SNIA NVM Programming Model

Paul von Behren

Intel Corporation

Overview

- Why is NVM programming important?
- What is NVM (relative to the programming model)
- The programming modes covered in the model
  - Modes for devices providing block storage behavior
  - Modes for devices providing memory behavior
- Wrap-up
  - Key takeaways for developers

Why is NVM Programming Important?

- From the perspective of file system and applications, most NVM hardware emulates hard disks
  - With some NVM specific behavior
- But applications may benefit from SW extensions tailored to NVM hardware
  - For example, ATA Trim command to enhance SSD endurance
  - Requests from users/developers to enhance SW stack to enable access to NVM extensions
  - Emerging persistent memory may benefit from new programming approach

SNIA NVM Programming TWG

- Members:
  - Calypso Systems, Cisco, Contour Asset Management, Dell, EMC, FalconStor, Fujitsu, Fusion-io, HP, HGST, Hitachi, Huawei, IBM, IDT, Inphi, Intel, Intuitive Cognition Consulting, LSI, Marvell, Micron, Microsoft, NEC, NetApp, OCZ, Oracle, PMC-Sierra, Qlogic, Red Hat, Samsung, SanDisk, Seagate, Sony, Symantec, Tata Consultancy Services, Toshiba, Viking, Virident, VMware
- Charter:
  - Develop specifications for new software programming models as NVM becomes a standard feature of platforms
- Initial Deliverable: SNIA *NVM Programming Model*

What is NVM?

- As used in the SNIA *NVM Programming Model*, NVM is:
  - Any type of non-volatile memory
  - Includes disk form factor SSDs
    - Using SATA, SCSI or vendor-specific commands
  - Includes NVM PCIe cards
    - Using NVM-Express, SCSI-Express, SATA Express, vendor-specific or other commands
  - Also includes emerging persistent memory (PM) hardware
    - Such as NVDIMMs

NVM Programming Model

- Specification being created by the SNIA NVM Programming TWG
- Goal: define common behavior allowing applications to use NVM optimally
  - Encourage OSVs and storage vendors implementing the NVM in storage stacks to support this behavior
- Behavior covers extensions to existing block addressable NVM software stacks and new approach for persistent memory
- Addresses software above the hardware-specific drivers
  - In other words, utilizes (but does not compete with) SCSI, ATA and NVM-Express

Goals for 1.0 specification:

- Functionality identified as high-priority to application developers
  - Primary focus – power failure recovery
- System-call level (C language focus)
- For 1.0, TWG opted to defer management-only and diagnostic behavior
- Other items deferred
- Plans to continue work on deferred items

Programming Model vs. API

- OSVs own their kernel APIs
  - Cannot define one API for multiple OS platforms
  - Next best thing is to agree on overall model
    - With OSV collaboration
    - Then engage OSV to define and implement API
- Similar situation in user-space
  - A common API doesn't always make sense
    - Example: the UNIX* versus the Windows* event models
    - Windows tends to C++; POSIX OSes tend to C
  - Ultimately: want OSV to ship and maintain the API
- *NVM Programming Model* defines behavior without specifying an API
  - Implementations are expected to provide "mapping documents"

NVM Programming Model

- Four modes representing common storage related service points (details in next few slides)
- Each mode defines a set of actions and attributes
  - Define behavior, not API
- Actions/attributes may map to SCSI/ATA/NVMe behavior
- Actions/attributes map to unique behavior
  - Avoid overloading: SCAR rather than “WRITE LONG with a specific set of options”
  - Different terms in related standards (e.g. DISCARD maps to ATA TRIM, SCSI UNMAP, and NVMe DEALLOCATE)

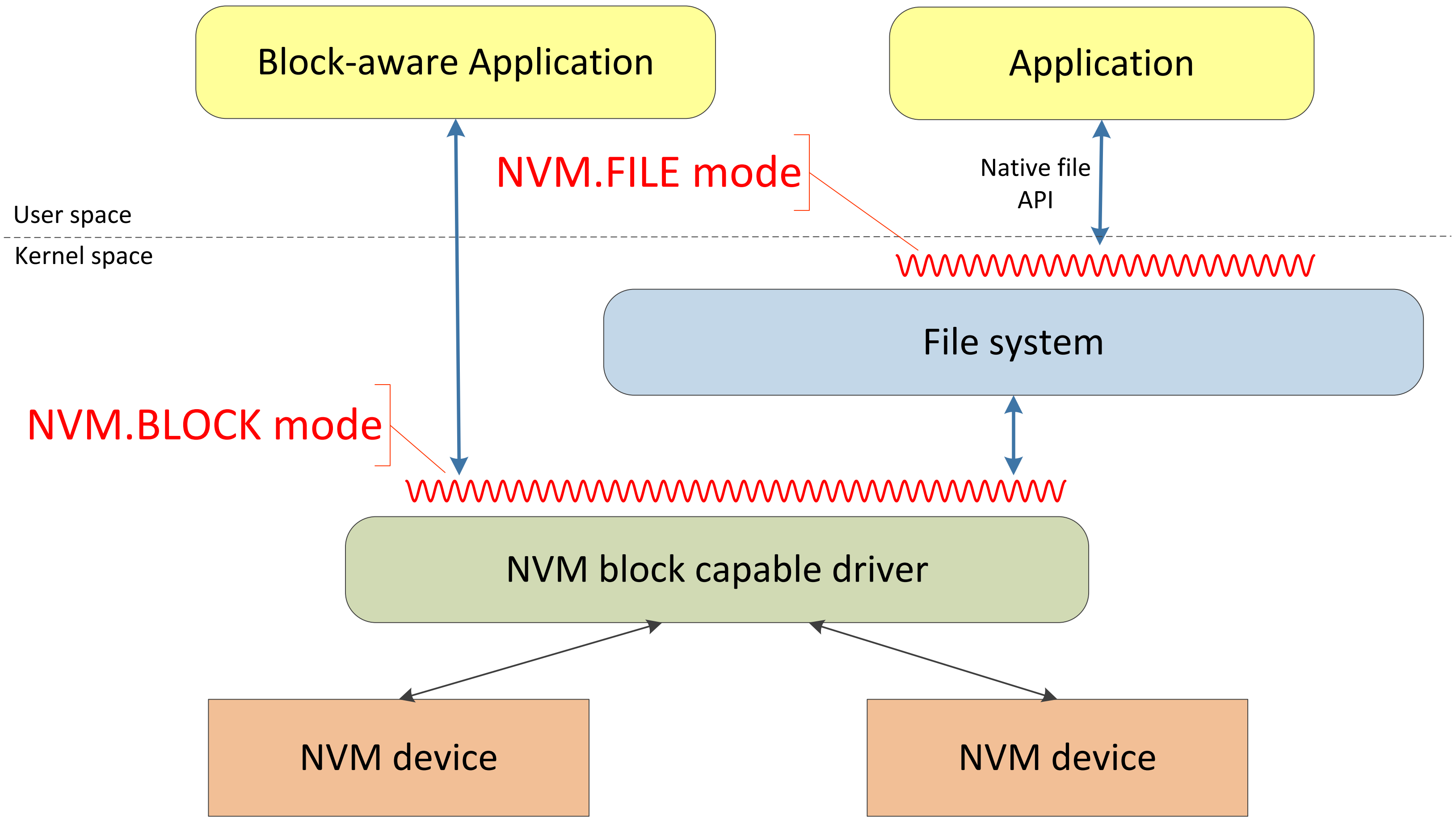
NVM Programming Modes

- *NVM Programming Model* addresses four modes
  1. NVM.FILE: Applications using file systems optimized for block storage (HDDs/SSDs)
  2. NVM.BLOCK: File systems, other kernel components (and advanced applications) optimized for block storage
  3. NVM.PM.FILE: Applications using file systems optimized for persistent memory
  4. NVM.PM.VOLUME: File systems optimized for persistent memory
- More modes may be added later (for new types of HW)

Block storage modes

- NVM.FILE mode - primary targets:
  - Applications striving to optimize I/O for NVM storage
  - Mission-critical applications needing to assure data on disk is consistent even when write operations are interrupted by power failures
- NVM.BLOCK mode – primary targets:
  - File systems
  - Other operating system (OS) components (such as hibernation) that directly use storage
  - Applications that use block commands directly (without a file system)

NVM.BLOCK/FILE Modes



NVM.FILE: Granularities

- Additional device granularities (such as block sizes)
  - SSDs typically have different physical and logical block sizes
    - And have commands allowing software to determine these sizes
  - There are additional granularities of interest to software
    - Does the device implementation assure that write operations smaller than a given size will not be "torn" due to a power failure?
    - Does the device have a checksum that applies to multi-block granules? If so, it may be impossible to read any block in the checksum granule, if one block is unreadable.
    - These granules may not be the same as logical or physical block size
    - Software can use information about these granularities to place data in a way that survives certain types of failures

NVM.FILE : Atomic Write

- Atomic writes are an emerging NVM device feature that assures data is consistent when a write operation is interrupted (for example, power failure)
  - Device assures that all blocks are written or no blocks are written
- Applications may write data twice to achieve the same assurance for devices that don't support atomic write
  - On restart, the application uses the two copies to recover
- Some NVM devices also support atomic write to *non-sequential* block addresses
  - Common application behavior: e.g., update data record and metadata

NVM.BLOCK Enhancements

- Atomic Write Behavior and Granularities
  - Similar to what's defined for NVM.FILE
- SCAR action
  - Provides a way for one software component to mark blocks so that other software components see an error when they attempt to read the blocks.
    - For example, asynchronously to a user request, application determines data inconsistent; it uses SCAR to mark the data. If the user causes the data to be re-rewritten before being read, then the SCAR contrition is removed.
  - SCSI pseudo unrecovered errors (created by WRITE LONG)
  - The error condition is removed when the blocks are written

NVM.BLOCK: Discard

- SSDs often have strategies to help increase endurance and performance by organizing the way blocks are written
  - *Wear leveling* is more effective when a higher percentage of blocks are known to be unused by the OS
  - Discard (AKA Trim) commands provide a way for software to inform the SSD that blocks are no longer in use
- First generation of discard commands met objectives for wear leveling
  - But the results of subsequent reads were not predictable
  - A read of a discarded block might return zeros, old data, a device-specific pattern
    - Applications generally don't read discarded blocks, but diagnostic and backup software might

NVM.BLOCK: Discard

- Standards (and devices) are being updated to address the non-deterministic read behavior
  - And defining ways to allow software to determine which discard variants are supported
- *NVM Programming Model* defines behavior for software to
  - determine what types of discard commands a device supports
  - issue discard commands
  - determine whether a block has been discarded

Persistent memory characteristics

- Within the scope of this presentation, PM is:
  - Not tablet-like memory for entire system
  - Not flash (as commonly used today)
- PM is Byte-addressable (from programmer's perspective)
  - Contrast with block addressable SSDs – updating a few bytes on disk requires software to read, modify, then update the block
- Load/store access
  - Not demand-paged
- Memory-like performance
  - Would reasonably stall a CPU load waiting for PM
- For modeling, think: Battery-backed DRAM
  - But HW may use other form-factors

Impact of new PM programming modes

- What types of applications benefit from PM-aware programming?
  - In-memory databases
    - No need to wait for memory to be re-populated from disk at startup
  - Applications using small data objects
    - For example: cloud software object stores, key/value or NoSQL databases
  - Persistent caches
    - Caches get "smarter" over time – learn what data benefits most from residing in cache
    - With volatile memory, cache SW must re-learn after each restart

Integrating PM with software

- Software can't use PM without modification
  - Current practice: applications allocate memory by size
    - Memory contents zero or undefined; applications must initialize
    - Memory allocated from pool; may not get same memory across a restart
- Three approaches to enable SW use of PM
  1. Hide PM from legacy applications, make it available for specific applications through non-standard APIs
     - Great for first generation SW; explore PM capabilities, limit issues
     - SW apps typically wait for OS integration to use new hardware
  2. Make the PM appear to be "virtual SSDs"
     - Use NVM.BLOCK/NVM.FILE modes; existing apps work without modification
     - Doesn't allow apps to take advantage of PM features
  3. Create new programming modes for optimal use of PM …

Goals for new PM modes

- Today's programming mode for applications that save/update data
  - Use volatile memory as workspace
    - When allocated, volatile memory is effectively empty
    - Copy previously saved data from files to volatile memory
  - Applications make updates to volatile memory,
    - Then save updates to files
- Proposed application programming mode for PM
  - Re-connect to same PM previously used
  - Don’t need the steps to move data between memory and files
  - Use existing OS and file system behavior where appropriate

Adapting malloc() for PM

- malloc(len) allocates len bytes and returns a pointer
  - The allocated memory may come from different physical memory addresses each time malloc is called
  - No parameter to provide a name representing the same memory (and content) previously used
- A better approach for PM is for the caller to specify a name (maps to the PM's physical address) plus optional length, offset, …

Memory-mapped Files

- This goal for accessing a named PM “chunk” led to revisiting *memory mapped files*
  - Existing feature of block file systems
  - POSIX mmap() / Windows MapViewOfFIle()
- Application usage (basic database use case)
  - App mmaps the file to virtual memory
  - App loads stores records relative to address returned by mmap
    - Load/store - memory-access commands: memcpy, variable assignment, …
  - File system pages file in/out of process memory as needed
    - Application can msync() to force flush to disk
  - App can restart and continue

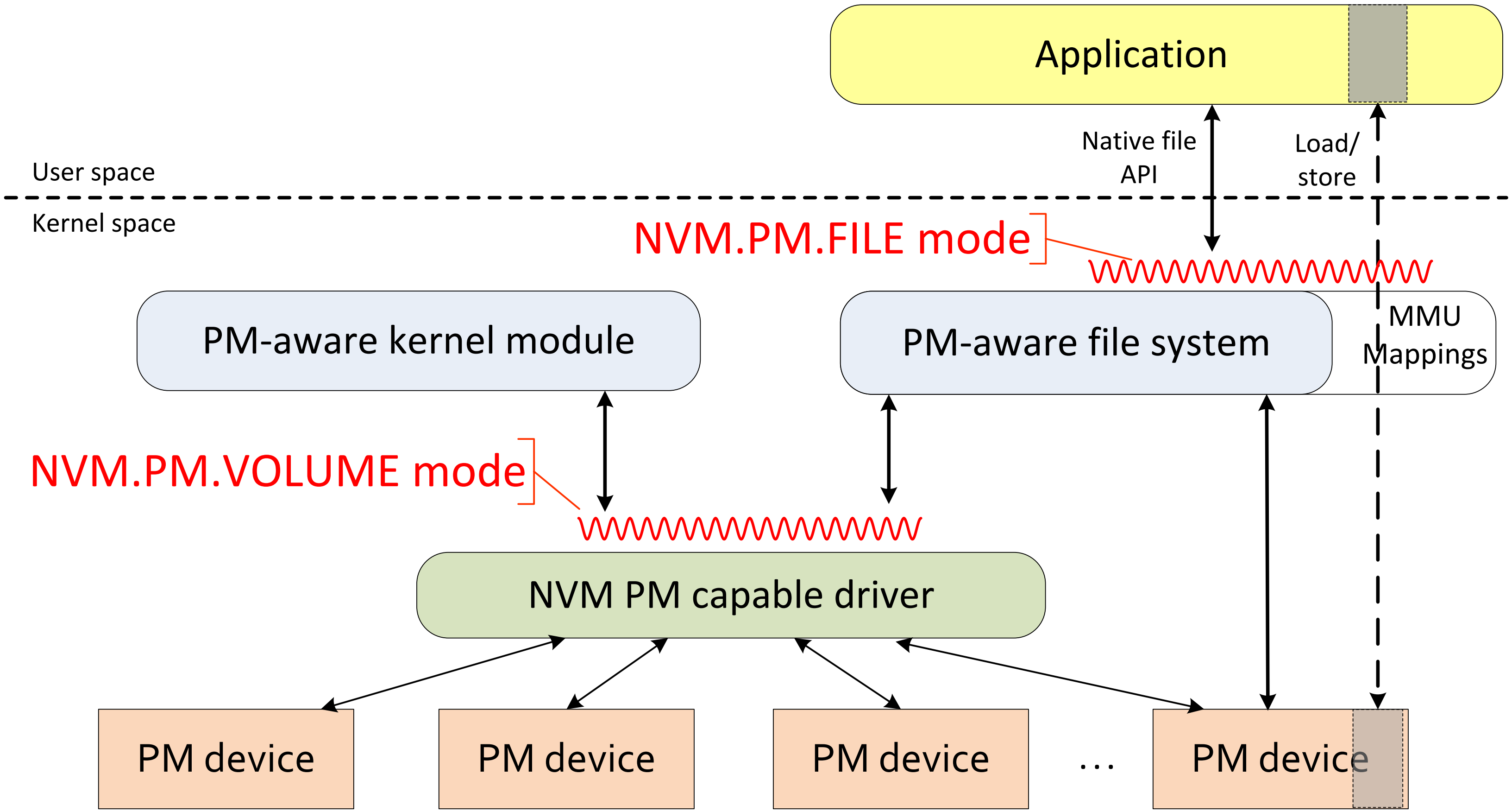
Memory Mapped Files and PM

- The basic usage still applies
  - App uses something like mmap() to access named PM volume
  - Load/store access to PM
  - Something like msync() needed to flush data from cache lines and PM device caches
- But there are differences
  - The file system does not page data to/from disk
  - Block storage msync() aligned to MMU pages sizes (typically 4K)
  - PM flush alignment is hardware specific; may align to cache lines
  - Memory errors are reported asynchronously

Persistent Memory Modes

- NVM.PM.FILE mode - primary targets:
  - Applications wishing to assure saved data is consistent even when operations are interrupted by power failures
  - Applications pursuing optimal PM performance
- NVM.PM.VOLUME mode - primary targets:
  - PM-aware file systems
  - Other PM-aware OS components

Persistent Memory Modes



NVM.PM.FILE actions

- MAP: unlike native mmap, MAP enables direct load/store access. Interactions with native APIs may be undefined.
- SYNC: range not required to be MMU page aligned
- In general, MAP and SYNC work like native APIs, but are optimized for PM
- OPTIMIZED_FLUSH: extends SYNC to support multiple byte ranges
- OPTIMIZED_FLUSH_AND_VERIFY: adds read/verify
- GET_ERROR_INFO: Generic interface to details of PM errors

NVM.PM.VOLUME mode

- GET_RANGESET
  - Allows an OS component (like a file system) to map volume-relative addresses to physical memory addresses
- SYNC actions
  - Hardware-independent interface to flush address ranges
  - Supports smaller byte ranges than POSIX msync
- Granularities
  - Similar to NVM.BLOCK – allow OS components to place data to prevent single PM error from impacting two copies of same data
- Discard
  - Similar to NVM.BLOCK – allow OS components to tell PM controller that a range of bytes is no longer in use

Wrap-up: Status of specification

- TWG is reviewing 1.0 “release candidate”
  - A few months to complete SNIA-wide approval
- Draft version is available
  - Download location on last slide
- TWG is now evaluations next steps
  - For example, higher-level interfaces for applications (e.g., language support)
  - Management behavior
  - New behavior in storage standards (e.g. access hints)
  - Many of these are mentioned in deferred behavior annex in spec

Key takeaways for developers

- NVM devices are introducing new features that help software meet end-user requests
  - Some are enhancements to legacy block behavior
    - Atomic write, information about granularities
  - New programming model for PM hardware
- Optimal use of PM probably requires revisiting SW design
- Consider how NVM features can be used in your products
- Follow progress in implementation support for the *NVM Programming Model*

What's Next for TWG

- TWG is now planning next steps
  - revisiting work we deferred
  - Write-ordering constraints
  - Remote access to NVM
  - Higher-level atomic write
  - NVM management
  - …
- BOF 7:00 tonight –gathering input on next steps

How to help

- Please review the spec and send comments
  - Is the NVM Programming Model covering key behavior? If not, what's missing?
    - Especially interested in feedback from application developers
  - SNIA members should send comments to TWG directly, non-members can use feedback link in spec
    - Questions about comment process? nvmptwg-info@snia.org
- Better yet, join TWG
  - We are now in the process of defini9ng next areas of work
  - Questions about joining? nvmptwg-info@snia.org

More information

- SNIA Solid State Initiative portal:
  - Up-to-date information about the *NVM Programming Model*, related materials and software
  - Links to approved versions will be added to this portal
  - http://snia.org/forums/sssi/nvmp
- Draft specification may be downloaded here:
  - http://snia.org/sites/default/files/NVMProgrammingModel_v1r5DRAFT.pdf
- Questions? Comments?
  - nvmptwg-info@snia.org