PLANNING AHEAD FOR PERSISTENT MEMORY

Rob Dickinson
Datacenter Solutions Group

October 17, 2017
ABOUT ME

NVM Software Architect at Intel
Specializing in persistent memory and databases

- github.com/RobDickinson
- linkedin.com/in/robfromboulder
- @robfromboulder
- robert.a.dickinson@intel.com
I'M NOT ONE OF THOSE INTEL ENGINEERS IN A BUNNY SUIT
EAT  SLEEP  CODE >>>  BUILD COOL STUFF

REPEAT  VALIDATE ON REAL SAMPLES

SHARE BEST PRACTICES

CONTRIBUTE TO OPEN SOURCE

MAKE THE WORLD A BETTER PLACE
Persistent Memory

Software Architects

Traditional Memory & Storage
FOUR GUIDELINES FOR PERSISTENT MEMORY

- Persistent memory is not exactly like DRAM nor SSD
- Persistent memory will be used in different ways by different applications
- Persistent memory will be used in concert with memory and storage (aka hybrid architectures)
- Persistent memory exposes applications to some classic problems
WHAT IS PERSISTENT MEMORY?

compared to DRAM and SSD
**INTEL® PERSISTENT MEMORY**

**New type of memory:**
- Persistent
- 6 TB per two-socket system
- Cheaper than DRAM

**Product available:**
- SSDs in 2017
- Intel DIMMs for next-gen platforms in 2018
## Compared to Memory and Storage

<table>
<thead>
<tr>
<th>Aspect of Comparison</th>
<th>Memory (DRAM)</th>
<th>Persistent Memory</th>
<th>Storage (SSD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>GB</td>
<td>TB</td>
<td>TB</td>
</tr>
<tr>
<td>Power-Safe Durability</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Access Model (native units)</td>
<td>Byte Addressable (CL)</td>
<td>Byte Addressable (CL/ECC)</td>
<td>Blocks (4K)</td>
</tr>
<tr>
<td>Latency</td>
<td>Nanoseconds</td>
<td>Nanoseconds</td>
<td>Microseconds</td>
</tr>
<tr>
<td>Wear Leveling</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cost</td>
<td>$$$</td>
<td>$$</td>
<td>$</td>
</tr>
</tbody>
</table>
PROGRAMMING FOR PERSISTENT MEMORY
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

MEMORY-MAPPED FILES

APPLICATION
LOAD/STORE ACCESS
USER SPACE

APPLICATION
PAGE FAULT ACCESS

APPLICATION

KERNEL SPACE

DRAM
FLUSHING CPU CACHES
EMULATING PERSISTENT MEMORY (LINUX)

1. Use /dev/shm

2. Set up DAX device on DRAM
   http://pmem.io/2016/02/22/pm-emulation.html

*** but remember emulation is an upper bound!
CLASSIC PROBLEMS RE-EXPOSED
Having direct access to persistent memory, bypassing the kernel and filesystem, brings classic problems into userspace (where programmers aren't used to seeing them)
1. MEMORY ALLOCATION & GC

Regular memory allocators aren't appropriate
A leak in persistent memory remains leaked
2. TRANSACTIONS

Probably the hardest problems of all...
Preventing torn updates using transactions
Includes reversing allocations/deallocations
3. LOCKING

Locks naturally creep into persistent structures
Abandoned locks must be reset
4. "SBD" ERROR DETECTION & HANDLING

Coping with media failures or corruption
Drivers & filesystems hide these problems
INTRODUCING NVML
Persistent Memory Programming

Libraries and Examples for Persistent Memory Programming

http://pmem.io  andy.rudloff@intel.com

Repositories 22  People 45  Teams 6  Projects 0

Pinned repositories

**nvml**
Active development tree for the NVM Library

- [C] 287  [V] 149

**syscall_intercept**
The system call intercepting library

- [C] 131  [V] 23

**vtrace**
Tool tracing syscalls in a fast way using eBPF linux kernel feature

- [C] 12  [V] 4

**pmemfile**
Userspace implementation of file APIs using persistent memory.

- [C] 8  [V] 11

**pyvm**
Forked from perone/pyvm

Python bindings for the NVML Non-volatile memory for Python.

- Python 4  [V] 3

**ndctl**
Utility library for managing the libnvram (non-volatile memory device) sub-system in the Linux kernel

- [C] 23  [V] 15
NVML: A SUITE OF OPEN-SOURCE OF LIBRARIES

Support to use Persistent memory as volatile memory

Posix-like API to access Persistent Memory

Provides interface for arrays of pmem-resident blocks, all the same size, that are atomically updated.

Provides interface for persistent memory allocation, transactions and general facilities

Provides interface to create a persistent memory resident log file.

Key Value data store for Persistent Memory

Support Transactions

Fully Validated
In Development
For volatile usage

Libraries

Libpmem

Low level support for local persistent memory

Low level support for remote access to persistent memory

Application

Load/Store

User Space

Pmem-Aware File System

NVDIMM

Proxies

Link to Open Source
http://pmem.io/nvml/

Link to Intel Developer Zone
• Data Persistence
  • pmem_persist flushes and calls sfence

• Data Consistency
  • Supports Transactional/Atomic Operations

• Persistent Memory Allocation
  • Persistent Memory Exposed as Object(memory) Pools
    • Uses Memory-Mapped Files
    • Uses Pool Identification (Layout)
  • Locating Objects in the Pool
    • Root Object: Anchor to which data structures are attached
    • Known Location in the Persistent Memory Pool
  • Ensures No Persistent Memory Leaks

• Persistent Memory Error Handling
using namespace nvml::obj;

struct root{
    p<int> pint;
    p<int> pint2;
    p<int> pint3;
};

// Do the transactional allocation using C++ Lamda function

transaction::exec_tx (pop,[&]{{
    root_obj = make_persistent<root>();
    root_obj->pint = 5;
    root_obj->pint2 = 6;
    root_obj->pint3 = root_obj->pint + root_obj->pint2;
});//

• Uses make_persistent function
• Allocates memory and constructs objects in-place
• All operations in this block are transactional. Allocations and assignments are done atomically.
SOFTWARE PATTERNS FOR PERSISTENT MEMORY
1. WAL ACCELERATION

Put WAL on persistent memory
Leave main data storage on SSD
2. PERSISTENT CACHE

Slower but larger than DRAM cache
Never needs to be reloaded from SSD
3. IN-MEMORY DATASTORES

Overcoming traditional challenges:
1) Limited by cost & capacity of DRAM
2) Recovery time increases with dataset size

Faster recovery = greater agility
4. CONVERGED READS AND WRITES

Usually to be minimized or avoided
Even a small amount of writes drives up queue depth on a SSD (testing with fio and 4k blocks)
5. HYBRID DATASTORES

Some (hot) data is in DRAM
Some (warm) data is in persistent memory

(more complex variations use SSDs too)
github.com/pmem/pmemkv
Let's skip mapping objects to JSON or SQL
Let's have native objects & references instead
struct KVSlot {
    uint8_t ph;           // Pearson hash for key
    uint32_t ks;          // key size
    uint32_t vs;          // value size
    persistent_ptr<char[]> kv; // buffer for key & value
};

struct KVLeaf {
    p<KVSlot> slots[LEAF_KEYS];  // array of slot containers
    persistent_ptr<KVLeaf> next; // next leaf in unsorted list
};

struct KVRoot {   // persistent root object
    persistent_ptr<KVLeaf> head;  // head of linked list of leaves
};
7. NATIVE PERSISTENT COLLECTIONS

Use non-volatile collections like volatile Expect these for your preferred language (eventually...)
CALL TO ACTION
GETTING STARTED WITH PERSISTENT MEMORY

- Consider persistent memory as its own unique tier
- Look for break-out opportunities to hybridize your architecture
- Use libraries to assist in safely using persistent memory
  - pmem.io
  - github.com/pmem/nvml/
  - software.intel.com/en-us/persistent-memory
- Start with emulation, but validate & tune on a real platform