Persistent Memory: The Value to HPC and the Challenges

November 12, 2017

Andy Rudoff
Principal Engineer, NVM Software
Intel Corporation
Intel Persistent Memory

New Type of Memory

- Persistent, Large Capacity & Byte Addressable
  - 6 TB per two-socket system
- DDR4 Socket Compatible
  - Can Co-exist with Conventional DDR4 DRAM DIMMs
- Demonstrated at SAP Sapphire and Oracle Open World 2017
- Cheaper than DRAM
- Availability
  - 2018
Optimized System Interconnect

Reach full potential of 3D XPoint™ Technology by connecting it as Memory

Sources: “Storage as Fast as the rest of the system” 2016 IEEE 8th International Memory Workshop and measurement, Intel® Optane™ SSD measurements and Intel P3700 measurements, and technology projections
Definition of *Persistent Memory*

Byte-addressable
- As far as the programmer is concerned
- Load/Store access
- Not demand-paged

Memory-like performance
- **Would reasonably stall a CPU load waiting for pmem**

Probably DMA-able
- Including RDMA

For modeling, think: Battery-backed DRAM
The Value of Persistent Memory

Data sets addressable with no DRAM footprint
- At least, up to application if data copied to DRAM

Typically DMA (and RDMA) to pmem works as expected
- RDMA directly to persistence – no buffer copy required!

The “Warm Cache” effect
- No time spend loading up memory

Byte addressable

Direct user-mode access
- No kernel code in data path
Programming Models
Programming Model: At least four meanings...

1. Interface between HW and SW
2. Instruction Set Architecture (ISA)
3. Exposed to Applications (by the OS)
4. The Programmer Experience
Programming Model:
At least four meanings...

1. Interface between HW and SW

2. Instruction Set Architecture (ISA)

3. Exposed to Applications (by the OS)

4. The Programmer Experience
Programming Model (meaning 1): HW to SW Interface
Programming Model (meaning 1): HW to SW Interface

Result:
Persistent Memory hardware accessed like memory (cache coherent).
Described by ACPI on x86.
NFIT (NVDIMM Firmware Interface Table)

First introduced in ACPI 6.0

- Describes the NVDIMMs in a system
- Separates NVDIMM memory from system main memory
- Describes persistency model
- Describes interleaving
  - Lots of interesting considerations here
  - Who should see this (which layers of SW?)
- Continuing to evolve to cover more use cases

NFIT is created by the BIOS

- BIOS must understand each supported NVDIMM type
JEDEC NVDIMM Types

**NVDIMM-N**
- Standardized
- Memory mapped DRAM. Flash is not system mapped.
- Access Methods -> byte- or block-oriented access to DRAM
- Capacity = DRAM DIMM (1's - 10's GB)
- Latency = DRAM (10's of nanoseconds)
- Energy source for backup
- DIMM interface (HW & SW) defined by JEDEC

**NVDIMM-F**
- Vendor Specific
- Memory mapped Flash. DRAM is not system mapped.
- Access Method -> block-oriented access to NAND through a shared command buffer (i.e. a mounted drive)
- Capacity = NAND (100's GB- 1's TB)
- Latency = NAND (10's of microseconds)

**NVDIMM-P**
- Proposals in progress
- Memory-mapped Flash and memory-mapped DRAM
- Two access mechanisms: persistent DRAM (−N) and block-oriented drive access (−F)
- Capacity = NVM (100's GB- 1's TB)
- Latency = NVM (100's of nanoseconds)


Author: Arthur Sainio
Programming Model:
At least four meanings...

1. Interface between HW and SW

2. Instruction Set Architecture (ISA)

3. Exposed to Applications (by the OS)

4. The Programmer Experience
Programming Model (meaning 2): Instruction Set Architecture (ISA)
Result:
Stores flushed from CPU cache, globally-visible ➔ Persistent (on x86)
The Data Path

CORE  CORE  CORE  CORE
L1    L1    L1    L1
L1    L1    L1    L1
L2    L2    L2    L2

MOV

L3

Memory Controller

NVDIMM

NVDIMM
Hiding Places

MOV

Core L1 L1 L2

Core L1 L1 L2

Core L1 L1 L2

Core L1 L1 L2

L3

Memory Controller

NVDIMM

Memory Controller

NVDIMM
How the Hardware Works

Core

L1
L1
L2
L3

CPU CACHES

CLWB + fence
-or-
CLFLUSHOPT + fence
-or-
CLFLUSH
-or-
NT stores + fence
-or-
WBINVD (kernel only)

DIMM

WPQ

ADR
-or-
WPQ Flush (kernel only)

Custom
Power fail protected domain
indicated by ACPI property:
CPU Cache Hierarchy

Minimum Required
Power fail protected domain:
Memory subsystem
## Visibility versus Power Fail Atomicity

<table>
<thead>
<tr>
<th>Feature</th>
<th>Atomicity</th>
</tr>
</thead>
</table>
| Atomic Store  | 8 byte powerfail atomicity  
|               | Much larger visibility atomicity                                         |
| TSX           | Programmer must comprehend XABORT, cache flush can abort                  |
| LOCK CMPXCHG  | *non-blocking* algorithms depend on CAS, but CAS doesn’t include flush to
|               | persistence                                                              |

- Software must implement all atomicity beyond 8-bytes for pmem
- Transactions are fully up to software
Programming Model:
At least four meanings...

1. Interface between HW and SW

2. Instruction Set Architecture (ISA)

3. Exposed to Applications (by the OS)

4. The Programmer Experience
The Storage Stack (50,000ft view...)

- Management UI
- Management Library
- Application
- Standard Raw Device Access
- Standard File API
- File System
- Driver
- Storage
A Programmer’s View
(not just C programmers!)

```c
fd = open("/my/file", O_RDWR);
...
count = read(fd, buf, bufsize);
...
count = write(fd, buf, bufsize);
...
close(fd);
```

“Buffer-Based”
A Programmer’s View (mapped files)

```c
fd = open("/my/file", O_RDWR);
...
base = mmap(NULL, filesize, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, 0);
close(fd);
...
base[100] = 'X';
strcpy(base, "hello there");
*structp = *base_structp;
...
```

“Load/Store”
Memory-Mapped Files

What are memory-mapped files really?
- Direct access to the **page cache**
- Storage only supports block access (paging)

With load/store access, when does I/O happen?
- Read faults/Write faults
- Flush to persistence

Not that commonly used or understood
- **Quite powerful**
- Sometimes used without realizing it
OS Paging

Application

User Space

load/store access

page fault access

Kernel Space

DRAM

NVDIMM

Application

Application

Application

...
Programming Model (meaning 3): Exposing to Applications

Management UI

Application

Standard Raw Device Access

Application

Standard File API

Application

Standard File API

Load/Store

User Space

"DAX"

Kernel Space

Management Library

File System

NVDIMM Driver

NVDIMM

File System

pmem-Aware File System

MMU Mappings

Intel

DCG
Data Center Group
The SNIA NVM Programming Model
Defining The Programming Model

40+ Member Companies

SNIA Technical Working Group
Defined 4 programming modes required by developers

Spec 1.0 developed, approved by SNIA voting members and published

- Interfaces for PM-aware file system accessing kernel PM support
- Interfaces for application accessing a PM-aware file system
- Kernel support for block NVM extensions
- Interfaces for legacy applications to access block NVM extensions

Programming Model: At least four meanings...

1. Interface between HW and SW

2. Instruction Set Architecture (ISA)

3. Exposed to Applications (by the OS)

4. The Programmer Experience
Programming Model (meaning 4): The Programmer Experience

Result: Safer, less error-prone, idiomatic in common languages
Java PersistentSortedMap

PersistentSortedMap employees = new PersistentSortedMap();
...
employees.put(id, data);

No flush calls. Transactional. Java library handles it all.

See “pilot” project at: https://github.com/pmem/pcj
What Lies Between:

Memory-Mapped Files

High-Level Language Support

Standard Flush

- msync()
- FlushViewOfFile()
- FlushFileBuffers()

PersistantSortedMap

employees.put(id, data);
NVM Libraries: pmem.io
C/C++ on Linux and Windows

- Open Source
  - [http://pmem.io](http://pmem.io)
- libpmem
- libpmemobj
- libpmemblk
- libpmemlog
- libvmem

NVM Libraries
Summary: What the Basic Model Provides (and what it does not provide)

- Raw user space access to a mapped area of persistent memory
  - “Your terabytes start at address X, have fun!”
- Hard problems programmers have to solve:
  - Allocation: something like malloc/free, new/delete
  - Transactions: prevent torn updates
  - Type safety:
    - Disallow pointing from pmem to DRAM
    - Remember type of thing in pmem
  - Position independence: allow mapping address to change
  - Memory error handling
    - Simpler, less error-prone programming
      - With tools to help find errors
Persistent Memory and HPC
Volatile Usages

Use pmem for its capacity, not for its persistence
- pmem will be larger than DRAM
- pmem will be cheaper than DRAM
- But...
- pmem will be slower than DRAM

HPC applications can manage data placement
- “Hot” data structures in DRAM
- Others in pmem (still faster than paging from storage)
libmemkind

https://github.com/memkind/memkind

- Provides familiar “malloc” interface
- For different “kinds” of memory
  - Includes NUMA nodes, HBM, pmem
- No need to track type, the “free” interface figures it out
Persistent Usages

Synchronous persistence
- Changes are persisted as they are made (transactionally)
- Best recovery semantics, worst performance overhead

Eventually persistent
- More complex programming ("sync points")
- Better performance

Snapshots
- Really more of a block/sequential use case
- Holds state locally while sending it to a server
- Performs well
libpmemcon: “close-to-open” Consistency

Behaves like malloc/free at run time
- No need for transactions
- No need for flushing
- Normal pointers

Clean shutdown saves heap state
Open fails unless shutdown was clean
- Or if can’t map at same place

Optimizes for most common case
Examining Challenges of Load/Store
Application Memory Allocation

- Well-worn interface, around for decades
- Memory is gone when application exits
  - Or machine goes down

\[
\text{ptr} = \text{malloc(} \text{len} \text{)}
\]
Application NVM Allocation

- Simple, familiar interface, *but then what?*
  - Persistent, so apps want to “attach” to regions
  - Need to manage permissions for regions
  - Need to resize, remove, …, **backup** the data
Visibility versus Persistent

It has always been thus:
- open()
- mmap()
- store...
- msync()

pmem just follows this decades-old model
- But the stores are cached in a different spot
pmem Programming Challenges: Flushing, Atomicity, Transactions

No Page Cache to Flush
CPU Caches still need flushing

- msync()
-FlushViewOfFile()
-FlushFileBuffers()
-Work as expected

pmem_persist()
User-Mode Instructions
POSIX Load/Store Persistence

```c
open(...);
mmap(...);

strcpy(pmem, "andy");

msync(pmem, 5, MS_SYNC);
```
Optimized Flush (only use when safe!)

open(...);
mmap(...);

strcpy(pmem, "andy");

pmem_persist(pmem, 5);
libpmem Load/Store Persistence

```c
open(...);
mmap(...);
strcpy(pmem, "andy");
pmem_persist(pmem, 5);
```
Crossing the 8-byte Store

open(...);
mmap(...);

strcpy(pmem, "andy rudoff");

pmem_persist(pmem, 12);

*crash*

Which Result?

1. "\0\0\0\0\0\0\0\0\0..."
2. "andy\0\0\0\0\0\0..."
3. "andy rud\0\0\0\0\0..."
4. "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0..."
5. "andy rudoff\0"
Position Dependence

Can pointers work across sessions?

Will a pmem file be mapped at the same address every time?
What is libpmemobj?

General purpose pmem transactions

pmem-aware memory allocator

Some common operations made atomic

Application doesn’t worry about HW details

- Library handles CPU cache flushing
- Library comprehends size of atomic stores

Lots of examples in the pmem.io source tree

This is probably the library you want
Application uses libpmemobj API

- Transactions entirely memory-centric user space code

- Much faster, but…

- App had to change
Libpmemobj Replication: Application Transparent (except for performance overhead)

Same API with and without replication

NVDIMM

NVDIMM
C Programming with libpmemobj
Transaction Syntax

TX_BEGIN(Pop) {
    /* the actual transaction code goes here... */
}

TX_ONCOMMIT {
    /*
     * optional - executed only if the above block
     * successfully completes
     */
}

TX_ONABORT {
    /*
     * optional - executed if starting the transaction fails
     * or if transaction is aborted by an error or a call to
     * pmemobj_tx_abort()
     */
}

TX_FINALLY {
    /*
     * optional - if exists, it is executed after
     * TX_ONCOMMIT or TX_ONABORT block
     */
}

TX_END /* mandatory */
Properties of Transactions

```c
TX_BEGIN_PARAM(Pop, TX_PARAM_MUTEX, &D_RW(ep)->mtx, TX_PARAM_NONE) {
    TX_ADD(ep);
    D_RW(ep)->count++;
} TX_END
```

- **Powerfail Atomicity**
- **Multi-Thread Atomicity**
- Caller must instrument code for undo logging
Persistent Memory Locks

- Want locks to live near the data they protect (i.e. inside structs)
- Does the state of locks get stored persistently?
  - Would have to flush to persistence when used
  - Would have to recover locked locks on start-up
    - Might be a different program accessing the file
  - Would run at pmem speeds

- PMEMmutex
  - Runs at DRAM speeds
  - Automatically initialized on pool open
C++ Programming with libpmemobj
libpmemobj

Application

Load/Store

API

API

API

atomic operations

transactions

locks

lists

allocator

libpmemobj

libpmem

memory-mapped pmem
C++ Queue Example

Application

root object:
class pmem_queue

pmem pool "myfile"

struct pmem_entry
C++ Queue Example: Declarations

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>persistent_ptr&lt;T&gt;</td>
<td>Pointer is really a position-independent Object ID in pmem.</td>
</tr>
<tr>
<td></td>
<td>Gets rid of need to use C macros like D_RW()</td>
</tr>
<tr>
<td>p&lt;T&gt;</td>
<td>Field is pmem-resident and needs to be maintained persistently.</td>
</tr>
<tr>
<td></td>
<td>Gets rid of need to use C macros like TX_ADD()</td>
</tr>
</tbody>
</table>
C++ Queue Example: Transaction

```c++
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;
        }
    });
}
```
Summary
## State of Ecosystem Today

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Detection of NVDIMMs</td>
<td>ACPI 6.0+</td>
</tr>
<tr>
<td>OS Exposes pmem to apps</td>
<td>DAX provides SNIA Programming Model</td>
</tr>
<tr>
<td></td>
<td>Fully supported:</td>
</tr>
<tr>
<td></td>
<td>• Linux (ext4, XFS)</td>
</tr>
<tr>
<td></td>
<td>• Windows (NTFS)</td>
</tr>
<tr>
<td>OS Supports Optimized Flush</td>
<td>Specified, but evolving (ask when safe)</td>
</tr>
<tr>
<td></td>
<td>• Linux: <strong>unsafe</strong> except Device DAX</td>
</tr>
<tr>
<td></td>
<td>• MAP_SYNC fixes this</td>
</tr>
<tr>
<td></td>
<td>• (safe in NOVA)</td>
</tr>
<tr>
<td></td>
<td>• Windows: <strong>safe</strong></td>
</tr>
<tr>
<td>Remote Flush</td>
<td>Proposals under discussion</td>
</tr>
<tr>
<td></td>
<td>(works today with extra round trip)</td>
</tr>
<tr>
<td>Deep Flush</td>
<td>In latest specification (SNIA NVMP and ACPI)</td>
</tr>
<tr>
<td>Transactions, Allocators</td>
<td>Built on above via libraries and languages:</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://pmem.io">http://pmem.io</a></td>
</tr>
<tr>
<td></td>
<td><strong>Much more language support to do</strong></td>
</tr>
<tr>
<td>Virtualization</td>
<td>All VMMs planning to support PM in guest</td>
</tr>
<tr>
<td></td>
<td>(KVM changes upstream, Xen coming, others too...)</td>
</tr>
</tbody>
</table>
Summary

Persistent Memory

- Emerging technology, game changing, large capacity
- New programming models allow greater leverage

NVM Libraries

- http://pmem.io
- Convenience, not a requirement
- Transactions, memory allocation, language support
- More coming

We don’t know all the answers yet

- The next few years are going to be pretty exciting!
References

- SNIA NVM Programming Model
  - http://www.snia.org/forums/ssi/nvmp
- Open Source NVM Libraries
  - http://pmem.io