SNIA NVM Programming Model: Optimizing Software for NVM

Paul von Behren
Intel Corporation
SNIA NVM Programming TWG co-chair

Objectives / Overview

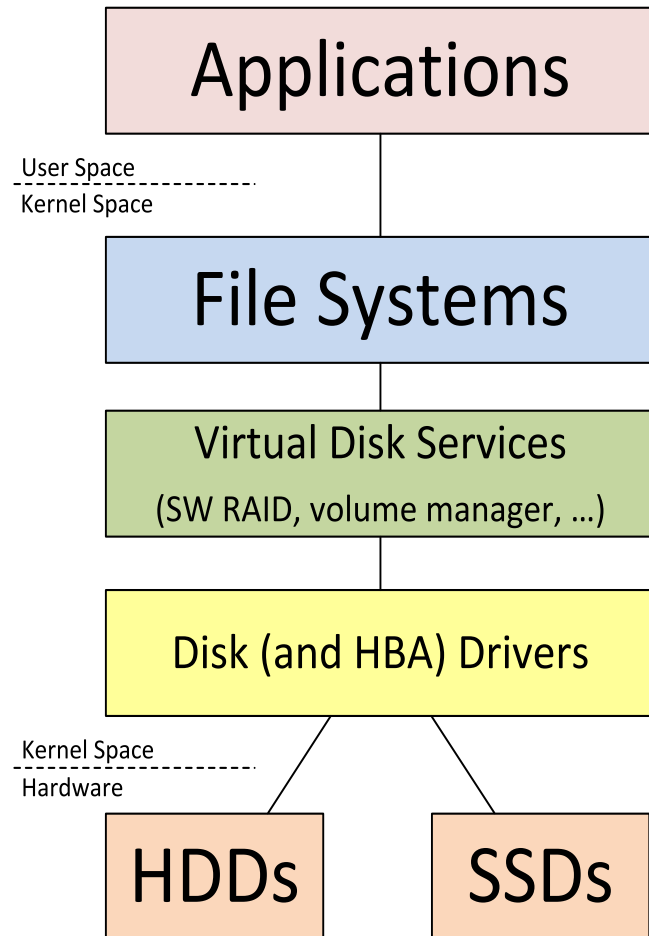
- Provide a glimpse of
  - How various types of software utilize today's NVM
    - No software experience necessary!
  - Software impact of enhancements for SSDs
  - Approaches for SW to utilize persistent memory
  - The SNIA *NVM Programming Model*
- Wrap-up

NVM Programming Model Goals

- NVM vendors (and standards) extending HW features
  - Often driven by application developer requests
  - Which dives a need for new/updated software interfaces
- Encourage a common software ecosystem
  - Without limiting ability to innovate
- SNIA is creating the *NVM Programming Model*
  - Published spec describing high level approach to using NVM in software
  - Stopping short of defining an API
  - More details about this spec later

Today's Disk IO Stack

- SSDs treated (nearly) identically to HDDs
  - *disk* refers to HDD or SSD
- Diagram depicts IO (not management) transactions
- Depicts control (not data) flow
  - Examples will look at data flow
- Control flows top-to-bottom
  - app sends IO command to file system,
  - flows through layers until, …
  - driver sends commands to disks



Programming impact - starting at the bottom of the stack (drivers)

- Where software addresses SCSI, ATA, NVMe and vendor-specific commands to HW
  - HW commands are out of scope for the *NVM Programming Model*
- Drivers provide a HW-independent (OS-specific) API to other kernel components
  - Common abstraction to all disks
  - An important interface for the programming model
    - The ***block mode*** discussed later
  - API serves as a front-end to DMA operations moving data between memory to disks

Middle of the stack – non-driver kernel components

- Kernel disk consumers utilize ***block mode*** to move data between memory and disks
  - They do not use HW (e.g., SCSI) commands
- Primary kernel disk consumer is file systems
  - File systems provide byte-addressable, hardware-independent interface (***file mode***) to applications
- Kernel hibernation logic also uses block mode

Applications – top of the stack

- File mode - applications view storage as either
  - byte- addressable files (basic apps)
  - ranges of blocks in files (advanced block-aware apps)
- Basic approach use by most applications
  - File systems hide block alignment and provide cache
  - Transparent cache masks flush to disk
- Block-aware approach disables file system cache
  - Provide ability to assure data is flushed to disk
  - Application must handle block/MMU-page alignment
  - Frequently used by mission-critical apps

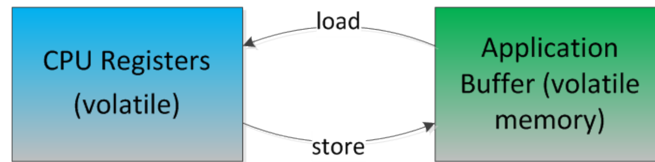
Flash Memory SUMMIT

What we mean by ***mode***

- Modes are approaches taken by certain classes of software
  - Not strictly defined in programming languages – more of a tactic rather than an API
  - In stack diagram – modes consist of the commands/behavior between layers in the stack
    - Add level abstraction to SW higher in the stack
- The exact commands (APIs) vary across OSes, but the concepts are similar
  - As used here, "programming mode" discusses the behavior without using a specific API
- Many modes (and variants)
  - We will look at commonly used modes

Memory Access Programming Mode (also known as "load/store access")

- Application uses volatile memory as a temporary place to store data
- Applications processes data using CPU registers



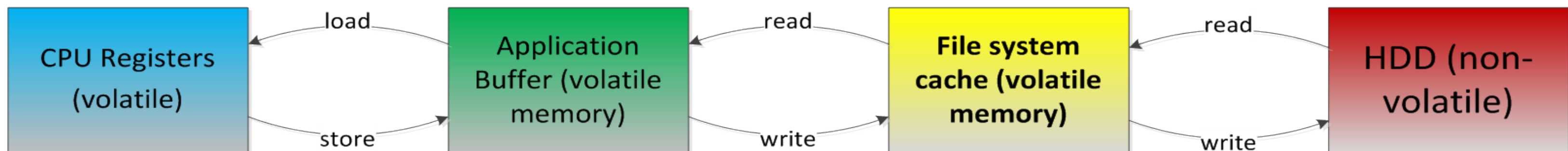
- Language run time environment hides the CPU/memory boundary
  - The program defines variables to hold data (this allocates memory)
  - Later when the program modifies the variables, the run-time environment loads data to CPU registers, modifies it, and stores it back to memory

Memory Access Example

- Consider an application that does not use storage – app displays the system temperature read from a sensor
  - The application allocates memory by defining variables to hold the data read from the sensor and the space needed to create a human-readable message
  - Application reads sensor data into variables (memory)
  - The sensor reports the temperature relative to a sensor-specific offset; the application adjusts by the offset
    - Variable/memory contents loaded to CPU register, modified, and stored to memory
  - The application uses the adjusted temperature to create a message
    - Again, load variable to CPU register, modify, then store back to memory
  - The temperature is not saved to disk by this application

File programming mode for disks

- Load/Store access still applies
- To assure the data is durable across system and application restarts, applications write the selected memory contents to files (persisted to disk)
- Hidden from application, the file system utilizes a cache (also stored in volatile memory) to improve access to disks



- The application is told a write command is complete as soon as the data is copied to FS cache; write to disk happens later
- A read command may require a disk I/O is requested blocks are not already in cache

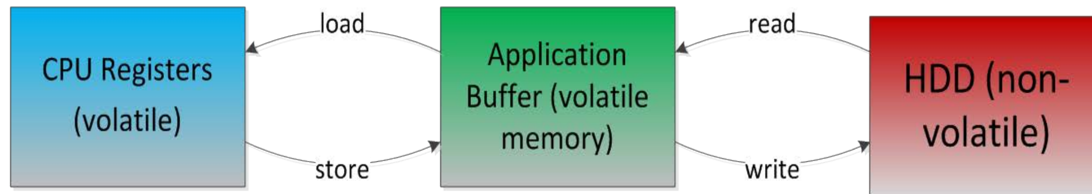
Basic file mode example

- Prerequisites
  - Basic application to update one record (of many) residing on disk; the records are not aligned on block boundaries
  - User provides the info needed to update the record
- App allocates memory to hold the record
- App asks FS to read the record by file name, address within the file, length, and allocated memory address
  - FS may need to read blocks containing the record into FS cache; then the record is copied to app's memory
- App modifies the record in memory
- App asks the FS to write the record to same location
  - FS may need to read blocks to FS cache. The app's record in memory is copied to FS cache and a write-to-disk is requested
- App exits, record may not have been written to disk (yet)

Flash Memory SUMMIT

Block-aware application mode

- Application goals include optimized performance and/or control over actual commit to disk
- Application disables file system cache
  - App aligns IO requests to MMU page sizes
  - App may manage its own cache



- Note: Although most apps use basic mode, block-aware apps are often critical to businesses
  - databases, HPC, big data
  - may drive storage purchase decisions

Block-aware Application Example

- Update a record on disk
- Prerequisites
  - Same as before: the records are not aligned, user provides info required to update the record
- App allocates memory to hold all the blocks containing the record
  - Record may not start or end at block boundary
- App asks the FS/driver to read records to memory
- App updates the record in memory
- App asks the FS/driver to write blocks to same location on disk
- (unlike basic example), record has been written to disk

Memory Mapped Files

- Application maps a file to virtual memory
  - App asks FS to mirror virtual memory and disk blocks
- Application does not do FS read/write commands
  - Requests to update mapped virtual memory cause FS to issue disk IO in the background
  - App may issue sync commands to force synchronization
- Often used by block-aware applications

Emerging NVM extensions

- The modes previously described apply to all HDDs and SSDs
- What NVM-specific extensions are in the works?
  - Trim/Unmap
  - Atomic writes
  - Discovery of granularities
  - Persistent memory modes

Trim/Unmap background

- Trim(ATA)/Unmap(SCSI) commands already in place, but are good examples of commands (partially) specific to SSDs
- On HDDs, each SW write to block # X is directed to the same physical block
  - Ignoring infrequent block reallocation
  - The same block can be rewritten frequently
  - HDDs don't require a command for SW to say "I'm done with these blocks" – SW stops doing IO until something triggers reuse of the blocks

Trim/Unmap background (continued)

- On SSDs, a block must be erased prior to being re-written
  - Erases are time-consuming, background garbage collection (GC) minimizes latency to subsequent write commands
  - Erasing is done on (device-specific) groups of blocks; GC logic will relocate non-erased blocks in order to fee up an entire group
- Pre-Trim, SSDs didn't have enough info to differentiate valid blocks from blocks that SW knows will never be read
  - E.g. admin created a partition; moved contents to a larger partition; and not (yet) reused the old space.
- Trim allows SW to give hints to SSD garbage collection
  - No need for GC to move trimmed blocks
  - May free up entire group for erase without moving any blocks
  - Or at least minimize the number of moves
  - Moving blocks may cause additional erases, which reduces endurance

Trim/Unmap – the extension

- Proposals to improve SCSI Unmap
  - NVM Programming Model proposes SW extensions based on SCSI committee proposal
  - Original implementations were hints to SSDs – not clear what to expect if a different app read an unmapped block
    - Access to data that should be protected?
    - All zeros?
  - Proposal - more specific about the behavior of subsequent reads
    - Allow options, but let SW determine (and optionally control) which option is in effect
  - Also, a new command to get the status of a block

Atomic Writes

- Mission-critical applications have strategies to write data in a way that allows graceful recovery after power (or other) failures
  - Worst-case problem: torn write – part (but not all) of the data being written is on the disk after a failure
- Atomic Writes ensure that operations are completed in their entirety or not at all
  - Highly desired feature from application developers
- Some devices have a write atomicity unit
  - writes of this size or smaller are considered atomic
- Other devices have atomic write commands
  - may also provide assurances about ordering relative to other commands

Discovery of granularities

- SSDs often have 4096 byte blocks, but claim to have 512 byte blocks for compatibility
- SSDs allow applications to discover logical and physical block size
  - But definitions of "logical" and "physical" are fuzzy
- Applications may wish to know
  - Atomic write unit (if supported)
  - ECC size (in case of ECC failure, all bytes in this range become unavailable)
  - Granule SW should use to avoid read-modify-write inside SSD

Persistent Memory (PM)

- From a SW perspective: PM hardware is accessed by SW similarly to systems memory, but contents are preserved across restarts
  - PM may be physically like system memory – e.g. NVDIMMs
  - May be PCIe card with appropriate driver/flash-translation-layer
- At this time, PM hardware is relatively expensive and tends to be designed for special-purpose applications
- It seems inevitable that prices will drop ands PM will be viable for general-purpose application use
- Which leads to the question – how should SW access PM?

Potential SW views of PM

- One approach is to use PM like an SSD – provide a driver that allows apps to use the existing file and block-aware modes
  - Existing apps work unmodified
- Recall disk stack diagram – drivers provide common abstraction to disks
  - A driver can be used to make a PM address range appear to be a disk
- Existing file systems and applications should work without modification

Disadvantages of PM emulating disks

- Memory does not have blocks and memory-to-memory transfers don't require MMU page alignment; minimize size access tied to CPU cache line (typically 8 bytes)
  - Consider an application updating a 16-byte field
  - With a disk emulation mode, the file system reads 4096 bytes, modifies the 16-byte filed, then writes 4096 bytes
- Memory does not have HDD latencies
  - Minimal benefit in read-ahead or grouping writes
  - No benefit in using file system cache

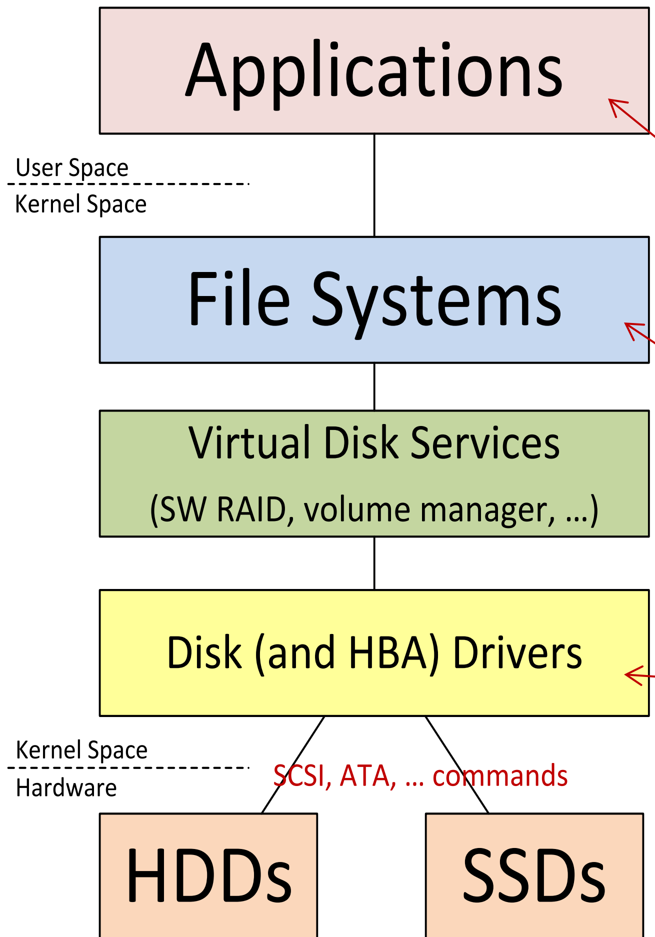
New programming mode for PM

- Recall the file and block mode examples; the application:
  - allocates "empty" volatile memory from pool
  - reads content from disks into memory
  - uses load/store mode to manipulate data in memory
  - writes selective memory contents to disk for persistence
- Redo as a PM-aware programming mode: the application
  - accesses the same PM it used previously
  - uses load/store mode to manipulate data in memory
  - That's it! No need to move data to/from disk (real or emulated)

New programming mode for PM

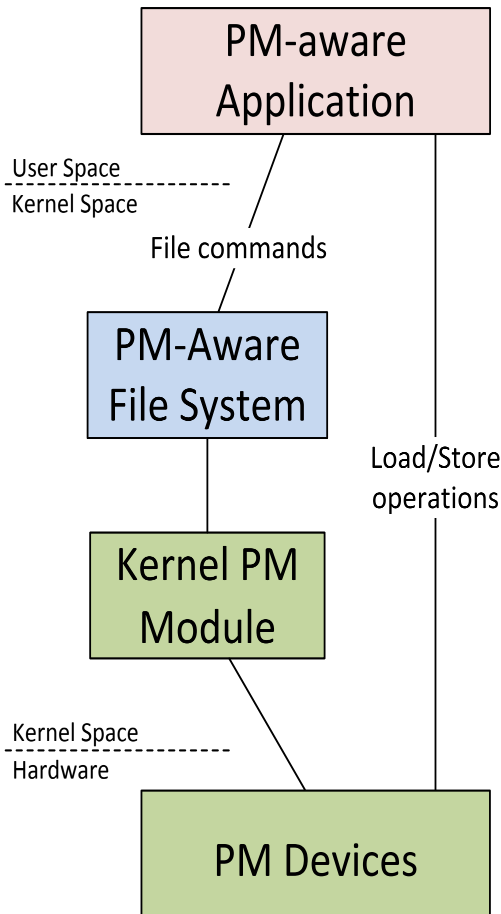
- From application perspective, this PM-aware mode is similar to memory-mapped files
  - File systems (mostly) hide the synchronization between memory and disk
  - Provide sync command so application can flush pending write to disk
- Tricky bit: allowing the application to access the same memory it used previous
  - Also, prevent unauthorized access to memory content
- Proposal: use a file system as a way to name PM regions
  - Provides access control – existing tools and procedures for controlling file access apply
  - Existing applications using memory mapped files require minimal changes to use this approach

Key takeaways for Programming SSDs



- SSDs "mostly" treated like disks
- Most SW treats disk storage in a HW-independent approach
  - Applications operate on files (byte addressable) or block ranges within files
  - Kernel components (file systems, …) operate on block ranges (MMU page size granularity)
- Knowledge of disk hardware (SCSI, ATA, vendor-specific, …) limited to disk drivers

Key takeaways for PM



- PM-aware software benefits from a leaner stack - depicted to the left
  - Path using PM-Aware File System needed
    - To verify app is allowed to access PM file
    - When app uses file operations
  - Load/Store between app and PM has minimal overhead
- PM may be treated like a legacy SSD
  - PM vendor provides a disk driver to the disk stack on the previous slide
  - Compatibility with existing applications and file systems

Flash Memory SUMMIT

Wrap Up

- SNIA's *NVM Programming Model*
  - Proposing ways for SW to utilize NVM extensions
    - Including proposals for utilizing emerging persistent memory
  - Spec organized in terms of Modes presented here
  - Draft specification now available for public review
- More info…
  - TWG portal page with link to draft: http://snia.org/forums/sssi/nvmp
  - Questions/comments? nvmptwg-info@snia.org