Introducing the Programming Model of Intel Optane DC Persistent Memory

Describes how the properties of persistent memory differ from conventional storage devices

Introduces the programming model for persistent memory

Explains the challenges that face software developers based on the new programming model

Provides a basic coverage of the supporting libraries for persistent memory

Peng Liu

Abstract

The paper introduces the programming model for Intel Optane DC Persistent Memory. The paper describes the challenges to application programming to best take advantage of the benefits of persistent memory, and provides an overview of the supporting libraries to ease the development work.

The target audience for this paper is developers who are considering using persistent memory in their applications. It is also valuable for people who are involved in related activities and want an overview of this new technology.

Contents

Introduction

Intel Optane DC Persistent Memory (DCPMM or PMEM) is a new generation of nonvolatile memory (NVM) technology that is fast enough for processors to access stored data directly, without high latency and without a tremendous reduction in performance. Like flash memory, PMEM is nonvolatile storage; yet, like Dynamic Random Access Memory (DRAM) it is also byte-addressable using regular memory instructions.

This change in architecture blurs the line between traditional storage devices and DRAM. Since it can be used as either, PMEM offers great flexibility, however, it also creates a challenge for software developers.

Three key challenges faced by software developers are the following

- First, existing operating systems are designed for a strict bifurcation of devices into memory (fast, random-access, volatile structures erased on reboot), and storage (persistent, slow, block-based devices). Neither of these approaches exposes the full power of PMEM to programmers.

  To take full advantage of PMEM, a new programming model[1] is introduced as the standard method, which requires a disruptive change to the way programs are written.
- Second, PMEM does not typically replace either memory or storage. Instead, it is a third tier used in conjunction with memory and storage. Program design needs to take the storage level changes into consideration.
- Third, there are still many programming issues unique to PMEM to be addressed, which is by no means a trivial task. Supporting libraries have been developed (and are continued to be developed) for various PMEM use scenarios, as we discuss in this paper.

Programming Model

Like DRAM and storage, PMEM needs to manage space allocation so it can be shared by different applications. Unlike volatile memory, the application needs to name regions that are allocated so the application can find them after a restart. In addition, regions of PMEM need permissions to control which applications have access to their contents. Rather than reinvent the wheel, the PMEM programming model uses standard file semantics to provide naming and permissions.

To access a storage device, an application has several choices: the standard way, direct I/O, or memory-mapped I/O, as shown in Figure 1 on page 4. DMA operations are inevitable for data transfer using all methods except for Direct Access (DAX).

Devices like hard drives and SSDs cannot be read/written like memory. For them, it is necessary to set up a Direct Memory Access (DMA) transfer, and the transfer itself is nontrivial. Data is usually cached by kernel in page caches.

[1] Andy Rudoff. Programming models for emerging non-volatile memory technologies. USENIX ;login:, 38(3), June 2013.

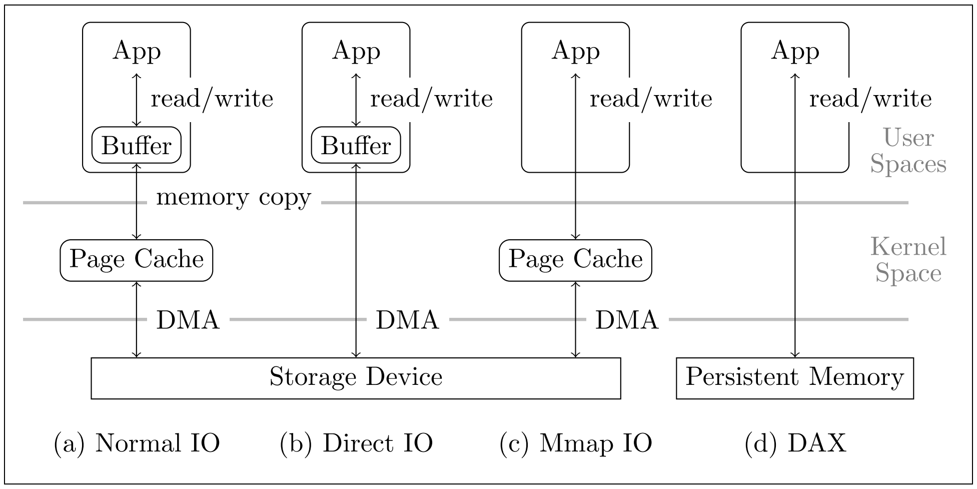

Figure 1 Comparison of device accessing methods

Because PMEM is directly accessible and fast, with throughput similar to DRAM, it cannot benefit from page caches as much as mechanical hard drives. Moreover, the usual I/O stack of block devices is not efficient enough.

We need a new way to expose PMEM directly to the user without any intervening layers. This is done by file memory-mapping plus DAX, which is a file system feature that enables applications to access PMEM directly, without using the system page cache as it would for normal, storage-based files.

Figure 2 shows the PMEM model which describes how applications interact with file system and PMEM to complete read/write operations.

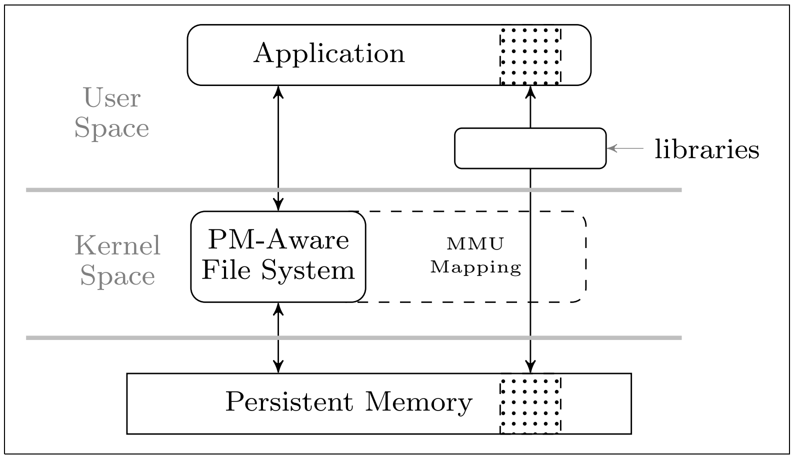

Figure 2 PMEM programming model

Since PMEM is memory-mapped, accesses to PMEM is completed merely by memory read/write; and because the underlying file system supports DAX, those accesses are the same as memory access in that they go directly to the PMEM device.

Using Linux as an example, the typical process of the model would be:

1. Build a file system supporting DAX on PMEM devices and mount it with DAX enabled
2. Create a file in the file system for the application's use
3. The application opens the file and memory-maps it into its address space

4. The application reads and writes the memory-mapped area to access the PMEM

Programming challenges

The PMEM programming model is only half of the story. It handles the direct accessibility of PMEM by setting up the data access path, but there are more complications when using non-volatility of PMEM.

Making changes durable

With volatile memory, squeezing the last drop of performance is the main concern, and hence caches and buffers are used all along the way from processors to memory. Since PMEM is also connected to processors, the caches and buffers are applied to it as well. To avoid data loss caused by power failure or system crash, the data must be flushed back to durable media frequently and quickly.

For memory-mapped synchronization, the standard API is `msync`, which flushes pages from a page cache. However, since PMEM doesn't use page cache, an application only needs to flush the CPU caches associated with the *persistent domain* (PD), a term used to describe that portion of a platform's data path where stores are safe from power-failures.

Figure 3 shows the data path taken by a store (MOV) to PMEM. The larger dashed box in the figure represents the PD on a platform that is able to flush the write pending queue (WPQ) automatically on power-fail.

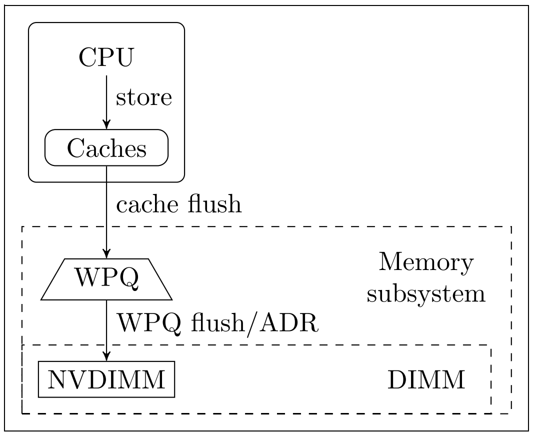

Figure 3 Persistent domain

The platform-level feature that performs this flushing is called Asynchronous DRAM Refresh (ADR). On machines without ADR, PD is the smaller dashed box in the figure. PMEM is usually mapped with cache enabled for better performance. Thus the store typically ends up in the CPU caches. Cache flush instructions can move stores out of the CPU caches, but the data may be left in the WPQ. The data is still at risk of being lost on machines not capable of ADR. Therefore, PCOMMMIT was added to the x86 instruction set to ensure that stores reach the PMEM DIMM.

PCOMMMIT is no longer needed, because all current platforms support PMEM and ADR. Therefore, cache flushing instructions alone are enough to ensure stores reach the PD. Instead, two new optimized cache flushing instructions, Cache Line Write Back (CLWB) and CLFLUSHOPT (Optimized CLFLUSH) [2], have been introduced to provide a high-performance cache flushing method on ADR-capable platforms.

Figure 4 shows an example of an instruction sequence for storing values (10 and 20) to PMEM with ADR supported.

```
MOV X1, 10  } Store 10 to X1, 20 to X2
MOV X2, 20  }
...         } Stores to X1 and X2 are globally
MOV R1, X1  } visible, but still potentially volatile
...
CLWB X1     } X1 and X2 flushed from caches
CLWB X2     }
...
SFENCE      } Ensures stores are ordered
            } ahead of any following code
```

Figure 4 CPU cache flushing

Making changes atomic

With multi-threaded applications, any data structure in memory accessed by one thread is at risk of change by another thread. This results in a partially complete update being returned to the former thread. In this context, it is said that atomicity is broken.

Broken atomicity is commonly solved by lock functions that can make ongoing changes invisible to other threads. It is more difficult to keep atomicity with PMEM, because a store may be interrupted by something like a power failure. For volatile memory, the memory state after reboot does not matter much because it itself is volatile. But the contents of PMEM are left in an inconsistent state and must be repaired by software.

On Intel processors, 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 undone, and there is no single instruction that will solve this issue. The software must arrange for the update to be transactional, by building on the eight-byte power-fail-atomic store provided by hardware.

Persistent Memory allocation

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 such as C's `malloc()` assume memory is volatile. Therefore, on program start-up it offers no way to reconnect with a persistent heap and takes no steps to ensure the heap is consistent in the face of failure. This makes space allocation a requirement for persistent memory programming.

Making data position-independent

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 somehow be usable even when the file is mapped to 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 PM-resident data structures.

Persistent Memory Development Kit

The Persistent Memory Development Kit (PMDK) is designed to solve the challenges described in the preceding section. It lives in user space as shown in Figure 2 on page 4, and comprises a growing collection of libraries and tools tuned and validated on both Linux and Windows. Most of these libraries are developed for specific use cases, and therefore can be used by applications as necessary.

The libraries build on the DAX feature and work with any PMEM that provides the PMEM programming model in Figure 2 on page 4. They are all open source, BSD-licensed, and developed on GitHub in the following PMDK project:

`https://github.com/pmem/pmdk`

The libraries are written in C, making them potentially adaptable for various languages. They are described in the subsections below.

For more information about PMDK, see the Persistent Memory Programming web site:

`https://pmem.io`

libpmem: Basic persistence support

The `libpmem` library is small and fairly simple. It targets the issue described in “Making changes durable” on page 5. It provides low-level PMEM support including detecting which types of flush instructions are supported by the CPU, The library also provides performance-tuned routines for copying ranges of persistence memory using the best instruction choices for the platform.

Most PMDK libraries depend on libpmem. Developers wishing to create their own persistent memory algorithms will find this library useful. However, most developers will likely use a higher-level PMDK library and let that library call libpmem for them. An example of PMEM support for `memcached` using `libpmem` directly is available on GitHub at the following page:

`https://github.com/lenovo/memcached-pmem`

libpmemobj: General-purpose allocations and transactions support

If your application has no special memory usage concerns and it needs more support than `libpmem` gives, then `libpmemobj` is probably the library you want. Because it is for general purposes and provides tools for solving the issues described in these sections:

- "Making changes atomic" on page 6
- "Persistent Memory allocation" on page 6
- "Making data position-independent" on page 7

The libpmemobj library provides a transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming.

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).

Libraries for specific use cases

Besides `libpmem` and `libpmemobj`, there are various libraries with target-specific use cases.

- `libpmemblk`

  This library supports arrays of pmem-resident blocks, all the same size, that are atomically updated. For example, a program keeping a cache of fixed-size objects in pmem might find this library useful.

- `libpmemlog`

  This library provides a pmem-resident log file. This is useful for programs like databases that append frequently to a log file.

- `libvmem`

  This library turns a pool of persistent memory into a volatile memory pool, similar to the system heap but kept separate and with its own malloc-style API.

- `libvmmalloc`

  This library transparently converts all the dynamic memory allocations into persistent memory allocations. This allows the use of persistent memory as volatile memory without modifying the target application.

- `libpmempool`

  This library provides support for offline pool management and diagnostics. Currently it provides only "check" and "repair" operations for pmemlog and pmemblk memory pools, and for BTT devices.

Usage scenarios

In practice, PMEM can be used to store persistent objects using `libpmemobj`. Prototypes exist that enable using PMEM as follows:

- Implementing (simple) mysql storage engine with `libpmemobj`

  http://pmem.io/2015/06/02/obj-mysql.html

- Redis, enhanced to use PMDK's `libpmemobj` (limited prototype):

  https://github.com/pmem/redis

- Redis, using persistent memory

  https://github.com/pmem/pmem-redis

PMEM can also be used as a cache layer for the underlying storage devices in SAP HANA, Apache Kudu, and others.

- Apache Kudu persistent memory enabled block cache:

  http://pmem.io/2017/04/03/cloudera-kudu-pmem-enabled-block-cache.html

Author

Peng Liu is a Linux Engineer at the Lenovo Data Center Group in Beijing, China. He joined the OS team in 2017. His main interests are storage and memory management kernel subsystems. Currently he is working on persistent memory-related Linux kernel development.

Thanks to the following people for their contributions to this project:

- Mark T. Chapman
- Pei Yue
- David Watts, Lenovo Press

Notices

This document was created or updated on August 23, 2019.

Send us your comments via the **Rate & Provide Feedback** form found at
http://lenovopress.com/lp1194

Trademarks