Quill: Exploiting Fast Non-Volatile Memory by Transparently Bypassing the File System

Louis Alex Eisner Todor Mollov Steven Swanson
UCSD CSE Technical Report #CS2013-0991
Computer Science and Engineering
University of California, San Diego

Abstract

Fast non-volatile memories will soon make their appearance on the processor memory bus. They offer the potential for extremely low-latency, high-bandwidth access to persistent files. However, existing interfaces will impose large system call and file system overheads on those accesses, squandering the memories' performance. At the same time, we would like to leverage existing file systems as much as possible, to reduce the cost of adopting these new storage technologies.

We present a user space library, called Quill, that avoids system call overheads on most accesses by interposing on file operations and converting them into memory operations. Quill enforces the protections that the file system provides by transferring protection information into the process's page table and accessing files directly via load and store instructions. We describe the Quill system and compare it to a conventional software stack accessing the same storage device.

1 Introduction

Emerging non-volatile memories such as spin-torque transfer, phase change, and memristor-based memories 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.

Placing fast, non-volatile memories on the memory bus raises a host of design issues, and several proposals examine extensive changes to the file system [4] and the basic abstractions that applications use to manipulate non-volatile data [3, 9]. These proposals provide useful features (in particular transactional semantics), but they require significant changes to the operating system and/or the applications to leverage them.

We propose a simpler approach that will expose most of the performance that these technologies offer while requiring only minor changes to the file and operating systems and no changes to the application.

Our system, called *Quill*, takes advantage of the fact that these fast non-volatile memories will appear as pages in the processor's physical address space. To provide access to files, Quill maps those physical pages directly into the application's virtual address space, so no paging is necessary. Quill interposes on IO system calls in user space and translates them into operations on the resulting memory region. As a result, for most accesses, applications see both the latency and bandwidth of DRAM while using a normal file-based interface.

Quill comprises two components. The first is a generic file IO interposition layer called *Nib* that allows a library to take over file IO operations for specific file descriptors and forward requests to a handler that implements file access functions. We are planning to release Nib as a general purpose tool. The second component is a Nib handler that implements the Quill functionality.

This paper describes Quill and Nib in detail and evaluates the impact of using memory-mapped access to expose the performance that fast non-volatile memories can provide. We find that Quill can improve performance by up to 7x compared to conventional operating system-based access to the same fast non-volatile memory.

The remainder of this paper is organized as follows. Section 2 describes Quill, Nib and the changes required in the kernel to support Quill. Section 3 evaluates Quill, and Section 4 places Quill in the context of other projects. Section 5 summarizes our conclusions.

2 Quill and Nib

Quill and Nib interpose on accesses to files that reside in non-volatile memories attached to the processor's memory bus. We assume that the memory appears as a RAM disk-like block device with the critical difference that its contents are persistent across restarts. For operations that do not affect the NVM block device (e.g., `read()`s from network sockets) Quill invokes the normal system calls.

The Quill handler translates calls which target the NVM block device into memory-only operations, avoiding operating system overheads. Quill uses processor's fast memory protection hardware (i.e., the TLB) to enforce file system permissions on accesses.

In our system we use DRAM as a stand-in for advanced non-volatile memories. The libraries transparently link into the application using LD_PRELOAD, allowing for extremely efficient control transfer without requiring applications to be modified or even recompiled.

Below we describe the necessary file system support, Quill, and Nib in more detail.

2.1 The file system

Quill requires the file system to support "execute-in-place" memory mapping, or XIP. For block devices whose backing storage resides in the processor's physical address space, XIP changes the behavior of `mmap()`. Instead of setting up page table entries that allow the operating system to page a file's contents between the block device and the buffer cache in DRAM, XIP maps the block device's physical pages directly into the application's address space and transfers file protection information into the process's page table. As a result, load and store instructions affect the contents of the block device directly, so there is no paging overhead. XIP got its start (and its name) by allowing embedded systems to run executables directly out of NOR flash. Access was read-only, since writes to NOR flash are very slow. Advanced, non-volatile memories (like PCM or STTM) write quickly enough to make read/write access feasible.

Linux does not provide XIP support by default, but it is available as a patch to the ext2 file system and the `brd` Linux block ram driver under Linux 2.6.32.

The changes required to `ext2` and `brd` are small – just 194 lines total. This suggests that adding support for XIP to other file systems is tractable.

2.2 Quill

The Quill library 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.

When an application accesses a file, Quill takes over, opening the file and using `mmap()` to map the file's contents into the application's address space. In doing so, it effectively transfers the permission and extent information from the file system's data structures into the application's page table, thereby leveraging the processor's fast memory protection hardware (i.e. the TLB) to locate the file's data and protect its contents.

After that, calls to `read()` and `write()` translate to some simple checks of the file state and a call to `memcpy()` to transfer data between application buffers and the memory mapped file. Similarly, other operations such as `lseek()` simply update Quill's internal state. Consequently, for most operations no calls to the operating system are required, avoiding costly context switching, OS, and file system overheads.

Quill 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 Quill 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. It also means that updates to the file's modification time do not occur as they would with normal accesses.

To be useful, Quill must mimic the behavior of the normal POSIX file interface, including 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 Quill 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 aggressively tested Quill'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 Quill and glibc. For our deterministic workloads, Quill behaves identically to POSIX.

2.3 Nib

Nib is a key component of Quill, and it provides all the functionality that is not related to translating file operations into memory operations. It handles interposing on the system calls, deciding which file descriptors to target (in Quill's case, it targets descriptors for files on XIP devices), and tracking the creation, destruction, and duplication of file descriptors.

Nib is a general-purpose library and Quill is just one of its applications. The core of Nib is a small software layer called the *hub* that forwards file operation to one or more handlers that actually process file operations (e.g., Quill is a handler). Below, we describe the hub, the handler interface, and then give some examples of useful handlers we have developed to support this work and that demonstrate Nib's general usefulness.

The hub The hub's job is to route application requests to the correct handler and to track the creation and duplication of file descriptors. To route requests, it maintains a map between active file descriptors and handlers. When the application calls `open()`, the hub uses a programmer-provided function to decide which handler should be responsible for operations on the resulting file descriptor. For Quill this function uses `fstat()` to determine whether the file resides on an XIP block device.

If hub encounters a file descriptor it is unaware of, it passes requests to the default POSIX handler that performs normal system calls. Unknown file descriptors include, for instance, those that describe network sockets.

The handlers The handlers are responsible for performing programmer-specified actions in response to accesses to a descriptor. Each handler implements a selection of POSIX functions related to I/O. On each call to Nib, the hub redirects the call to the handler implementation specified by the user.

The handler functions have access to all the arguments to the original system call and its handler's private state. A handler can either perform the operation itself (e.g., performing a `read()`) or perform some processing and pass it down to the next handler. This allows the programmer to create composable "pass through" handlers, for instance, to log accesses.

We have created a useful library of the handlers (and stacks of handlers), including a simple debug handler similar to `strace` and POSIX handler that implement the normal POSIX operations.

The most useful "utility" handler we have developed is for testing. The test handler automates the process of testing handler implementations. The Test handler creates a shadow copy of each file as it is opened. On subsequent calls, the handler executes every operation twice: once on the original file, and once on the shadow copy, using different handlers for each. It then compares the results and return codes of the two handlers. If they do not match, it signals an error. We use it verify that new handlers (e.g., the Quill handler) match the behavior of a reference implementation (e.g., the POSIX handler). It has been invaluable for testing.

So far, Nib only tracks file accesses, but extending it to handle directory and other IO operations would be straightforward.

3 Results

This section evaluates Quill's performance using a collection of microbenchmarks and database workloads. We begin by measuring Quill's bandwidth and latency relative the conventional OS-based interface. For all the experiments in this section, we use a 64 GB ramdisk to emulate a storage device built from fast, non-volatile memory on the processor's memory bus.

3.1 Latency

Quill adds overhead to each file access because the libraries must route the request to correct handler. This latency replaces the file system and operating system latency that normal system calls incur.

The total latency for a 4 KB read operation through Quill is 1.55 μs, on average. Of this, 0.43 μs is overhead in the hub routing the request to the Quill handler, checking permissions, and acquiring the necessary locks. The remaining 1.13 μs is the copy from the mapped file into the user's buffer. For comparison, a read through the conventional interface takes 3.42 μs on average.

In some cases, Quill reduces performance. In particular file operations that modify file system metadata are slower, since Quill must make a system call to effect the changes. The most common of these operations are appends to a file, since they must update the file's size. Under Quill a 1 byte append takes 0.860 μs. Un-

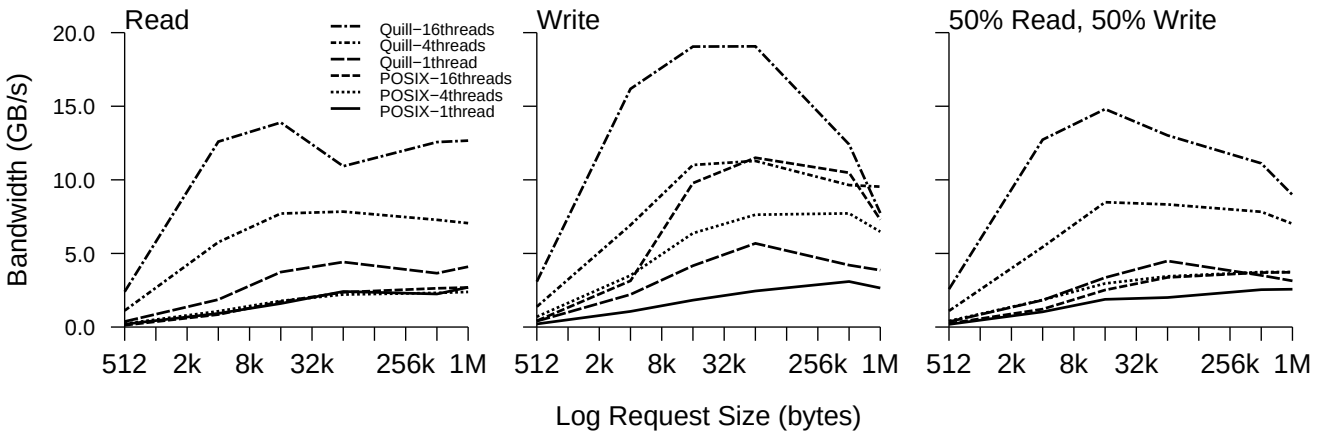

Figure 1: **Quill bandwidth** These graphs show aggregate bandwidth for various request sizes and access types. Quill improves bandwidth by up to 7×.

der the conventional interface it requires just 0.742 μs. For larger appends, the gap is reversed: A 4 KB append under Quill takes 1.07 μs, under the OS it takes 1.99 μs.

Quill performs significant file read ahead. When Quill `mmap()`s a file it creates a memory mapping twice the size of the file. If the file size grows, Quill does not need to update the mapping; it just needs to increase the file size. If the file eventually exceeds the mapping size, Quill doubles the size of the mapped region. As a result, some appends incur larger overheads, but, on average, the overhead is small.

3.2 Bandwidth

Placing storage on the processor's memory bus should enable high bandwidth. For instance, our dual-socket test machines have three DDR3 channels per socket for an aggregate 51.2 GB/s of memory bandwidth.

Figure 1 quantifies the bandwidth available from our ramdisk via the conventional OS-based interface and via Quill. Each graph shows the performance through the file system for Quill and the standard POSIX interface. The three figures measure performance for reads, writes, and a 50/50 combination of the two. The graphs measure the aggregate bandwidth of 1, 4, and 16 threads performing random accesses at many different sizes (as shown on the x axis). The accesses are all to a single 55 GB file.

Quill out-performs POSIX by a wide margin in all cases. The gains are especially large for small accesses. Quill can sustain 6.1 million 512 byte accesses per second compared to 845 thousand for POSIX. Two aspects of Quill's write performance are particularly interesting. First, performance is higher for writes than reads. We believe this is because the processor does not need to wait for stores to complete, as it must for loads. Second, performance for writes drops off precipitously for accesses over 64 KB in size for 16 threads. Here, the culprit is coherence traffic. When two threads happen to access the same portion of the file, coherence traffic slows the dramatically and increases the probability that another thread will start accessing the same region before they complete. The result is continual thrashing of lines between the processors' caches.

For Quill, bandwidth is better for writes than for reads because the processor does not need to wait for stores to complete, as it must for loads.

For this reason, the performance gap between 512 byte requests and 4 KB requests is very large: each 512 byte request must fetch and modify a page, while writes greater than 4k do not.

Quill's interaction with the memory system determines, in large part, its performance. The size of the file (or subset of the file) that Quill is accessing is critical in this regard. Figure 2 measures this effect by performing random 4 KB reads on various sized files. For files that fit in the processor's 8 MB L3 caches, Quill can sustain over 25 GB/s. As file size grows, performance quickly tapers off. The data also show that POSIX performs surprisingly well for writes to small files.

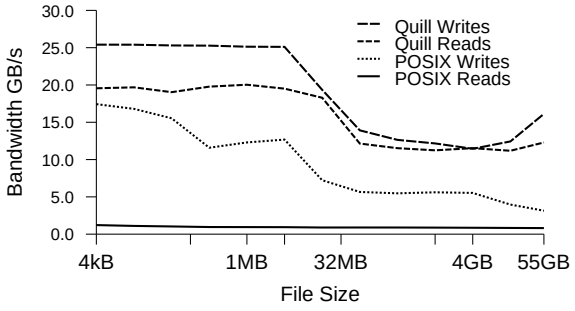

Figure 2: **File size and bandwidth** Larger files put more pressure on the TLB, as a result bandwidth rises as file size shrinks.

4 Related work

The notion of using fast, non-volatile solid-state memories on the processor's memory bus as a storage device has received a great deal of attention in recent years. BPFS [4] redesigns the file systems to leverage a memory-based interface to storage and provide strong transactional guarantees and improved performance. BPFS still requires system calls for file access. The Quill approach is complementary – Quill would improve performance for file data access while BPFS would accelerate updates to file system meta data. However, Quill dose not currently support the transactional guarantees that BPFS provides.

Other systems have exposed non-volatile memories on the processor bus but provide novel interfaces to the data. Rio Vista [6] and recoverable virtual memory [7] 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 groups [3, 9, 8] have developed frameworks for storing persistent objects in these memories. Like Quill, these schemes expose persistent memory directly to the applications, but they also require programmers to rewrite applications to take advantage of them. In return, the libraries offer more sophisticated, memory-like semantics.

Many research projects have taken parts or all of file systems and placed them in user space. In most cases, the goal is easy, safe extensibility – a programmer can extend an existing file system or implement a new one without modifying the operating system. Systems of this type include FUSE [5], Ufo [2] and the work of Zadox et. al. [10]. Quill also moves some file system functionality into user space, but the goal is performance rather than extensibility.

IEEE Standard 1285 [1] uses memory mappings for I/O devices rather than going through a dedicated I/O channel. However, these mappings exist at the kernel rather than user level. As a result, each transaction still incurs the cost of a protection boundary crossing. Furthermore, this standard does not follow standard POSIX semantics. Leveraging its potential speedups require non-trivial changes to the applications.

5 Conclusion

Quill allows programs to access files stored in fast, non-volatile memories without going through the operating and file system while still preserving the protections they provide. We have shown that Quill can reduce latency for small accesses by up 2x and improve bandwidth by 7x. Quill demonstrates that it is possible to expose most of the performance of these memories for file access without significant changes to the operating or file system.

References

[1] Ieee standard for scalable storage interface (s/sup 2/i). *IEEE Std 1285-2005*, 2006.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: the ufo global file system. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 1997. USENIX Association.

[3] 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.

[4] 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.

[5] http://fuse.sourceforge.net/.

[6] 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.

[7] 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.

[8] 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.

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

[10] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 1999. USENIX Association.