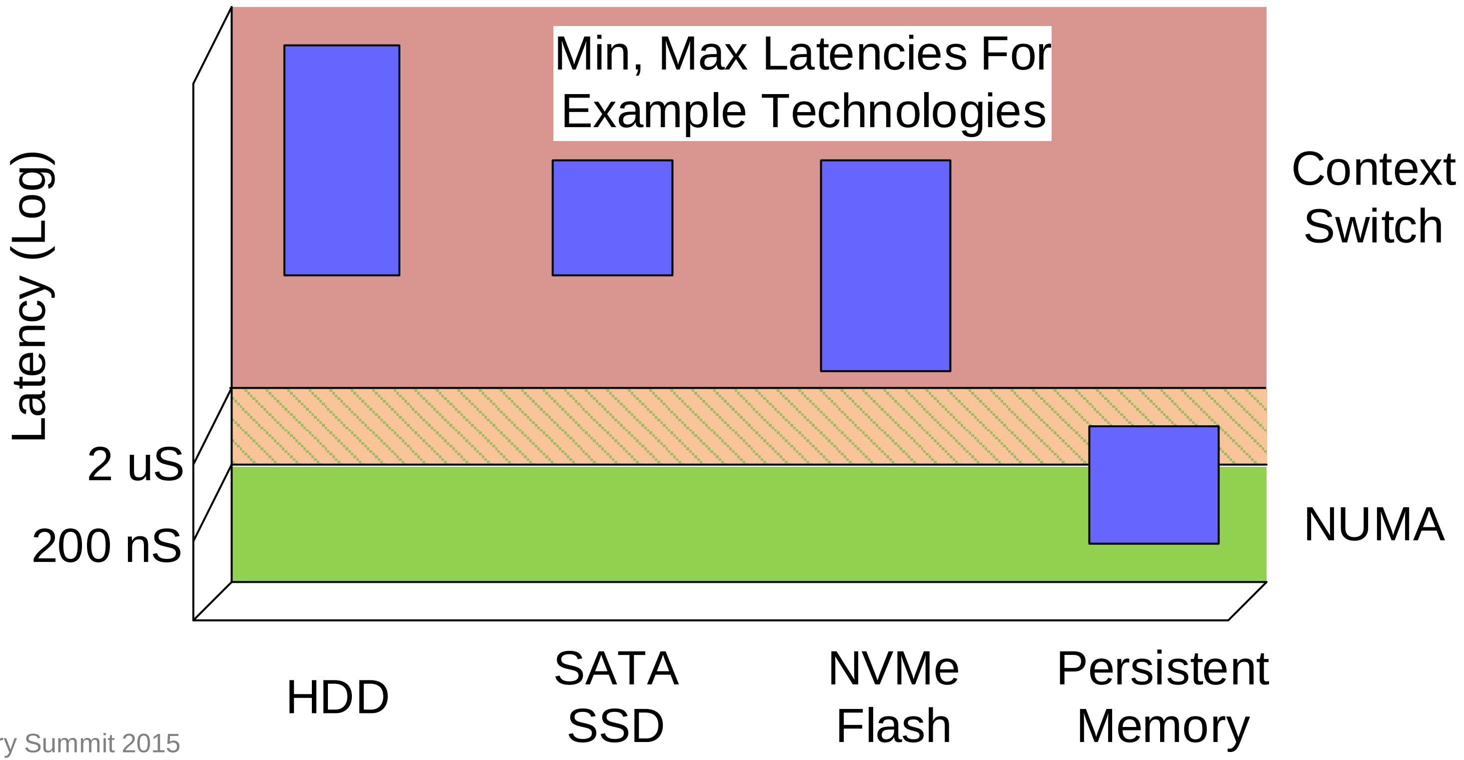
Programming for Non-Volatile Memory

Doug Voigt,
Hewlett Packard (Enterprise)

Contents

- Implications of the NVM Programming Model
  - Map and Sync, Opt flush and verify, Pointers, Atomicity, Exception Handling
- Persistent Memory Data Structures
  - Atomic updates, PM Allocation, Data structure library, transactions
- High Availability
  - Remote Opt Flush, Recovery scenarios, Application level backtracking

Latency thresholds cause disruption



Persistent Memory (PM) is a type of Non-Volatile Memory (NVM)

- Disk-like non-volatile memory
  - Appears as disk drives to applications
  - Accessed as traditional array of blocks
- Memory-like non-volatile memory
  - Appears as memory to applications
  - Applications store data directly in byte-addressable memory

"Persistent memory" refers to memory-like non-volatile memory

SNIA NVM Programming Model

- Version 1.1 approved by SNIA in March 2015
  - http://www.snia.org/tech_activities/standards/curr_standards/npm
- Expose new block and file features to applications
  - Atomicity capability and granularity
  - Thin provisioning management
- Use of memory mapped files for persistent memory
  - Existing abstraction that can act as a bridge
  - Limits the scope of application re-invention
  - Open source implementations available
- Programming Model, not API
  - Described in terms of attributes, actions and use cases
  - Implementations map actions and attributes to API's

Block Access NVM

No Application Functionality Change

Implications of the NVM Programming Model for Persistent Memory Applications

Persistent memory modes

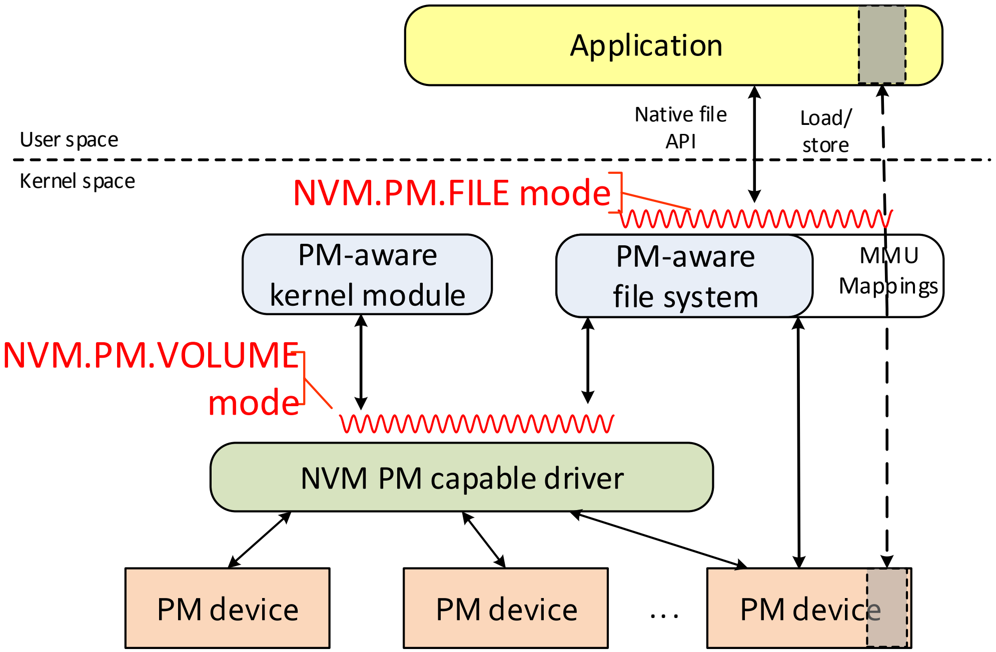
Use with memory-like NVM

NVM.PM.VOLUME Mode

- Software abstraction to OS components for Persistent Memory (PM) hardware
- List of physical address ranges for each PM volume
- Thin provisioning management

NVM.PM.FILE Mode

- Describes the behavior for applications accessing persistent memory Discovery and use of atomic write features
- Mapping PM files (or subsets of files) to virtual memory addresses
- Syncing portions of PM files to the persistence domain



Map and Sync

- Map
  - Associates memory addresses with open file
  - Caller may request specific address
- Sync
  - Flush CPU cache for indicated range
  - Additional Sync types
  - Optimized Flush – multiple ranges from user space
  - Optimized Flush and Verify – Optimized flush with read back from media
- Warning! Sync does not guarantee order
  - Parts of CPU cache may be flushed out of order
  - This may occur before the sync action is taken by the application
  - Sync only guarantees that all data in the indicated range has been flushed some time before the sync completes

Pointers

How can one persistent memory mapped data structure refer to another?

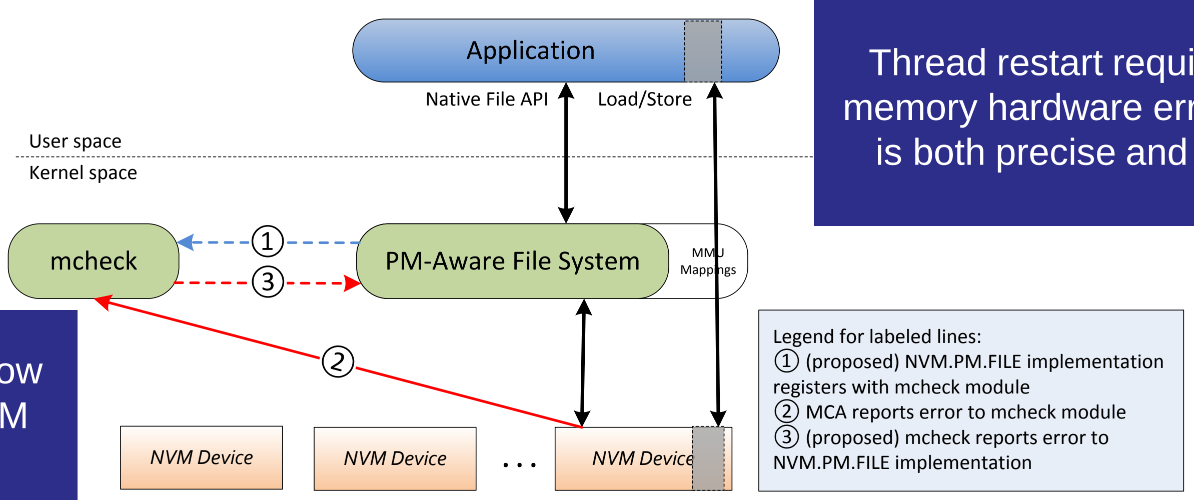
- Use its virtual address as a pointer
  - Assumes it will get the same address every time it is memory mapped
  - Requires special virtual address space management
- Use an offset from a relocatable base
  - Base could be the start of the memory mapped file
  - Pointer includes namespace reference

Failure atomicity

- Current processor + memory systems
  - Guarantee inter-process consistency (SMP)
  - But only provide limited atomicity with respect to failure
    - System reset/restart/crash
    - Power Failure
    - Memory Failure
- Failure atomicity is processor architecture specific
  - Processors provide failure atomicity of aligned fundamental data types
  - Fundamental data types include pointers and integers
  - PM programs use these to create larger atomic updates or transactions
  - Fallback is an additional checksum or CRC

Error handling – exceptions instead of status

Figure 1 Linux Machine Check error flow with proposed new interface



Thread restart required unless memory hardware error detection is both precise and contained

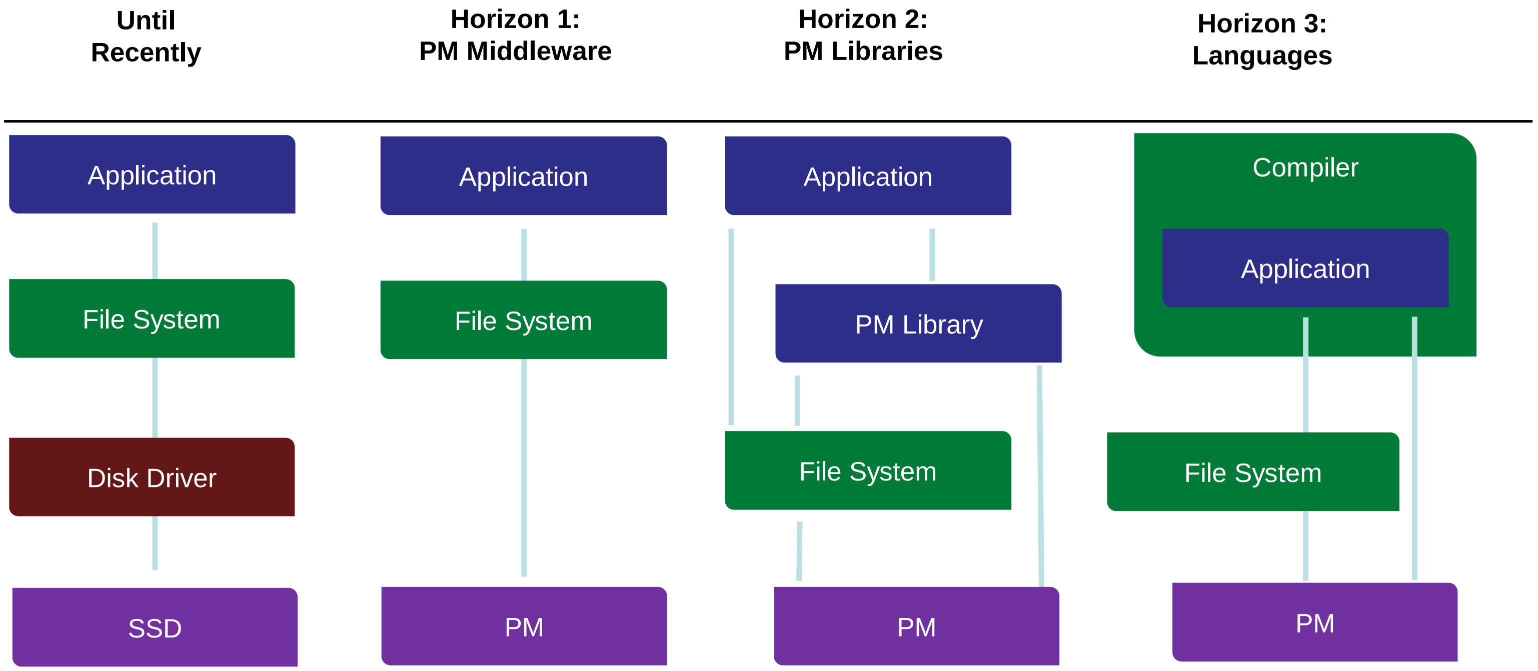
New machine check flow to integrate file and PM level recovery

Precise: exact memory location(s) are identified
Contained: instruction execution can be resumed (RTI)

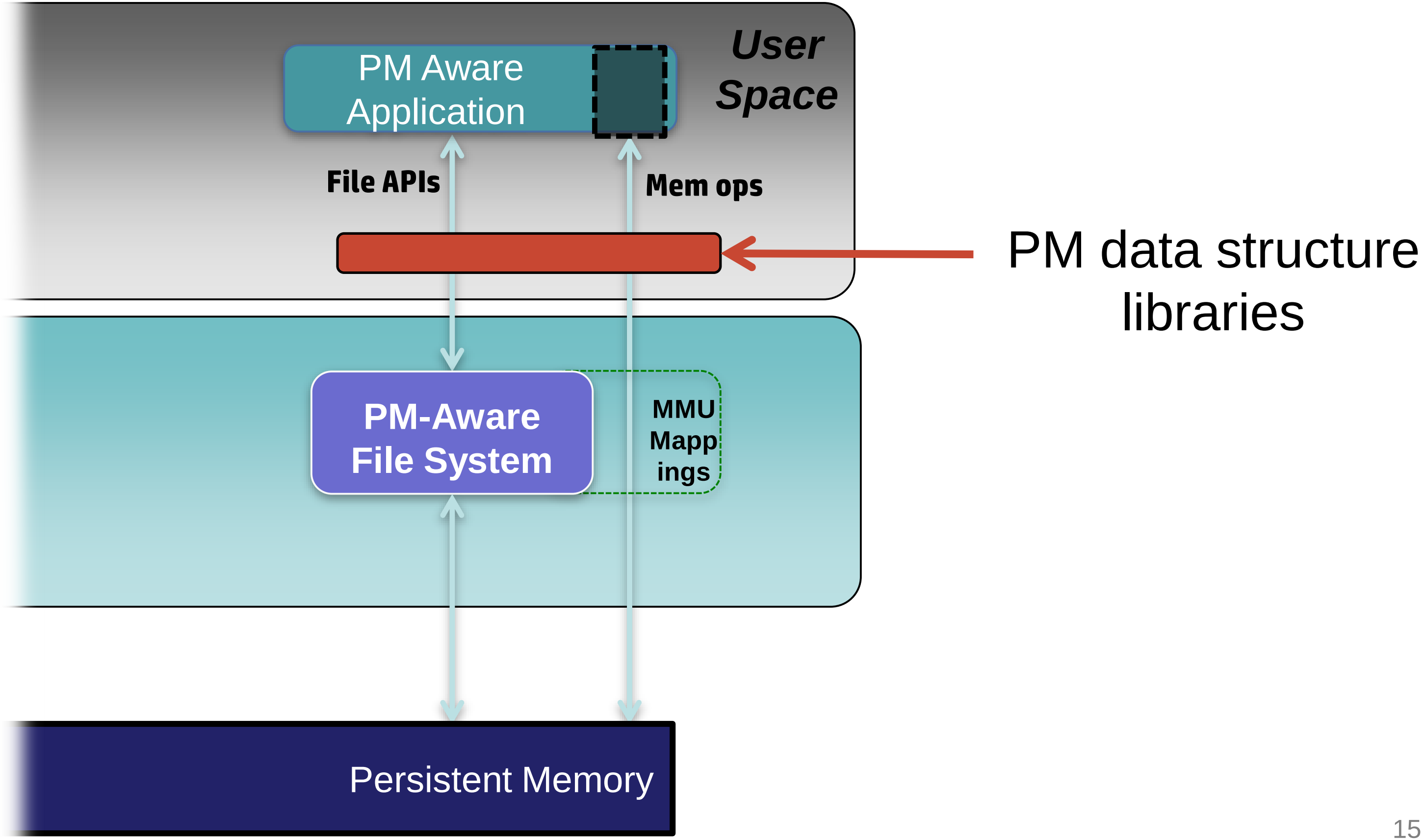
Application gets exception if file level recovery fails or backtracking is needed

Persistent Memory Data Structures

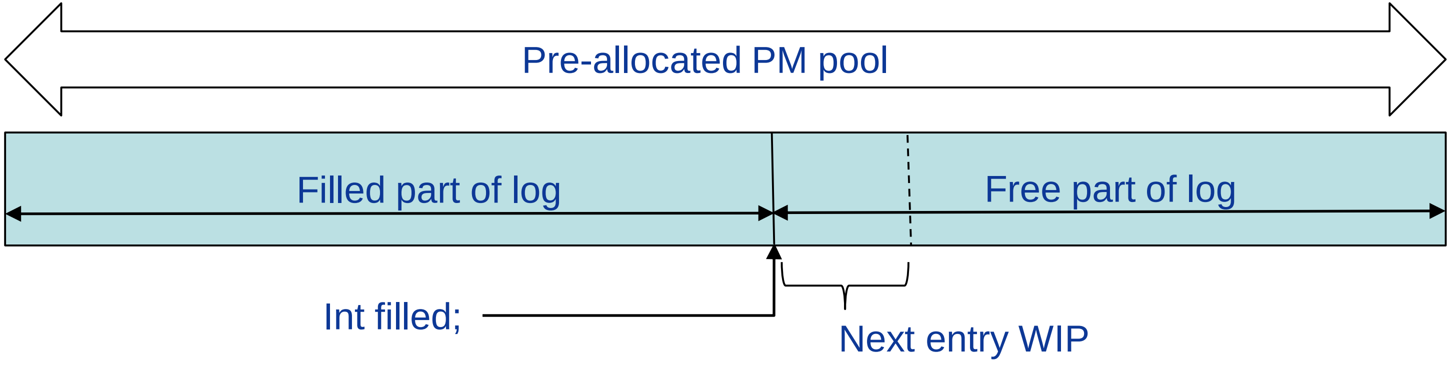
Application horizons



PM data structure libraries



Trivial example: append only log



Append pseudocode:

<Create new log entry in free space>

Sync(new entry);

filled = filled + size(new entry); # Atomic update to fundamental data type

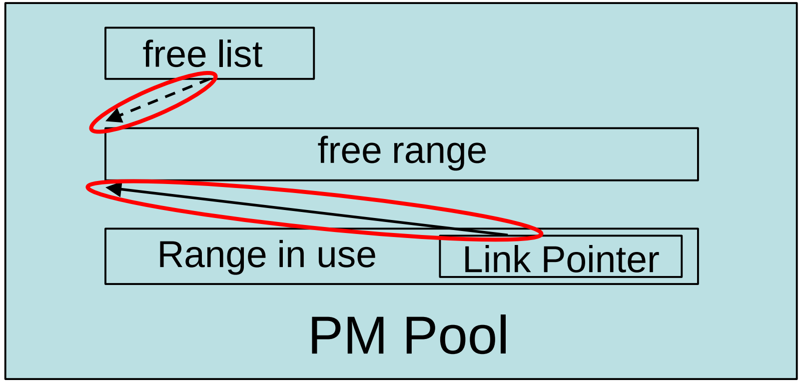
Sync(filled);

Transactional allocation

- Pmalloc – Allocate space for persistent data structures
  - Allocates ranges of memory mapped PM from a pool or file
  - PM memory allocation survives power loss
  - PM space management information (free list) must be persistent
- PM allocation must be atomic
  - Failure before completion of data structure creation must roll back allocation
  - Requires a common anchor object for transactions and space management

Linked list example

Free list and link pointers must be updated atomically



Link pseudocode:

<Temporarily allocate free range for new item>

<Create new item in temporarily allocated space>

<Transactionally update link pointer and free list>

Larger transactions

- Atomic updates to arbitrary data structures
  - Transactions delimited by Begin, End indicators
  - Ranges to be atomically updated are registered using add_range
  - Transaction object tracks and manages ranges
    - Capture pre-image and roll back on abort
    - Sync/Flush data to persistence domain on commit
- Groups of data structures can participate
  - Within the same PM pool
  - Cataloged under a common root

Pmem.io Library

- http://pmem.io/nvml
- PM assist functions
  - Map, Sync, Allocation
- PM Data Structures
  - Log, Block
- PM Object
  - Root, Transactions, Type Safety and more

Language vs. Library

- Features similar to pmem can be integrated into standard programming languages
  - More convenient
  - More sophisticated
  - Safer

http://www.hpl.hp.com/techreports/2013/HPL-2013-78.pdf

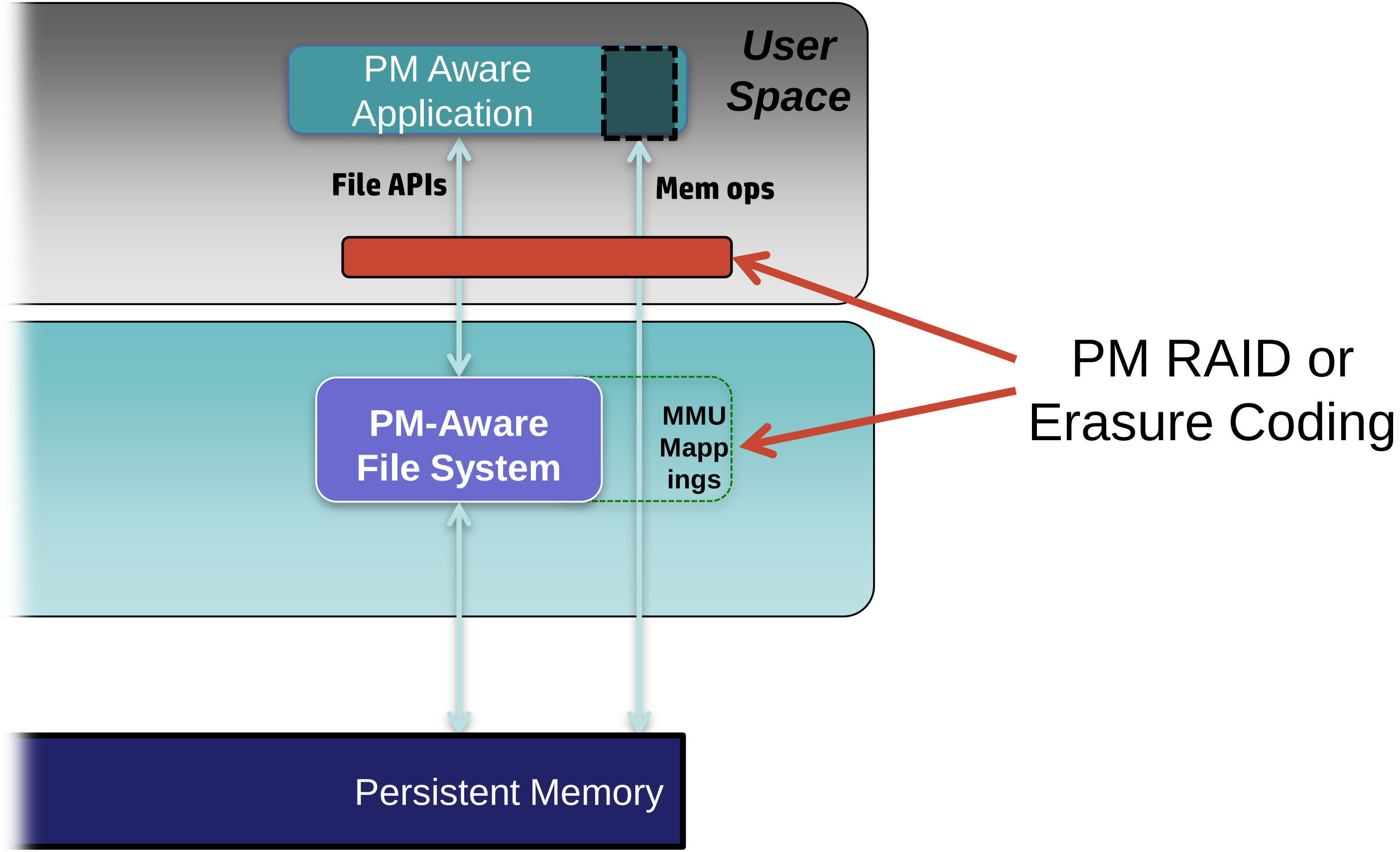
Failure atomic code sections based on existing critical sections

http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf

NVM region file management, transactions with locks, heap management

Failure Recovery

PM fault tolerance



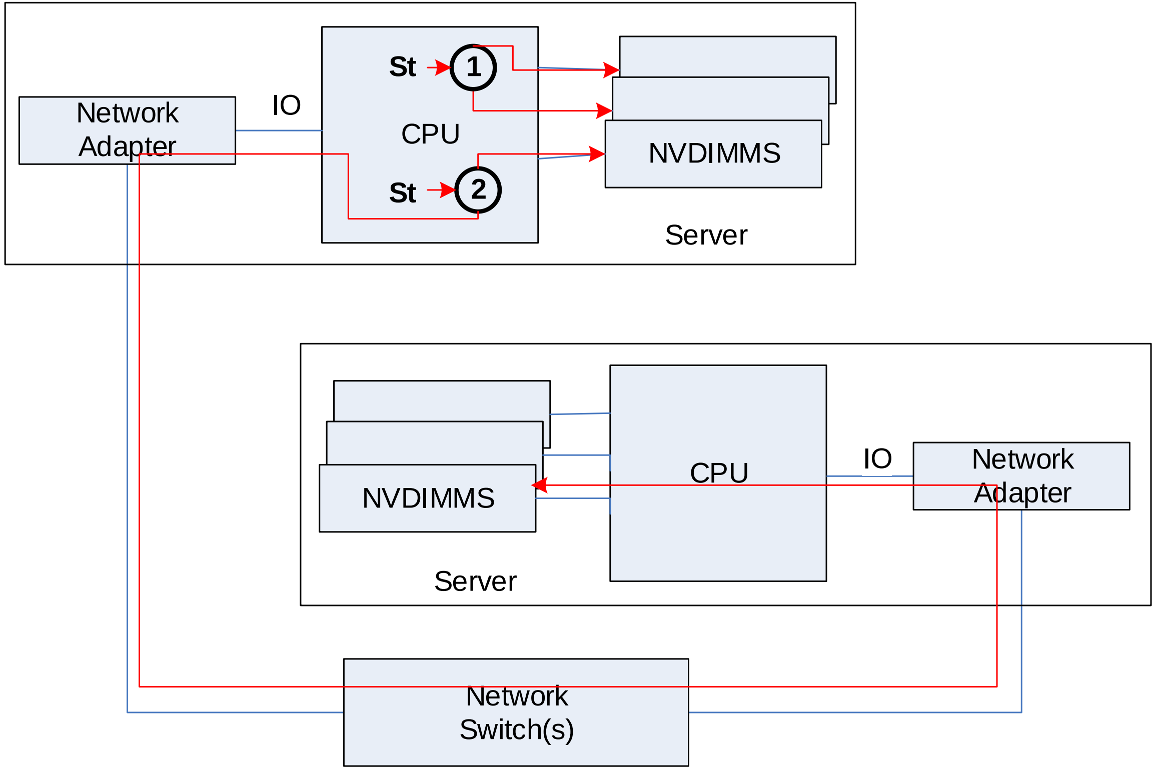
Durability and Availability

Durability

- Ability to (eventually) recover data after failure
- e.g. Local mirroring (1)
- Does not guarantee continuous access

Availability

- Ability to continuously access data regardless of failure
- Requires cross-node redundancy (2)
- Availability requires durability



Recovery AND Consistency

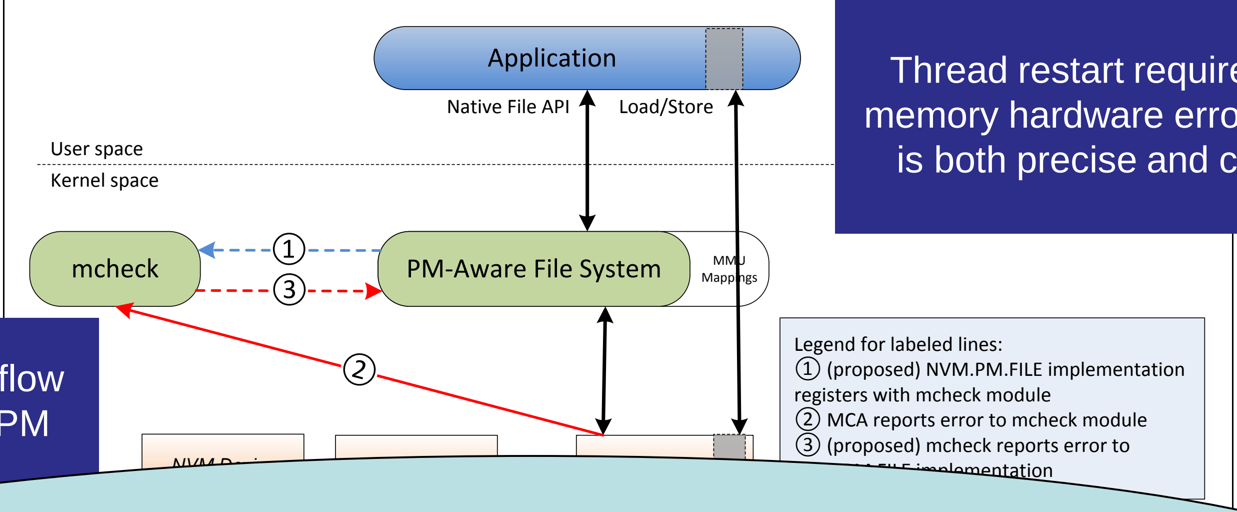
- Application level goal is recovery from failure
  - Requires robust local and remote error handling
  - High Availability (as opposed to High Durability) in today's systems requires application involvement.
- Consistency is an application specific constraint
  - Uncertainty of data state after failure
  - Crash consistency
  - Higher order consistency points such as transactions
  - Atomicity of Aligned Fundamental Data Types

Remote Access for High Availability

- SNIA NVMP TWG work in progress
  - Use today's RDMA to explore this use case
  - Agnostic to specific implementation (IB, ROCE, iWARP)
  - Optimal implementation may not always be RDMA
- Recommends Remote OptimizedFlush network service
  - Goal is to minimize latency
  - Requires at least 2 round trips with today's implementations
  - Main issue is assurance of durability at remote site.
- New RDMA completion type helps
  - Proposed in Open Fabrics Alliance IO working group
  - Delays RDMA completion until data is in the remote persistence domain
  - Likely component of remote optimized flush implementation

Error handling – Remember this?

Figure 1 Linux Machine Check error flow with proposed new interface



Thread restart required unless memory hardware error detection is both precise and contained

New machine check flow to integrate file and PM level recovery

Precise: exact memory location(s) are identified

Contained: instruction execution can be resumed (RTI)

or backtracking is needed

Backtracking recovery

- Occurs when PM state is recovered to a recent consistency point
  - Created by remote optimized flush or transaction
  - Requires work in progress to be reconciled by the application
- Detection
  - During an exception
  - During a system or application restart
- Application Response
  - Transaction roll forward or roll back and retry
  - Consistency checking and correction

Recovery scenarios with precise and contained exceptions

- In line recovery
  - When the primary copy of data is lost, the data is recovered during a memory exception without any application disruption
  - Requires stronger replication order than sync or optimized flush
- Backtracking recovery
  - When the primary copy of data is lost, transaction(s) involving the data must be adjusted by the application (roll forward or back)
  - Best case recovery if the secondary copy is not guaranteed to be sufficiently up to date to allow direct replacement

Recovery scenarios without precise and contained exceptions

- Local application restart
  - When the primary copy of data is lost the application must restart on the same server
  - Data is recovered during the restart and must adhere to a consistency mode from which the application is designed to recover with an acceptable RPO.
- Application Failover
  - A node running an application and/or data access is lost so the application must fail over to another node.
  - The data on the new node must adhere to a consistency mode from which the application is designed to recover with acceptable RPO

Application recovery scenarios

Scenario	Redundancy freshness	Exception	Application backtrack without restart	Server Restart	Server Failure
In Line Recovery	Better than sync	Precise and contained	NA	No	No
Backtracking Recovery	Consistency point	Imprecise and contained	Yes	No	No
Local application restart	Consistency point	Not contained	No	NA	No
		NA	NA	Yes	No
Application Failover	Consistency point	NA	NA	NA	Yes

Review

- Implications of the NVM Programming Model
- Persistent Memory Data Structures
- High Availability

Thank You