HARDWARE

Programming Models for Emerging Non-Volatile Memory Technologies

ANDY RUDOFF

Andy Rudoff is an Enterprise Storage Architect at Intel. He has more than 25 years of experience in operating systems internals, file systems, and networking. Andy is co-author of the popular *UNIX Network Programming* book. More recently, he has focused on programming models and algorithms for Non-Volatile Memory usage. andy.rudoff@intel.com

Upcoming advances in Non-Volatile Memory (NVM) technologies will blur the line between storage and memory, creating a disruptive change to the way software is written. Just as NAND (Flash) has led to the addition of new operations such as TRIM, next generation NVM will support load/store operations and require new APIs. In this article, I describe some of the issues related to NVM programming, how they are currently being resolved, and how you can learn more about the emerging interfaces.

The needs of these emerging technologies will outgrow the traditional UNIX storage software stack. Instead of basic read/write interfaces to block storage devices, NVM devices will offer more advanced operations to software components higher up in the stack. Instead of applications issuing reads and writes on files, converted into block I/O on SCSI devices, applications will turn to new programming models offering direct access to persistent memory (PM). The resulting programming models allow applications to leverage the benefits of technological advances in NVM.

The immediate success of these advances and next generation NVM technologies will rely on the availability of useful and familiar interfaces for application software as well as kernel components. Such interfaces are most successful when key operating system vendors and software vendors agree on an approach, terminology, and a strategy for widespread adoption. I will describe some of the more interesting changes on the horizon for NVM programming and outline new solutions to address these changes. Finally, I'll explain how the industry is driving commonality for these interfaces using a Technical Work Group (TWG) recently formed by the Storage Networking Industry Association (SNIA).

NVM Programming Models

Although there are surely countless possible programming models for using NVM, I'll focus on the four most relevant models. The first two represent the most common storage interfaces in use for many decades, which I will call NVM Block Mode and NVM File Mode. The remaining two models, which I will call PM Volume Mode and PM File Mode, specifically target the emergence of persistent memory.

NVM Block Mode

The diagram in Figure 1 represents a portion of a common software stack, where the dashed red line represents the interface providing the NVM Block Mode programming model.

There are many variations on the exact details of the software stack both above and below the dashed red line in Figure 1. The point of the diagram is to illustrate how the interface works, not to focus on a specific set of software components using it. As shown, the NVM Block Mode programming model provides the traditional block read/write interface to kernel modules such as file systems and, in some cases, to applications wanting to use the block device directly (for example, by opening /dev/sda1 on a Linux system).

Why is this decades-old interface interesting for a discussion of NVM Programming? Advances in NVM technology make it interesting by motivating change to an interface that

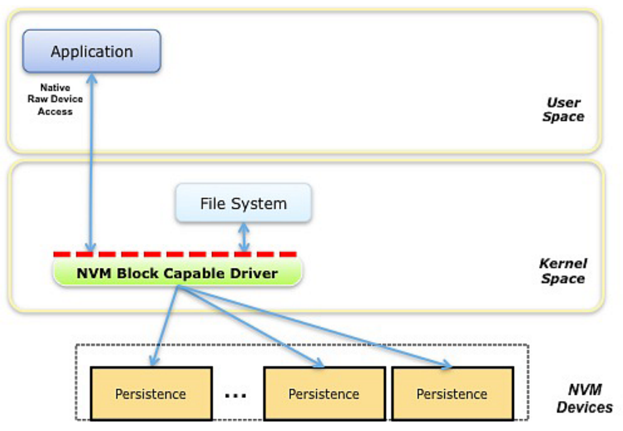

Figure 1: The NVM Block Mode interface, depicted by the red dashed line

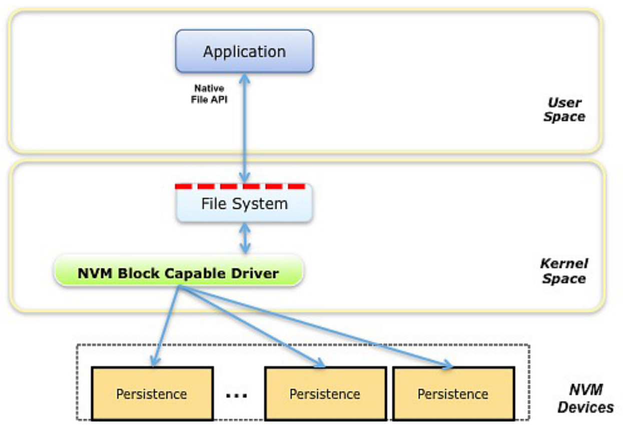

Figure 2: The NVM File Mode interface, providing the usual file operations enhanced with additional NVM operations

has otherwise barely changed in many years. A fairly recent but widely adopted example is the addition of software support for the TRIM command on the modern solid state drive (SSD). The TRIM command allows file systems to inform an SSD which blocks of data are no longer in use. Although useful for virtual arrays that support thin provisioning, this information was not necessary at the basic drive level until the emergence of Flash drives, where the wear-leveling management required by the drive can benefit from it [1].

Just as the simple block read/write interface to the driver required extensions to support TRIM, other emerging NVM features, such as support for atomic operations, will require similar broadening of the interfaces [2]. Additionally, simply exposing certain attributes of an NVM device to applications may prove just as useful. Applications can optimize I/O for performance using information on optimal I/O sizes, supported granularity of I/O, and attributes such as powerfail write atomicity. By arriving at common definitions for these extended operations and attributes, the NVM industry can provide a more effective ecosystem for software writers to develop NVM-aware applications that better leverage NVM features across multiple system types. Exactly how this ecosystem is created is covered later in this article.

NVM File Mode

Figure 2 illustrates the NVM File Mode programming model. Again, the red dashed line depicts the interface of interest, and the rest of the diagram is simply one possible layout of software components to show how the interface is used.

In this mode, a common file system such as ext4 on Linux uses a block-capable driver in the usual fashion. As with NVM Block Mode, this long-standing programming model will gain some incremental additions to the standard file API to allow applications to take advantage of advances in NVM.

For an example of how the NVM File Mode can evolve to benefit applications, consider the double write technique used by the MySQL database. This technique is used to protect database tables from corruption due to system interruption, such as a power failure. These tables are typically stored in files, and the double write technique is used to protect MySQL from partial page writes, that is, the write of a page of data that is torn by a system interruption. If the MySQL application were able to discover that writes of up to a certain size (the database page size) are guaranteed untearable by a system interruption, the double writes could be avoided [3]. Providing an interface for applications to request the powerfail write atomicity of the underlying NVM allows applications like MySQL to discover these attributes automatically and modify their behavior accordingly. Without this interface system, administrators must determine obscure attributes of the storage stack and edit MySQL configuration files to inform the application of these attributes.

PM Volume Mode

In Figure 3, the block diagram looks similar to NVM Block Mode, but here the device is not just NVM, but PM-Capable NVM. To be PM-capable means the NVM is usable directly via the processor load and store instructions. Although one might argue that any storage element might be connected to the system in a way the processor can load directly from it, the practicality of stalling a CPU while waiting for a load from technology such as NAND Flash prevents such a direct connection. But more advanced NVM technology, as well as innovative caching techniques, is allowing a generation of PM devices to emerge.

PM Volume Mode, as shown in the diagram, allows a kernel component to gain access to the persistence directly. The diagram shows a PM-Aware Kernel Module communicating with the NVM driver. This interface allows the kernel module to fetch the physical address ranges where the PM is accessed. Once the

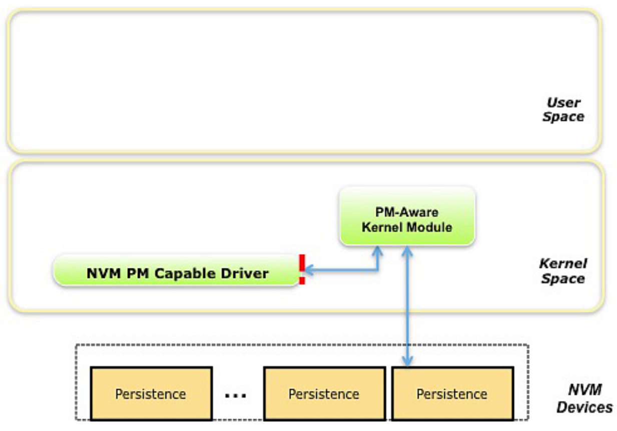

Figure 3: The PM Volume Mode interface, allowing a PM-Aware Kernel Module to look up the physical addresses in the volume

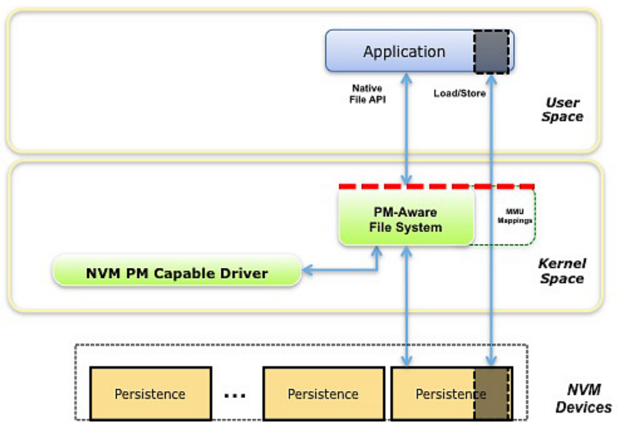

Figure 4: The PM File Mode interface, depicted by the red dashed line, providing the standard file operations but with memory mapped files going directly to NVM

kernel has that information, it need not ever call back into the NVM driver, instead accessing the PM directly as shown by the blue arrow in the diagram connecting the PM-Aware Kernel Module directly with the persistence in the NVM device. This fairly raw access to the PM allows the kernel module to add its own structure to the ranges of persistence and use it however it chooses. Examples include using the PM as a powerfail-safe RAID cache, a persistent swap space, or, as we'll discuss next, a PM- Aware File System.

A product providing PM may also provide NVM Block Mode, or any of the other modes; these programming models are not mutually exclusive, I am simply describing them separately because they are independent of each other.

PM File Mode

Our fourth NVM programming model is shown in Figure 4. PM File Mode is similar to the NVM File Mode we described earlier, but in this case the file system is specifically a PM-Aware File System.

Notice the interfaces associated with this programming model (the red dashed line again). The PM-Aware File System typically provides all the same file APIs as a traditional file system. In fact, a PM-Aware File System is likely created by enhancing an existing file system to be PM-aware. The key difference is in what happens when an application memory maps a file. As shown by the far right blue arrow in the diagram, memory mapping a file allows the application direct load/store access to the PM. Once the mapping is set up, accesses to the PM bypass any kernel code entirely since the MMU mappings allow the application physical access. This diverges from a traditional file system where memory mapped files are paged in and out of a page cache.

Why a file system? Why does this programming model center around the file APIs? This will be explained in the next section where I focus on persistent memory.

Persistent Memory

Now that all four NVM programming models have been described, I'll turn to the details of persistent memory. PM deserves special attention because, unlike the incremental improvements occurring with the NVM Block and NVM File modes, PM offers a much more disruptive change. Just as the ecosystem reacted to the change from faster clock rates on single CPUs to higher core counts (forcing performance-sensitive applications to revise their algorithms to be multithreaded), PM will cause the ecosystem to rethink how data structures are stored persistently. PM offers a combination of persistence and the ability to access the data structures without first performing block I/O and then converting the blocks of data into memory-based structures. As with any new technology, the benefits of PM come with a set of new and interesting challenges.

Allocation of Persistent Memory

Every C programmer is familiar with the standard malloc interface for dynamically allocating memory:

```
ptr = malloc(len);
```

Given a length in bytes, an area of RAM is returned to the calling process. This well-worn interface is simple and easy to use, although one could argue it is also easy to misuse, causing hours of debugging memory leak and memory corruption issues. But with so many decades of use and millions of lines of C code depending on malloc, a natural way to expose PM seems to be simply adding another version of malloc:

```
ptr = pm_malloc(len);  /* the naïve solution */
```

This simple solution gives the application programmer a choice between allocating RAM (using malloc) and PM (using pm_malloc), which seems like a fine solution on the surface but quickly falls short on further examination. Presumably the application was allocating PM in order to store something in it persistently, since that's the whole point. So the application will need a way to get back to that range of PM each time it is run, as well as each time the system is rebooted or power cycled. To allow this, the PM must be given a name that the application can provide to reconnect with it.

Many naming schemes for PM are possible, from some sort of numeric object ID to URL-like strings. But once the PM is named, the next issue immediately comes up: How to determine if an application has permission to connect to an area of PM? Like naming, many permission schemes are possible, but as you delve into the management of PM, you start to find even more issues, such as how does the system administrator change the permissions on PM? How are old areas of PM removed or renamed? Even more importantly, how are areas of PM backed up to protect against hardware failure? For traditional storage, the file system semantics provide answers to all these questions, so even though PM is much more like system memory, exposing it like files provides a convenient solution. The file API provides a natural namespace for PM ranges—ways to create, delete, resize, rename the ranges—and many off-the-shelf backup tools will simply work. The net effect of this approach is that if an application wants volatile memory, it calls malloc, and if it wants PM, it opens (or creates) a file on a PM-Aware File System and uses the mmap API to map it into its address space.

Making Changes to Persistent Memory Durable

With volatile memory, there's no need to worry about the durability of stores because all memory-resident information is assumed lost when the application or system shuts down. But with storage, that data stored is often cached and must be committed to durable media using some sort of synchronization API. For memory mapped files, that API is msync [4]. Although a strict interpretation of the traditional msync call is that it flushes pages of data from a page cache, with PM the application has direct load/store access without involving the page cache. The msync call for PM is instead tasked with flushing the processor caches, or any other intermediate steps required to make sure the changes are committed to the point of being powerfail safe.

Position-Independent Data Structures

With PM available to applications, for those applications to store data structures such as arrays, trees, heaps, etc. is convenient. On start-up, the application can use the file APIs described above to memory map PM and immediately access those data structures; however, there's an issue around position-independence of the data structures as shown in Figure 5.

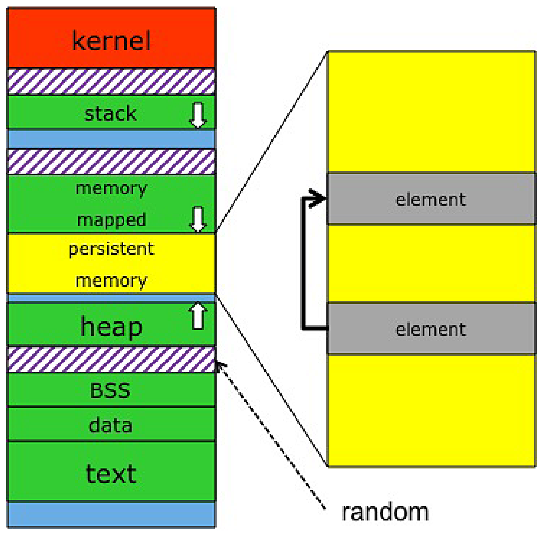

Figure 5: Typical process address space layout, with slightly different positions each run due to the randomly sized areas

On the left side of the diagram, the typical address space layout of a process on a UNIX system is shown. Because PM is accessed as memory mapped files, it gets mapped into the area with the other memory mapped files, such as shared libraries (growing downwards). The striped areas on many systems are the spaces between ranges, such as stack and memory mapped files, and the exact sizes of the striped areas are often random. This is a security feature designed to make some types of attacks more difficult [5]. For data structures stored in PM, the result is that any pointers, like the one depicted on the right side of the diagram, will be invalid between any two runs of the application. Of course, this isn't a new problem; storing absolute pointers in a memory mapped file has always been problematic, but the emergence of PM means this is expected to be a much more common problem to solve.

The obvious solution, to only store relative pointers in PM, can be somewhat error prone. Every pointer dereference must account for the fact that the pointer is relative and add in some sort of base offset. Higher-level languages with runtime virtual machines, such as Java, or languages without explicit pointers, may be able to handle this transparently, which is an area of research, but the first goal is to expose PM in the basic low-level system call and C environment. One potential answer is the idea of based pointers, a feature available in some C compilers, such as Microsoft's C++ compiler [6]. With this feature, a new keyword, `__based`, is added to the language so that declarations such as this linked list example are possible:

```
void *myPM;

struct element {
      …
      struct element __based (myPM) *next;
}
```

The result is that when the PM file is memory mapped, the location of the PM area is stored in the pointer `myPM`, and due to the `__based` declaration, every time the next field is dereferenced, the compiler generates code to adjust it by the value of `myPM`, creating a convenient position-independent pointer for the programmer.

So far I've described only one of the many issues around position-independent data structures and the storing of data structures in PM. Fortunately, there is quite a bit of research going on in academia on this topic, and two bodies of work demand special mention here. The NV-Heaps work [7] and the Mnemosyne project [8] both attack the issue described here in different and innovative ways. These works also look into language extensions and other runtime solutions to these problems and are recommended reading for anyone interested in PM.

Error Handling

The main memory of a computer system is typically protected against errors by mechanisms such as error correcting codes (ECC). When that memory is used by applications, such as memory allocated by calling malloc, applications do not typically deal with the error handling. Correctable errors are silently corrected—silently as far as the application is concerned (the errors are often logged for the administrator). Uncorrectable errors in which application memory contents are corrupted may be fixed by the OS if possible (for example, by re-reading the data from disk if the memory contents were not modified), but ultimately there are always cases in which the program state is known to be corrupted and it is not safe to allow the program to continue to run. On most UNIX systems, the affected applications are killed in such cases, the UNIX signal SIGBUS most often being used.

Error handling for PM starts off looking like memory error handling. Using Linux running on the Intel architecture as an example, memory errors are reported using Intel's Machine Check Architecture (MCA) [9]. When the OS enables this feature, the error flow on an uncorrectable error is shown by the solid red arrow in Figure 6, which depicts the mcheck module getting notified when the bad location in PM is accessed.

As mentioned above, sending the application a SIGBUS allows the application to decide what to do; however, in this case, remember that the PM-Aware File System manages the PM and that the location being accessed is part of a file on that file system. So even if the application gets a signal preventing it from

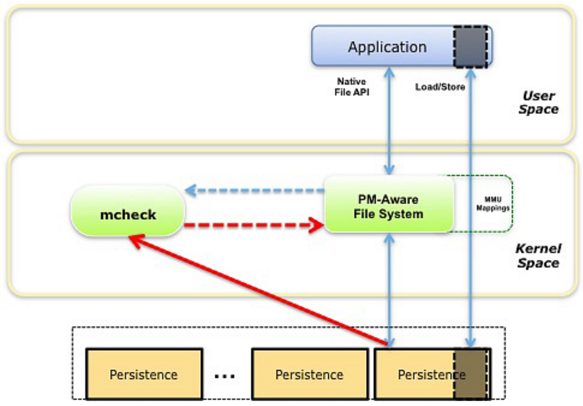

Figure 6: The machine check error flow in Linux with some proposed new interfaces depicted by dashed arrows

using corrupted data, a method for recovering from this situation must be provided. A system administrator may try to back up the rest of the data in the file system before replacing the faulty PM, but with the error mechanism we've described so far, the backup application would be sent a SIGBUS every time it touched the bad location. In this case, the PM-Aware File System needs a way to be notified of the error so that it can isolate the affected PM locations and then continue to provide access to the rest of the PM file system. The dashed arrows in Figure 6 show the necessary modification to the machine check code in Linux. On start-up, the PM-Aware File System registers with the machine code to show it has responsibility for certain ranges of PM. Later, when the error occurs, the PM-Aware File System gets called back by the mcheck module and has a chance to handle the error.

Here I've provided an abbreviated version of the error-handling story for PM. This is still a developing area and I expect the error-handling primitives to continue to evolve.

Creating an Ecosystem

The rapid success of PM and other emerging NVM technologies depends on creating an effective ecosystem around new capabilities as they become available. If each operating system vendor and hardware vendor creates its own divergent API for using these features, the ability of software vendors, kernel programmers, and researchers to exploit these features becomes limited. To avoid this, a group of industry leaders has worked with SNIA to create the NVM Programming Technical Work Group. Here is how the TWG describes itself:

The NVM Programming TWG was created for the purpose of accelerating availability of software enabling NVM (Non-Volatile Memory) hardware. The TWG creates specifications, which provide guidance to operating system, device driver, and application developers. These specifications are vendor agnostic and support all the NVM technologies of member companies. [10]

The TWG is currently working on a specification describing the four NVM programming models I covered in this article. The specification will cover the common terminology and concepts of NVM, including PM, and it will describe the semantics of the new actions and attributes exposed by emerging NVM technology. But the TWG intentionally stops short of defining the APIs themselves. This approach of providing the semantics but not the syntax is done to allow the operating systems vendors to produce APIs that make the most sense for their environments. The TWG membership includes several operating system vendors that are actively participating in the definition of the programming models. In fact, in the few months that the TWG has existed, a remarkable number of companies have joined. As of this writing, the membership list is: Calypso Systems, Cisco, Dell, EMC, FalconStor, Fujitsu, Fusion-io, Hewlett-Packard, Hitachi, Huawei, IBM, Inphi, Integrated Device Technology, Intel, LSI, Marvell, Micron, Microsoft, NetApp, Oracle, PMC-Sierra, QLogic, Samsung, SanDisk, Seagate, Symantec, Tata Consultancy Services, Toshiba, Virident, and VMware. (This list illustrates the scale of the collaboration and will surely be out-of-date by the time this article is published.)

Summary

A software engineer will see countless incremental improvements in hardware performance, storage capacity, etc. through a long career. That same career will witness high impact, game-changing developments only a few times. The transition of NVM from something that looks like storage into something that looks more like memory is one such disruptive event. By pulling the industry together to define common ground, we can enable software to rapidly and fully exploit these new technologies. The SNIA NVM Programming Technical Work Group is our effort to make this happen, and it has gained considerable industry traction in just a few months.

References

[1] Intel, Intel High Performance Solid-State Drive—Advantages of TRIM: http://www.intel.com/support/ssdc/hpssd/sb/CS-031846.htm.

[2] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, D.K. Panda, “Beyond Block I/O: Rethinking Traditional Storage Primitives,” 17th IEEE International Symposium on High Performance Computer Architecture (HPCA-17), February 2011, San Antonio, Texas.

[3] Peter Zaitsev, “Innodb Double Write,” MySQL Performance Blog (Percona): http://www.mysqlperformanceblog.com/2006/08/04/innodb-double-write/.

[4] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 edition—msync: http://pubs.opengroup.org/onlinepubs/009695399/functions/msync.html.

[5] Shacham et al., “On the Effectiveness of Address-Space Randomization,” Proceedings of the 11th ACM Conference on Computer and Communications Security, 2004, pp. 298-307.

[6] Microsoft Developer Network, Based Pointers (C++): http://msdn.microsoft.com/en-us/library/57a97k4e(v=vs.80).aspx.

[7] J. Coburn et al., “NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories,” The 16th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’11), March 2011, Newport Beach, California.

[8] Haris Volos, Andres Jaan Tack, Michael M. Swift, “Mnemosyne: Lightweight Persistent Memory,” The 16th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’11), March 2011, Newport Beach, California.

[9] Intel, Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, Chapter 15, March 2013: http://download.intel.com/products/processor/manual/325462.pdf.

[10] Storage Networking Industry Association, Technical Work Groups: http://www.snia.org/tech_activities/work/twgs.