Persistent Memory Programming

ANDY RUDOFF

Andy Rudoff is a Senior Principal Engineer at Intel Corporation, focusing on non-volatile memory programming. He is a contributor to the SNIA NVM Programming Technical Work Group. His more than 30 years' industry experience includes design and development work in operating systems, file systems, networking, and fault management at companies large and small, including Sun Microsystems and VMware. Andy has taught various operating systems classes over the years and is a co-author of the popular *UNIX Network Programming* textbook. andy.rudoff@intel.com

In the June 2013 issue of *;login:*, I wrote about future interfaces for non-volatile memory (NVM) [1]. In it, I described an NVM programming model specification [2] under development in the SNIA NVM Programming Technical Work Group (TWG). In the four years that have passed, the spec has been published, and, as predicted, one of the programming models contained in the spec has become the focus of considerable follow-up work. That programming model, described in the spec as NVM.PM.FILE, states that persistent memory (PM) should be exposed by operating systems as memory-mapped files. In this article, I'll describe how the intended persistent memory programming model turned out in actual OS implementations, what work has been done to build on it, and what challenges are still ahead of us.

The Essential Background on Persistent Memory

The terms *persistent memory* and *storage class memory* are synonymous, describing media with byte-addressable, load/store memory access, but with the persistence properties of storage. In this article, I will focus on persistent memory connected to the system memory bus, like a DRAM DIMM, creating a class of non-volatile DIMMs known as NVDIMMs.

To further clarify what I mean by persistent memory, I am only speaking about NVDIMMs that allow software to access the media as memory (some NVDIMMs only support block access and are not covered here). This provides all the benefits of memory semantics, like CPU cache coherency, direct memory access (DMA) by other devices, and cache line granularity access which programmers can treat as byte-addressability. To provide these semantics, the media must be fast enough that it is reasonable to stall a CPU while an instruction is accessing it. NAND Flash, for example, is too slow to be considered persistent memory by itself, since access is typically done in block granularity and it takes long enough that context switching to allow another thread to do work makes more sense than stalling. Where hard drive accesses are typically measured in milliseconds, and NAND Flash SSD accesses are measured in microseconds, persistent memory accesses are measured in nanoseconds. Depending on the exact type of media, an NVDIMM may not be as fast as DRAM, but it is in the neighborhood.

Some NVDIMM products on the market today use DRAM as the media at runtime but automatically back up the contents to NAND Flash on power loss and restore the contents when the power returns. These products provide DRAM performance but also require additional components and an energy source to save the data, giving them a lower per-DIMM capacity and higher cost per gigabyte than DRAM. Emerging non-volatile media, like the 3D XPoint technology announced jointly by Intel and Micron in 2015, promises higher capacity at a price point lower than DRAM. Multiple terabytes per CPU socket are expected, making persistent memory interesting on multiple fronts: persistence, capacity, and cost [3].

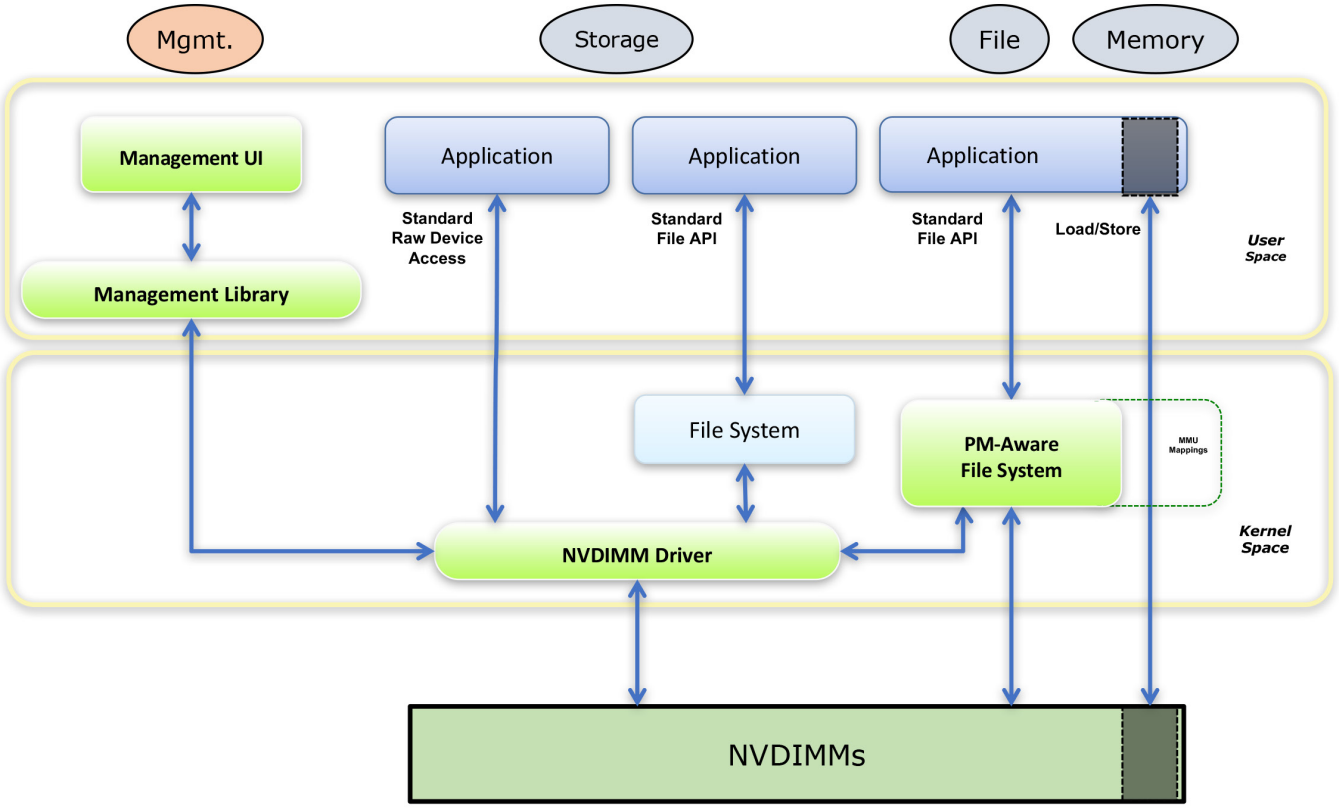

Figure 1: The SNIA persistent memory programming model

The Persistent Memory Programming Model

How does an application get access to persistent memory? Unlike volatile memory, the application needs a way to connect with specific persistent contents; persistent memory isn't anonymous like volatile memory; regions need names so applications can find them, just like files. And also like files, regions of persistent memory need permissions to control which applications have access to the persistent information. The entire point of the persistent memory programming model specified by the SNIA TWG was to recommend that operating systems use standard file semantics to provide naming, permissions, and memory-mapping of persistent memory.

Now that this has been implemented in multiple operating systems, including Linux and Windows, it seems very obvious, and you might wonder why a specification was even necessary. But four years ago when I wrote the first *;login:* article, there were multiple competing ideas on how to expose persistent memory, and software vendors were in danger of having to decide between incompatible programming models from different products. Instead, the ecosystem has unified nicely around the model shown in Figure 1.

The NVDIMM shown at the bottom of the figure represents the persistent memory installed in the system, potentially spread across many NVDIMMs, and potentially interleaved (striped) for performance by the memory controller. On Intel-based systems, the BIOS creates a table called the NVDIMM Firmware Interface Table (NFIT) that enumerates the NVDIMMs installed. This table was added to the ACPI specification in version 6.0 and continues to evolve as NVDIMMs evolve. As shown in the figure, some driver (or collection of drivers) consumes the NFIT information and takes ownership of the persistent memory, exposing it to management software (left side of the figure), potentially exposing it as traditional block storage which is emulated by the driver (middle part of the figure), and exposing it directly to applications through a *persistent memory aware file system* (the right side of the figure).

DAX

My definition of a *persistent memory aware file system*, like the one shown in Figure 1, is a file system that allows direct access to persistent memory without using the system page cache as it would for normal, storage-based files. This feature has been named DAX by the operating systems folks, short for *Direct Access*. Conveniently, both Linux and Windows use the same term for the same feature.

The persistent memory programming model, and the corresponding DAX feature, says persistent memory files can be mapped into memory using standard calls like `mmap()` on Linux or `MapViewOfFile()` on Windows. This results in the far-right arrow on Figure 1, where the application has direct load/store access to the persistence. Once these mappings are set up (and after any initial minor page faults that may be required to create the mappings in the MMU), this provides the shortest possible code path to persistence, allowing the applications to perform loads and stores on the persistent media directly with no kernel involvement. No interrupts, no context switching, no kernel code at all is required for media access.

Making Stores Persistent

Just as persistent memory is accessed using standard memory-mapped files, the steps for making changes persistent follow the same standards. On Linux (actually any POSIX-compliant system), the range-based `msync()` call or file-based `fsync()` call may be used to ensure changes are persistent. On Windows, the combination of `FlushViewOfFile()` and `FlushFileBuffers()` is used. These calls create a *store barrier*, a point after which the program can assume the previous changes it made to the persistent memory are actually persistent. Historically, this store barrier required the operating system to find dirty pages in the system page cache, flushing them to block storage, such as a disk. But since persistent memory doesn't use the page cache, the operating system need only flush the CPU caches, as appropriate, to get changes into the *persistence domain*. I define the persistence domain as the point along the data path taken by stores where they are considered persistent because that point is *power fail safe* (see Figure 2).

The dashed box in Figure 2 shows the persistence domain required by Intel platforms supporting persistent memory. At the platform level, any stores inside the dashed box are either on the DIMM, or still in the *write pending queue* (WPQ) in the memory controller, on their way to the DIMM. Either way, platforms supporting persistent memory are required to have enough stored energy to flush any stores inside the dashed box all the way to persistent media on power loss. This feature, flushing the stores the rest of the way on power failure, is known as *asynchronous DRAM refresh* (ADR) and has been a requirement of NVDIMM products since they first appeared a few years ago.

At the x86 instruction level, simply executing a store instruction is not enough to make data persistent, since the data may be sitting in the CPU caches indefinitely and could be lost by a power failure. Additional cache flush actions are required to make the stores persistent. The following table describes how each of these works.

Looking at Figure 2 and the instructions in the Table 1 might make you wonder why Intel didn't just make the CPU caches part

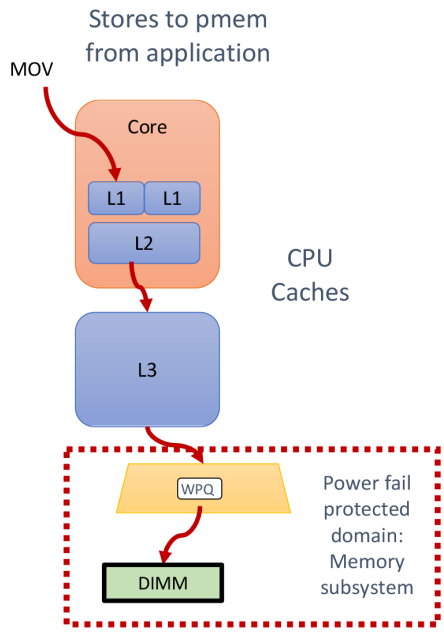

Figure 2: The path taken by a store, and the persistence domain (dashed box)

of the persistence domain. This is technically possible, producing the situation shown in Figure 2 but with the dashed box now including the CPU caches.

The problem with extending the persistence domain to include the CPU caches is that the x86 caches are quite large, and it would take more energy than the capacitors in a power supply can practically provide. This usually means the platform would have to contain battery. Requiring a battery for every server supporting persistent memory is not practical at this time, but it is certainly possible for companies, such as appliance vendors who ship custom hardware, to include a battery in their product. This would allow the cache flush instructions described in Table 1 to be skipped, but the SFENCE instruction would still be required as a store barrier—stores should be considered persistent only when they are globally visible, and that's what the SFENCE ensures.

Because some appliance vendors plan to use batteries as I've described, and because I hope that all platforms will someday include the CPU caches in the persistence domain, a property is being added to ACPI so that the BIOS can notify the operating system when the CPU flushes can be skipped. This allows the operating system to implement calls like `msync()` in the most optimal way.

User Space Flushing to Persistence

With the exception of WBINVD, the instructions I described in Table 1 are supported in user mode by Intel CPUs. Flushing a cache line using CLWB (or CLFLUSHOPT or CLFLUSH) and using non-temporal stores are all supported from user space.

CLFLUSH	This instruction, supported in many generations of CPU, flushes a single cache line. Historically, this instruction is serialized, causing multiple CLFLUSH instructions to execute one after the other, without any concurrency.
CLFLUSHOPT (followed by an SFENCE)	This instruction, newly introduced for persistent memory support, is like CLFLUSH but without the serialization. To flush a range, software executes a CLFLUSHOPT instruction for each 64-byte cache line in the range, followed by a single SFENCE instruction to ensure the flushes are complete before continuing. CLFLUSHOPT is optimized (hence the name) to allow some concurrency when executing multiple CLFLUSHOPT instructions back-to-back.
CLWB (followed by an SFENCE)	Another newly introduced instruction, CLWB stands for *cache line write back*. The effect is the same as CLFLUSHOPT except that the cache line may remain valid in the cache (but no longer dirty, since it was flushed). This makes it more likely to get a cache hit on this line as the data is accessed again later.
NT stores (followed by an SFENCE)	Another feature that has been around for a while in x86 CPUs is the non-temporal store. These stores are "write combining" and bypass the CPU cache, so using them does not require a flush. The final SFENCE instruction is still required to ensure the stores have reached the persistence domain.
WBINVD	This kernel-mode-only instruction flushes and invalidates every cache line on the CPU that executes it. After executing this on all CPUs, all stores to persistent memory are certainly in the persistence domain, but all cache lines are empty, impacting performance. In addition, the overhead of sending a message to each CPU to execute this instruction can be significant. Because of this, WBINVD is only expected to be used by the kernel for flushing very large ranges, many megabytes at least.

Table 1: x86 cache flush instructions for use with persistent memory

This could allow the flushing to persistence directly from user space, without calling into the kernel, a feature documented in the SNIA programming model spec as *Optimized Flush*. The spec describes Optimized Flush as optionally supported by the platform, depending on the hardware and operating system support. Despite the CPU support, it is important for applications to only use Optimized Flush when the operating system says it is safe to use. The operating system may require the control point provided by calls like `msync()` when, for example, there are changes to file system metadata that need to be written as part of the `msync()` operation.

Support for safe userspace flushing is an evolving feature in the current implementations. At the time of this writing, the DAX support in Windows, provided by the NTFS file system, includes unconditional support for Optimized Flush. Windows programs can ensure stores to persistent memory are persistent using instruction sequences like CLWB + SFENCE. On Linux, the two file systems that support DAX, ext4 and XFS, do not currently consider userspace flushing safe. While hoping to work out interfaces with these file systems that tell applications when Optimized Flush is safe, it is an ongoing discussion. Other file systems, like NOVA [4], a research project from UCSD, are designed from the start to support Optimized Flush but are not considered production ready yet. As an interim solution, Linux provides Device-DAX [5], which allows an application to open a persistent memory device (without a file system), memory map it, and utilize userspace flushes to make stores persistent.

To insulate application programmers from this complexity, and to keep them from having to research the current state of affairs while programming for persistent memory, the `libpmem` library provides a function which tells the application when Optimized Flush is safe. Programmers are strongly encouraged to use libpmem to make this determination and to use userspace flushing only when it is safe, falling back on the standard method of flushing stores to memory mapped files otherwise. The libpmem library is also designed to detect the case of the platform with a battery I described above, turning flush calls into simple SFENCE instructions instead. I've got much more to say about libraries below, and all the libraries I describe build on this logic to make sure they transparently depend on the most optimal type of flushing available to the program.

Persistent Memory Challenges

When a modern program changes any data structure in memory, the question of *atomicity* comes up. Is it possible for another thread to access the data structure and see the change only partially complete? With multithreaded programming, this issue is commonly solved using locks to protect data structures. Sometimes it is solved by using instruction sequences that guarantee atomicity in hardware. These issues have been around for years and are very familiar to programmers, library writers, and high-level language designers. In this context, the term *atomicity* really refers to *visibility*, protecting the changes made by one thread from becoming visible by other threads until the changes are complete. Adding persistent memory into this picture, the requirements change from simple atomicity to something more

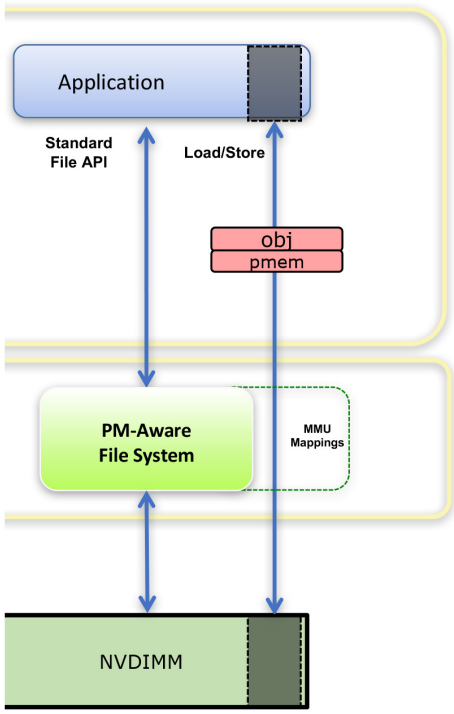

Figure 3: Using the libpmemobj library, which in turn uses the primitives in libpmem

like the ACID semantics required for database transactions on storage [6]. Not only do we want to keep other threads from seeing an incomplete change, we want to handle changes that are interrupted by power failures, program crashes, or exceptions. Everyone who starts writing programs to use persistent memory seems to immediately come to this conclusion: we need transactions that are power fail safe.

Before persistent memory existed, if a store was interrupted by something like power failure, the resulting memory state didn't matter much because it was volatile. But with persistent memory, it is important to understand what is guaranteed by hardware and what is left to software. On Intel, only an eight-byte store, aligned on an eight-byte boundary, is guaranteed to be failure atomic. That means if the store is interrupted by a power failure, the memory contents will contain the previous eight bytes, or the new eight bytes, but not some combination of the old and new data. Anything larger than eight bytes can be torn by power failure and must be handled by software. For example, if you want to update two eight-byte pointers in your program, and you want it to happen atomically, protecting those pointers with a lock will only help you prevent other running threads from seeing the partial update. A power failure might leave the update partially done, and there's no single instruction that will solve that—software must arrange for the update to be transactional by building on the eight-byte power-fail-atomic store provided by hardware. The logic for creating these transactions is a bit tricky, which points to the need for libraries or language features to provide them.

Another persistent memory challenge is more basic: managing the space. Since persistent memory regions are exposed as files, the file system primarily manages that space. But once the file is memory-mapped by an application, what happens within that file is completely up to the application. Functions like C's `malloc()` assume memory is volatile, offering no way on program start-up to reconnect with a persistent heap and taking no steps to make sure the heap is consistent in the face of failure. This adds space allocation to our list of requirements for persistent memory programming.

The need for location-independence is another challenge. Although it is technically possible to require that a range of persistent memory is always mapped at exactly the same address in a program, it can become impractical when the sizes of other mapped items change. A security feature known as *Address Space Layout Randomization* (ASLR) additionally causes operating systems to randomly adjust where libraries and files are mapped. Location-independence means that when one data structure in persistent memory refers to another using a pointer, that pointer must be somehow usable even when the file is mapped at a different address. There are several ways to achieve this, such as relocating pointers after mapping, using relative pointers instead of absolute pointers, or by using some type of *Object ID* (OID) to refer to persistent memory-resident data structures.

The NVM Libraries

The libraries produced by my team at Intel are designed to solve the challenges described above. They are meant as a convenience, not as a requirement for persistent memory programmers. Although I refer to them collectively by the single name NVML, they are really a suite of six libraries (with additional libraries already under development). The libraries are all open source, BSD-licensed, and developed in the open on GitHub. I'll describe the libraries here, but much more information is available at http://pmem.io, including man pages, blog entries, and lots of example code.

The libraries are written in C and are validated and ready for use on 64-bit Linux and Windows systems. Some Linux distros already contain the libraries in their repositories, allowing them to be installed with simple package management commands. Otherwise, you can clone the GitHub tree and use `make install` to install the library from source (details are on the Web site [7]).

Since these are C libraries, it is possible to call them from various languages. When using C, we provide some macros to try to help

catch common persistent memory programming errors, but C macros are never a replacement for full language integration. The C++ support recently released in libpmemobj (http://pmem.io/nvml/libpmemobj/; see below) is the cleanest, least error-prone way we have to do persistent memory programming. For this reason, if you're just beginning to explore persistent memory programming, the C++ examples are the best place to start.

Here's an overview of the suite of libraries in NVML. Many examples are available in the examples directory of the source on GitHub, but to save space I will limit my examples to the most commonly used library, libpmemobj.

libpmem: Basic Persistence Support

The libpmem library is small and fairly simple, containing the code that detects which types of flush instructions are supported by the CPU, as well as performance-tuned routines for copying ranges of persistence memory using the best instruction choices for the platform. As mentioned above, a routine that tells the caller whether Optimized Flush is safe is supplied (this routine is called `is_pmem()` for historical reasons—perhaps `optimized_flush_available()` would have been a better name in hindsight).

Even if you decide not to use any of the libraries I describe below, you might still decide to use libpmem (or steal the code) just to avoid the tedious development of code that detects supported instructions, the correct use of non-temporal stores, etc.

libpmemobj: General-Purpose Allocations and Transactions

This is probably the library you want. As you might guess, the "obj" in the name is short for `object`, but by that I mean the variable-sized blob of data referred to by the term *object storage,* not the class with methods in an object-oriented language. Figure 3 shows where this library sits in the programming model. Like all the persistent libraries in the NVML suite, this library builds on the primitives provided by libpmem.

The libpmemobj library allows persistent memory *objects* to be allocated in a way that is power fail safe, allows referring to them by Object IDs (OIDs), which are location-independent, and allows making an arbitrary number of changes atomic by encompassing the changes in a transaction. The library is multithread safe and optimized for multithread scalability (by doing things like maintaining per-thread allocation caches).

As mentioned above, the C++ support in this library provides the cleanest, easiest-to-use interfaces, so I'll use a C++ example. The classic persistent memory example is to link something into a linked list (a *queue* in this example, taken verbatim from the `queue.cpp` example in the NVML examples area), where multiple operations are required to be done as a transaction. The example code below starts by creating a class which defines the struct `pmem_entry`, the entries on the queue:

```
class pmem_queue {

        /* entry in the list */
        struct pmem_entry {
                persistent_ptr<pmem_entry> next;
                p<uint64_t> value;
        };
        /* … */
```

Notice the `persistent_ptr` smart pointer template. This indicates a pointer to an object in persistent memory, namely the next item in the persistent queue. These are the location-independent OIDs I mentioned earlier. Also notice the `p<>` persistent property in the above declaration, used to indicate fields that reside in persistent memory. The result of these C++ declarations is that the code to atomically allocate a new entry, initialize it, and link it into the queue can be done as follows:

```
        /*
         * Inserts a new element at the end of the queue.
         */
        void
        push(pool_base &pop, uint64_t value)
        {
             transaction::exec_tx(pop, [&] {
                  auto n = make_persistent<pmem_entry>();

                  n->value = value;
                  n->next = nullptr;

                  if (head == nullptr) {
                           head = tail = n;
                  } else {
                           tail->next = n;
                           tail = n;
                  }
             });
        }
```

The above push operation is transactional. More specifically, the code in the C++ *lambda*, indicated by `[&] {…}`, is transactional, meaning if the program or the machine crashes during the execution of that code, libpmemobj automatically rolls any partially done changes back (this includes the allocation done by the `make_persistent` call).

There are many more details available for this example, as well as others, on the pmem.io Web site. The main point of the short example above is to show that, with no compiler or language changes, libpmemobj provides a flexible allocation and transaction mechanism for persistent memory.

libpmemblk and libpmemlog: Support for Specific Use Cases

In addition to libpmemobj and its flexible transaction support, two other libraries target specific use cases. The library libpmemblk is written specifically to maintain a large array of persistent memory blocks, all the same size. This is useful, for example, when an application is managing a block cache. The block size provided by the library is flexible, supporting blocks 512-bytes and larger.

Similarly, the library libpmemlog is written for a specific use case where the application frequently appends to a private log file, one that is read rarely, like during crash recovery. This library takes the relatively long file system append path through the kernel and turns it into a very short memory copy in persistent memory, followed by an atomic pointer adjustment.

Both of these specific use cases are easily solved using the more flexible libpmemobj, but the point of libpmemblk and libpmemlog is they provide APIs that constrain the caller, allowing the library to assume specific cases and optimize for them.

libmemkind: The Volatile Use of Persistent Memory

With the large capacity and cheaper-than-DRAM price points expected for emerging persistent memory products, many volatile use cases have come up. These are cases where the application places some data structures in persistent memory to avoid a large DRAM footprint, but the application doesn't really care that the memory is persistent—it is just using it as a second tier of volatile memory. When NVML was first developed, we created a library called libvmem ("vmem" for *volatile memory*). Since then, another more general library for volatile use cases has been open sourced on GitHub [8]. Some projects have already been written to our libvmem interfaces, but for all future development of volatile use cases, we recommend using libmemkind.

Conclusion

The ideas I outlined in 2013 have come true and have matured into a fairly complete programming model, resulting at the operating system level in the DAX feature for both Windows and Linux (and potentially other operating systems beyond the scope of this article). Next, libraries have been built on that basic model to provide application developers with a menu of APIs to choose from as they leverage the benefits persistent memory has to offer. There's still a long list of interesting and fruitful work to be done, integrating persistent memory support into additional languages and libraries (see our GitHub area at https://github.com/pmem for numerous works-in-progress in this space).

References

[1] A. Rudoff, "Programming Models for Emerging Non-Volatile Memory Technologies," *;login:*, vol. 38, no. 3 (June 2013): https://www.usenix.org/system/files/login/articles/08_rudoff_040-045_final.pdf.

[2] "SNIA NVM Programming Technical Work Group": http://www.snia.org/forums/sssi/nvmp.

[3] "3D XPoint™ Technology Revolutionizes Storage Memory": https://www.youtube.com/watch?v=Wgk4U4qVpNY.

[4] J. Xu and S. Swanson, "NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*: https://www.usenix.org/system/files/conference/fast16/fast16-papers-xu.pdf.

[5] Dan Williams, "Device-DAX": https://lists.gt.net/linux/kernel/2434768.

[6] T. Haerder, A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys,* vol. 15, no. 4 (December 1983), pp. 287–317.

[7] NVML install instructions: https://github.com/pmem/nvml/blob/master/README.md.

[8] libmemkind: https://github.com/memkind.