NVML: Implementing Persistent Memory Applications

Paul von Behren / Intel Corporation

SNIA Legal Notice

- The material contained in this tutorial is copyrighted by the SNIA unless otherwise noted.
- Member companies and individual members may use this material in presentations and literature under the following conditions:
  - Any slide or slides used must be reproduced in their entirety without modification
  - The SNIA must be acknowledged as the source of any material used in the body of any document containing material from these presentations.
- This presentation is a project of the SNIA Education Committee.
- Neither the author nor the presenter is an attorney and nothing in this presentation is intended to be, or should be construed as legal advice or an opinion of counsel. If you need legal advice or a legal opinion please contact your attorney.
- The information presented herein represents the author's personal opinion and current understanding of the relevant issues involved. The author, the presenter, and the SNIA do not assume any responsibility or liability for damages arising out of any reliance on or use of this information.

NO WARRANTIES, EXPRESS OR IMPLIED. USE AT YOUR OWN RISK.

Abstract

NVML: Implementing Persistent Memory Applications

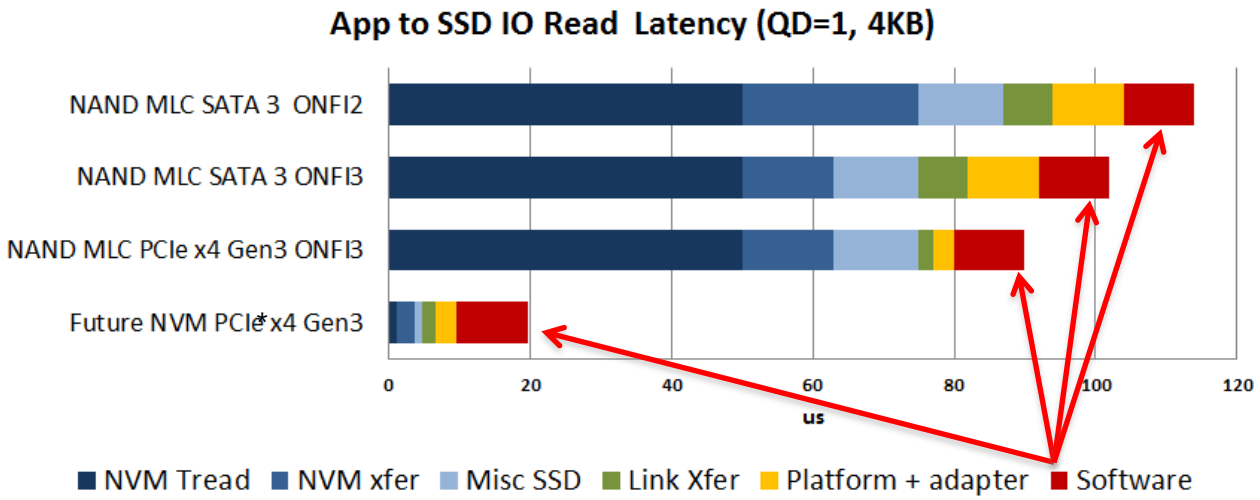
NVML is an open-source library that simplifies development of applications utilizing byte-addressable persistent memory (PM). The SNIA NVM Programming Model describes basic behavior for a persistent memory-aware file system enabling applications to directly access persistent memory. NVML extends the SNIA programming model providing application APIs that help applications create and update data structures in persistent memory avoiding pitfalls such as persistent memory leaks and inconsistencies due to unexpected hardware or software restarts. This tutorial includes an overview of persistent memory hardware (NVDIMMs) and the SNIA NVM Programming Model, then describes the APIs provided by NVML and examples showing how these APIs may be used by applications.

Hardware supporting persistent memory (PM)

- Yesterday: battery backed RAM
- Today: NVDIMMs with DRAM and flash
  - On power down, RAM copied to flash; on power up, copy back to RAM
- Emerging NVDIMMs: Phase Change Memory, Memristor, many others
  - Offer ~ 1000x speed-up over NAND, closer to DRAM
- Characteristics as seen by software
  - Load/Store (memory instructions) accessible
  - Would reasonably stall CPU for a load instruction
  - No paging (at least not by the OS)

Software access to PM

- Could treat PM like disks/SSDs
  - Existing software works, faster than with flash
  - But we still have block stack latency (Intel SSD study)



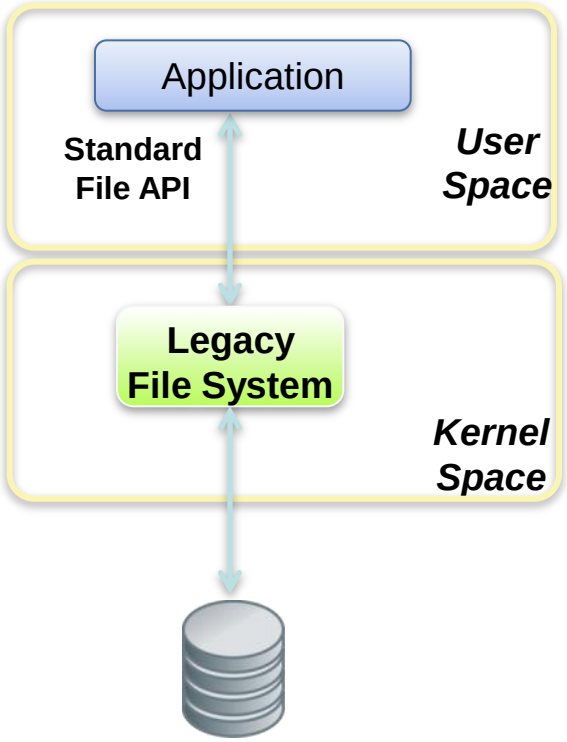
- With Next Generation NVM, hardware is no longer the bottleneck

SNIA goals

- Define a programming model for direct access to PM
  - No kernel code in data path, use existing load/store instructions
- Use a general approach for different types of PM hardware
  - Use existing OS solutions where appropriate
    - E.g., use existing file permissions rather than invent something new
- Specify behavior, not a specific API
  - Allow OS developers to implement APIs appropriate to the OS
- Support application developer goals for power-fail safe atomicity

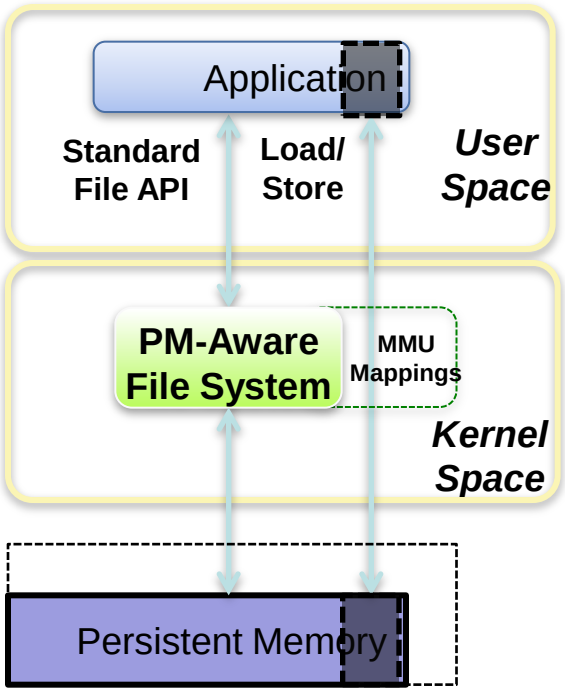
Legacy memory mapped files

- Background: memory mapped files backed by block devices
  - POSIX mmap(), Windows MapViewOfView()
  - Disk file mapped to virtual memory
  - Paged to memory when referenced
  - Msync() flushes dirty pages to disk
- Programming Model (POSIX)
  - mmap()
  - load/store commands to virtual memory
  - msync() to assure changes are persistent



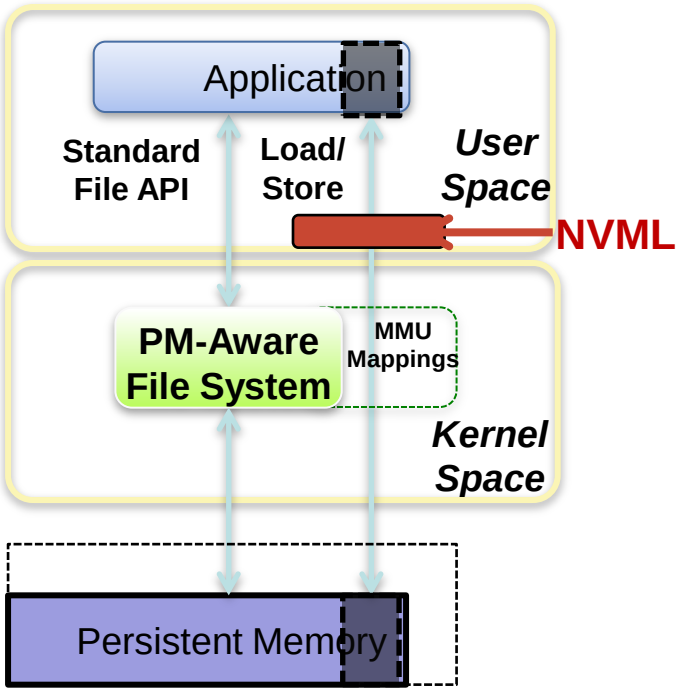
Persistent memory programming model

- With PM, no paging between persistence and volatile memory
- Memory map command causes PM file to be mapped to app's virtual memory
- Sync command flushes CPU cache
  - And PM device flush as needed
- Load/Store commands directly access PM
- Standard file API still works
  - Perhaps less performant



Role of NVML

- Open-source library that simplifies development of apps using byte-addressable PM.
- Builds on the SNIA model
  - APIs to help developers use PM
- Works with existing CPU hardware
  - Support new features as they emerge
- Doesn't require language extensions
  - We recognize that they will improve developer experience
  - But we don't want to wait for standard language support, or drive developers to non-standard languages



Libraries provided by NVML

- **libpmem** provides low level persistent memory support. The libraries below are implemented using **libpmem**.
- **libpmemobj** provides a transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming
- **libpmemblk** supports arrays of PM-resident blocks, all the same size, that are atomically updated
- **libpmemlog** provides a PM-resident log file
- See the libpmem page for documentation and examples.
- **libvmem** turns a pool of persistent memory into a volatile memory pool, similar to the system heap

Libpmem: low level PM support

- Flush-to-persistence support
- Includes PM optimized memmove and friends
- Other NVML libraries are implemented using **libpmem**
- May be used without using other NVML libraries

```
#include <libpmem.h>

cc ... -lpmem (or -lpmem_debug)

int pmem_is_pmem(void *addr, size_t len);
void pmem_persist(void *addr, size_t len, int flags);
void pmem_flush(void *addr, size_t len, int flags);
void pmem_fence(void);
void pmem_drain(void);
```

Libpmemobj: transactional object store

- General purpose (unlike the specialized libpmemblk, libpmemlog, libvmem)
- "Object" does not imply some specific object store implementation; refers to any memory container for data
- Library maps entire memory pool (direct access file) into program's address space
- Programmer designs the desired layout
  - Defines data structures
  - Uses libpmemobj to coordinate all accesses

libpmemobj:
guiding design principles

- Programmer can load/read any data structure without having to copy it
  - Load instructions directly access data from its resting place in PM
- Program can write to a data structure directly after telling the library the structure is changing as part of a transaction
  - Library maintains undo log, rolls back interrupted transactions on recovery

libpmemobj:
guiding design principles

- Programmer is responsible for multi-threaded locking
  - But locking/unlocking can be tied to transactions for programming convenience
- All pointers in pmem are really Object IDs
  - OIDs can point between pmem pools
  - OIDs can be mapped to different memory addresses each time the programs runs and they still work correctly

libpmemobj:
macros to help using OIDs

```
struct node {
        OID_TYPE(struct node) next;
        int val;
};
```

- Declares a linked list structure with a `next` pointer that is an OID
  - Macros will use `struct node` for type checking at compile time to make sure `next` is always used as a pointer to that type

libpmemobj:
transaction example

```
/* “pop” is a pool handle… */
TX_BEGIN(pop)
    newnode = TX_ALLOC(struct node, 1);
    DIRECT(newnode)->val = 123;
    DIRECT(newnode)->next = NULL;
    DIRECT(list_head)-> next = newnode;
TX_END
```

- The effects of operations between `BEGIN/END` happen fully or not at all (including memory allocation)

libpmemobj:
providing atomic operations

- Library provides some common operations
  - Atomic with respect to other threads
  - Atomic with respect to power loss – style interruptions
  - Don’t need transactions to use these
- These operations often eliminate the need to use a transaction
  - Or eliminate the need for a nested transaction

libpmemobj:
atomic list operations

- Move element between two lists
  - MT safe, atomic
  - After crash, element will be on one list
- Allocate an element and place on a list
  - After crash, element will be on list or never allocated
- Remove an element from a list and free
  - After crash, element will be on list or freed

Libpmemblk: PM carved into blocks

- Pool is divided up into a specific chunk size
- Single block writes to the pool are atomic
- Ideal usage: user space PM cache
- APIs

```
PMEMblkpool *pmemblk_open(const char *path, size_t bsize);
PMEMblkpool *pmemblk_create(const char *path, size_t bsize,
            size_t poolsize, mode_t mode);
void pmemblk_close(PMEMblkpool *pbp);
size_t pmemblk_nblock(PMEMblkpool *pbp);
int pmemblk_read(PMEMblkpool *pbp, void *buf, off_t blockno);
int pmemblk_write(PMEMblkpool *pbp, const void *buf, off_t blockno);
int pmemblk_set_zero(PMEMblkpool *pbp, off_t blockno);
int pmemblk_set_error(PMEMblkpool *pbp, off_t blockno);
```

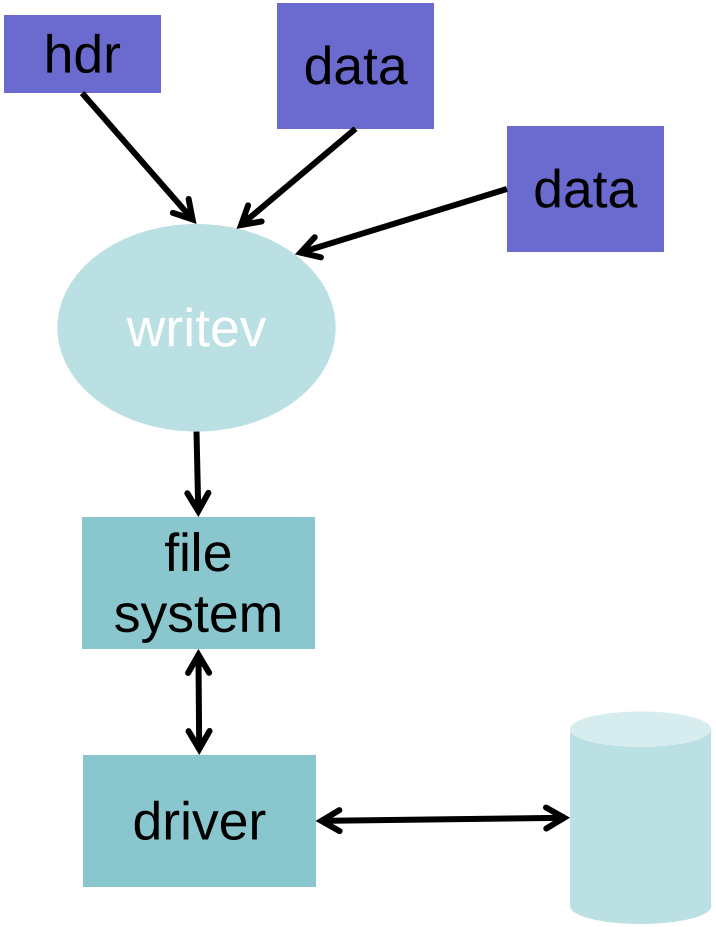
libpmemlog: (append-mostly) log

- Common usage: append entries to log file
- Append operation very efficient
- Read through (for log shipping) also optimized
- APIs

```
PMEMlogpool *pmemlog_open(const char *path);
PMEMlogpool *pmemlog_create(const char *path,
          size_t poolsize, mode_t mode);
void pmemlog_close(PMEMlogpool *plp);
int pmemlog_append(PMEMlogpool *plp, const void *buf, size_t count);
int pmemlog_appendv(PMEMlogpool *plp,
       const struct iovec *iov, int iovcnt);
off_t pmemlog_tell(PMEMlogpool *plp);
void pmemlog_rewind(PMEMlogpool *plp);
void pmemlog_walk(...);  /* walk log and call callback function */
```

libpmemlog : legacy append APIs

- The writev() system call is often used:
  - `writev(fd, iov, iovcnt)`
  - Handy for grabbing header, data from separate locations in memory
- Not atomic
  - Well, POSIX says “atomic with respect to other reads and writes”
  - Certainly not power-fail atomic
- Fairly long code path
  - Includes file system
  - Potentially multiple trips through the block stack for metadata updates



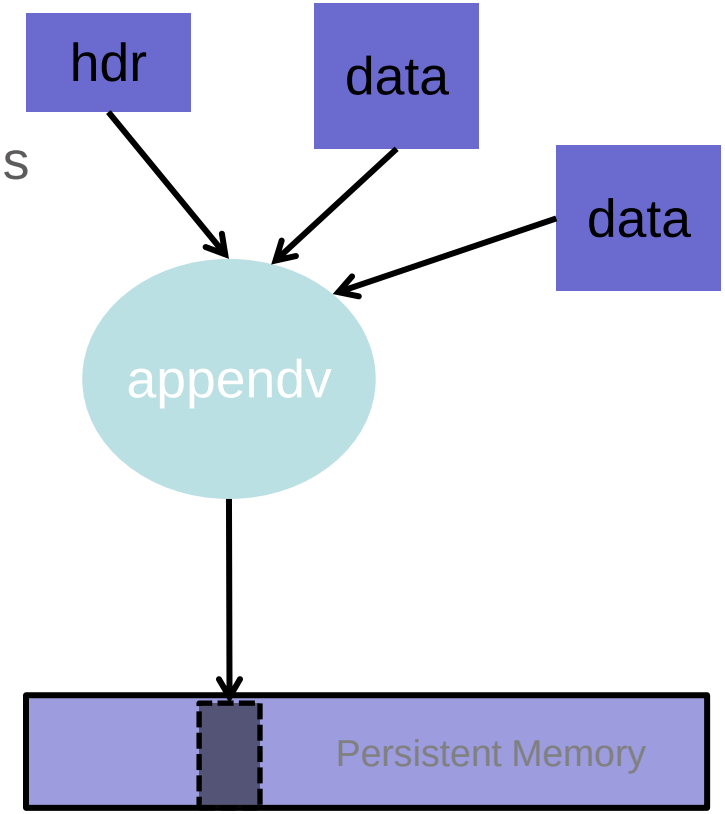
The PMEMlog API

APIs

```
PMEMlog *pmemlog_map(int fd);
void pmemlog_unmap(PMEMlog *plp);
size_t pmemlog_nbyte(PMEMlog *plp);
int pmemlog_append(PMEMlog *plp, const void *buf, size_t count);
int pmemlog_appendv(PMEMlog *plp, const struct iovec *iov, int iovcnt);
off_t pmemlog_tell(PMEMlog *plp);
void pmemlog_rewind(PMEMlog *plp);
void pmemlog_walk(PMEMlog *plp, size_t chunksize ,
          int (*process_chunk)(const void *buf, size_t len, void *arg),
          void *arg);
```

Algorithm converted for PM

- `pmemlog_appendv(plp, iov, iovcnt)`
- Atomic "gather append"
  - Uses direct access store to PM
  - And appropriate CPU flush operations
  - No system calls



libvmem: volatile memory allocator

- Allows caller to
  - use PM as volatile memory via malloc/free-like calls
  - Leverage capacity
- Doesn’t bother flushing for durability
- Vmem pools “reset” on application restart
- We are adding facility to intercept malloc/free in unmodified code
  - Uses LD_PRELOAD

On being a good citizen

- The library never:
  - Exits
  - Forks or Joins threads
  - Uses signals
  - Calls select()
- Caller can supply:
  - Custom malloc(), etc.
- Debug version of the library:
  - Traces all calls, errors, lots of details
  - Includes assertion checking

The SNIA model and implementation status

- *NVM Programming Model* spec published in 2012
  - Update in SNIA member review
    - clarifies assumptions for kernel; and CPU behavior
  - http://www.snia.org/tech_activities/standards/curr_standards/npm
- Linux support progressing
  - Has been in development review for a year
  - Hoping for integration kernel in 1-2 months
  - Expecting inclusion in experimental distros in 3-4 months
  - Implemented as DAX (direct access) mount option for EXT4
    - `mkfs.ext4 /dev/pmem0`
    - `mount -o dax /dev/pmem0 /mnt/pmem/`
    - Support for other file systems started

NVML status and more information

- NVML is still under development and is not yet ready for production use
  - libpmemobj in early development, other modes are implemented
  - Stable API planned for end of March 2015
  - Stable implementation planned for September 2015
- http://pmem.io/nvml/
  - has blog articles and links to NVML source
- https://github.com/pmem/nvml/
  - The source repo
- https://github.com/pmem
  - Related work: POC adapts of OSS, valgrind macros

Attribution & Feedback

The SNIA Education Committee thanks the following Individuals for their contributions to this Tutorial.

Authorship History

Paul von Behren: February 17, 2015

Updates:

Additional Contributors

Andy Rudoff

*Please send any questions or comments regarding this SNIA Tutorial to **tracktutorials@snia.org***