Accelerating Storage Access with Persistent Memory

Abstract

Persistent memories are appearing on the horizon and soon they will sit on the processor memory bus. They will allow software to access persistent data with extremely low latency and high bandwidth. However, existing storage architectures introduce significant overhead that can dominate the low-latency of persistent memories.

In this paper, we present a new, layered storage architecture called Chell that uses fast non-volatile memory to accelerate conventional file access. Chell can operate in two different modes: In direct access mode, Chell gives applications direct access to file stored in a file system that resides in non-volatile main memory. In Chell's caching mode, it uses non-volatile main memory as a cache for a file system that resides on a more conventional storage device (e.g., a hard drive or SSD). In both modes, Chell provide a userspace POSIX-compliant library interface that directly maps non-volatile main memory into the application's address space. From there, file accesses become simple (and very fast) memory copy operations.

Our results shows that with Chell direct access mode, it reduces the 4 KB read/write latency by 19% over PMFS, and improves Berkeley-DB throughput by 6%. Chell cache mode improves the cache hit latency for 4 KB reads by 35% and 50% for writes over page cache, and improves Berkeley-DB performance by up to 4.8×. Finally, we discuss how the antiquated semantics of the POSIX file interface limits Chell's efficiency, and we suggest some changes that would improve performance.

1. Introduction

Emerging non-volatile memories such as spin-torque transfer, phase change, and memristor-based memories [4, 5, 6, 1, 29] promise to revolutionize IO performance and how systems manage and provide access to persistent state. The most aggressive proposals for integrating these technologies place them on the processor's memory bus alongside or replacing conventional DRAM, leading to non-volatile main memory (NVMM)-based systems [30, 20, 26, 18]. The close coupling of compute and non-volatility in NVMM systems will force system designers to rethink how software interacts with storage, if applications are going to reap the benefits of NVMM.

NVMMs invite a range of different interface designs, depending on the amount of NVMM available, the scale of the overall storage system, and the need (or lack thereof) to support legacy applications. Researchers have proposed specialized file systems for NVMMs **[11, 10]**, non-volatile caches [22, 16], and new programming models [8, 25]. However, these interfaces will only be successful if they can provide access to NVMM without sacrificing the performance it can provide.

In this work we present *Chell*, a system that uses NVMM to accelerate access to file data. Chell operates in two modes: The first mode, *Chell-Direct*, targets systems that store entire file systems in NVMM. These might include future mobile systems or servers with large NVMM capacity. The goal of Chell-Direct is to provide direct access to file contents without incurring the substantial latencies that operating and file system-mediated access impose.

Chell-Direct achieves this by transparently replacing conventional system calls with memory operations: In response to `open()`, Chell-Direct memory maps the physical pages that make up the file directly into the application's address space and can then implement calls to `read()` and `write()` with userspace memory-to-memory copy operations. Chell-Direct is implemented as a userspace library, *libChellD*, that dynamically links into the application and interposes on system calls. As a result, it requires neither modification nor re-compilation of the application.

The second approach, Chell-Cache, focuses on systems that must access large storage volumes and are equipped with modest amounts of NVMM. Chell-Cache provides these systems with an NVMM-based persistent, write-back cache of a larger, conventional storage volume.

Chell-Cache builds upon Chell-Direct by adding a kernel module that caches the contents of a conventional file system in the system's NVMM. The Chell-Cache kernel module, *the Chell cache manager*, manages and maintains the cache while the Chell-Cache userspace library, *libChell*, provides direct access to the cached contents.

This paper describes and evaluates both of Chell's modes. It examines the design challenges that each one presents and explains the motivation for the design decisions we made. We evaluate both systems using a combination of micro- and macro-benchmarks and find that comparing to PMFS [11], Chell-Direct reduces 4 KB read/write latency by 19% and improves Berkeley-DB throughput by 6%. Chell-Cache reduces 4 KB access latency by 35% for reads 50% for writes, and improves Berkeley-DB throughput by up to 4.8× over Linux page cache.

We also use the results to illustrate the negative impact that legacy file-based interfaces impose, and, based on these findings, we propose changes that would allow for simpler, faster access to file data. These changes would allows applications to more fully benefit from NVMM performance without requiring invasive changes in how they access and manage non-volatile data.

The remainder of the paper is organized as follows. Section 2 introduces Chell-Direct architecture and implementation, and Section 3 evaluates the performance of Chell-Direct.

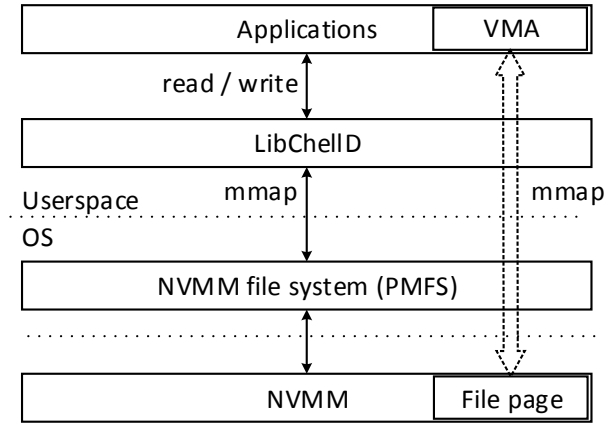

Figure 1: Chell-Direct architecture: by combining NVMM with the ability to map files into userspace, Chell-Direct provides applications with a fast, POSIX-compliant interface to data stored in NVMM.

Section 4 gives a overview of Chell-Cache architecture, and describes the system implementation details of Chell-Cache. Section 5 evaluates the performance of Chell-Cache, and Section 6 introduces our effort to improve the POSIX interface. Section 7 introduces related work and Section 8 summarizes our conclusion.

2. Chell-Direct: accelerating file access on NVMM

Chell-Direct accelerates access to files stored in file systems that reside solely in NVMM. It gives applications direct access to a file's contents without requiring any interaction with the operating system in the common case. As a result, it can eliminate both the system call overhead required to enter the kernel and the file system overhead required to locate stored data and perform permission checks. Below we describe the architecture and implementation of Chell-Direct.

2.1. Chell-Direct

Figure 1 shows the high-level architecture of Chell-Direct. The system is equipped with NVMM that appears in the system's physical address space. An NVMM-aware file system (e.g., PMFS [11]) manages the NVMM space and allows users to create and manage files that reside in the NVMM. When an application requests that a file be `mmap()`'ed into its virtual address space, the NVMM-aware file system maps the NVMM pages directly into the virtual address space. No paging or copying between NVMM and DRAM is necessary. In the Linux kernel this kind of `mmap()` implementation is called *direct access (DAX)*.

Chell-Direct is implemented as an user space library, libChellD, that transparently interposes on file access system calls. Chell-Direct exploits DAX `mmap()` to make common-case access to file data fast. When an application opens a file, Chell-Direct intercedes, and `mmap()`s the file's contents into the application's address space. This provides direct access to the file's data and transfers the permission and extent information from the file system's data structures into the application's page table, leveraging the processor's fast memory protection hardware to locate the file's data and protect its contents.

When the application calls `read()` and `write()`, Chell-Direct translates them into calls to `memcpy()` to transfer data between application buffers and the memory mapped file. Other operations such as `lseek()` simply update the state (e.g., the current file pointer) that Chell-Direct maintains about each open file. Consequently, common-case accesses require no calls to the operating system, avoiding costly system calls and operating system and file system overheads.

Chell-Direct cannot eliminate all interactions with the operating system. In particular, any operations that modify file system meta data still must enter the OS. This means that Chell-Direct helps performance less for appends than for normal write operations, since it needs to make a system call to extend both the file and the in-memory mapping. Also, the updates to the file's modification time do not occur as they would with normal accesses. However, many performance-intensive applications already disable file modification and access time updates to improve performance.

Below we discuss the POSIX compatibility and implementation details of Chell-Direct.

2.2. POSIX Emulation

Chell-Direct must mimic the behavior of the normal POSIX file interface. It must implement the inheritance of file descriptors across calls to `fork()` and their (selective) persistence across calls to `exec()`. Furthermore, `dup()` and `dup2()` can create file descriptors that are aliases for one another. The library must also enforce access restrictions (e.g., disallowing `write()` calls if the file opened read-only, even if the file's permissions allow modification). Implementing this functionality requires Chell-Direct to duplicate much of the information that the kernel would usually manage. This includes file descriptor permission information, the close-on-exec flag, file position information, and file descriptor aliasing information.

We have tested Chell-Direct's fidelity to glibc's implementation of the POSIX interface under Linux using a battery of short tests, the applications described in Section 3, and a random file operation generator. Our testing system (described below) compares the return value and resulting data for every file operation to detect any variation between Chell-Direct and glibc. In our tests, Chell-Direct behaves identically to POSIX.

For file descriptors that point to files that don't reside in NVMM or that represent network sockets and other resource, Chell-Direct passes requests to the default POSIX functions that perform normal system calls.

Chell-Direct avoids the need to modify application source code or recompile by using LD_PRELOAD. This allows Chell-Direct transparently link into the applications and interpose

on calls to libc.

The Chell-Direct POSIX emulation layer is built to be extensible and can be used to implement a variety of interesting utilities. For instance, we used it to test Chell-Direct's POSIX emulation by forwarding requests to both Chell-Direct's implementation and the default POSIX versions (but operating in a different set of files) and comparing the results.

2.3. Implementation

After the POSIX emulation determines which data an read or write targets, the Chell-Direct translates the file operation into an operation on the memory mapped contents of the file.

For each file being accessed via Chell-Direct, Chell-Direct divides the file into 2 MB *chunks*. A B-tree store information about the file's `mmap()ed` chunks. The key is the file offset, and the value is the mapped address and length. For each access, Chell-Direct queries the tree for the mapping information. If it finds the data, it performs a `memcpy()` to transfer data between application buffers and the memory mapped NVMM pages.

If the file offset is not present in the B-tree, Chell-Direct uses `mmap()` to map the data into the application's addresss space. Then, it updates the B-tree and performs the `memcpy`.

Chell-Direct `mmap()`s 2 MB at a time, because `mmap()` is expensive and when the file size changes, it may be necessary to remap the entire file. Mapping two megabyte chunks amortizes the cost of mapping and reduces the need for remapping. It also limits the number of memory-mapped regions the kernel must maintain. Each `mmap()` request allocates a virtual memory area (vma) in application's address space, and Linux kernel limits each process to have up to 65,536 vmas. If the requests are small and frequent, `mmap()`ing each of them will exhaust this number very quickly. For small requests, the 2 MB `mmap()` also works as a prefetching mechanism, reduce the overhead for small sequential requests.

Chell-Direct issues `mmap()` with the MAP_POPULATE flag set to force the kernel to populate the application's page table immediately. Without the flag, the first access to each page causes a page fault, degrading performance. We examine costs and benefits of this approach in detail in Section 3.

2.4. Supported file systems

Chell-Direct relies on the DAX-style behavior of `mmap()` that maps NVMM directly into the application's address space. DAX-like functionality has existed in some file systems for many years. It used to be called *eXecute In Place*, or XIP, and mostly found use in embedded systems that needed to execute code directly from NOR flash devices. DAX capabilities are already present in PMFS [11] and efforts are underway to add it to other file sysetms [15].

3. Results: Chell-Direct

This section evaluates Chell-Direct's performance using a collection of microbenchmarks and database workloads. We begin by measuring Chell-Direct's latency and bandwidth relative to the conventional OS-based interface. For all the experiments in this section, we use 8 GB of DDR3-1600 DRAM to emulate NVMM. We run PMFS to provide file system access to the NVMM.

3.1. Latency

Chell-Direct adds overhead to each file access because its userspace library implements POSIX compatibility and must distinguish between accesses that Chell-Direct should perfom and accesses should it forward to the OS. This latency replaces some of the file system and operating system latency that normal system calls incur.

Figure 2 shows the latency of Chell-Direct for different request sizes comparing to PMFS. All the pages Chell-Direct and PMFS accessed are loaded into memory before measurement begins to ensure that they are in the page table to eliminate the `mmap()` and page fault handling overhead. The total latency for a 4 kB read operation through Chell-Direct is 0.39 μs on average, compared to 0.48 μs for an access to PMFS via the kernel. For a 4 kB write, Chell-Direct takes 0.51 μs, and PMFS takes 0.63 μs. Chell-Direct reduces 4 kB access latency by 19% over PMFS. For requests smaller than 4 kB, Chell-Direct's performance relative to PMFS improves, because writes to the file system involve metadata changes.

In some cases, Chell-Direct reduces performance. In particular file operations that modify file system metadata are slower, since Chell-Direct incurs its own overhead *and* must make a system call. Appends are the most common example. Under Chell-Direct, a 4 kB append takes 1.3 μs. Under the conventional interface, it requires just 1.1 μs.

Figure 3 measures the overhead of `mmap()` and page fault handling and quantifies the trade-offs between pre-loading the page tables with MAP_POPULATE and loading pages on demand. It shows the latency breakdown of for the first access to a page with Chell-Direct. In the test we use Chell-Direct to read a 2 MB file with a sequence of 4 kB requests. Since Chell-Direct `mmap()`s 2 MB at a time, there is only one `mmap()` call for the file. The graph shows the amortized cost for accessing a single page, the total `mmap()` latency is 512 times larger. With MAP_POPULATE disabled, `mmap()` does not populate the pages to the page table, and the amortized `mmap()` latency is low: only 2.6 ns for each 4 kB access. With MAP_POPULATE enabled, the amortized cost of `mmap()` is 88 ns for each page. However, the access latency for each page with MAP_POPULATE enabled is 310 ns versus 612 ns with MAP_POPULATE disabled. This means servicing the page miss takes 302 ns. These data suggest that using MAP_POPULATE is important to achieving good performance, *if most pages will eventually be accessed*.

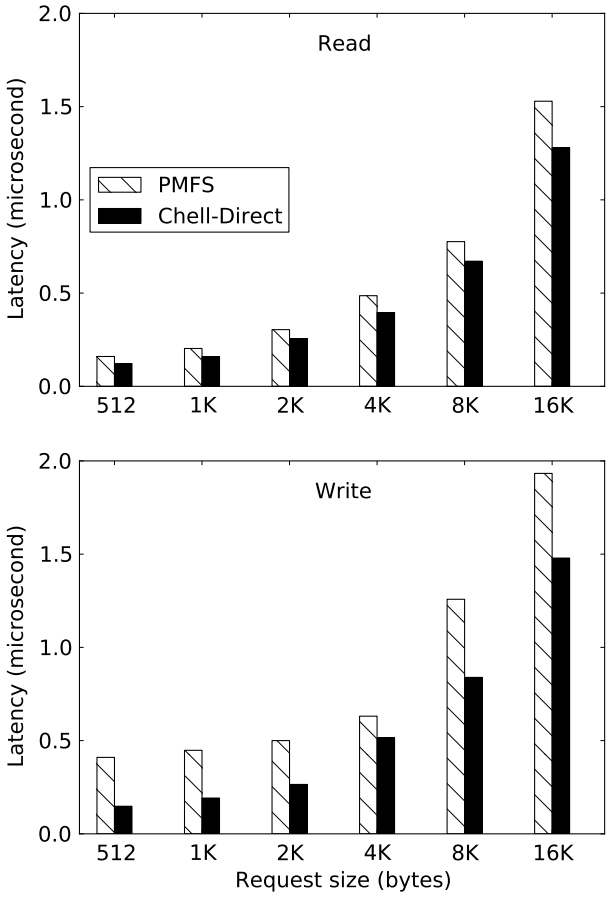

Figure 2: Chell-Direct latency for different request sizes compared to PMFS with pages loaded on demand. The cost of mmap() is amortized across the 512 pages in the 2 MB file.

Based on the values we measured for our system, if at least 28% pages of the 2 MB chunk will be accessed, then enabling MAP_POPULATE improves performance.

3.2. Bandwidth

Placing storage on the processor's memory bus should provide high bandwidth access to data. We use Xdd [28] to measure bandwidth of Chell-Direct. Figure 4 quantifies the bandwidth available from our NVMM device via PMFS and via Chell-Direct. Each graph shows the performance for sequential and random 4 kB accesses with three different read/write ratios: 100% read , 50/50% read/write, and 99% writes. We did not test 100% writes, because files opened for write-only access cannot be `mmap()ed`. The three figures measure the aggregate bandwidth of 1, 4, and 16 threads.

For a single thread, Chell-Direct out-performs POSIX by between 4% and 21%, and Chell-Direct performs better on writes than reads. For four threads and 16 threads, Chell-Direct outperforms PMFS in writes by up to 2.7× for four threads and 4.2× for 16 threads. We believe this is because

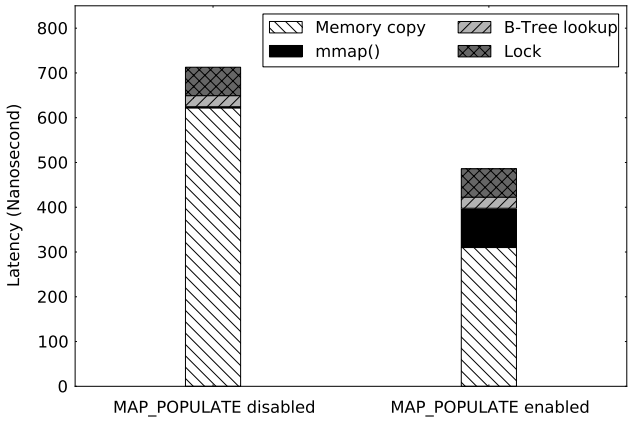

Figure 3: Chell-Direct 2 MB file read latency breakdown for each 4 kB access, with the MAP_POPULATE flag set. The cost of mmap() is amortized across the 512 pages in the 2 MB file.

	PMFS	Chell-Direct
	(Ops per second)	
Berkeley-DB Btree	27160	29019

Table 1: Chell-Direct Berkeley DB performance

PMFS applies strict synchronization to provide write atomicity, and limits the aggregate write bandwidth for multi-thread applications. However, for performance sensitive applications, relaxing write atomicity in return for higher performance is common choice [2]. For reads, Chell-Direct and PMFS performance are very similar.

3.3. Macro-benchmarks

To measure application-level performance we ran Berkeley-DB Btree on top of Chell-Direct. In the test, Berkeley-DB Btree application reads and writes records to a 5.5 GB database file, and writes logs to 10 MB log files. The workload comprises 35% reads and 65% writes. The average read request size is 511 bytes, and average write request size is 431 bytes. The application is running with single thread.

Chell-Direct improves the performance by 6% compared to PMFS. Amdahl's law limits the gains: NVMM is fast enough that the bottleneck is no longer in the device I/O. Over the test period, only about 25% time is spent on I/O, and Chell-Direct saves about 20% time on I/O operations.

4. Chell-Cache: Accelerating Legacy Storage with NVMM

Chell-Direct is useful if the entire file system resides in NVMM, but many storage systems are too large to fit in NVMM. To improve performance of these systems, Chell provides an alternative mode of operation called Chell-Cache. Chell-Cache uses NVMM to provide a reliable, consistent, write-back cache of data that resides in an existing file system.

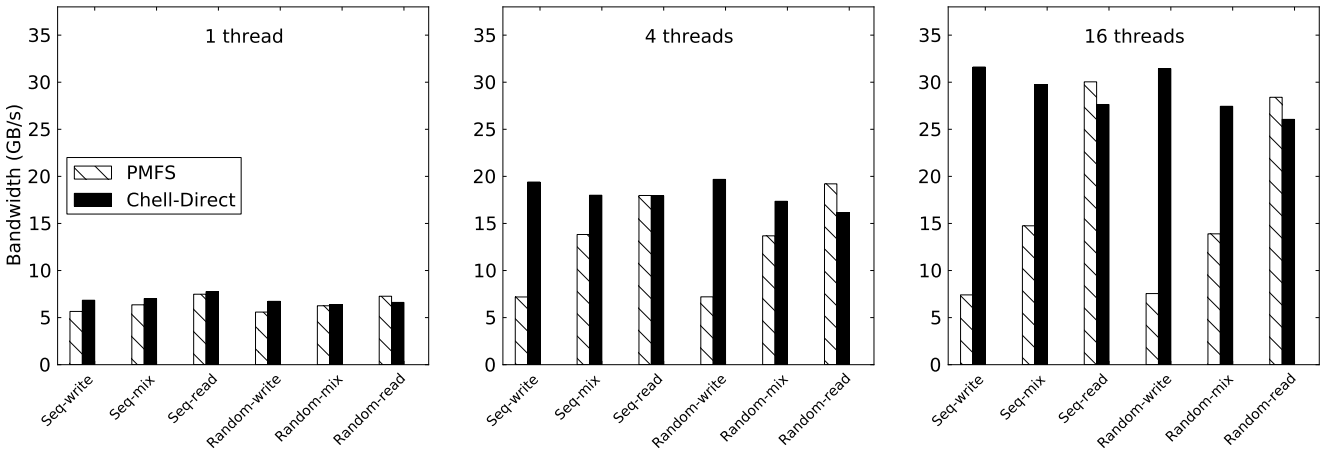

Figure 4: Chell-Direct xdd bandwidth

Chell-Cache provides fast access to the cached data using the same techniques that Chell-Direct uses but it accesses cached data rather than actual file data. This section and the next describe the differences between Chell-Cache and Chell-Direct.

Figure 5 shows the system architecture of Chell-Cache. Like Chell-Direct, there is an userspace library, libChell to handle cache hits, but Chell-Cache also adds a kernel module, the Chell cache manager, to manage the contents of the cache and interact with the file system on the backing store device.

The only difference in libChellD and libChell is in how they detect and react to a miss. LibChellD detects a miss when it finds that its B-tree does not contain an entry for the file location the application is trying to access. This occurs the first time the application accesses each chunk. Its response is simple, issues an `mmap()` to map the chunk.

LibChell can also experience a miss when an entry is missing from its B-tree. But a miss can also manifest itself as a segmentation fault, if the data was mapped into userspace but the kernel had to evict it to make room for other data. In both cases, libChell issues an `ioctl()` to the Chell cache manager requesting that it load the data from the backing store (if needed), and then map the data into the application's address space.

The Chell cache manager manages the data cached in the NVMM. It copies data between cache pages and the backing store device, memory maps the requested cache pages to the application's address space, and handles cache write back and eviction.

When the Chell cache manager receives the `ioctl` request from libChell, it first checks the file Inode permission to make sure it is a valid request. Then it checks if the requested data is in the cache, and loads it, if required. Next, the Chell cache manager copies data from the user buffer to the cache pages if it is a write, or from the backing store device to the cache pages and user buffer if it is a read. Then it calls `mmap()` to map the cache pages to the application's address space and returns the mapped address and length to libChell.

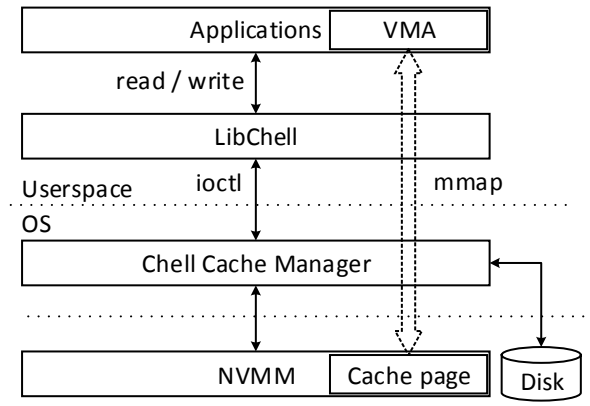

Figure 5: Chell-Cache architecture: Chell-Cache caches files stored on slow storage devices in NVMM, provides a fast, POSIX-compliant interface to applications.

By default Chell-Cache requires all the applications to access the NVMM cache via libChell. Otherwise, if one application accesses a cached file via Chell-Cache, while another application accesses the same file on the backing store device via POSIX write, Chell-Cache and the file system on the backing store device will have an inconsistent view about the file. To prevent this inconsistency, the Chell cache manager interposes at the block device driver for the backing store and rejects accesses that touch cached data but that are not from applications using libChell.

4.1. Chell-Cache Implementation

The Chell-Cache kernel module, the Chell cache manager, manages the system's NVMM memory as a cache for a conventional file system. The file system could reside on a disk or SSD. Chell-Cache must address several challenges to make the cache safe and efficient. These include fast cache pages allocation and lookup, efficient cache eviction and replacement policy, and cache recovery from system failure. In the fol-

lowing sections we describe the implementation details about Chell-Cache.

4.1.1. Chell-Cache space management The space allocation and management system in the Chell cache manager is based on PMFS. It allocates NVMM in 4 kB pages to minimize fragmentation and wasted space for small files. For file systems that store mostly large file, larger 2 MB or 1 GB pages would improve TLB performance, reduce the number of page faults, and shrink the page table.

Chell-Cache uses *cache nodes (Cnode)* to manage cached files. Cnode is similar to a file system Inode: It contains the meta information for the cached contents of a file, including the file size and file access time. For each file in the backing store device with data in the cache, there is a corresponding Cnode in the Chell cache manager. Figure 6 shows the data layout of cache space. The NVMM is divided into three sections: the superblock, the journal area, and the Cnode and data pages. The superblock contains information about the cache's layout and is the starting point for crash recovery.

The Chell cache manager uses two maps to organize the Cnodes and cache pages. *Inode-to-Cnode (I2C)* maps Inodes from the backing store to Cnodes in Chell-Cache. Each Cnode has a map called *Cnode-to-File (C2F)* that tracks the location of cached data in NVMM. Both maps are implemented as B-trees.

When the Chell cache manager starts up, it remaps the whole NVMM region into the kernel's address space. This gives it access to superblock region. The superblock contains a root pointer to the I2C map. If the Chell cache manager is initialized on an empty NVMM, it allocates a new I2C map. If the Chell cache manager is loaded to recover a existing Chell-Cache system, the superblock uses the root pointer to find the root of the I2C map.

The Chell cache manager creates an entry in the I2C map on demand for each file that an application accesses. The newly-added Cnode contains a pointer to the Cnode's C2F map, and the Chell cache manager populates it as it brings data for the file into the cache.

4.1.2. Cache mmap policy To efficiently perform `mmap()` operations, Chell-Cache registers a page fault handler to translate a virtual address to a physical address in NVMM cache.

When a page fault occurs, the OS virtual memory manager (VMM) triggers the handler, and the handler returns the corresponding cache page to the VMM, so that the cache page will be mapped directly to the application's address space. This avoids the overhead of the page cache and the block layer for normal `mmap()` operations and eliminates a `memcpy()` in the Chell cache manager. Like Chell-Direct, Chell-Cache divides the cached file into 2 MB chunks and `mmap()` with 2 MB granularity.

If the application appends a file, Chell-Cache need to send the request to the backing store device since the request changes the metadata. However, if the append size is large (over 32 kB), Chell-Cache does not send the write

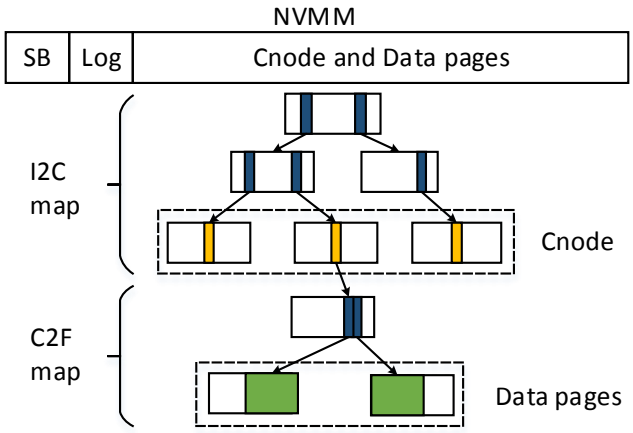

Figure 6: Chell-Cache space layout: The Chell cache manager uses two maps to manage NVMM space: Inode-to-Cnode map and Cnode-to-File map

request to the backing store device directly, instead it issues a `fallocate()` to the file system on the backing store, so the file system will allocate disk space for the append data. If the `fallocate()` succeeds, Chell-Cache simply writes the request data to the cache pages in NVMM, eliminating the overhead to write to backing store device. This requires the file system on the backing store device to support `fallocate()`, but the major Linux file systems (e.g., ext4, btrfs [21], and xfs [13]) support it. The append size threshold also affects the performance, as for small requests, the overhead of `write()` is smaller than `fallocate()`. We show the impact of `fallocate()` threshold in Section 5.2.

4.1.3. Cache eviction When the cache is full, the Chell cache manager must evict some cache pages to make room for new requests. Since cache pages may be shared and `mmap()`ed to multiple applications' address spaces, it is essential to remove all the old mappings of these cache pages before filling them with new data: otherwise the applications will access invalid or protected data. The Chell cache manager removes the mappings from all the applications' address spaces and then writes back the data to the backing store device if the cache page is dirty.

LibChell does not know whether a cache page has been evicted or not. If libChell performs a `memcpy()` on an evicted cache page where the mapping has been removed, a segmentation fault will occur. To deal with this situation, libChell installs a signal handler to intercept and handle the segmentation faults. Upon receiving a SIGSEGV signal from the handler, libChell removes missing memory region from its B-tree and tries again. The access will miss, and control will pass to the Chell cache manager.

4.1.4. Replacement policy The Chell cache manager decides which Cnode will serve as the victim when an eviction is necesssary. the Chell cache manager uses an approximate LRU policy to select the victim.

the Chell cache manager maintains a linked list of Cnodes to track LRU information. Whenever the Chell cache manager accesses an Cnode, the Chell cache manager moves the Cnode to the tail of the list. During an eviction the Chell cache manager chooses the head of the list.

The LRU information is imperfect, because the Chell cache manager is unaware of many accesses to the cached data, since libChell accesses the NVMM directly on cache hits. Since we managed the LRU list as a queue, the result is a blend of LRU and FIFO.

4.1.5. Dirty data tracking Chell-Cache is a write-back cache, so when the Chell cache manager evicts cache pages, it must know which data in the cache is dirty, so it can write it back correctly. Since libChell writes to memory mapped cache pages directly from user space, the Chell cache manager cannot detect changes to the cached data on its own. Fortunately, the OS already tracks dirty virtual memory pages: when a memory page is written, the hardware set the dirty bit in the page table entry (PTE) of the page, and Chell-Cache leverages this facility. Before evicting a cache page, the Chell cache manager looks up the page table entry (PTE) in all the mappings of the page. If the PTEs has its dirty bit set, the Chell cache manager will write the cache page back to backing store.

4.1.6. Consistency and recovery Chell-Cache uses undo logging to ensure that the cache metadata remains consistent in the event of a crash. Whenever the Chell cache manager needs to make a change to a Cnode, allocate a Cnode, or re-allocate pages, the Chell cache manager records the data in the log and makes it persistent before writing the new data in place.

Every metadata change is organized as a transaction. The Chell cache manager starts a transaction by allocating the required number of log entries in the journal. Then, the Chell cache manager fills the log entries with old metadata. Each log entry is written to the journal and made persistent by flushing it to NVMM before new data is written in-place. After all the updates are finished, the Chell cache manager commits the transaction by writing a special "finish" log entry to the journal to invalidate the undo information.

If there is a power failure or system crashes, the Chell cache manager can recover the cache when the system reboots. During recovery, the Chell cache manager locates the log area using the superblock, identifies uncommitted transactions and applies the undo information in the log to restore the metadata. After all the uncommitted transactions have been undone, Chell-Cache is in a consistent state.

5. Results: Chell-Cache

In this section we evaluate Chell-Cache's performance using a collection of micro- and macro-benchmarks.

5.1. Experimental setup

We use the same system for this evaluation as we did in section 3. We use a 1 GB file to test Chell-Cache performance. To compare it with Linux page cache, we also ran the same tests directly on the backing store device and disabled any DIRECT I/O option to make sure page cache was used. We set the cache size to 8 GB and use a 120 GB SSD as backing store device, mounted with an ext4 file system. We repeated each test three times.

5.2. Microbenchmarks

To evaluate Chell-Cache's performance characteristics we used a combination of custom-built microbenchmarks, XDD [28], and IOzone [14].

Figure 8 shows the 4 kB access latency of Chell-Cache. Accessing the cache pages directly from the user space allows Chell-Cache to reduce read/write operation latency. To perform a 4 kB read operation, Chell-Cache takes about 0.52 μs on average, while a 4 kB read from page cache takes 0.79 μs. For 4 kB write, Chell-Cache takes 0.62 μs, and page cache takes 1.24 μs.

Chell-Cache reduces latency by 35% for 4 kB reads and 50% for 4 kB writes, because entering the kernel takes a significant part of the whole operation for small memory accesses. By bypassing the kernel, Chell-Cache eliminates this overhead.

Figure 7 compares the bandwidth of Chell-Cache and page cache for sequential and random 4 kB accesses in three different read/write combinations: 100% reads, 50/50% reads and 99% writes.

For single-threaded bandwidth, Chell-Cache outperforms the page cache for all three workloads, with improvements ranging from 3% for random read to 4.1× for sequential write. With four threads, Chell-Cache outperforms page cache by 19% for random read and boost sequential writes for 9×. With 16 threads, Chell-Cache outperforms page cache by 74% for random read and 5.6× for sequential writes.

IOzone [14] is a filesystem benchmark tool that provides many different types of operation patterns. We use it to confirm the results from xdd and measure a wider range of access patterns. We evaluated the performance for read, write, re-read, re-write, random read/write, read backwards, record rewrite and strided read.

Figure 9 shows the bandwidth of Chell-Cache and page cache for different IOzone tests, with 4 kB access size and 1 GB file size. Chell-Cache improves bandwidth over page cache in all test cases, from 29% (random read) to 103% (random write).

When an application appends to a file, Chell-Cache needs to notify the file system on the backing store device that the metadata of the file has changed. There are two ways to do this: one is to simply send the write request to the file system, and the other is to call `fallocate()` to allocate space for the file on the backing store device, then write the data to NVMM cache.

Figure 10 shows the impact of `fallocate()` to Chell-Cache performance with different request sizes. As we expect, for small request (less than 32 kB), the overhead of `write()` is smaller than `fallocate()` and copy data to cache. For

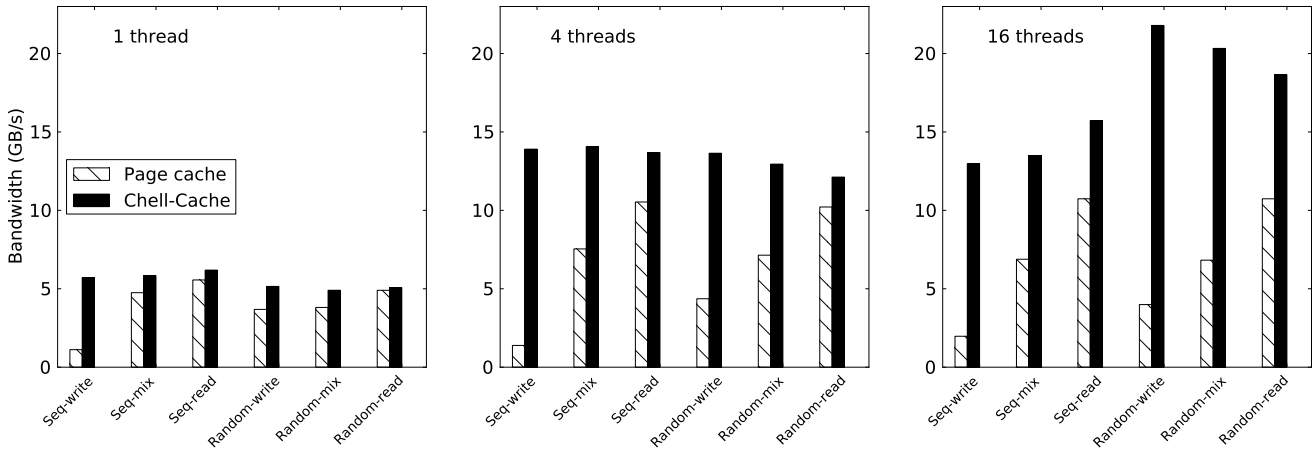

Figure 7: Chell-Cache performance: xdd

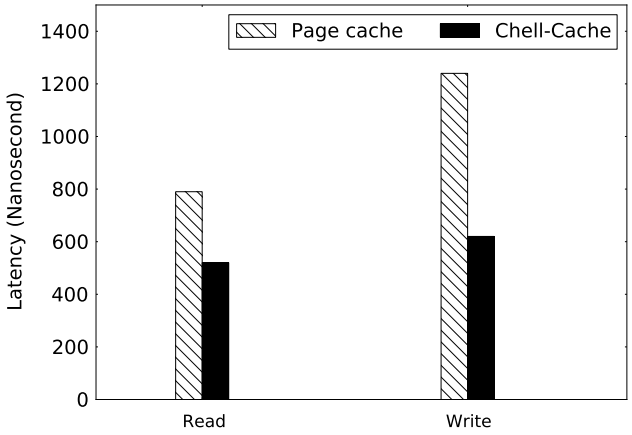

Figure 8: Chell-Cache 4kB read/write latency comparing with page cache

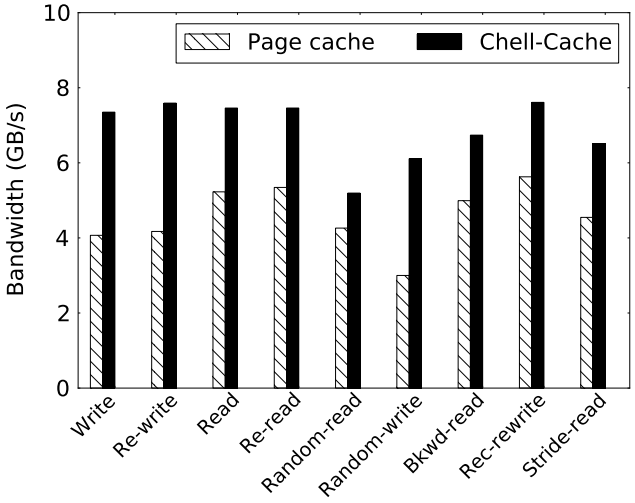

Figure 9: Chell-Cache performance for IOzone with different read/write patterns

requests equal to or larger than 32 kB, using `fallocate()` is more efficient. In Chell-Cache we use 32 kB as a threshold to trigger `fallocate()` for file appendings.

5.3. MacroBenchmarks

We use Berkeley-DB Btree application to evaluate the performance of Chell-Cache with real applications. The application runs with a single thread, two threads, four threads, or eight threads. Berkeley-DB uses multiple threads to perform database lookup and update operations.

Figure 11 shows the performance of Berkeley-DB Btree application on Chell-Cache. Chell boosts the throughput by 3.7× to 4.8× compare with page cache, and close to the performance of Berkeley-DB Btree running directly on PMFS.

Berkeley-DB calls `fdsync()` for each write operation to sync file data to storage devices to ensure the data persistency. For each `fdsync()`, libChell flushes the affected cachelines, while the Chell cache manager ensures the cache metadata remains consistent in the event of a system crash. This significantly reduces the sync latency and improves application-level performance.

We also run Filebench [12] workloads with Chell-Cache. Filebench is a file system and storage benchmark, it includes several useful macro-benchmarks to emulate real applications. In the test we show the normalized performance of Chell-Cache comparing to page cache, with different average file size.

Figure 12 shows the performance of Chell-Cache on filebench. We choose three workloads: fileserver, webserver and varmail. The fileserver workload creates files, append to them, read them and then close and delete. With average file size of 128 kB, Chell-Cache gets 77% performance of the page cache. This is because we `mmap()` files in 2 MB chunks, and small file does not provide an opportunity to amortize the

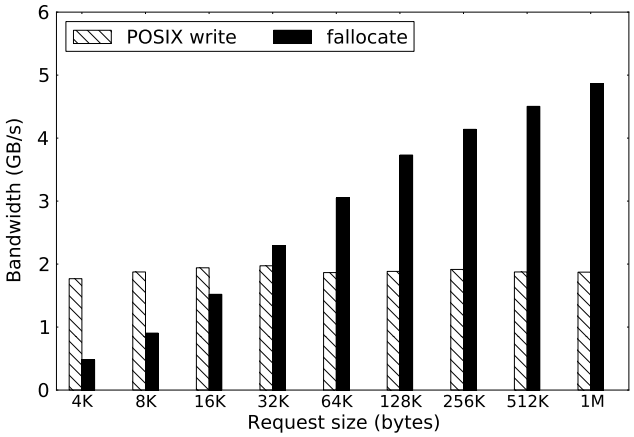

Figure 10: Chell-Cache performance with fallocate() to allocate space for file appends

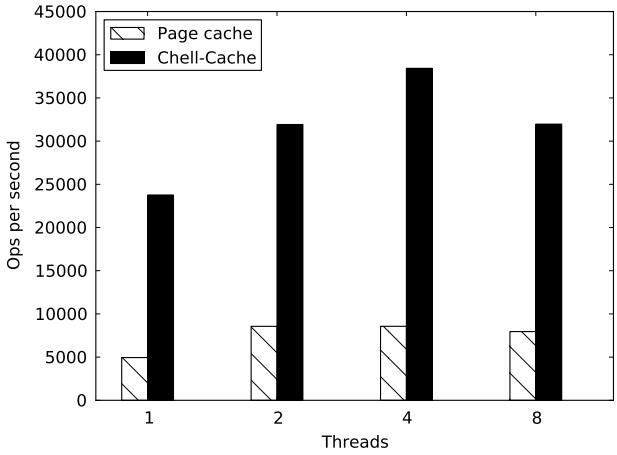

Figure 11: Chell-Cache performance: Berkeley-DB

cost of the `mmap()`. The fileserver workload also creates many new files, and all the data be written to the backing store device. When the average file size increases, Chell-Cache shows better performance than page cache: for 512 kB files it outperforms page cache by 31% and for 2 MB files the performance gain is 91%. Similarly, with webserver workload, Chell-Cache outperforms page cache by 28% for average file size of 2 MB; with varmail workload, Chell-Cache performs even better, improves the performance from 82% (128 kB) to 3.4× (512 kB) over page cache. This is because varmail workload uses `fsync()` to make the file persistent, and with Chell-Cache the `fsync()` is much more efficient.

6. Fixing the POSIX File Interface

Both Chell-Direct and Chell-Cache transparently provide a POSIX-compatible interface, so no changes to the application are needed. This means, however, that applications must use POSIX-style file access functions and some of these are a poor match for NVMM. In particular, the POSIX interface requires the programmer to specify the buffer that `read()` copies data to. POSIX places no constraints on the size or alignment of the buffer, so a copy operation is usually unavoidable. For slow disks and even SSDs, the cost of this copy is small.

But for Chell-Direct and Chell-Cache, the relative cost is much larger. Figure 13 shows the latency breakdown of 4 kB read and write operations of Chell-Direct with page table populated and page fault overhead eliminated. The memory copy accounts for 76% for 4 kB read and 77% for 4 kB write. For Chell-Cache, the situation is similar. This high overhead of memory copying makes the POSIX interface itself a major impediment to exploiting the performance of NVMM.

To eliminate it, we propose a new file read interface optimized for NVMM-based storage systems. The interface provides a new function:

```
ssize_t get_read_buf(int fd, char **buf, size_t count);
```

The interface is similar to the conventional `read`, except that calls to `get_read_buf` return a buffer to programmer via the `buf` parameter. The `*buf` contains a portion of the target file (`fd`), starting from current file offset and extending for up to `count` bytes. The return value is the actual length of the memory region. No copying is necessary. We added the get_read_buf interfaces based to Chell-Direct.

Figure 14 compares the performance of the normal POSIX interface and `get_read_buf` for a sequential read workload running on PMFS with Chell enabled. The copy that the POSIX interface requires causes latency to increase linearly with request size. `get_read_buf` does not do any copying, so latency is nearly constant. For 4 kB request, get_read_buf outperforms POSIX for 4.2×, and for 1 MB request, `get_read_buf` achieves performance gain up to 700×.

7. Related Work

The emergence of fast, non-volatile memories and the prospect of widely-deployed NVMM has motivated many projects.

BankShot [3] implements a fast SSD caching system and allows applications to access the SSD cache directly without going to kernel, and implements file system permission check in hardware to provide protection to files. It also supports dirty data tracking and LRU eviction in hardware.

eNVy [26] proposes a system that attaches flash to the memory bus, and using SRAM to achieve high bandwidth and low latency. WSP [18] utilizes NVMM's persistent property, flushes processor registers and caches to NVMM on power failures to reduce the system recovery time. Some vendors have produced non-volatile main memory devices, they are either small capacity and byte-addressable [17], or large ca-

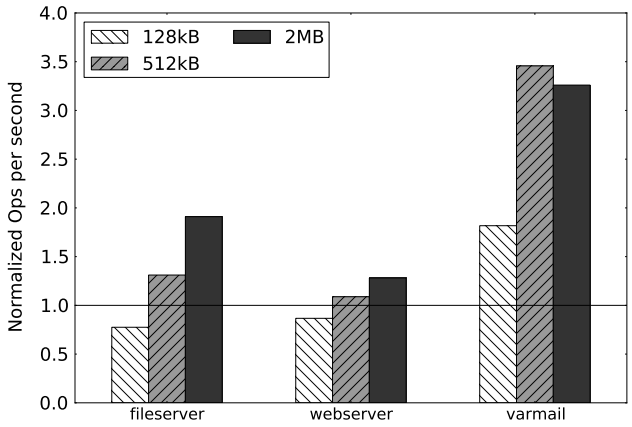

Figure 12: Chell-Cache filebench performance: comparing to normalized page cache performance with different mean file sizes

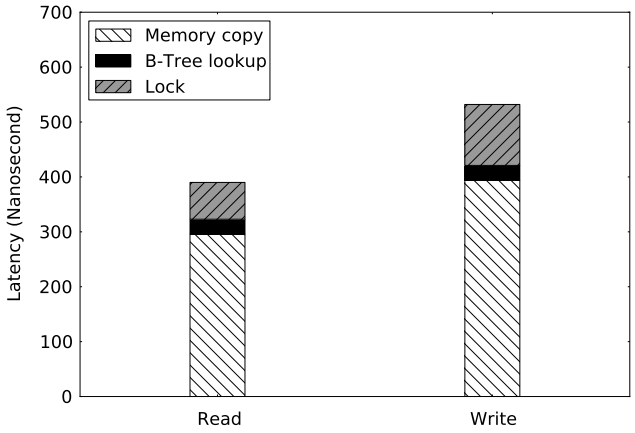

Figure 13: Chell-Direct 4 kB read/write latency breakdown. Memory copy accounts for 76% of total latency. Page table is populated on file open, so the page fault overhead is eliminated.

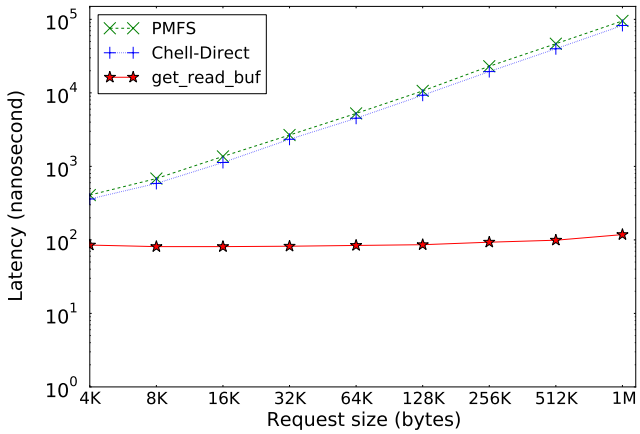

Figure 14: get_read_buf interface latency for different request sizes: latency of PMFS and Chell-Direct increases with request size, while get_read_buf has constant latency

pacity but block-addressable [23]. With emerging non-volatile technologies, we can expect large capacity, byte-addressable NVMMs to appear.

Several systems have exposed non-volatile memories on the processor bus and provide novel interfaces to the data. Rio Vista [16] and recoverable virtual memory [22] are among the oldest systems in this class. Rio Vista exposed raw battery-backed DRAM to the application but made no provisions for file-based access. RVM provided a simple transactional interface to raw non-volatile memory. More recently, several projects [9, 25, 24] produced frameworks for storing persistent objects in these memories. Like Chell, these schemes expose persistent memory directly to the applications, but they also require programmers to rewrite applications to take advantages of them.

Some researchers integrate NVMM into existing systems with minimal impacts. PMBD [7] leverages Linux kernel's block layer and presents NVMM as a block device to users. It protect the NVMM from stray writes by dynamically mapping NVMM pages into the kernel space only when needed, and uses non-temporal store (e.g., `movntq` and `clflush` in write-mode to provide write ordering. Although PMBD provides protection and ordered persistence, implementing NVMM as a block device undermines the low-latency and high-bandwidth properties of NVMM, as the performance is restricted by the block layer, which is designed for slow storage like hard disks. Chell-Direct and Chell-Cache works as a memory and allow applictions to access NVMM directly from user space to minimize the software overhead.

A great deal of research focuses on the file system architect for NVMMs. SCMFS [27] utilizes the OS VMM module to perform block management and layout for files in large contiguous virtual address regions. BPFS [10] is a file system designed for NVMM that provides tranactional guarantees. BPFS uses copy-on-write and eight-byte atomic updates to provide metadata and data consistency. However, BPFS does not support `mmap()`, and it relies on a hardware approach (epochs) to support store durability and ordering, which requires hardware modifications. PMFS [11] is a light-weight file system that exploits persistent memory's byte-addressability, avoiding the overhead of accessing memory via the block layer and enabling applications to access persistent memory directly. It also provides software-enforcebale guarantees of durability and ordering of writes while protecting persistent memory from stray writes. Unlike BPFS, PMFS utilizes pm_wbarrier primitive to flush PM stores to a power

fail-safe destination, and does not require hardware support.

Both Chell-Direct and Chell-Cacheses `mmap()` to accelerate NVMM access. Failure-atomic msync [19] proposes a atomic commit of memory pages change for `mmap()ed` files, by flushing memory pages to persistent store first and then write back to the destination. For Chell-Direct and Chell-Cache, as they are running on NVMM, there is no need to write back the data, and `msync()` is simely a CPU cache flush.

8. Conclusion

NVMMs will require significant changes to systems architectures if applications are to exploit the performance they can provide. We have described and evaluated Chell, a system that uses NVMM to accelerate file access. Chell-Direct provides direct access to file data for file systems that reside solely in NVMM, while Chell-Cache uses NVMM a persistent, write-back cache of a conventional file system. In both cases, Chell allows the application to access file data, in most cases, without interacting with the filesystem.

We found that Chell-Direct improves 4 kB access latency by up to 19% relative to file access through a file system built for NVMM and improves Berkeley-DB performance by 6% over PMFS. Chell-Cache can reduce 4 kB access latency by up to 50% and improve Berkeley-DB performance by up to 4.8× relative to file access using the system's file buffer.

Finally, we demonstrated a novel file IO interface that eliminates the need to copy data to and from userspace memory. Since the advent of NVMM will make such copies the dominant cost in accessing file data, eliminating them is necessary if applications are going to fully levelage NVMMs.

References

[1] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[2] R. Arpaci-Dusseau. The application/storage interface: After all these years, we're still doing it wrong. HotStorage'14, 2014.

[3] M. S. Bhaskaran, J. Xu, and S. Swanson. Bankshot: Caching slow storage in fast non-volatile memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 1:1–1:9, New York, NY, USA, 2013. ACM.

[4] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 385–395, New York, NY, USA, 2010. ACM.

[5] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[6] A. M. Caulfield and S. Swanson. Quicksan: A storage area network for fast, distributed, solid state disks. In *ISCA '13: Proceedings of the 40th annual international symposium on Computer architecture*, 2013.

[7] F. Chen, M. P. Mesnier, and S. Hahn. A Protected Block Device for Persistent Memory. Technical Report TR-14-01, LSU/CSC. http://www.csc.lsu.edu/ fchen/publications/abs/TR-14-01.html.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *To Appear: ASPLOS '11: Proceeding of the 16th international conference on Architectural support for programming languages and operating systems*, 2011.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

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

[12] Filebench file system benchmark. http://sourceforge.net/projects/filebench.

[13] S. G. International. XFS: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs.

[14] Iozone filesystem benchmark. http://www.iozone.org/.

[15] https://lwn.net/Articles/588218/.

[16] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 92–101, New York, NY, USA, 1997. ACM.

[17] Hybrid memory: Bridging the gap between dram speed and nand nonvolatility. http://www.micron.com/products/dram-modules/nvdimm.

[18] D. Narayanan and O. Hodson. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[19] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 225–238, New York, NY, USA, 2013. ACM.

[20] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[21] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[22] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 146–160, New York, NY, USA, 1993. ACM.

[23] J. Scaramuzzo. Reaching the final latency frontier (SMART storage systems). In *Proceedings of the 2013 Flash Memory Summit*, 2013.

[24] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2011. ACM.

[26] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 86–97, New York, NY, USA, 1994. ACM.

[27] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[28] XDD version 6.5. http://www.ioperformance.com/.

[29] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[30] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 14–23, New York, NY, USA, 2009. ACM.