM-based storage system is organized as a set of PCM chips placed in DDR-compatible DIMMs.

One limitation of this approach is that capacity will be restricted by the density of the chips residing on a DIMM. For example, a 2008 Samsung prototype PCM chip holds 512 Mb [9], so with 16 chips on a high-capacity DIMM, we could reach a capacity of 1 GB per DIMM right now. Combined with process technology and efficiencies from manufacturing at volume, which will further improve density and capacity, we expect to have enough capacity to provide a useful storage medium in the near future. If additional capacity is required, we can place larger quantities of PCM (hundreds of gigabytes) on the PCI Express bus in addition to the PCM on the memory bus; however, for the purposes of this paper, we assume that all PCM is accessible via the memory bus.

4.2 Wear Leveling and Write Failures

Although PCM has much higher write endurance than NAND flash, it will still wear out after a large number of writes to a single cell. The industry consensus as of 2007 is that PCM cells will be capable of enduring at least 10^8 writes in 2009 and up to 10^{12} by 2012 [1]. Even though these endurance figures are high compared to other non-volatile memories, placing PCM on the memory bus instead of an I/O bus may expose the cells to greater write activity and thus require *wear leveling*, which is a process that distributes writes evenly across the device to reduce wear on any single location. Although our file system does not specifically concentrate updates on one location (recall that most updates are committed locally and not at the file system root), there is the potential for some workloads to result in "hot" locations.

Fortunately, there are several approaches to wear leveling that can operate independently of our file system. First, we can design PCM arrays in ways that minimize writes, extending device lifetime from 525 hours to 49,000 hours (5.6 years) [9]. Second, several mechanisms have been proposed for applying wear leveling to PCM [18, 27]. In short, effective wear leveling can be implemented by using two techniques: within each page, wear is evened out by rotating bits at the level of the memory controller, and between pages, wear is evened out by periodically swapping virtual-to-physical page mappings. By choosing these shifts and swaps randomly, additional defense against malicious code can be provided. This work shows that it is possible to design reasonable wear-leveling techniques that are independent of BPFS.

When eventual failures occur, we expect to detect them using error-correcting codes implemented in hardware. For example, we can take advantage of existing error-correcting codes used for flash [11]. When PCM pages degrade beyond the ability to correct errors in hardware, the operating system can retire PCM pages, copying the data to a new physical page and then updating the page table. Of course, data can still be lost if sufficiently many bits fail; however, in this paper, we assume that PCM hardware will be designed with enough redundancy to make such failures negligibly rare.

4.3 Enforcing Atomicity

In order to enforce atomicity for 8-byte writes, we must ensure that in the case of a power failure, a write either completes entirely, with all bits updated appropriately, or fails entirely, with all bits in their original state.

We propose enforcing atomicity by augmenting DIMMs with a capacitor holding enough energy to complete the maximum number of write transactions ongoing within the PCM subsystem. Since all writes are stored temporarily in volatile row buffers on each DIMM before being written to PCM, having a capacitor on each DIMM ensures that all writes residing in the row buffers are completed. Although the memory controller fails to issue further commands, any in-progress writes will be guaranteed to complete, so that no 64-bit word is left in an intermediate state.

Note that unrecoverable bit failures can occur while performing the final writes during a power failure. As above, we assume that PCM devices provide enough redundancy to make such failures extremely unlikely. If additional reliability is required, the memory controller can be modified to write all in-flight writes to a backup location as well as to the primary location in the event of a power failure. This approach increases the chances of successful completion at the expense of additional capacitance.

The amount of power required to complete all in-flight writes is quite small, even for a mobile device. To write a logical zero, a PCM bit requires a current ramp down from 150 µA to 0 µA over 150 ns requiring 93.5 nF at 1.2 V. Similarly, to write a logical one, a PCM bit requires 300 µA over 40 ns, requiring 75 nF at 1.6 V. Assuming PCM row widths of 512 bits (one cache line), the total capacitance required would vary between 38.4 and 47.8 µF. To maintain stable power, the capacitor would need to be somewhat larger, with circuitry to provide a transient but stable output voltage as the capacitor discharges. On-chip decoupling capacitors can provide part of this charge; the total decoupling capacitance on the Alpha 21264 was 320 nF and the Pentium II contained 180 nF [16]. Discrete capacitive elements on the memory module can easily provide several µF of supplemental charge [22].

If desired, larger units of atomicity could be provided by integrating additional capacitors at the board level. We propose 64 bits because a single atomic pointer update can be used as a primitive in order to update even larger quantities of data, as shown in BPFS.

4.4 Enforcing Ordering

Modern caches and memory controllers can reorder writes on their way from the CPU to memory. For example, if a CPU writes address A and then address B, and both updates are stored in a write-back cache, then the new data at address B may be written back to main memory before the data at address A. Similar effects may also occur in memory controllers, where volatile buffers on each DIMM may be written back to primary storage in an arbitrary order. In fact, recent industrial memory controllers can explicitly reorder in-flight writes to improve locality and parallelism.

With volatile DRAM, these ordering issues are irrelevant. Cache coherence protocols and memory barriers such as the x86 `mfence` instruction ensure that all CPUs have a consistent global view of the state of memory, and as long as that consistent view is maintained, it does not matter when or in what order data is actually written back to DRAM. Indeed, these mechanisms do not currently enforce any such ordering. For example, if writes A and B are separated by an `mfence`, the `mfence` only guarantees that A will be written to the cache and made visible to all other CPUs via cache coherence before B is written to the cache; it does not ensure that write A will be written back to DRAM before write B.

With BPRAM in place of DRAM, though, the order in which writebacks occur is now important. For example, consider the sequence of operations for updating a 4 KB file in BPFS. First, a new 4 KB buffer is allocated in BPRAM, and the new data is written to that buffer. Then, a pointer in the file system tree is updated to point to that new buffer. At this point, all of this data is likely to be resident in the L1 or L2 cache but has not been written back to BPRAM. If the cache controller chooses to write back the pointer update before it writes back the 4 KB buffer, the file system in BPRAM will be inconsistent. This inconsistency is not visible to any currently-executing code, since existing cache coherence and

memory barrier mechanisms ensure that all CPUs see the updates in the correct order; however, if a power failure occurs before all data is written back to BPRAM, the file system will be inconsistent when the machine is rebooted. Thus, in order to ensure that the file system in persistent memory is always consistent, we must respect any ordering constraints when data is written to persistent memory.

There are a number of choices for enforcing ordering. One possibility is to use write-through caching (or to disable the cache entirely); unfortunately, doing so would be extremely slow. A second possibility is that we could flush the entire cache at each memory barrier in order to ensure that all data arrives in non-volatile memory in the correct order. However, flushing the cache is also quite costly in terms of performance and would have the side-effect of evicting the working sets of any other applications sharing the cache. A third possibility is that we could track all cache lines that have been modified during an operation so that we can flush only those lines that contain dirty file system data. This approach requires a large amount of software bookkeeping; not only is this bookkeeping expensive, but it also represents a poor division of labor between software and hardware, since the software must do a large amount of work to compensate for the deficiencies of what should ideally be a transparent caching mechanism.

In this paper, we propose a fourth alternative: allow software to explicitly communicate ordering constraints to hardware. By doing so, we once again allow hardware to transparently cache reads and writes to persistent data, but we ensure that the necessary ordering constraints are respected.

The mechanism we propose for enforcing ordering is called an *epoch barrier*. An epoch is a sequence of writes to persistent memory from the same thread, delimited by a new form of memory barrier issued by software. An epoch that contains dirty data that is not yet reflected to BPRAM is an *in-flight* epoch; an in-flight epoch *commits* when all of the dirty data written during that epoch is successfully written back to persistent storage. The key invariant is that when a write is issued to persistent storage, all writes from all previous epochs must have already been committed to the persistent storage, including any data cached in volatile buffers on the memory chips themselves. So long as this invariant is maintained, an epoch can remain in-flight within the cache subsystem long after the processor commits the memory barrier that marks the end of that epoch, and multiple epochs can potentially be in flight within the cache subsystem at each point in time. Writes can still be reordered within an epoch, subject to standard reordering constraints.

4.4.1 Epoch Hardware Modifications

Our proposed hardware support includes minor modifications to several parts of the PC architecture. In this section, we discuss how our modifications impact the processor core, the cache, the memory controller, and the non-volatile memory chips. Although hardware modifications are often a daunting proposition, we believe that these modifications represent straightforward extensions to existing hardware that will be essential for many uses of BPRAM.

First, each processor must track the current epoch to maintain ordering among writes. Each processor is extended with an *epoch ID counter* for each hardware context, which is incremented by one each time the processor commits an epoch barrier in that context. Whenever a write is committed to an address located in persistent memory, it is tagged with the value of the current epoch ID, and this information is propagated with the write request throughout the memory system. The epoch ID counter is 64 bits wide, and in a multiprocessor system, the epoch ID space is partitioned among the available hardware contexts (effectively using the top bits as a context ID). Thus, epoch IDs in a shared cache will never conflict.

Next, each cache block is extended with a *persistence bit* and an *epoch ID pointer*. The persistence bit indicates whether or not the cached data references non-volatile memory, and it is set appropriately at the time a cache line is filled, based on the address of the block. The epoch ID pointer indicates the epoch to which this cache line belongs; it points to one of eight hardware tables, which store bookkeeping information for the epochs that are currently in-flight.

We then extend the cache replacement logic so that it respects the ordering constraints indicated by these epoch IDs. The cache controller tracks the oldest in-flight epoch resident in the cache for each hardware context, and it considers any cache lines with data from newer epochs to be ineligible for eviction. In cases where a cache line from a newer epoch *must* be evicted, either because of a direct request or because no other eviction options remain, the cache controller can walk the cache to find older cache entries, evicting them in epoch order. The cache maintains bookkeeping information for each 4 KB block of cache data in order to make it easy to locate cache lines associated with each in-flight epoch.

This cache replacement logic also handles two important corner cases. First, when a processor writes to a single cache line that contains dirty data from a prior epoch, the old epoch must be flushed in its entirety—including any other cache lines that belong to that older epoch. Second, when a processor reads or writes a cache line that has been tagged by a different hardware context, the old cache data must be flushed immediately. This requirement is particularly important during reads in order to ensure that we capture any read-write ordering dependencies between CPUs.

Note that the coherence protocol does not change; existing coherence protocols will work correctly as long as our cache replacement policy is followed.

Finally, the memory controller must also ensure that a write cannot be reflected to PCM before in-flight writes associated with all of the earlier epochs are performed. To enforce this rule, the memory controller records the epoch ID associated with each persistent write in its transaction queue, and it maintains a count of the in-flight writes from each epoch. When each write completes, it decrements this counter, and it does not schedule any writes from the next epoch until the current epoch's counter hits zero.

The overall modifications to the hardware include four changes. First, we add one 64-bit epoch ID counter per hardware context. Second, we extend the cache tags by 4 bits: 3 for the epoch ID pointer and 1 for the persistence bit. Third, we augment the cache with 8 bit-vectors and counter arrays for fast lookup of cache lines in a given epoch (total overhead of 7 KB for a 4 MB L2 cache). Finally, we add capacitors to ensure that in-progress writes complete. The total area overhead for a dual-core system with 32 KB private L1 caches, a shared 4 MB L2 cache, and a maximum of 8 in-flight epochs is approximately 40 KB.

These changes are neither invasive nor prohibitive. First of all, the changes to the memory hierarchy are very simple compared to the complexity of the hierarchy as a whole. Second, the capacitor requires board-level support but applies well-known power supply techniques to ensure a temporary but stable supply voltage in the event of a power loss. Finally, the changes do not harm performance on the critical path for any cache operations. For example, the cache changes affect the replacement logic in the control unit only; they do not affect the macro blocks for the cache.

In addition, we believe that cache and memory controllers are likely to be modified to account for PCM timing constraints independent of these changes. Given that this hardware is likely to change anyway, and given that these changes can potentially support atomicity and ordering for a variety of software applications, these changes do not represent a significant additional overhead.

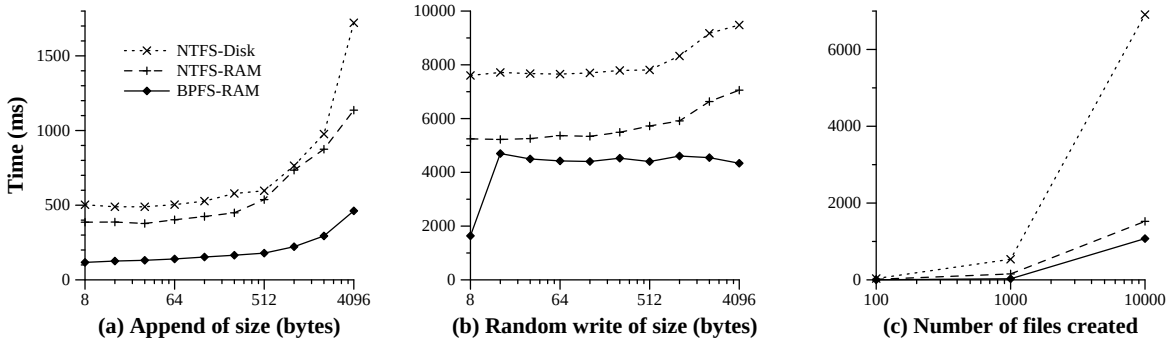


Figure 3: Microbenchmarks showing BPFS performance on appends, random writes, and creates.

5. EVALUATION

Our evaluation answers the following questions:

- Does BPFS perform better than a file system designed for disks?
- What are the performance benefits of our proposed hardware modifications?
- Will BPFS backed by PCM perform better than a system that uses a DRAM buffer cache plus a disk?

5.1 Methodology

Making a meaningful performance comparison between BPFS (on PCM) and NTFS[1] (on disk) presents a methodological challenge. On one hand, the lack of an existing hardware platform with a PCM-based storage system prevents us from evaluating BPFS on real hardware. On the other hand, we cannot simulate NTFS at the microarchitectural level. Instead, we adopt an alternative strategy to compare the two file systems. We first perform a set of tests on real hardware to measure the performance of NTFS backed by disk (with a DRAM buffer cache), NTFS backed solely by DRAM, and BPFS backed by DRAM. We then perform a second set of tests to measure the amount of traffic directed to persistent storage on BPFS by consulting the performance counters on our hardware platform. Next, we evaluate our proposed hardware features using a microarchitectural simulator. Finally, we utilize a simple (yet conservative) analytical model to estimate best-case and worst-case performance for BPFS on PCM and to show how BPFS performance will vary based on sustained PCM throughput. We plug the throughput figures observed in simulation into this model to estimate the common-case performance of BPFS on PCM.

When we run benchmarks on real hardware, we use a dual dual-core (2 chips, 2 cores per chip) 2 GHz AMD Opteron system with 32 GB of RAM running 64-bit Windows Vista SP1. We use an NVIDIA nForce RAID controller with two 250 GB 7200 RPM Seagate Barracuda disks, each with an 8 MB cache, running in a RAID-0 configuration. We measure time using `timeit`, which captures the wall clock time for each benchmark. We run NTFS in two configurations: a standard configuration backed by the striped disks, and a configuration backed by a RAM disk,[2] which is meant to represent an alternative where we simply run an existing disk-based file system in persistent memory. Unless otherwise noted, results are the mean of 5 runs, and error bars represent 90% confidence intervals. BPFS runs by allocating a contiguous portion of RAM within the kernel as its "PCM", and it stores all file system data structures in that segment. All other data structures that would normally reside in DRAM (e.g., the directory cache) are stored through regularly allocated memory within the kernel. We refer to NTFS backed by a disk as NTFS-Disk, and NTFS backed by a RAM disk as NTFS-RAM.

Our microarchitectural simulations use the SESC simulation environment [20] to model a dual-core CMP with 4 MB of L2 cache. This simulator models a 4-issue out-of-order superscalar core and has a detailed model of a DDR2-800 memory system. We augment SESC's DRAM model with a command-level, DDR2-compatible PCM interface, and we modify the cache subsystem to implement epoch-based caching, including all of the sources of overhead mentioned in the previous section. However, SESC is not a full-system simulator and cannot boot a real operating system, so we restrict these experiments to three I/O intensive microbenchmarks that stress different file system characteristics (e.g., data vs. metadata updates, complex operations that require multiple ordering invariants), and the PostMark [8] workload, a popular throughput benchmark for measuring file system performance. We run these simulations with a user-level implementation of BPFS, since the kernel-level implementation requires the rest of Windows to run.

5.2 Experimental Evaluation

In this section, we present an experimental evaluation of BPFS, which runs as a native Windows file system driver, compared to NTFS backed by a 2-disk striped RAID array (RAID 0), and NTFS backed by a RAM disk.

5.2.1 Microbenchmarks

We began our evaluation of BPFS by measuring the performance of three microbenchmarks that stress various aspects of the file system. The results in Figure 3(a) show the performance of BPFS compared to NTFS-Disk and NTFS-RAM when appending to a file 130,000 times. We ran the benchmark while varying the granularity of writes (i.e., the amount of data written per system call) from 8 bytes to 4 KB. BPFS is between 3.3 and 4.3 times faster than NTFS-Disk, and it is 2.5 and 3.3 times faster than NTFS-RAM.

Figure 3(b) shows the result of executing 1 million writes to random offsets of a 128 MB file. We ran multiple versions of the benchmark while varying the size of the writes to the file. We used the same seed to a pseudo-random number generator to maintain consistency of results across runs. BPFS is between 1.6 and 4.7 times faster than NTFS-Disk and between 1.1 and 3.2 times faster than NTFS-RAM. 8-byte writes are particularly fast on BPFS, since they can be done entirely in-place; all other writes in BPFS require a 4 KB copy in order to preserve atomicity.

[1] NTFS is the gold standard for Windows file systems. It is a journaling file system comparable to Linux's `ext3`.

[2] Available from `http://www.ramdisk.tk/`; these experiments used version 5.3.1.10.

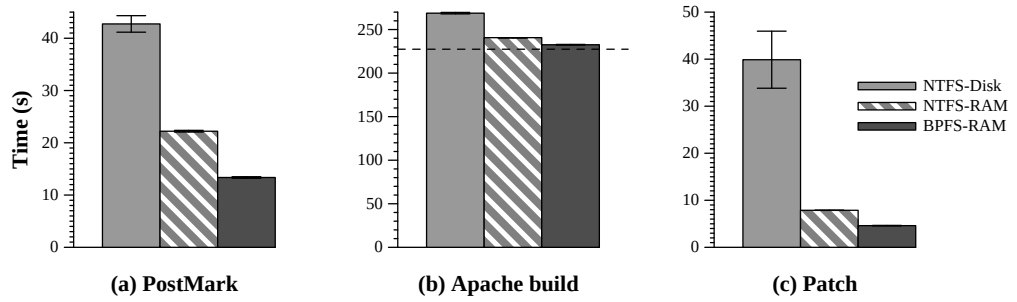


Figure 4: Large benchmark execution times. The dashed line for Apache indicates the approximate amount of time in computation.

Figure 3(c) shows the time to create 100, 1,000, and 10,000 files on BPFS, NTFS-Disk, and NTFS-RAM. BPFS is 6 to 18 times faster than NTFS-Disk. NTFS commits its log to disk during metadata operations such as file creation, and therefore its performance is penalized significantly. However, even when NTFS is running on a RAM disk, BPFS is 5.8 times, 5.4 times, and 1.4 times faster. BPFS shows its biggest advantages on small numbers of operations, but it still outperforms NTFS even for large groups of operations.

5.2.2 Throughput Benchmark

Our next benchmark is similar in spirit to the PostMark [8] file system benchmark, which emulates the workload of a news and mail server. License issues prevented us from using the original PostMark source code, so we wrote our own version of the benchmark, which creates 100 files, executes 5000 transactions on those files consisting of reads and writes, and then deletes all files. This benchmark is a good test of file system throughput, since it has no computation and executes as quickly as possible.

The results are presented in Figure 4(a). The first bar shows the time to execute the benchmark on NTFS-Disk, while the third bar shows the time to execute within BPFS. BPFS is 3.2 times faster than NTFS backed by disk. One reason for this result is that when a file is created, NTFS does a synchronous write to disk to commit a journal operation, whereas BPFS has no such overhead.

The second bar shows the performance of NTFS backed by a RAM disk. BPFS is 1.7 times faster than NTFS backed by a RAM disk when running the file system throughput benchmark, even though it is also providing stronger reliability guarantees.

5.2.3 Apache Build Benchmark

To see how BPFS compares to NTFS on a benchmark that overlaps computation with file system I/O, we created a benchmark that unzips the Apache 2.0.63 source tree, does a complete build, and then removes all files.

Figure 4(b) shows the results. BPFS executes the benchmark 13% more quickly than NTFS backed by a disk, and 3.4% faster than NTFS backed by a RAM disk. This improvement in performance is much lower than the other benchmarks. However, we found that when running the benchmark on BPFS, only 5.1 seconds of time was spent executing operations to the file system; the remainder of the time was compute bound. Figure 4(b) shows a dashed line to indicate the best-case performance of a file system on this benchmark; even if file operations were instantaneous, the maximum speedup over NTFS backed by a RAM disk is 6.5%.

5.2.4 Patch Benchmark

Our last benchmark for BPFS on RAM decompresses the Apache source tree and then runs a patch operation against the tree. The patch script examines each file in the source tree, looks for a certain pattern, replaces it as necessary, and then writes the entire file out to the file system, replacing the old version. This benchmark strikes a balance between the PostMark-like benchmark, which is throughput-bound, and the Apache benchmark, which is compute-bound.

The results can be seen in Figure 4(c). BPFS is 8.7 times faster than NTFS-Disk and is 1.7 times faster than NTFS-RAM. Thus, even when writing to RAM, the design of BPFS outperforms traditional disk-based file systems while still providing strong safety and consistency guarantees.

5.3 Simulation Evaluation

Now that we have explored the performance of BPFS on DRAM, we will present our results when running BPFS on the SESC simulator, in order to determine the costs and benefits of our hardware modifications. First, we compare BPFS running with our hardware modifications to BPFS where all writes to PCM are treated as write-through to ensure correctness. Second, we measure the amount of interference that results from placing PCM alongside DRAM on the memory bus; we want to ensure that traffic to PCM does not hurt the performance of DRAM itself. These simulations take into account all of the overhead of the hardware modifications described in the previous section.

5.3.1 Epoch-Based Caching

Figure 5 compares the performance of our epoch-based cache subsystem to a version that guarantees ordering by treating all file system writes as write-through. The results are normalized to the performance of the write-through scheme. On average, the epoch-based cache subsystem improves performance by 81% over write-through. The speedups are larger than 45% on all applications, with a minimum of 49% for PostMark and a maximum of 180% on the append benchmark. This performance improvement shows that epoch-based caching is capable of using the entire on-chip cache subsystem to coalesce writes to PCM. Moreover, by keeping enough data in the on-chip cache space, applications can quickly process I/O-intensive code regions at processor speeds, lazily performing PCM updates during CPU-intensive phases of execution. This benchmark demonstrates that our epoch-based cache subsystem provides a significant performance gain compared to write-through while preserving the same strong reliability guarantees that write-through provides.

5.3.2 PCM/DRAM Interference

Sharing a memory channel between DRAM and PCM ranks can result in interference when running multiprogrammed workloads. In particular, memory accesses that reference DRAM could get queued up and delayed behind slower PCM operations, significantly degrading the performance of the memory-intensive appli-

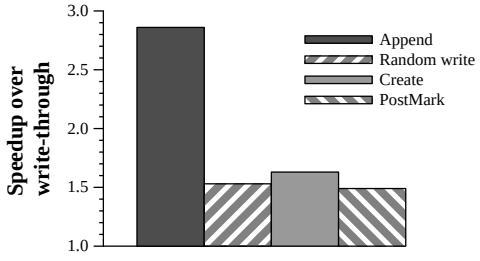


Figure 5: Speedup of epoch-based caching.

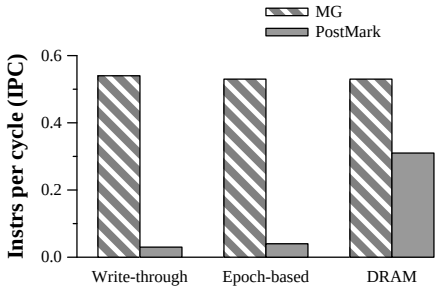


Figure 6: Interference between DRAM and PCM.

cations in the system. Fortunately, straightforward out-of-order command-scheduling policies such as FR-FCFS ("first-ready, first-come, first-served", which is our baseline) typically prioritize ready commands that can be issued in any given clock cycle, allowing DRAM requests to be serviced while PCM updates are in progress. Moreover, although the DDR timing constraints for PCM are considerably more costly than those for DRAM, many of these constraints are restricted to a single rank. Since PCM and DRAM are not interleaved within a single rank but are placed in different ranks, the interference due to timing constraints should be minimal.

To evaluate interference, we ran the PostMark benchmark using BPFS concurrently with the memory-intensive NAS MG benchmark [2] on our simulator. Figure 6 compares the instructions per cycle (IPC) of both benchmarks for various hardware models. Although the PostMark benchmark experiences speedups when moving from write-through to our epoch-based architecture and to a version of BPFS backed by DRAM, the NAS benchmark achieves roughly the same IPC value in all three cases. Thus, the amount of traffic to PCM does not affect the throughput of DRAM-intensive applications.

5.4 Analytical Evaluation

For the final part of our evaluation, we combined the results from our DRAM tests and from our simulations in order to predict the results of our benchmarks if BPFS were run on real PCM.

For each benchmark, we measured the total amount of traffic to memory during our DRAM-based tests. To measure read traffic, we used AMD's hardware performance counters to count the number of L2 data cache misses while running our driver. To measure write traffic, we added counters within our driver to tally the number of cache lines written during the test. To estimate each workload's execution time for a given value of sustained PCM throughput, we add the measured execution time on real hardware to the ratio of the traffic and throughput: Time(PCM) = Time(DRAM) + (Traffic / PCM throughput). We repeat this calculation for different throughput values between the highest (800 MB/s) and lowest (34 MB/s) sustained bandwidth possible with our memory controller under constant load. The ratios and throughput metrics are taken directly from the architectural simulation, and they capture the overhead of our hardware modifications.

These results are conservative in several respects. First, we count all L2 data cache misses in our driver, which includes misses to volatile data that would not be stored in PCM. Second, we assume that the amount of time to transfer this traffic across the off-chip interface would be fully reflected in the application's execution time (i.e., there is no overlapping of computation with memory I/O). And finally, we do not subtract the time spent writing to DRAM during our actual performance tests; thus, our results show the costs of PCM writes *in addition* to DRAM writes.

Figure 7 shows performance projections for BPFS on PCM at different values of sustained PCM throughput. On the Apache and Patch benchmarks, BPFS outperforms NTFS on disk regardless of the PCM channel utilization. The impact of the file system on end-to-end system performance is less significant on Apache as this workload overlaps computation with I/O significantly, and on Patch, BPFS outperforms NTFS by a factor of 4–8 depending on the sustained PCM throughput. On PostMark, BPFS once again outperforms NTFS on disk for all but the lowest values of sustained throughput, values that are observed only with pathological access patterns that exhibit little or no memory-level parallelism and spatial locality. In microarchitectural simulation, the observed throughput for this workload is approximately 480 MB/s, at which point BPFS outperforms NTFS-Disk by a factor of two, as shown by the diamond mark on the BPFS-PCM curve. (We do not show this point on the other graphs because we did not run them in microarchitectural simulation; however, BPFS-PCM performs better than NTFS-Disk at every point in these other tests.) These graphs also show the performance of NTFS-RAM, which is often similar to the expected performance of BPFS-PCM. However, NTFS-RAM is a purely volatile storage system; if we ran NTFS on PCM instead of DRAM, its performance would likely be significantly worse than that of BPFS on PCM.

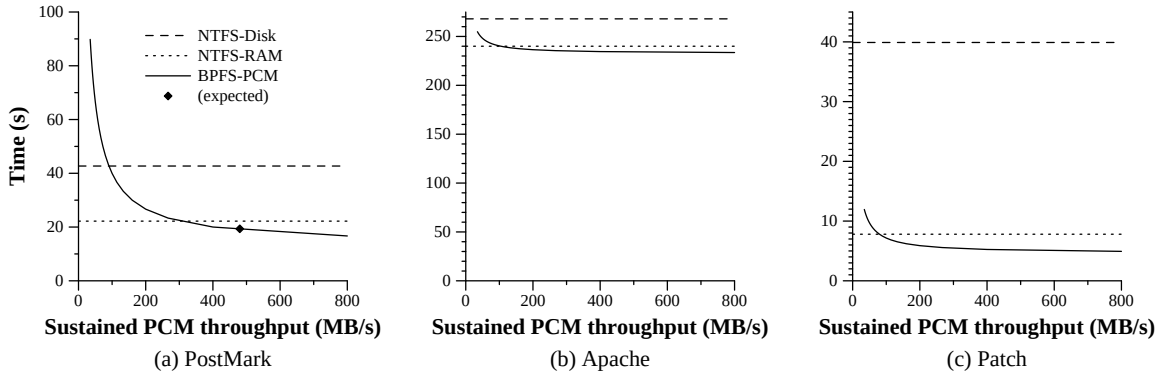
Overall, these conservative results suggest that BPFS on PCM is likely to outperform traditional disk-based file systems by a significant margin, while still providing strong safety and consistency guarantees.

6. RELATED WORK

To the best of our knowledge, BPFS is the first file system to implement short-circuit shadow paging on byte-addressable, persistent memory.

File systems. File systems have long been optimized for their intended medium. The Cedar file system [6] and the Sprite log-structured file system [21] are both classic examples of maximizing the amount of sequential I/O in order to take advantage of the strengths of disk. Likewise, file systems such as the Journaling Flash File System (JFFS) [25] tried to optimize writes into large blocks to lessen the impact of the program/erase cycle present in flash. We have taken a similar approach with BPFS, optimizing the design of the file system for the properties of PCM, most notably making use of fast, small, random writes and no longer buffering file system data or metadata in DRAM. Unlike these other file systems, BPFS takes advantage of new architectural features to provide strong file system correctness guarantees while still benefiting from the availability of PCM on the memory bus. We believe that the benefits of these architectural features are not specific to this file system—in fact, they will be required by any application that wants to provide guarantees based on non-volatile memory.

The file system most similar to BPFS is the WAFL [7] file system. WAFL stores the file system as a copy-on-write tree structure on disk. Whenever changes are reflected to disk, the changes "bub-



(a) PostMark (b) Apache (c) Patch

Figure 7: Projected performance of BPFS-PCM for various levels of sustained throughput.

ble up" to the root of the tree. By changing the root pointer, all of the changes are committed atomically. The authors note that the copy-on-write procedure is quite expensive, and therefore file system changes are kept in a log in NVRAM and are only occasionally reflected to disk. In contrast, BPFS places all data structures directly in BPRAM, and it uses short-circuit shadow paging to efficiently reflect changes to persistent storage individually and atomically. However, BPFS does not use copy-on-write to provide snapshots of previous versions, as WAFL does; since BPFS can perform in-place updates and commit new data at any point in the tree, providing versioning would require significant changes—and perhaps sacrifices in terms of safety, consistency, or performance.

Sun's ZFS [23] also shares a number of common features with BPFS. First, ZFS provides a copy-on-write mechanism similar to the one offered by WAFL. As with WAFL, ZFS batches up file system transactions into groups for efficiency, whereas BPFS can commit each file system operation individually and without copying all the way to the root. However, ZFS uses copy-on-write to perform versioning, which BPFS does not do. Second, ZFS uses checksums to detect file system corruption proactively, whereas BPFS relies on ECC to detect hardware failures.

Consistency and safety. In general, transaction processing systems have focused on using write-ahead logging or shadow paging to ensure the "ACID" properties for transactions [14]. BPFS focuses on shadow paging to maintain these properties, and because we have an atomic write and the ability to update persistent storage at a fine granularity, we can enforce the ACID properties for each file system call (with a few caveats for modification times) using a combination of atomic writes, epochs, and conventional file system locking. Another related area is lock-free data structures, which tend to be built from either a compare-and-swap operation or transactional memory [5]. However, because BPFS uses conventional locking to ensure isolation from concurrent threads, and because it has an atomic write enforced by hardware, neither of these primitives is necessary for the guarantees we want to provide.

BPFS improves safety guarantees by taking advantage of a high-throughput, low latency connection to BPRAM. However, writing to BPRAM is sill slower than writing to DRAM. External synchrony [15] hides most of the costs of synchronous disk I/O by buffering user-visible outputs until all relevant disk writes have been safely committed. We view this work as complementary to our own; as long as non-volatile storage is slower than volatile storage, then external synchrony can be used to hide the costs of synchronous I/O.

Non-volatile memory. Other storage systems have considered the impact of non-volatile memories. eNVy [26] presented a storage system that placed flash memory on the memory bus by using a special controller equipped with a battery-backed SRAM buffer to hide the block-addressable nature of flash. With PCM, we have a memory technology that is naturally suited for the memory bus, and we investigate ways to build more efficient file systems on top of this memory.

More recently, Mogul *et al.* [12] have investigated operating system support for placing either flash or PCM on the memory bus alongside DRAM. They describe several policies that could be used to anticipate future data use patterns and then migrate data between fast DRAM and slow non-volatile memory appropriately.

The Rio file cache [10] took a different approach to non-volatile memory by using battery-backed DRAM to store the buffer cache, eliminating any need to flush dirty data to disk. Rio also uses a simple form of shadow paging to provide atomic metadata writes. In contrast, BPFS does away with the buffer cache entirely, building a file system directly in BPRAM. Whereas Rio provides atomicity only for small metadata updates, BPFS guarantees that arbitrarily large data and metadata updates are committed atomically and in program order.

In the same vein as Rio, the Conquest file system [24] used battery-backed DRAM to store small files and metadata as a way of transitioning from disk to persistent RAM. In contrast, BPFS is designed to store both small and large files in BPRAM, and it uses the properties of BPRAM to achieve strong consistency and safety guarantees. Conquest's approach may be useful in conjunction with BPFS in order to use higher-capacity storage.

In general, battery-backed DRAM (BBDRAM) represents an alternative to using BPRAM. Most of the work described in this paper would also apply to a file system designed for BBDRAM—in particular, we would likely design a similar file system, and we could take advantage of the same hardware features. However, there are two main advantages that BPRAM has over BBDRAM. First, BBDRAM is vulnerable to correlated failures; for example, the UPS battery will often fail either before or along with primary power, leaving no time to copy data out of DRAM. Second, BPRAM density is expected to scale much better that DRAM, making it a better long-term option for persistent storage [19].

Finally, several papers have explored the use of PCM as a scalable replacement for DRAM [9, 18, 27] as well as possible wear-leveling strategies [18, 27]. This work largely ignores the non-volatility aspect of PCM, focusing instead on its ability to scale much better than existing memory technologies such as DRAM or flash. Our work focuses on non-volatility, providing novel software applications and hardware modifications that support non-volatile aspects of BPRAM.

7. CONCLUSION

In this paper, we have presented a design for BPFS, a file system for byte-addressable, persistent memory, as well as a hardware architecture that enforces the required atomicity and ordering guarantees. This new file system uses short-circuit shadow paging to provide strong safety and consistency guarantees compared to existing file systems while simultaneously providing significantly improved performance.

8. ACKNOWLEDGMENTS

Many thanks to our shepherd, Jeff Mogul, and to the anonymous reviewers for their helpful comments and feedback. Thanks also to Galen Hunt and Bill Bolosky for their insight and technical advice.

9. REFERENCES

[1] Process integration, devices, and structures. In *International Technology Roadmap for Semiconductors* (2007).

[2] BAILEY, D. ET AL. NAS parallel benchmarks. Tech. Rep. RNR-94-007, NASA Ames Research Center, 1994.

[3] BEDESCHI, F. ET AL. An 8Mb demonstrator for high-density 1.8V phase-change memories. In *VLSI Circuits* (2004).

[4] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1996).

[5] FRASER, K., AND HARRIS, T. Concurrent programming without locks. *Transactions on Computer Systems (TOCS) 25*, 2 (2007).

[6] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Symposium on Operating Systems Principles (SOSP)* (1987).

[7] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference* (1994).

[8] KATCHER, J. PostMark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance, 1997.

[9] LEE, B., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture (ISCA)* (2009).

[10] LOWELL, D. E., AND CHEN, P. M. Free transactions with Rio Vista. In *Symposium on Operating Systems Principles (SOSP)* (1997).

[11] MICHELONI, R., MARELLI, A., AND RAVASIO, R. BCH hardware implementation in NAND Flash memories. In *Error Correction Codes in Non-Volatile Memories*. Springer Netherlands, 2008.

[12] MOGUL, J. C., ARGOLLO, E., SHAH, M., AND FARABOSCHI, P. Operating system support for NVM+DRAM hybrid main memory. In *Hot Topics in Operating Systems (HotOS)* (2009).

[13] MOHAN, C. Repeating history beyond ARIES. In *Very Large Data Bases (VLDB)* (1999).

[14] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *Transactions on Database Systems (TODS) 17*, 1 (1992).

[15] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Operating Systems Design and Implementation (OSDI)* (2006).

[16] PANT, M. D., PANT, P., AND WILLS, D. S. On-chip decoupling capacitor optimization using architectural level prediction. *Transactions on Very Large Scale Integration Systems (TVLSI) 10*, 3 (2002).

[17] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. In *Symposium on Operating Systems Principles (SOSP)* (2005).

[18] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture (ISCA)* (2009).

[19] RAOUX, S. ET AL. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4 (2008).

[20] RENAU, J., FRAGUELA, B., TUCK, J., LIU, W., PRVULOVIC, M., CEZE, L., SARANGI, S., SACK, P., STRAUSS, K., AND MONTESINOS, P. SESC simulator, 2005. `http://sesc.sourceforge.net`.

[21] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *Transactions on Computer Systems (TOCS) 10*, 1 (1992).

[22] SMITH, L., ANDERSON, R., FOREHAND, D., PELC, T., AND ROY, T. Power distribution system design methodology and capacitor selection for modern CMOS technology. *IEEE Transactions on Advanced Packaging 22*, 3 (1999).

[23] SUN MICROSYSTEMS. ZFS. `http://www.opensolaris.org/os/community/zfs/`.

[24] WANG, A.-I. A., REIHER, P., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *USENIX Technical Conference* (2002).

[25] WOODHOUSE, D. JFFS: The journalling flash file system. In *Ottawa Linux Symposium* (2001), RedHat Inc.

[26] WU, M., AND ZWAENEPOEL, W. eNVy: A non-volatile, main memory storage system. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1994).

[27] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *International Symposium on Computer Architecture (ISCA)* (2009).