Persistent Memory: The Value to HPC and the Challenges

Andy Rudoff
Intel Corporation
andy.rudoff@intel.com

ABSTRACT

This paper provides an overview of the expected value of emerging persistent memory technologies to high performance computing (HPC) use cases. These values are somewhat speculative at the time of writing, based on what has been announced by vendors to become available over the next year, but we describe the potential value to HPC as well as some of the challenges in using persistent memory. The enabling work being done in the software ecosystem, applicable to HPC, is also described.

CCS CONCEPTS

• **Hardware** → **Memory and dense storage**; • **Software and its engineering** → *Software libraries and repositories*;

KEYWORDS

persistent memory, storage class memory, NVM programming

ACM Reference Format:
Andy Rudoff. 2017. Persistent Memory: The Value to HPC and the Challenges. In *MCHPC'17: MCHPC'17: Workshop on Memory Centric Programming for HPC, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3145617.3158213

1 INTRODUCTION

The High Performance Computing (HPC) community has taken notice of the emerging persistent memory products in the ecosystem. These products include non-volatile memory DIMMs (NVDIMMs) and Intel's persistent memory DIMM based on their 3D XPoint media[1]. We will summarize some of the more disruptive properties of persistent memory and how HPC applications might choose to leverage them. We will also describe the various programming challenges and what solutions are emerging to meet them.

2 PROPERTIES OF PERSISTENT MEMORY

2.1 Definition

The term *persistent memory* has been used by the Storage Networking Industry Associate (SNIA) to describe byte-addressable persistence that performs fast enough so that it is reasonable to stall a CPU while waiting for a load instruction to complete[7].

[1]3D XPoint is a trademark of Intel Corporation.

MCHPC'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5131-7/17/11…$15.00
https://doi.org/10.1145/3145617.3158213

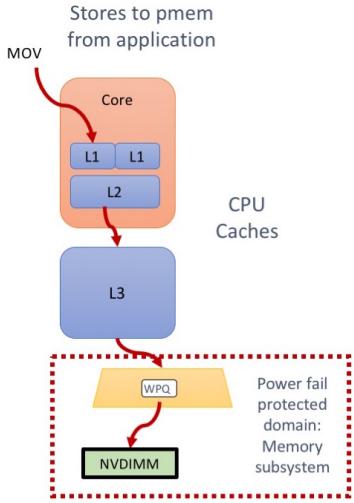

Figure 1: Flushing Stores to the Persistence Domain

While this may not be a rigid, formalized definition, it illustrates the primary differences with traditional storage: small accesses and the elimination of context switching while waiting for I/O to complete. Another common industry term for this type of media is *storage class memory*.

While one could argue whether a slow media with fast cache in front of it fits this definition, in this paper we focus instead on the programming models associated with the product, rather than the details of the media behavior. For our purposes, something is considered persistent memory if it supports the load/store programming model described below. The NVDIMM-N products in the market and the persistent memory DIMM announced by Intel meet this definition.

2.2 Programming Models

When we talk about programming models, there are multiple levels to consider. We start by describing the lowest level programming model, and work our way up the software stack.

2.2.1 Hardware Interface. The lowest level programming model is the interface to the hardware. For persistent memory, the media is delivered on a DIMM like traditional DDR memory, and it plugs directly into the processor memory bus. The result is a programming model similar to volatile memory, but with the additional requirement that the system must be able to identify the persistent memory to keep from treating it like the volatile system main memory. On the x86 platform, this identification is done through a

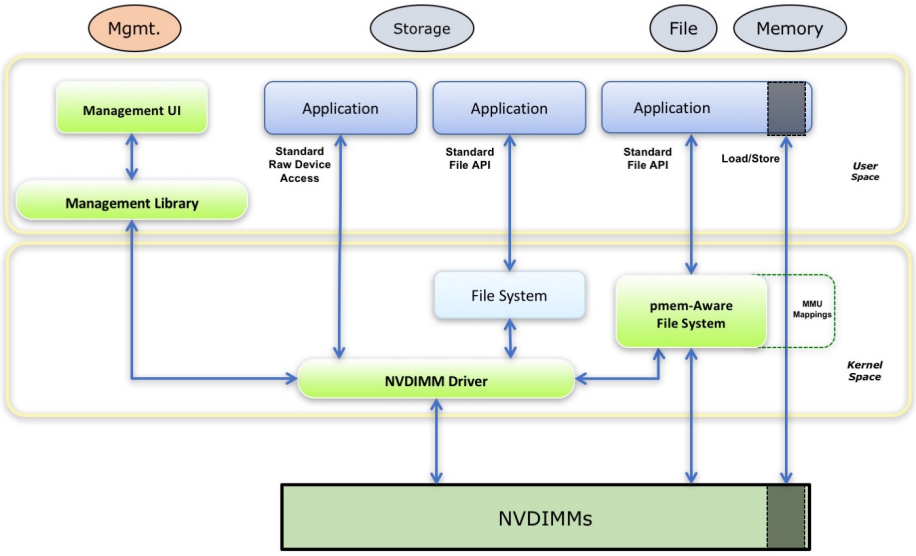

Figure 2: The SNIA Persistent Memory Programming Model

Table 1: Instructions for Flushing to Persistence

Instruction	Effect
`CLFLUSH`	Flush a cache line, serialized
`CLFLUSHOPT`	Flush a cache line, non-serialized
`CLWB`	Flush but allow to remain valid, non-serialized

standard table known as the NFIT (NVDIMM Firmware Interface Table), which was added to the ACPI 6.0 specification[3].

2.2.2 Instruction Level Interface. The next programming model to consider is at the instruction set level. As shown in Figure 1, stores to persistent memory may land in the CPU caches, where they are not considered persistent. Once stores are flushed from the CPU caches, and accepted by the x86 memory subsystem, they are considered in the *persistence domain*, as indicated by the dashed red box in the figure.

Table 1 lists the instructions used to flush stores to the persistence domain. The `CLFLUSH` instruction has existed for many generations of Intel CPU, but it is a *serialized* instruction, meaning flushing multiple cache lines will be performed serially, with one flush waiting to complete before starting the next flush. The `CLFLUSHOPT` instruction, introduced specifically for persistent memory, is non-serialized, so a loop of flushes will perform better due to concurrent flushing. Also introduced for persistent memory, the `CLWB` instruction flushes a cache line, but allows the line to remain valid in the cache instead of invalidating the cache line. This can improve performance if the cache line is accessed soon after it was flushed. All of these instructions are described by Intel's Software Development Manual[4].

2.2.3 Operating System Abstraction. The next programming model to consider is how persistent memory is made available to applications by the operating system. Figure 2 illustrates the persistent memory programming model specified by the Storage Networking Industry Association (SNIA)[7]. In this model, persistent memory (shown as the NVDIMM at the bottom of the figure), is managed by driver modules that provide access for manageability (left side of figure), traditional block usages (center of figure), and as load/store accessible persistent memory (right side of figure). For our discussion, the path on the right is the most interesting, where applications use standard file names and permissions to get access to ranges of persistent memory managed by a *pmem-aware file system*, as shown in the figure. After memory-mapping one of these files, the result is the blue arrow on the far right side of the figure, showing the persistence as part of the applications address space. This allows the application direct, load/store access.

The operating system feature that allows direct access to persistent memory has been named *DAX* by both Linux and Windows and is already publicly available in both operating systems. The result provides a familiar model, using standard interfaces to memory-map files, but can also be somewhat daunting, as an application developer is faced with the management of a raw range of (potentially terabytes of) persistent memory. For this reason, it has been critical to the software ecosystem to provide libraries and tools to enhance the software environment, as described below.

2.2.4 Programming Language Environment. When programming with persistent memory, or any large block of memory for that matter, the programmer is typically responsible for managing how that memory is allocated as data structures are placed in it. For volatile memory, allocation calls like the C language's `malloc` and `free` are commonly used. But for persistent memory, those interfaces are replaced with interfaces that can re-connect with the persistent data when an application restarts. This is made possible by having a *transactional* memory allocator.

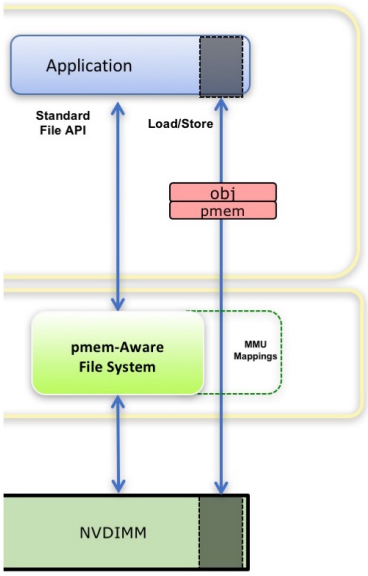

Figure 3: The libpmemobj and libpmem libraries.

Figure 3 shows where one such transactional library fits into the software stack. The library, `libpmemobj`, is a general-purpose persistent memory transaction library, available as open source code and bundled with several Linux distros[6].

2.3 The Value of Persistent Memory

Persistent memory can provide value to HPC workloads in multiple ways, based on the expected capacity, cost, performance, and the ability to offer load/store performance.

2.3.1 Capacity. At the time of writing, the commercially available persistent memory products provide 8 gigabytes to 16 gigabytes per DIMM. However, the DIMM announced by Intel promises a much larger capacity, totaling 6 terabytes on a two-socket server[1]. In addition, the expectation is that the persistent memory is cheaper than DRAM. Since HPC workloads tend to have a high demand for memory capacity, this makes persistent memory valuable as a way to provide that capacity more economically. Because of this, we expect some HPC use cases to use persistent memory purely for the additional capacity, while ignoring the fact that the media is persistent.

2.3.2 Avoid Paging. Although the emerging types of persistent memory media may not be as fast as DRAM, or as high-bandwidth as technologies like *high bandwidth memory* (HBM), it is still fast enough for direct load/store access, unlike storage. This provides value for storing large data structures (trees, hash tables, matrices, etc.) in a way where they are directly accessible without having to use *paging* to bring blocks of the data structure into DRAM. This not only avoids the paging delays, but it avoids the DRAM footprint that must evict other things from DRAM to make room for blocks being paged in.

2.3.3 Persistence. The fact that persistent memory holds its contents across power loss allows large, in-memory data structures to be present immediately when booting. For multi-terabyte data structures, this can save considerable start-up time. Additionally, the results of long, processor-bound calculations are less likely to be lost by system crash if they can be persisted as they are stored in memory.

3 HPC PERSISTENT MEMORY USE CASES

We break the HPC use cases into two broad categories, persistent and volatile. For each use case we describe what an HPC programmer can do to prepare for the availability of large-capacity persistent memory.

3.1 Volatile Use Cases

The volatile use cases are somewhat easier for the HPC programmer to leverage, since they are based on the familiar idea of volatile memory allocation. A library, such as `libmemkind`[5], is used to allocate both DRAM and persistent memory, and the application writer chooses which data structures are placed in each tier. DRAM is the smaller resource, but also the more performant, so "hot" data structures are placed there. Persistent memory is used for its capacity, so large data structures are placed there.

To prepare for the availability of persistent memory, the application changes can be made on any large memory system. Using socket-local memory to emulate the faster tier, and memory on remote sockets to emulate the persistent memory pool is an easy way to test the changes ahead of the media availability (`libmemkind` supports this).

3.2 Persistent Use Cases

Persistent use cases require more complex programming, especially when transactions are used to allow consistent data structures across program and system crashes. The `libpmemobj` library provides general-purpose persistent transactions and can be tested on normal DRAM during development[2].

4 CONCLUSIONS

This paper has provided a brief view into the expected ecosystem changes when large-capacity persistent memory becomes available. We described some of the potential values of persistent memory, the programming models for using it, and the ways we expect HPC applications to leverage it.

Due to the brief nature of this summary paper, we've only touched on the details surrounding the persistent memory ecosystem. We hope the links to more information will provide HPC application writers with a starting-point for preparing for persistent memory.

REFERENCES

[1] 2015. Intel Shows Off 3D XPoint Memory Performance. (2015). https://www.nextplatform.com/2015/10/28/intel-shows-off-3d-xpoint-memory-performance/
[2] 2016. How to Emulate Persistent Memory. (2016). http://pmem.io/2016/02/22/pm-emulation.html
[3] 2017. ACPI Specification Version 6.2. (2017). http://uefi.org/specifications
[4] 2017. Intel®64 and IA-32 Architectures Software Developer Manuals. (2017). https://software.intel.com/en-us/articles/intel-sdm
[5] 2017. MEMKIND. (2017). https://github.com/memkind/memkind
[6] 2017. The NVM Libraries. (2017). http://pmem.io
[7] 2017. The NVM Programming Model Specification. (2017). http://www.snia.org/forums/sssi/nvmp