Taking Advantage of Persistent Memory with Open Source PMDK

PEARC '19 Tutorial

July 29, 2019

Andy Rudoff, Intel

and material from Tom Talpey, Microsoft

Agenda

1:30pm – 2:15pm	Persistent Memory Concepts
2:15pm – 2:30pm	Linux Support
2:30pm – 2:40pm	Windows Support
2:40pm – 3:00pm	Demo: Persistent Memory in the Cloud
3:00pm – 3:15pm	**BREAK**
3:15pm – 4:00pm	Pulling it all together with libraries Allocation, Transactions, Replication
4:00pm – 5:00pm	Intel Profiling Tools (separate presentation)

Reminder: Materials delivered, including any skipped slides

About this presentation

- Andy Rudoff
  - Intel
- Material from Tom Talpey
  - Microsoft
- Repo containing examples & slides:
  - `https://github.com/andyrudoff/pearc19`

After this presentation, you will…

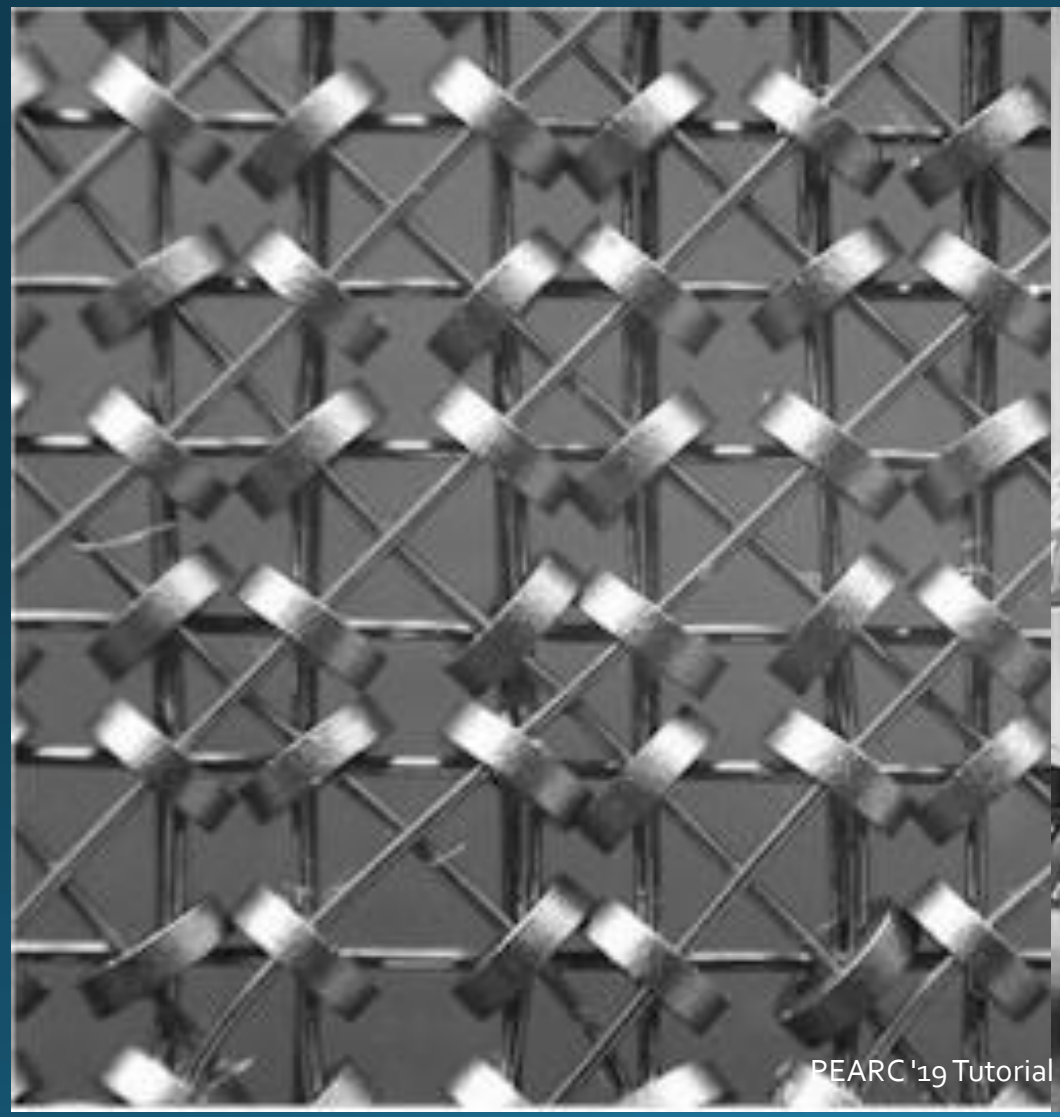
- Know what persistent memory is
- Understand why & when it gets used
- Be familiar with design and development decisions that software developers make when using pmem
- Have seen programming examples
- Know where to go when you need more information!

- Shameless plug: Join us at one of our "hackathons" for intense, hands-on programming experience with pmem

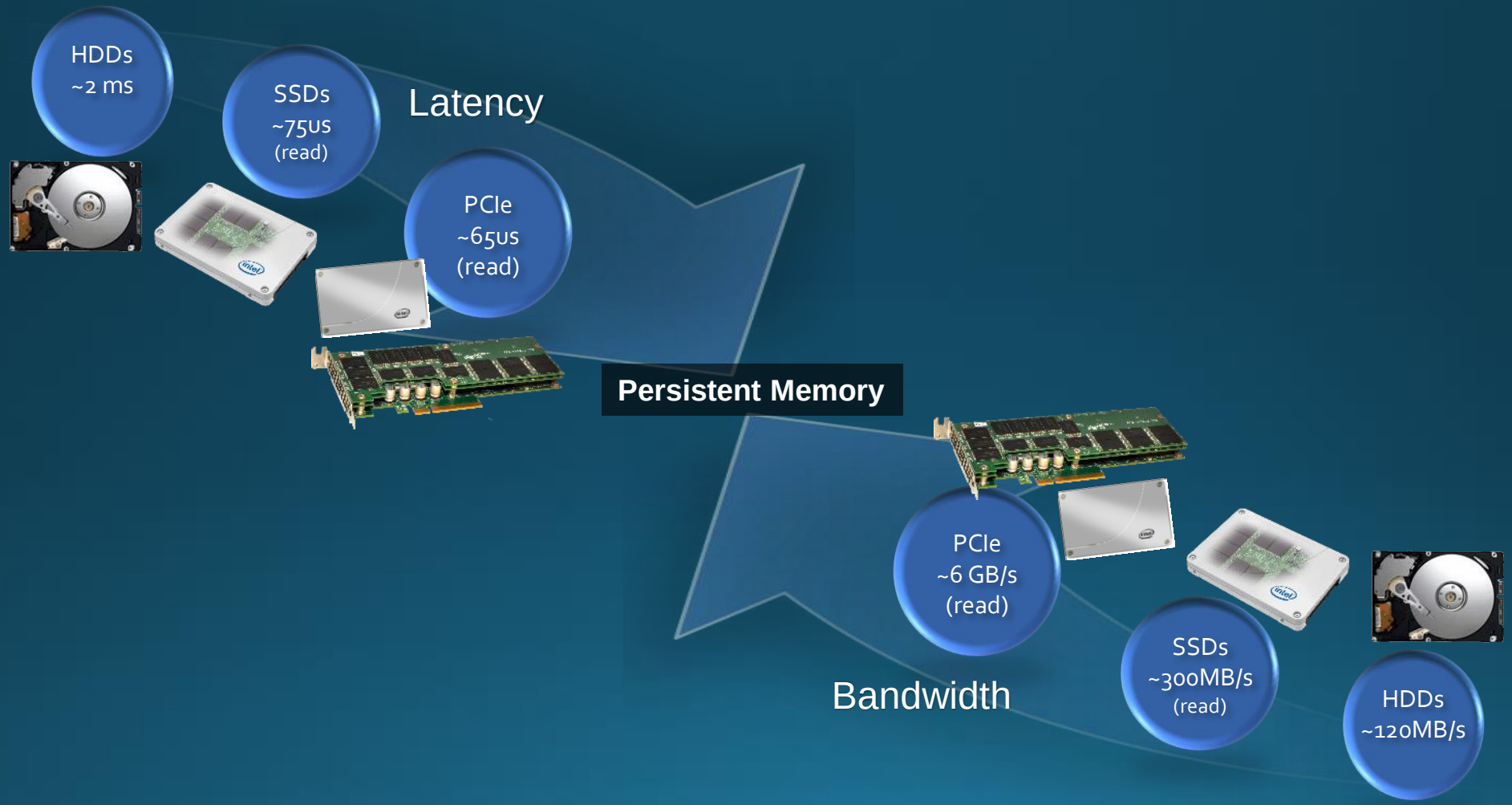
Persistent Memory

Concepts

Memory and Storage



Progression of Storage

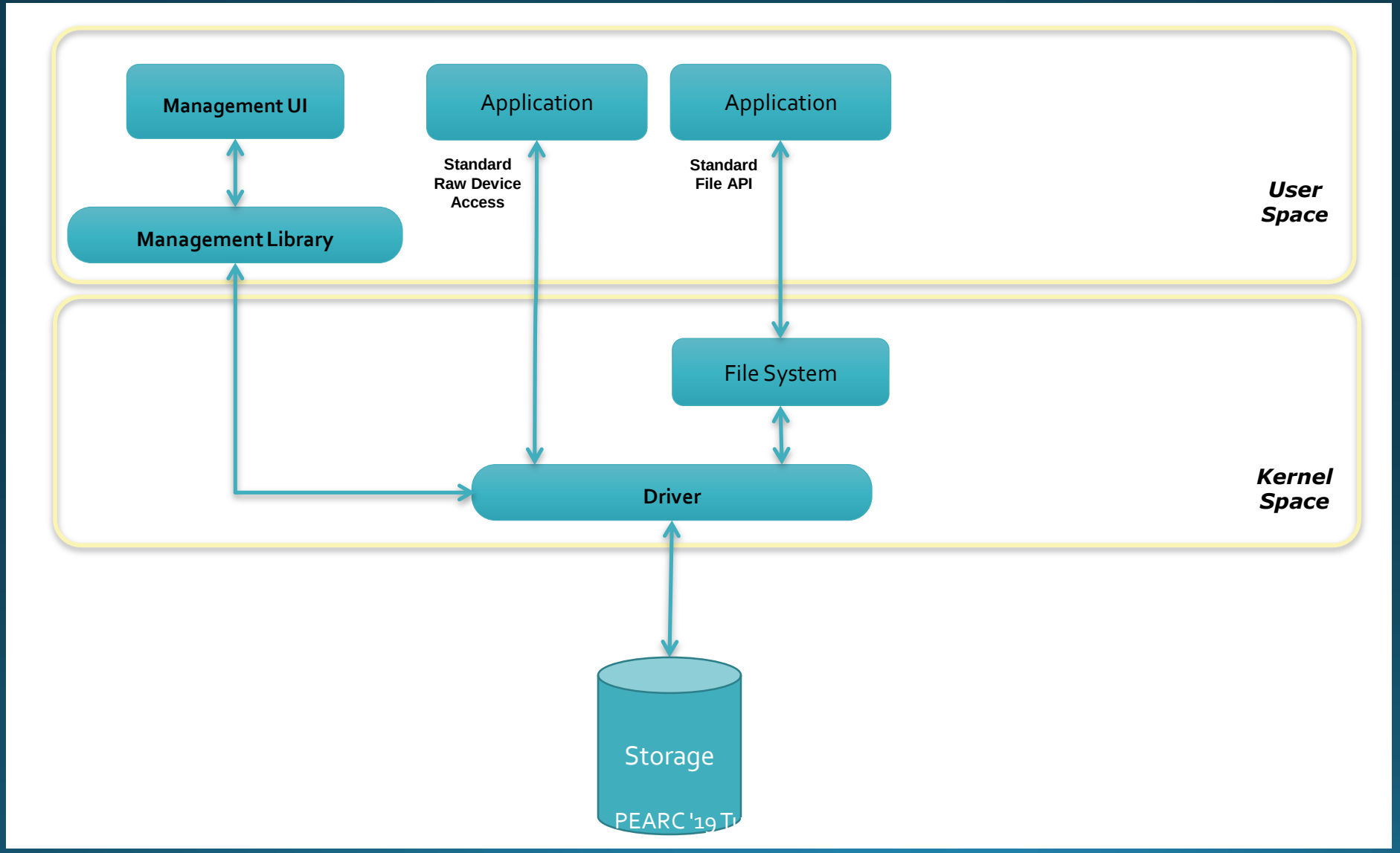


Numbers You Should Know

https://gist.github.com/jboner/2841832

L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2 cache, 200x L1
Compress 1K bytes with Zippy	3,000	ns	3 us		
Send 1K bytes over 1 Gbps network	10,000	ns	10 us		
Read 4K randomly from SSD*	150,000	ns	150 us		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	250 us		
Round trip within same datacenter	500,000	ns	500 us		
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000 us	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000	ns	10,000 us	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000	ns	20,000 us	20 ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150,000 us	150 ms	

The Storage Stack (50,oooft view…)



A Programmer's View

(not just C programmers!)

```
fd = open("/my/file", O_RDWR);
…
count = read(fd, buf, bufsize);
…
count = write(fd, buf, bufsize);
…
close(fd);
```

"Buffer-Based"

A Programmer's View (mapped files)

```
fd = open("/my/file", O_RDWR);
…
base = mmap(NULL, filesize, PROT_READ|PROT_WRITE,
                                    MAP_SHARED, fd, 0);
close(fd);
…
base[100] = 'X';
strcpy(base, "hello there");
*structp = *base_structp;
…
```

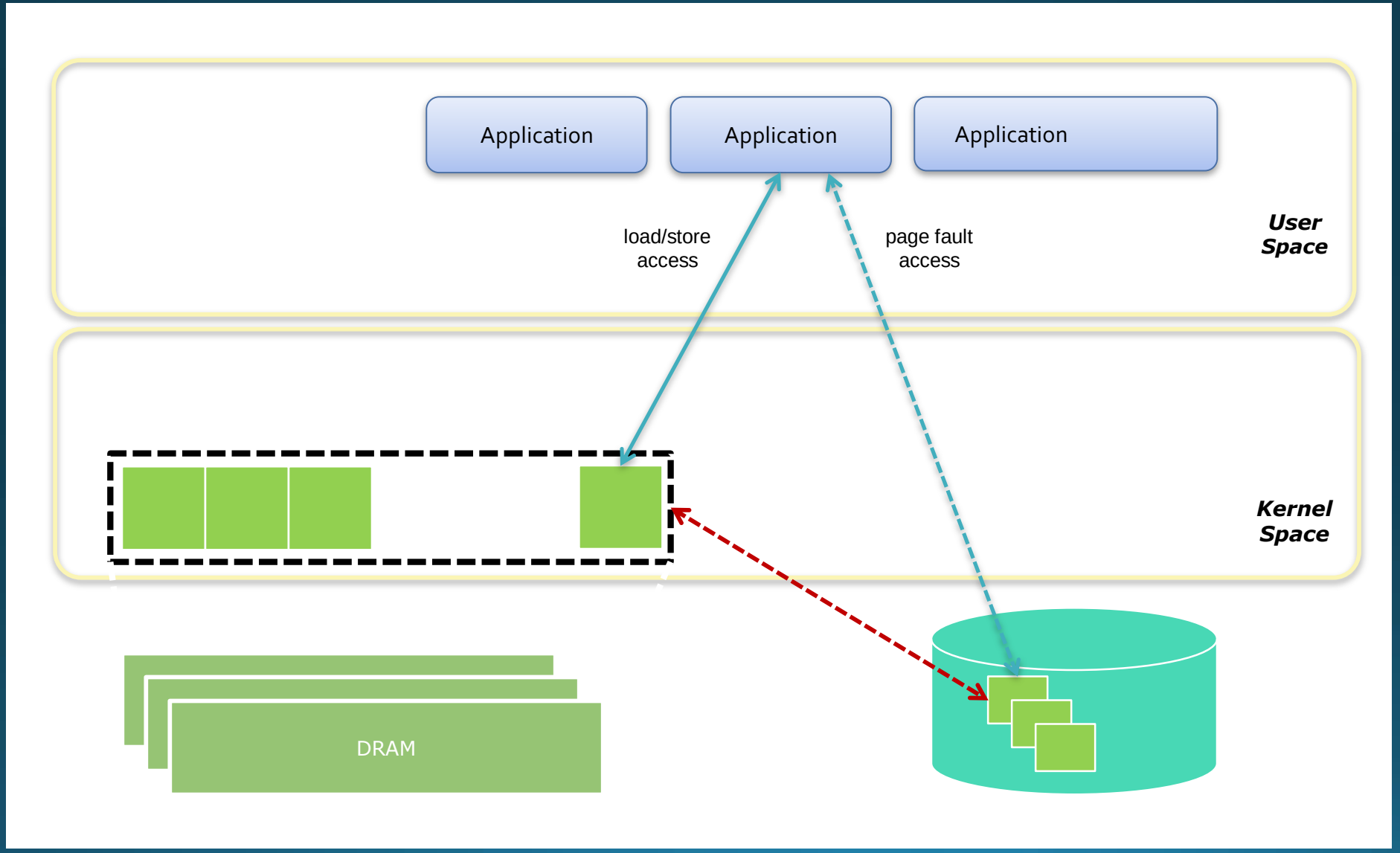
"Load/Store"

Memory-Mapped Files

- What are memory-mapped files really?
  - Direct access to the **page cache**
  - Storage only supports block access (paging)
- With load/store access, when does I/O happen?
  - Read faults/Write faults
  - Flush to persistence
- Not that commonly used or understood
  - Quite powerful
  - Sometimes used without realizing it

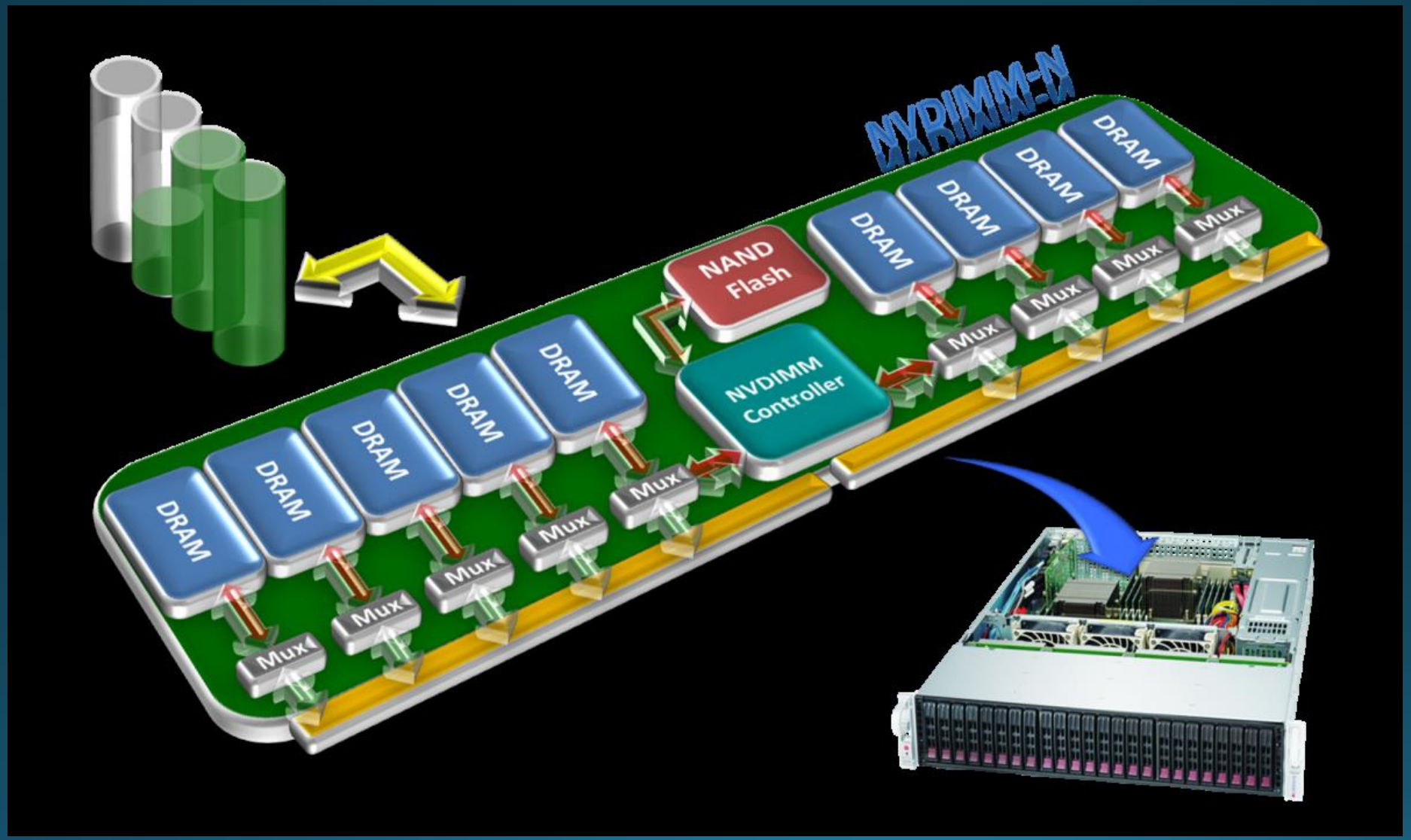
Good reference: http://nommu.org/memory-faq.txt

OS Paging

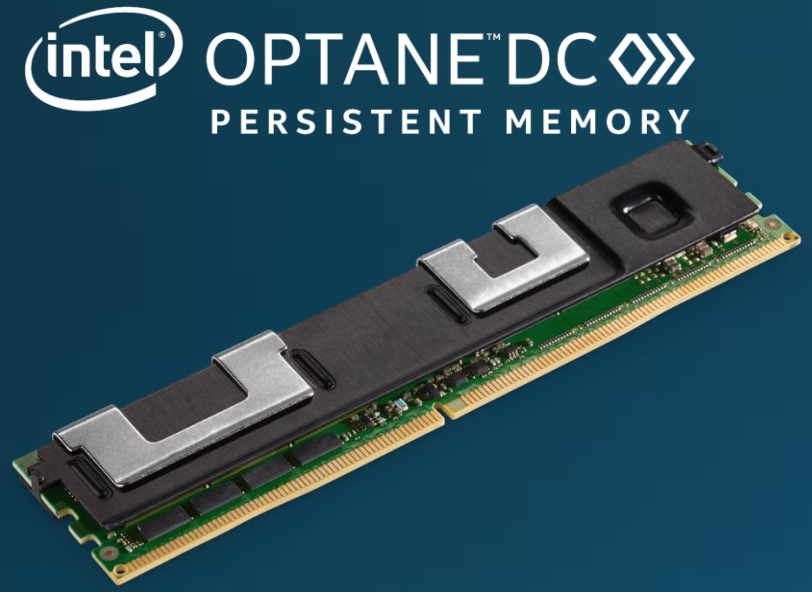

Application
Application
Application
load/store
access
page fault
access
User
Space
Kernel
Space
DRAM

Persistent Memory

NVDIMM-N



Source: SNIA

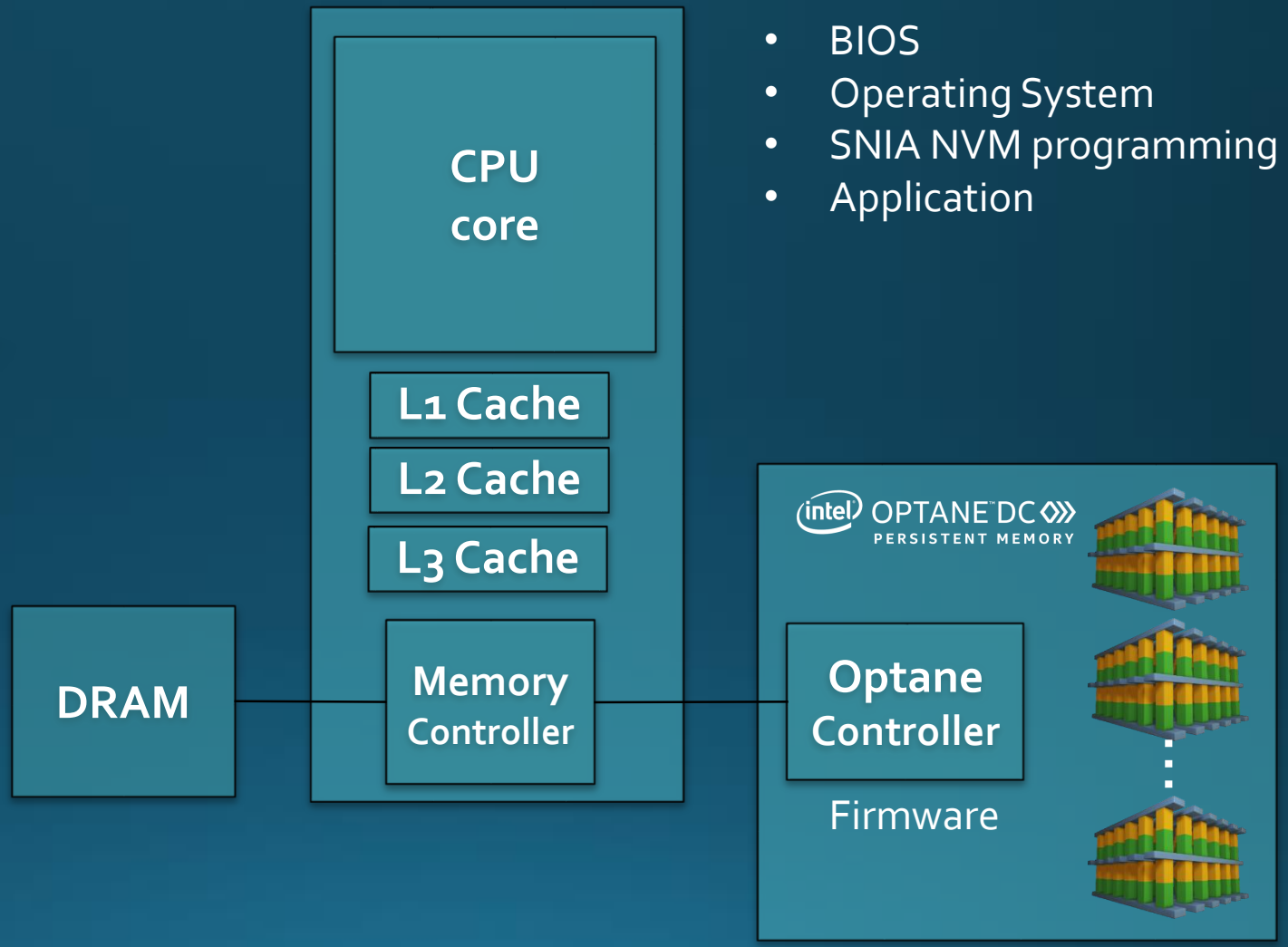


Direct Load/Store Access

Native Persistence

128, 256, 512GB

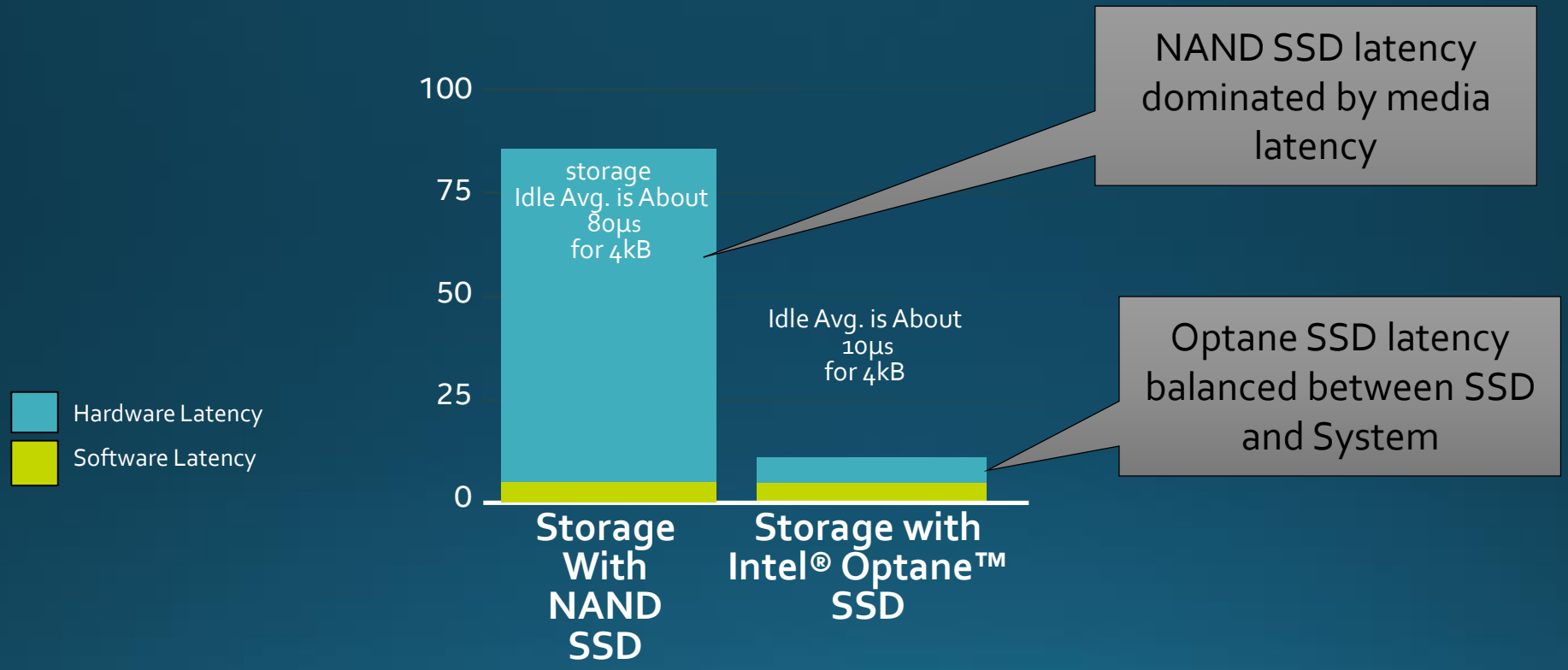
DDR4 Pin Compatible



- BIOS
- Operating System
- SNIA NVM programming Model
- Application

Motivation for the PM Programming Model?

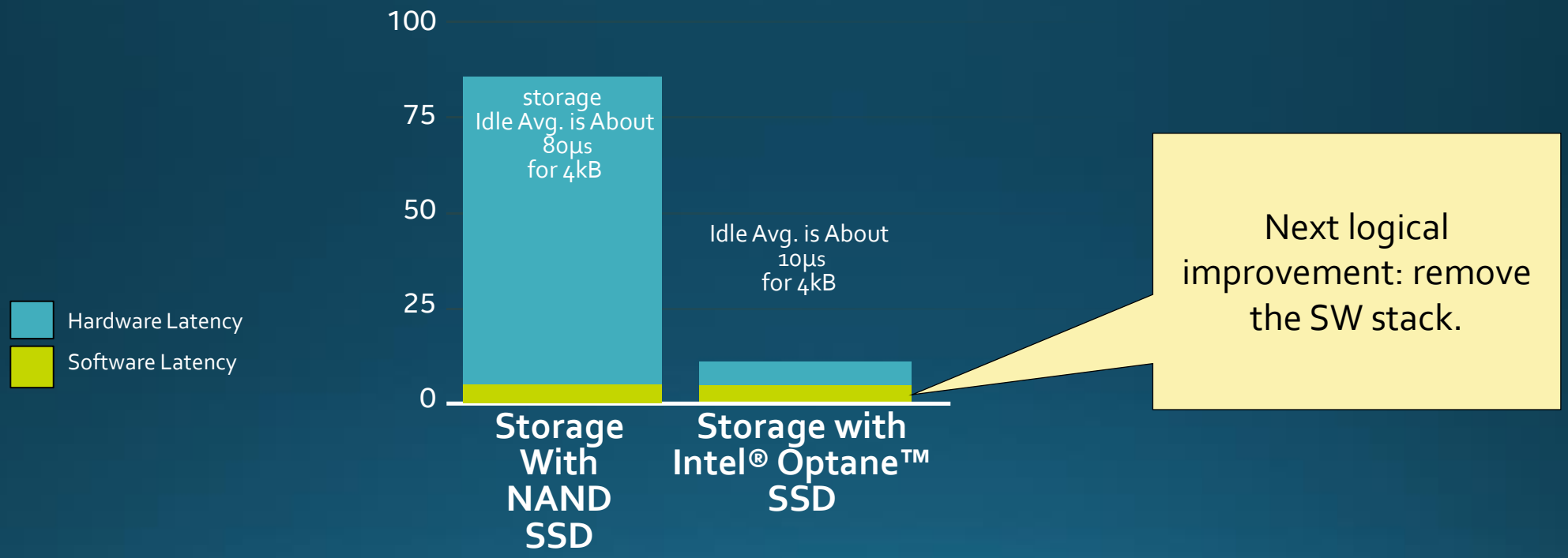
Idle Average Random Read Latency[1]



[1] Source – Intel-tested: Average read latency measured at queue depth 1 during 4k random write workload. Measured using FIO 3.1. Common Configuration - Intel 2U Server System, OS CentOS 7.5, kernel 4.17.6-1.el7.x86_64, CPU 2 x Intel® Xeon® 6154 Gold @ 3.0GHz (18 cores), RAM 256GB DDR4 @ 2666MHz. Configuration – Intel® Optane™ SSD DC P4800X 375GB and Intel® SSD DC P4600 1.6TB. Latency – Average read latency measured at QD1 during 4K Random Write operations using FIO 3.1. Intel Microcode: 0x2000043; System BIOS: 00.01.0013; ME Firmware: 04.00.04.294; BMC Firmware: 1.43.91f76955; FRUSDR: 1.43. SSDs tested were commercially available at time of test. The benchmark results may need to be revised as additional testing is conducted. Performance results are based on testing as of July 24, 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

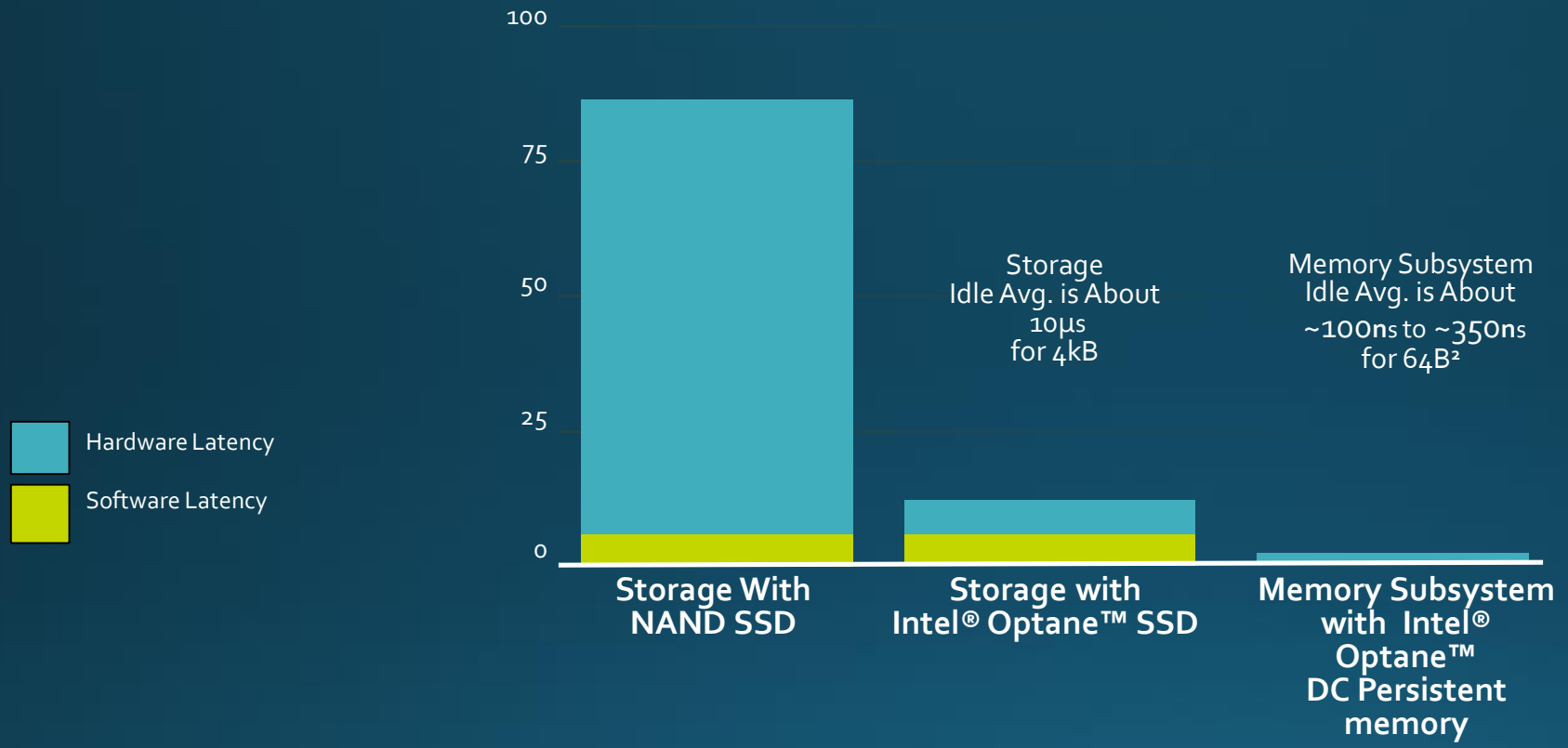
Motivation for the PM Programming Model?

Idle Average Random Read Latency[1]



[1] Source – Intel-tested: Average read latency measured at queue depth 1 during 4k random write workload. Measured using FIO 3.1. Common Configuration - Intel 2U Server System, OS CentOS 7.5, kernel 4.17.6-1.el7.x86_64, CPU 2 x Intel® Xeon® 6154 Gold @ 3.0GHz (18 cores), RAM 256GB DDR4 @ 2666MHz. Configuration – Intel® Optane™ SSD DC P4800X 375GB and Intel® SSD DC P4600 1.6TB. Latency – Average read latency measured at QD1 during 4K Random Write operations using FIO 3.1. Intel Microcode: 0x2000043; System BIOS: 00.01.0013; ME Firmware: 04.00.04.294; BMC Firmware: 1.43.91f76955; FRUSDR: 1.43. SSDs tested were commercially available at time of test. The benchmark results may need to be revised as additional testing is conducted. Performance results are based on testing as of July 24, 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Idle Average Random Read Latency[1]



[1] Source: Intel-tested: Average read latency measured at queue depth 1 during 4k random write workload. Measured using FIO 3.1. comparing Intel Reference platform with Optane™ SSD DC P4800X 375GB and Intel® SSD DC P4600 1.6TB compared to SSDs commercially available as of July 1, 2018. Performance results are based on testing as of July 24, 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

[2] App Direct Mode , NeonCity, LBG B1 chipset , CLX B0 28 Core (QDF QQYZ), Memory Conf 192GB DDR4 (per socket) DDR 2666 MT/s, Optane DCPMM 128GB, BIOS 561.D09, BKC version WW48.5 BKC, Linux OS 4.18.8-100.fc27, Spectre/Meltdown Patched (1,2,3, 3a)

Persistent Memory Definition

- Byte-addressable
  - As far as the programmer is concerned
- Load/Store access
  - Not demand-paged
- Memory-like performance
  - **Would reasonably stall a CPU load waiting for pmem**
- Probably DMA-able
  - Including RDMA

- For modeling, think: Battery-backed DRAM

pmem in This Presentation…

- Is **not** tablet-like memory for entire system
  - "Transparent Persistent Memory" is another topic
- Is **not** NAND Flash
  - At least not directly
    - perhaps with aggressive caching
- Is **not** block-oriented

Persistent Memory Attributes

- pmem does not
  - Surprise the program with unexpected latencies
    - No major page faults
  - Kick other things out of memory
  - Use the page cache unexpectedly

- pmem stores aren't durable until data is flushed
  - Is this a new, inconvenient attribute of PM?
  - Or is this something that's been around for decades?

- pmem may not always stay at the same address
  - Physically
  - Virtually

The Value of Persistent Memory

- Data sets addressable with no DRAM footprint
  - At least, up to application if data copied to DRAM
- Typically DMA (and RDMA) to PM works as expected
  - **RDMA directly to persistence – no buffer copy required!**
- The "Warm Cache" effect
  - No time spend loading up memory
- Byte addressable
- Direct user-mode access
  - No kernel code in data path

How Do I Use Persistent Memory?

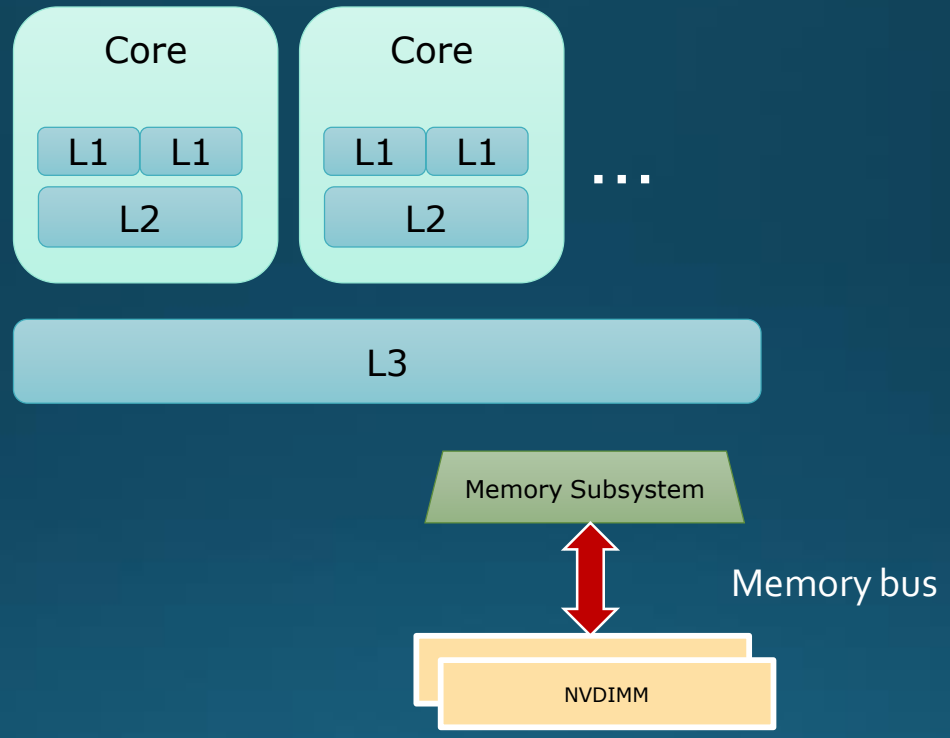
- For most people:
  - The system uses it **transparently**, or
  - Some middleware uses it **transparently**, or
  - The application uses it **transparently**
  - (But you had to purchase a system with pmem installed)
- But for the developer/technologist audience:
  - You design a solution that leverages pmem
  - The rest of this presentation deck is about that…

Programming Models

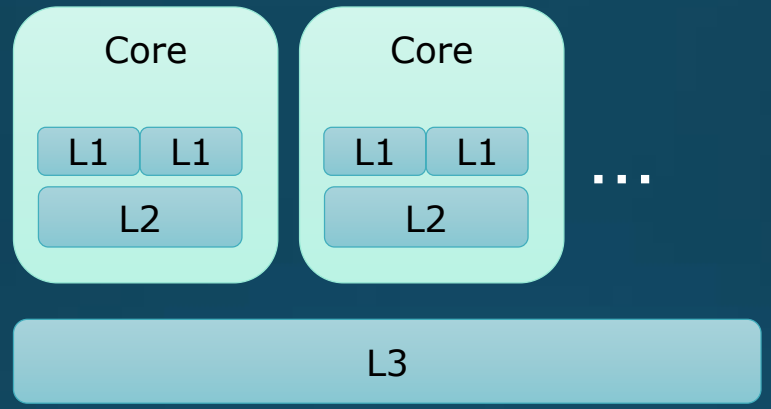
Programming Model: At least four meanings…

1. Interface between HW and SW
2. Instruction Set Architecture (ISA)
3. Exposed to Applications (by the OS)
4. The Programmer Experience

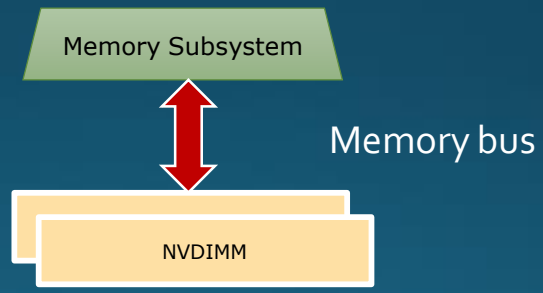
Programming Model (meaning 1): HW to SW Interface



Programming Model (meaning 1): HW to SW Interface



Result:
Persistent Memory hardware accessed like memory (cache coherent).
Described by ACPI on x86.



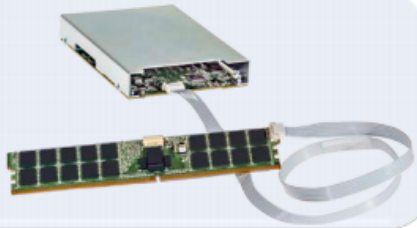
NFIT (NVDIMM Firmware Interface Table)

- First introduced in ACPI 6.0
  - Describes the NVDIMMs in a system
  - Separates NVDIMM memory from system main memory
  - Describes persistency model
  - Describes interleaving
    - Lots of interesting considerations here
    - Who should see this (which layers of SW?)
  - Continuing to evolve to cover more use cases
- NFIT is created by the BIOS
  - BIOS must understand each supported NVDIMM type

JEDEC NVDIMM Types

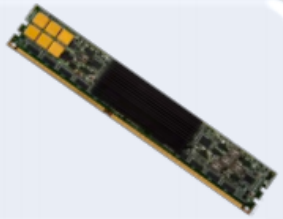
NVDIMM-N
Standardized

- Memory mapped DRAM. Flash is not system mapped
- Access Methods -> byte- or block-oriented access to DRAM
- Capacity = DRAM DIMM (1's -10's GB)
- Latency = DRAM (10's of nanoseconds)
- Energy source for backup
- DIMM interface (HW & SW) defined by JEDEC



NVDIMM-F
Vendor Specific

- Memory mapped Flash. DRAM is not system mapped.
- Access Method -> block-oriented access to NAND through a shared command buffer (i.e. a mounted drive)
- Capacity = NAND (100's GB-1's TB)
- Latency = NAND (10's of microseconds)

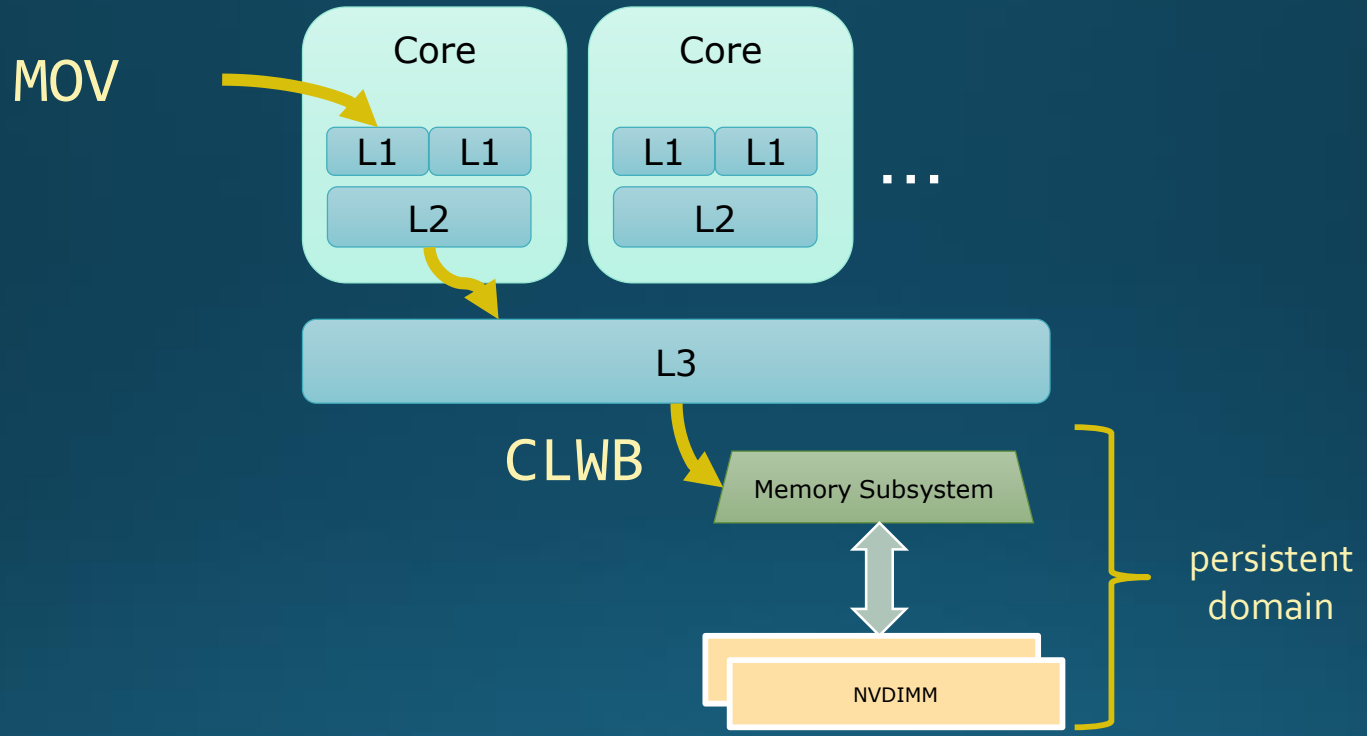


NVDIMM-P
Proposals in progress

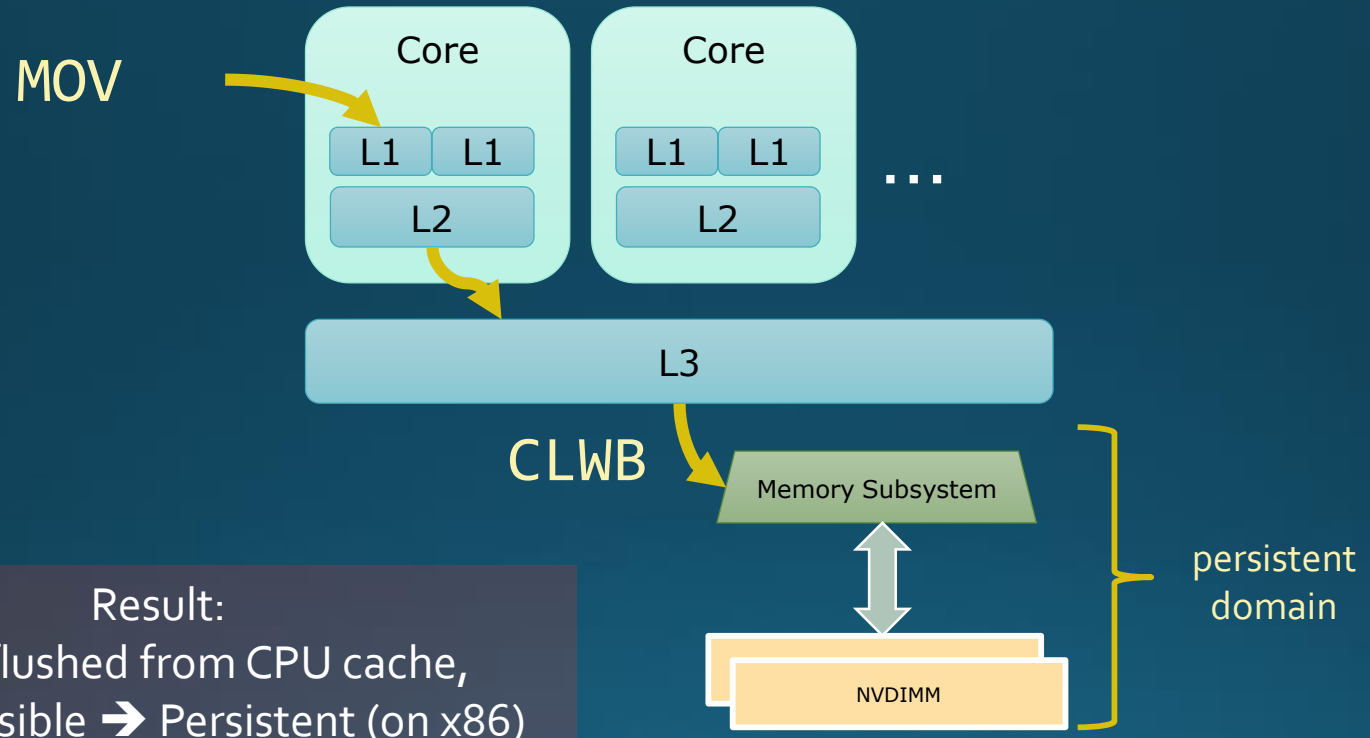
- Memory-mapped Flash and memory-mapped DRAM
- Two access mechanisms: persistent DRAM (–N) and block-oriented drive access (–F)
- Capacity = NVM (100's GB-1's TB)
- Latency = NVM (100's of nanoseconds)

Source: http://www.snia.org/sites/default/files/SSSI/NVDIMM%20-%20Changes%20are%20Here%20So%20What's%20Next%20-%20final.pdf
Author: Arthur Sainio

Programming Model (meaning 2): Instruction Set Architecture (ISA)

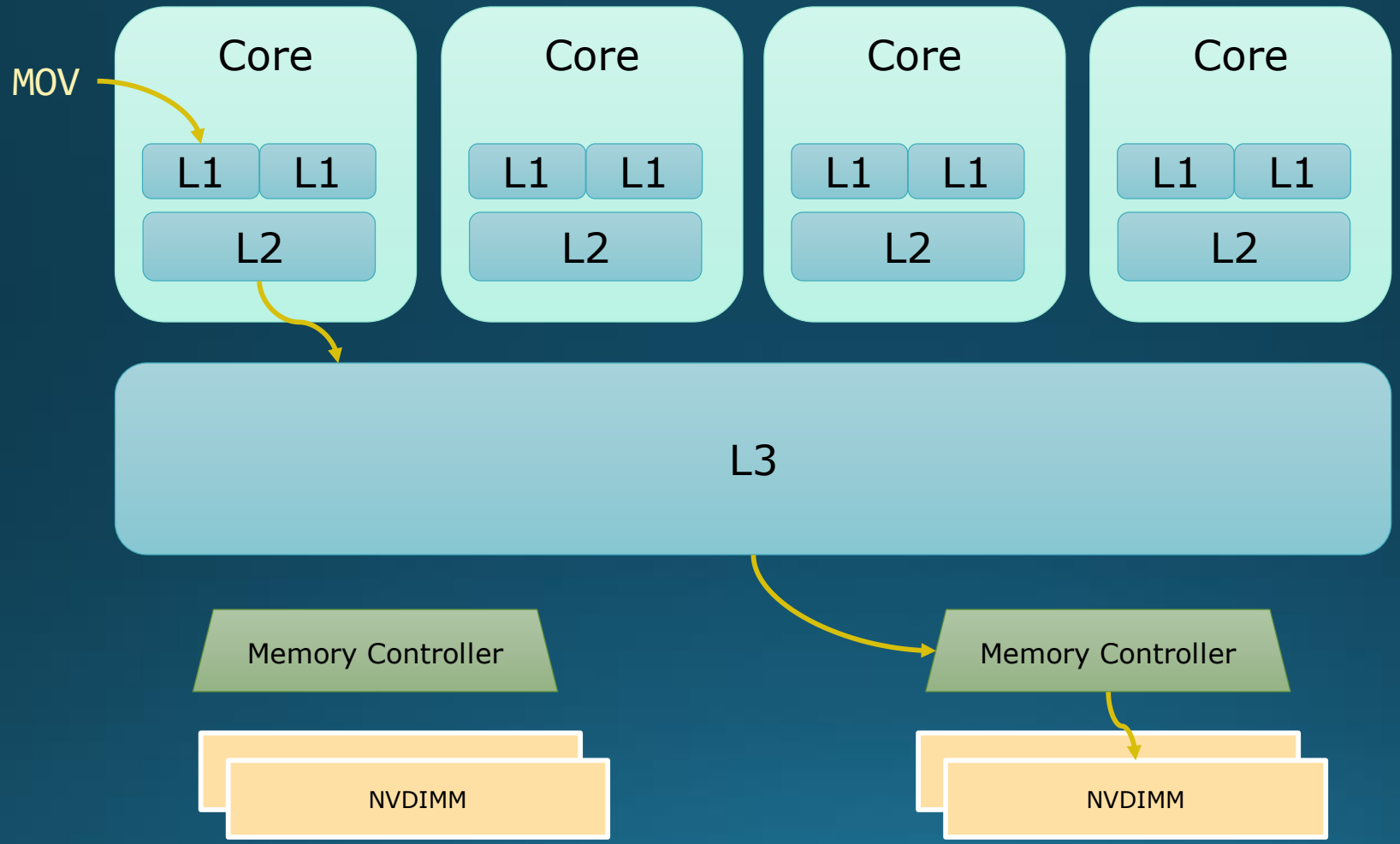


Programming Model (meaning 2): Instruction Set Architecture (ISA)

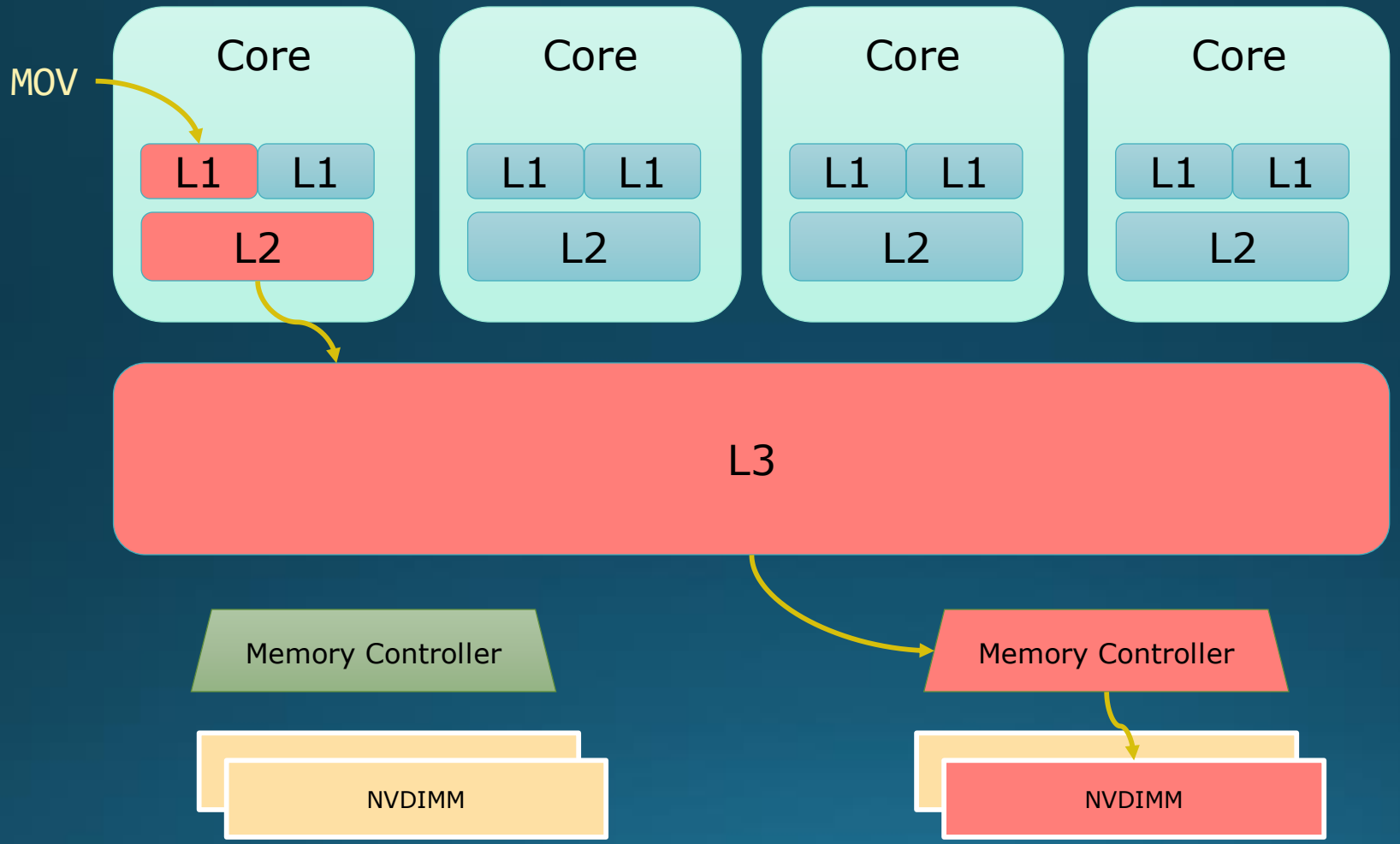


Result:
Stores flushed from CPU cache, globally-visible ➔ Persistent (on x86)

The Data Path



Where's Your Data? (immediately after MOV)

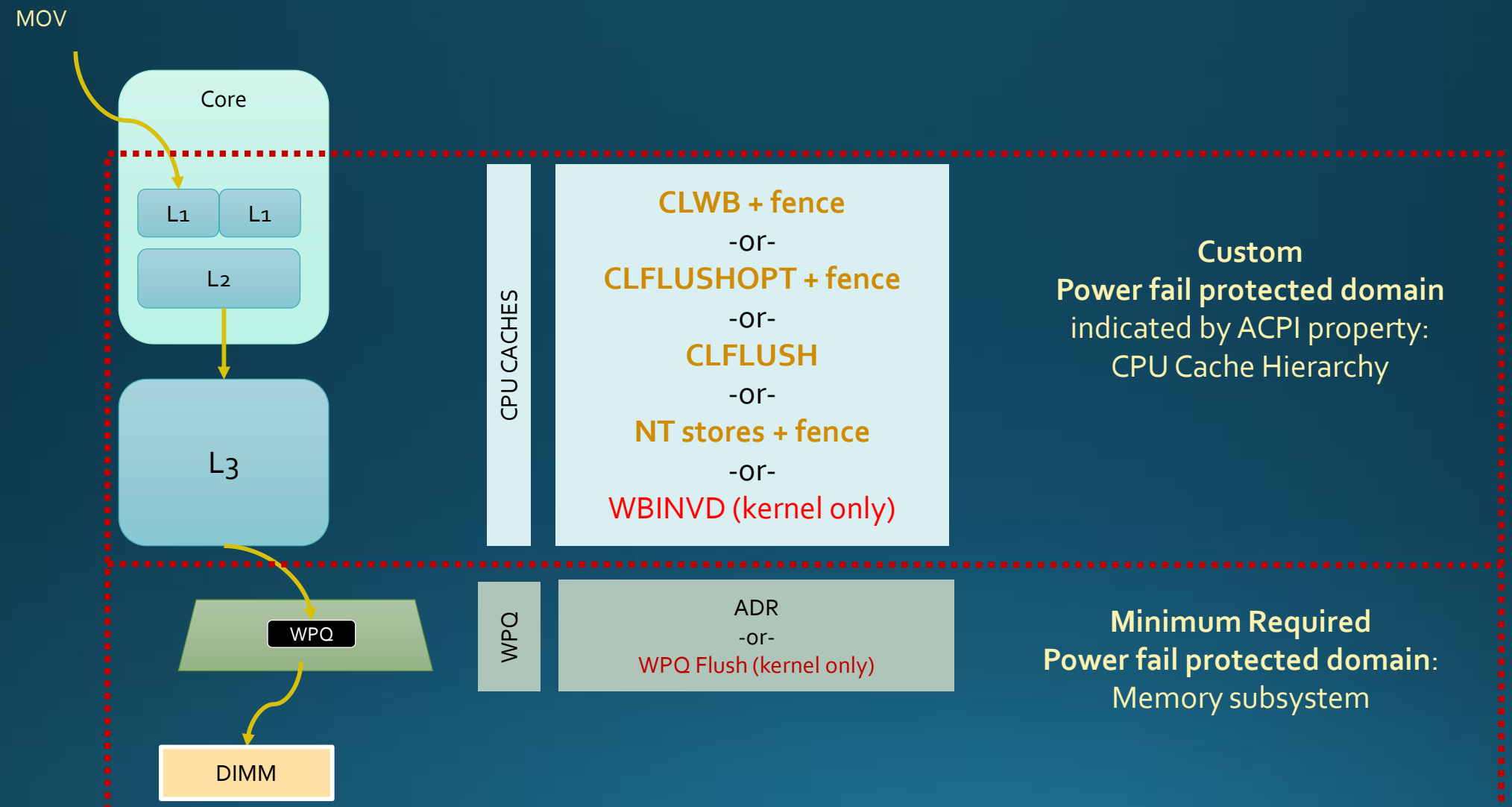


Flushing Stores From Caches

Instruction	Meaning
`CLFLUSH addr`	Cache Line Flush: Available for a long time
`CLFLUSHOPT addr`	Optimized Cache Line Flush: New to allow concurrency
`CLWB addr`	Cache Line Write Back: CPU allowed (not required) to leave value in cache for performance of next access

- Software must comprehend what flushing from user space really means
  - Only if OS says it is safe

The Persistent Domain (x86 example)



ADR

- Stands for “Asynchronous DRAM Refresh”
- Really means a platform-level feature that:
  - Kicks in on power loss/shutdown/crash
  - Flushes pmem stores that were accepted by the memory sub-sustem
  - DOES NOT flush anything else (CPU caches are not handled by ADR)
- For Intel platforms, ADR is **required** for persistent memory

Deep Flush

10.2.9 NVM.PM.FILE.DEEP_FLUSH

Requirement: mandatory if NVM.PM.DEEP_FLUSH_CAPABLE is set.

The purpose of this action is to provide the same persistency semantics as NVM.PM.FILE.SYNC, but performance may be sacrificed in order to flush persistent memory stores to the **most reliable persistence domain available to software** .

[…]

The result is a **higher expected reliability at the cost of flush performance**.

Visibility versus Power Fail Atomicity

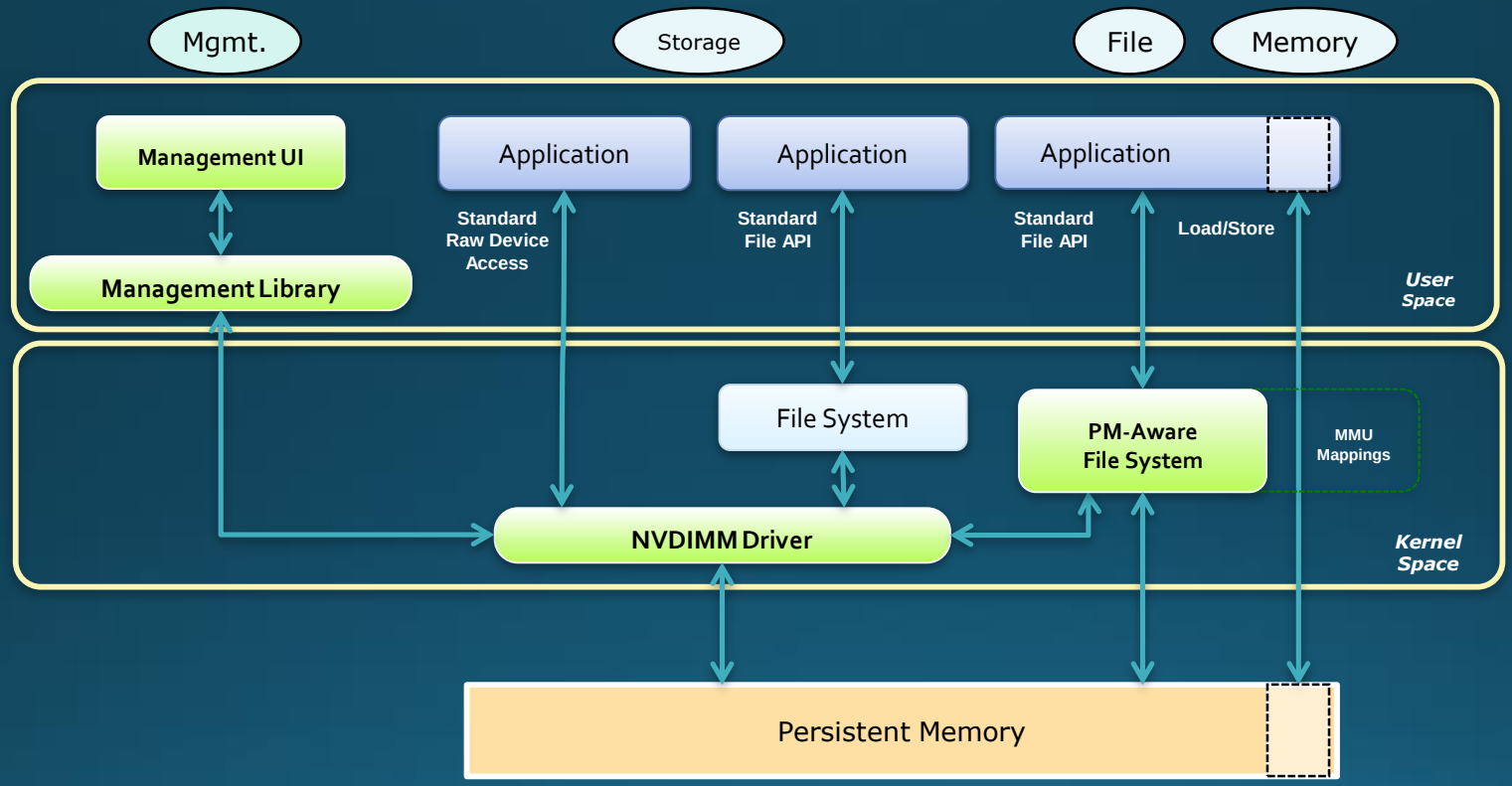
Feature	Atomicity
Atomic Store	8 byte powerfail atomicity Much larger visibility atomicity
TSX	Programmer must comprehend XABORT, cache flush can abort
LOCK CMPXCHG	*non-blocking* algorithms depend on CAS, but CAS doesn't include flush to persistence

- Software must implement all atomicity beyond 8-bytes for pmem
  - Transactions are fully up to software

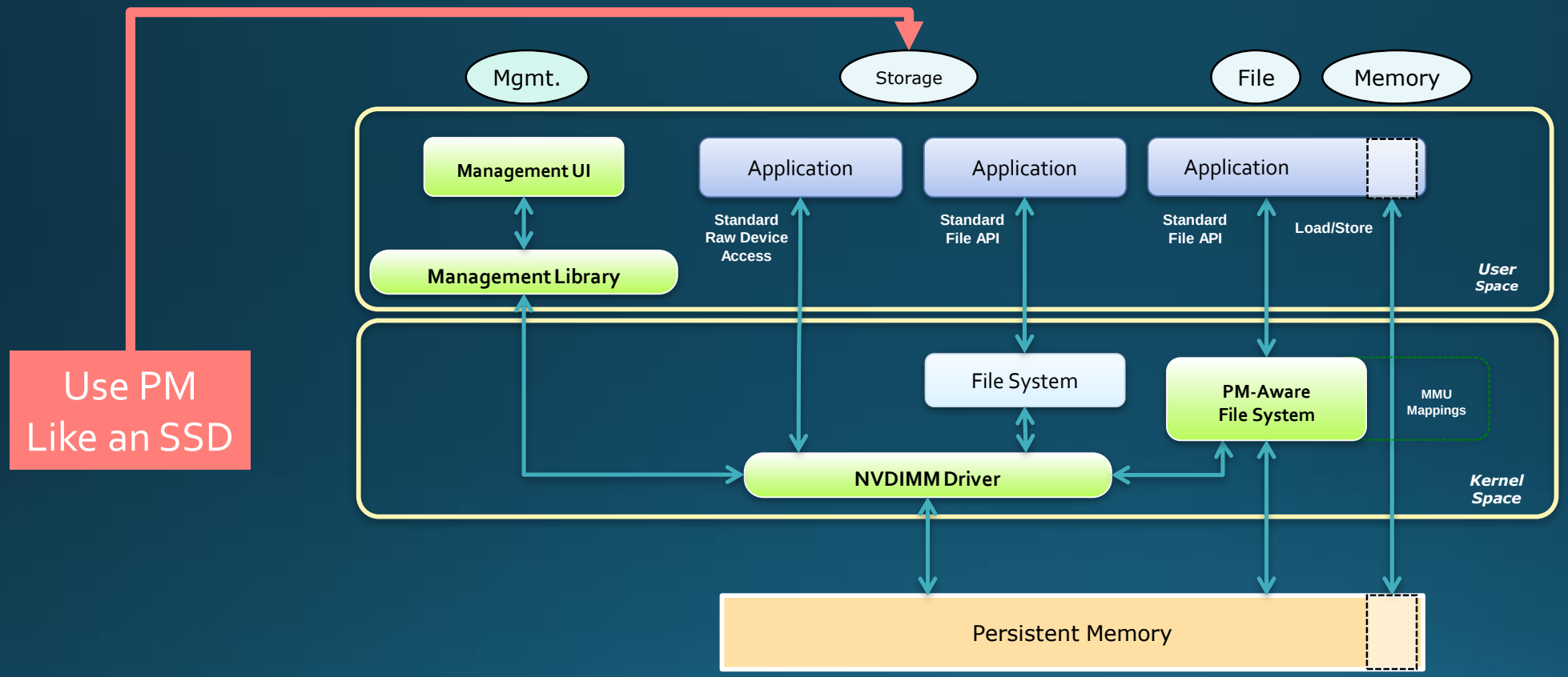
Programming Model (meaning 3): Exposing to Applications

- Requirements:
  - A way to name pmem, so can "re-connect" with data later
  - A permission model
  - A way to manage pmem regions
    - Create/delete
    - Back up
    - Preferably without new versions of all the storage commands
  - A common model, not vendor-lock-in
    - Protect the ISVs!

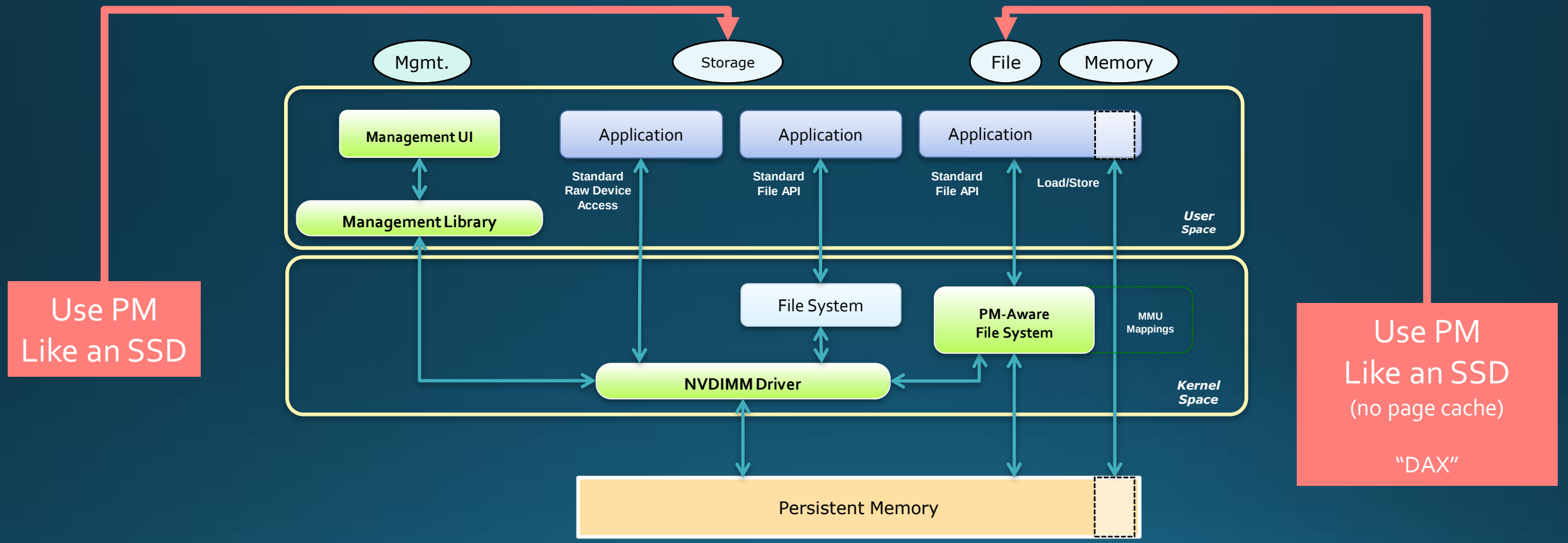
The SNIA NVM Programming Model



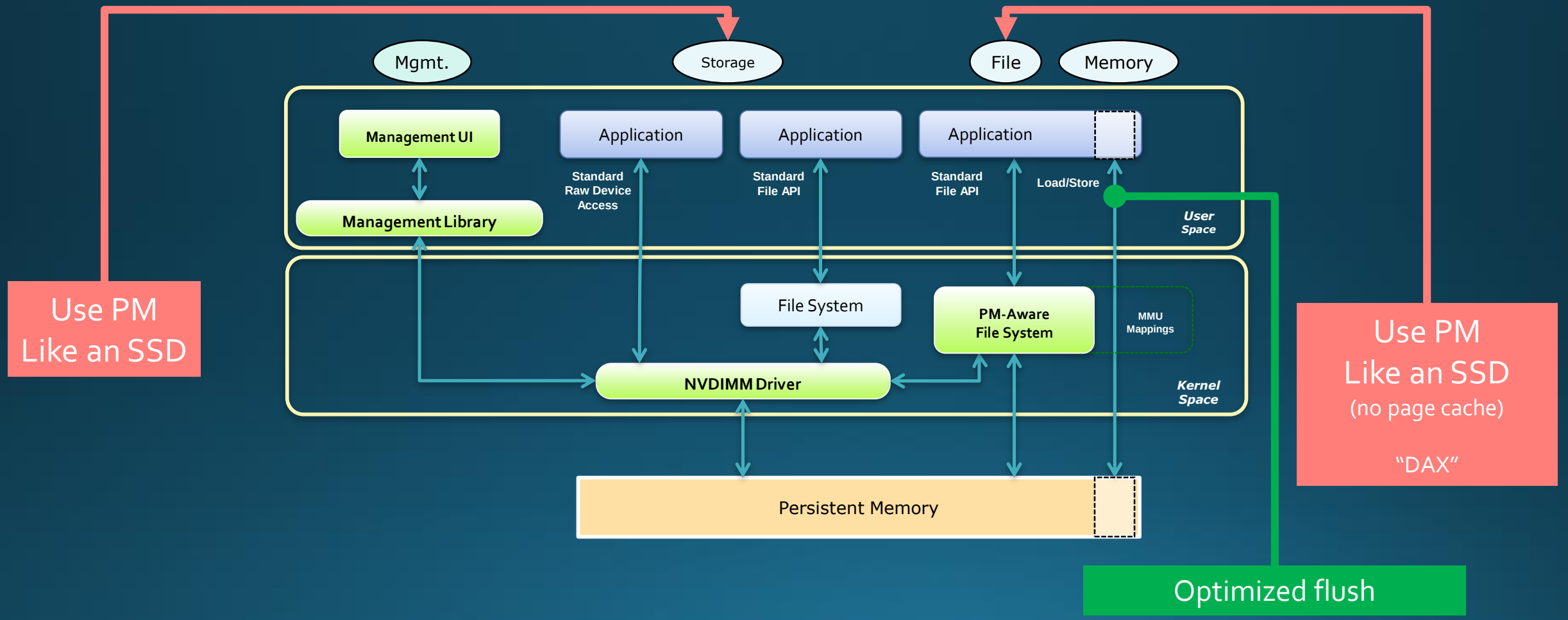
The Programming Model Builds on the Storage APIs



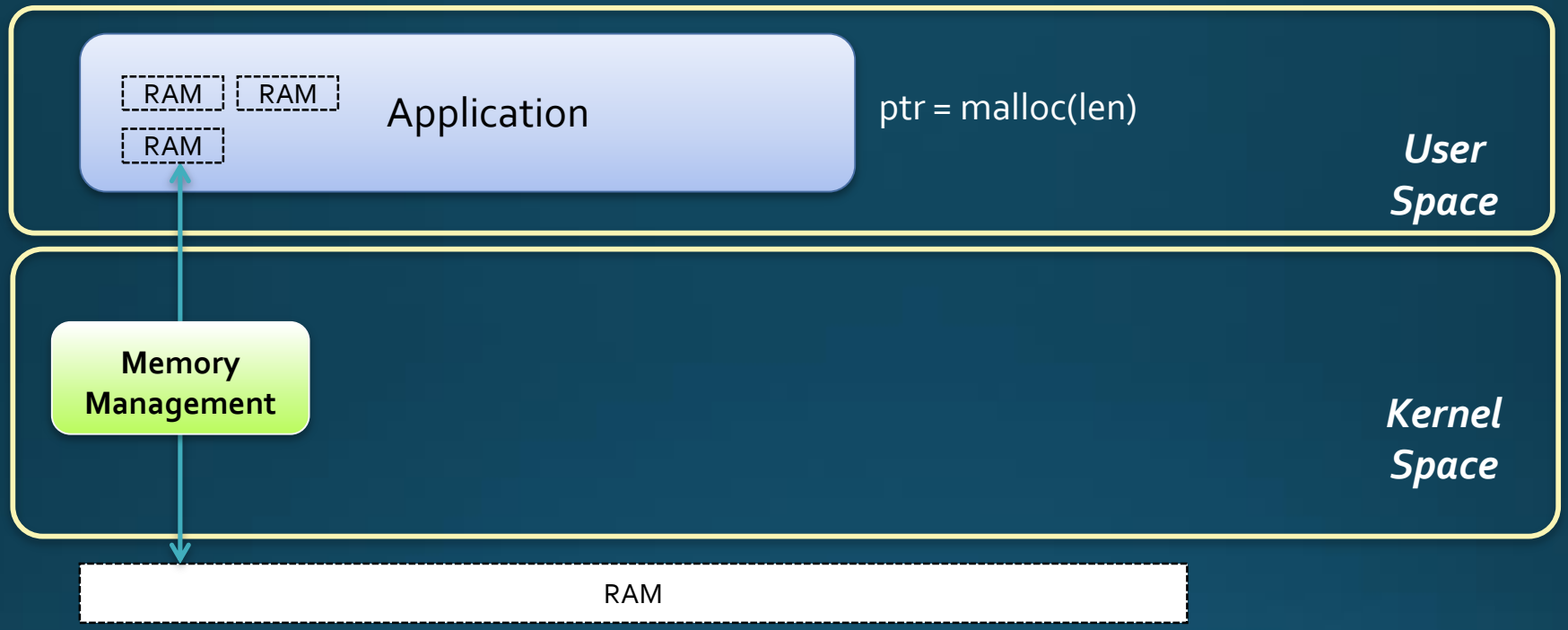
The Programming Model Builds on the Storage APIs



Optimized Flush is the Primary New API

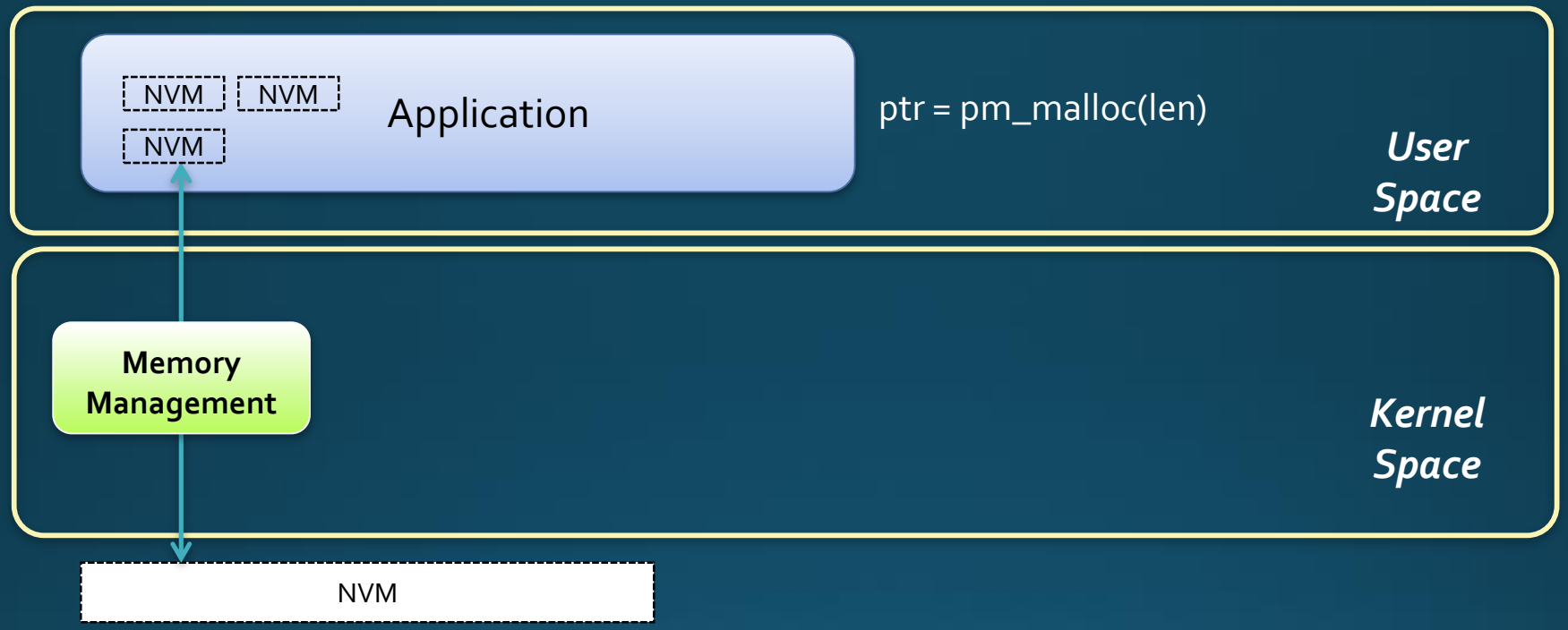


Application Memory Allocation



- Well-worn interface, around for decades
- Memory is gone when application exits
  - Or machine goes down

Application NVM Allocation



- Simple, familiar interface, *but then what?*
  - Persistent, so apps want to "attach" to regions
  - Need to manage permissions for regions
  - Need to resize, remove, …, **backup** the data

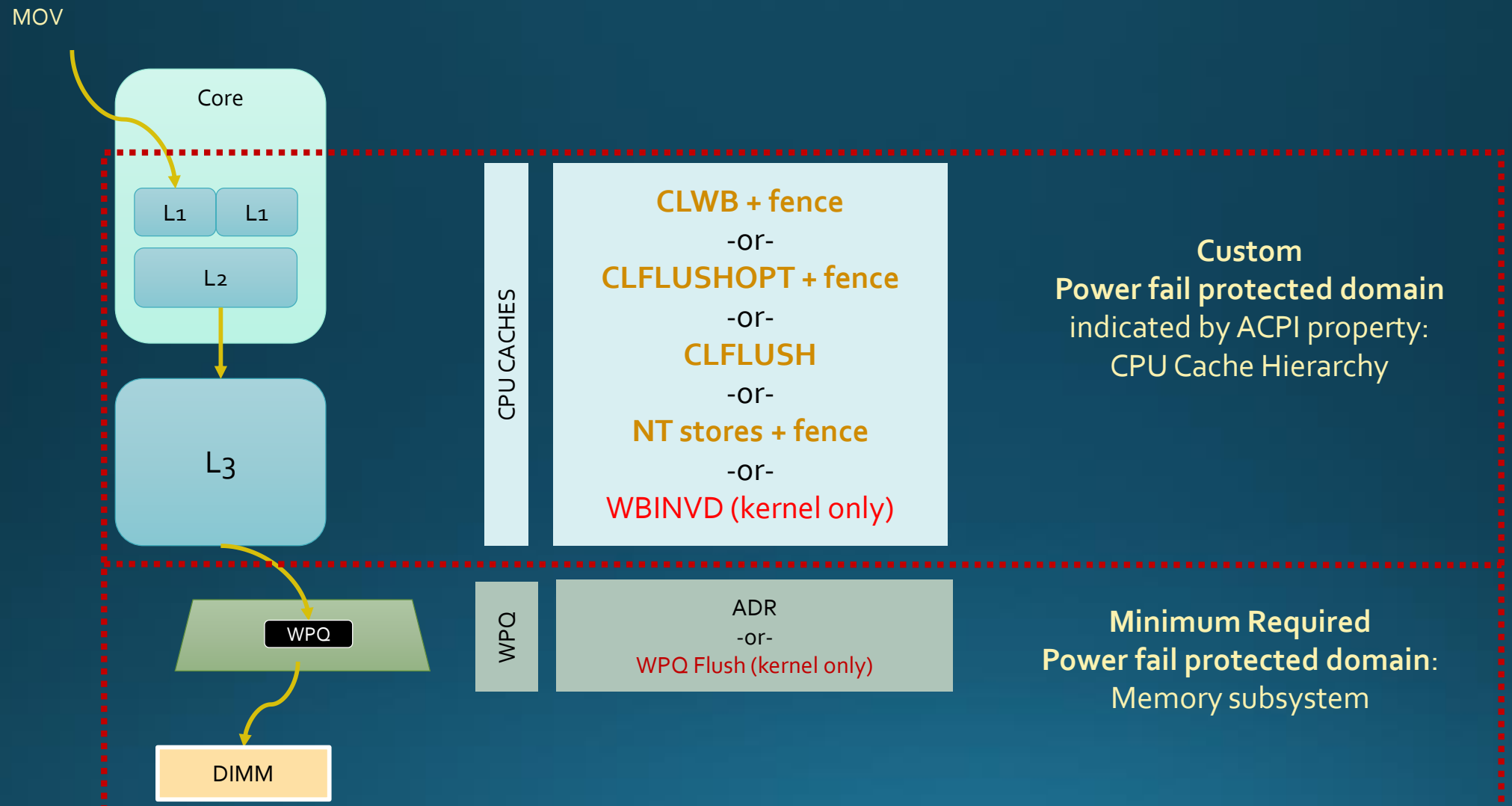
Visibility versus Persistent

- It has always been thus:
  - open()
  - mmap()
  - store…

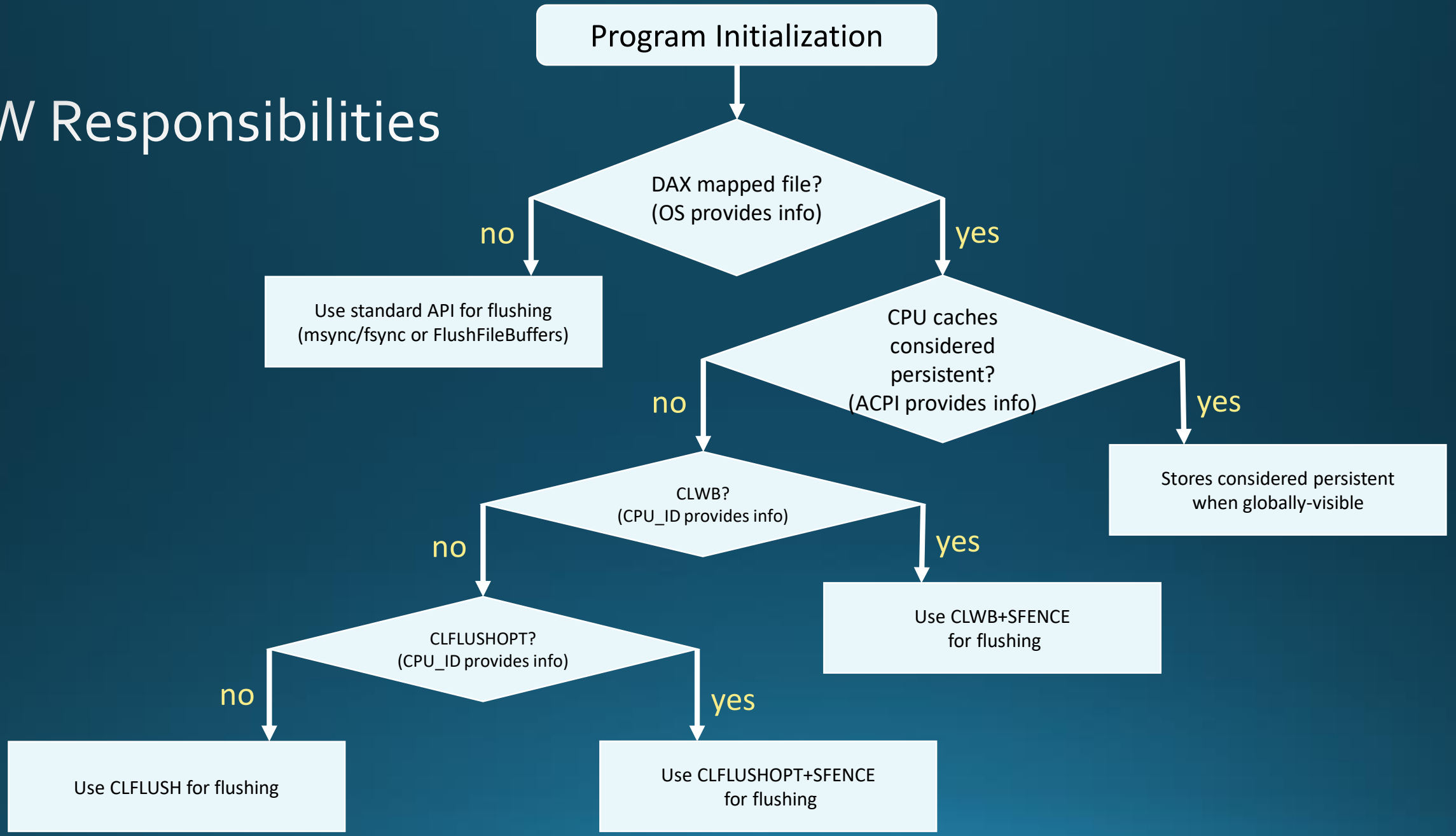
  - msync()

- pmem just follows this decades-old model
  - But the stores are cached in a different spot

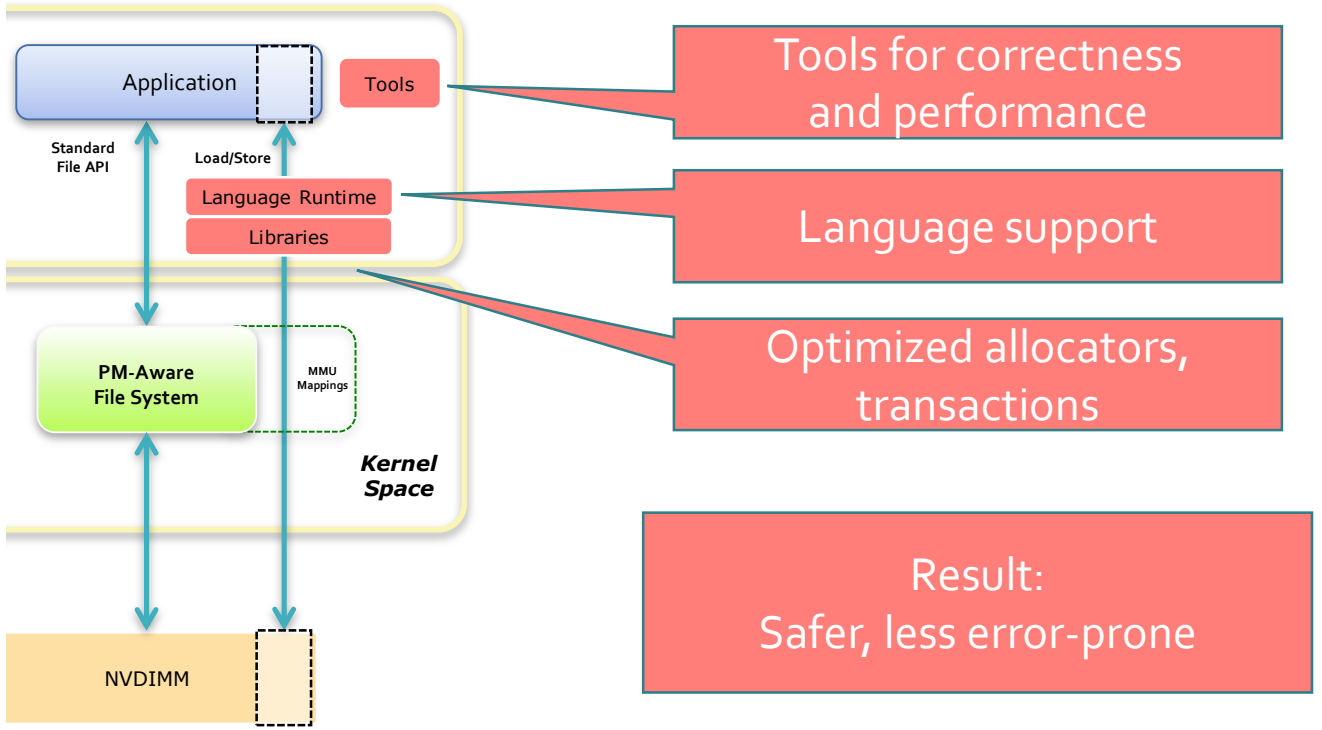
Remember this from earlier?



SW Responsibilities



Programming Model (meaning 4): The Programmer Experience



Java PersistentSortedMap

```
PersistentSortedMap employees = new PersistentSortedMap();
...
employees.put(id, data);
```

No flush calls.
Transactional.
Java library handles it all.

See “pilot” project at: https://github.com/pmem/pcj

What Lies Between These?

- Memory-Mapped Files

Standard Flush

```
msync()

FlushViewOfFile()
FlushFileBuffers()
```

- High-Level Language Support

```
PersistentSortedMap

employees.put(id, data);
```

Answer coming later!

Persistent Memory on Linux

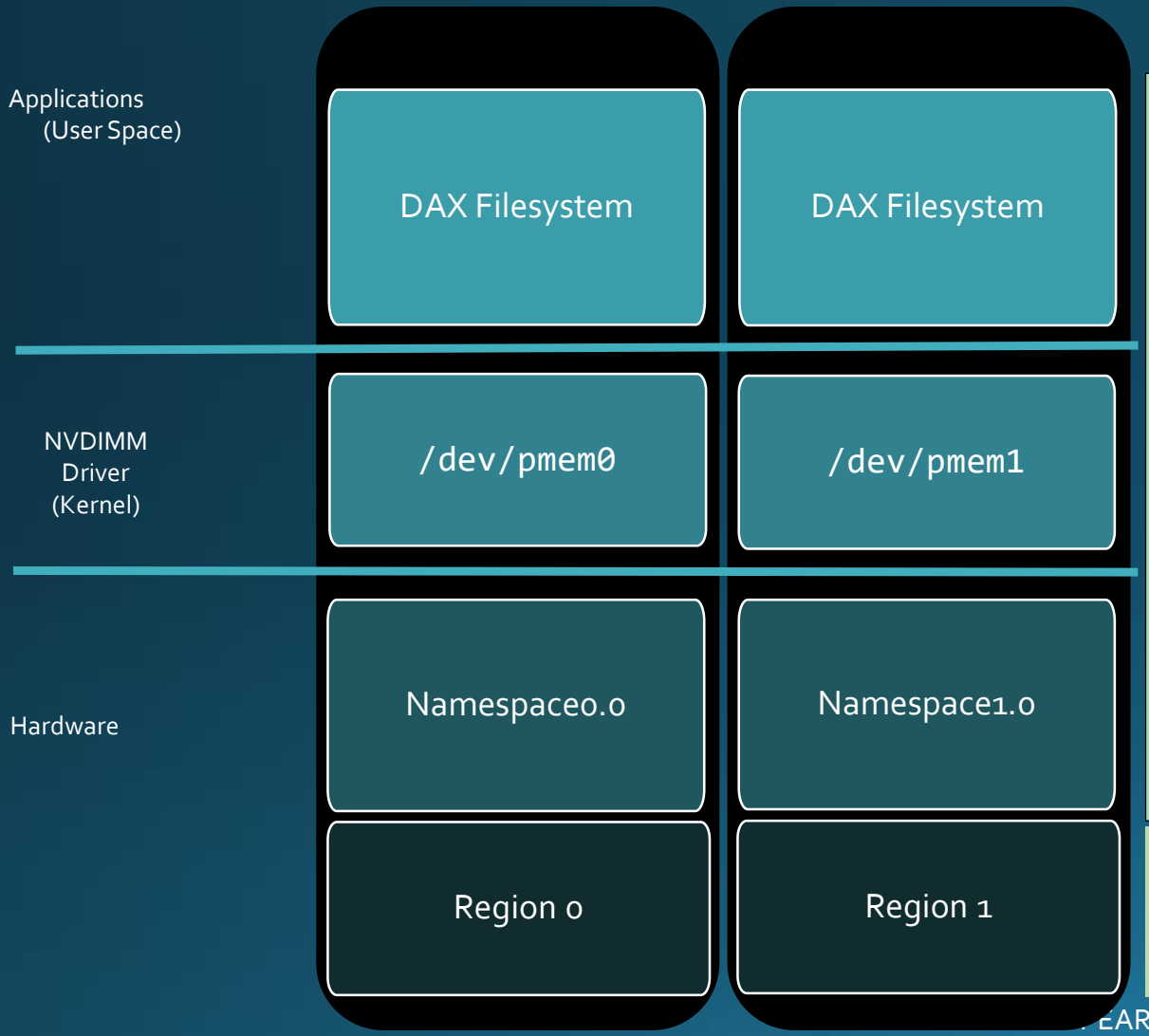
SW Enabling Summary

- Linux
  - Direct Access (DAX) changes upstream
    - Includes "pmem" stack
  - ext4 & XFS are pmem-aware file systems (others may follow)
  - Getting started guide: https://docs.pmem.io/
  - Kernel info: https://nvdimm.wiki.kernel.org/
- Virtualization
  - Using pmem inside a guest VM
  - KVM changes upstream, other VMMs in progress

Linux System Administration

- Making sure you have the kernel support you need:
- Managing pmem regions: `ndctl`
  - http://pmem.io/ndctl/
  - Generic NVMDIMM command
  - Vendor may also have vendor-specific management commands
    - Example: `ipmctl` https://github.com/intel/ipmctl
- Getting direct access (DAX)
  - `mount -o dax [...]`
- Managing blobs of persistent memory
  - Like files!
    - /bin commands like `cp`, `mv`, `chown`, `chmod`, `tar`, etc.

Typical Linux Config Steps




```
# mkfs.ext4 /dev/pmem0
# mkfs.ext4 /dev/pmem1
# mount -o dax /dev/pmem0 /pmemfs0
# mount -o dax /dev/pmem1 /pmemfs1
```

new

Defaults

```
# ndctl create-namespace --region region0 --mode fsdax
# ndctl create-namespace --region region1 --mode fsdax
```

Vendor Neutral

```
# ipmctl create -goal PersistentMemoryType=AppDirect
# reboot
```

Vendor Specific

Programming with Optimized Flush

- Use Standard unless OS says it is safe to use Optimized Flush
- On Windows
  - When you successfully memory map a DAX file:
    - Optimized Flush is safe
- On Linux
  - When you successfully memory map a DAX file with MAP_SYNC:
    - Optimized Flush is safe
  - MAP_SYNC flag to mmap() is new

Emulating Persistent Memory

- The programming model builds on memory-mapped files
  - So development on memory-mapped files work fine
    - PMDK will use msync() to flush to persistence
    - Non-optimal performance
  - Use any 64-bit Linux
- For benchmarking:
  - http://pmem.io/2016/02/22/pm-emulation.html
  - Distros with recent kernels are usually built with DAX/pmem
    - Avoid the kernel build!
    - Can also avoid building PMDK – install from distro's repo

Persistent Memory on Windows

Availability of Windows PM Support

- Client Workstation:
  - August, 2016: Windows 10 Anniversary Update
  - April, 2017: Windows 10 Creators Update
  - October, 2017: Windows 10 Fall Creators Update
  - April, 2018: Windows 10 April 2018 Update
  - October, 2018: Windows 10 October 2018 Update
- Server:
  - September, 2016: Windows Server 2016
  - November, 2019: Windows Server 2019
- Supported Hardware:
  - JEDEC NVDIMM-N
  - HPE Scalable Persistent Memory
  - Support for **Intel Optane DC Persistent Memory** when Hardware becomes available

DAX Support

- Supported since Windows Server 2016
- Advantages:
  - Improved IO performance by eliminating OS overhead
- Disadvantages:
  - Functionality loss due to elimination of OS hook points, examples:
    - Software Encryption
    - Software Compression
- Only NTFS supports DAX

Windows PM Block Mode Volume Support

Is fully backwards compatible

- Maintains existing storage semantics
  - Sector atomicity supported by the PM disk driver
- Fully compatible with existing applications and filter drivers
- Supported by all Windows file systems

Windows Hyper-V PM Support

- Available in Windows Server 2019
- Windows & Linux guests in generation 2 VMs see virtual PMEM (vPMEM) devices
  - Memory mapped files in the guest have direct access to PM hardware on the host
  - Full Win32 and PMDK support
- New VHD file type: .VHDPMEM

Windows PM Management Support

- Windows Server 2019 introduces Powershell support for managing physical and logical persistent memory devices
  - Ability to enumerate, create and delete logical persistent memory devices
  - Ability to enumerate and initialize physical persistent memory devices
  - Example cmdlets:

- Get-PmemDisk
- New-PmemDisk
- Remove-PmemDisk
- Get-PmemPhysicalDevice
- Initialize-PmemPhysicalDevice
- Get-PmemUnusedRegion

Windows PMDK Support

- Available since Windows Server 2016
- Defines a set of application API's for efficient use of PM hardware
  - Abstracts out OS specific dependencies
  - Underlying implementation uses memory mapped files
  - Makes its own atomicity guarantees
  - Works in both PM and non-PM environments
  - Simpler application development model
- Open source library available for Windows and Linux via GitHub
  - https://github.com/pmem/pmdk/

Windows Native PM Programming

Using DAX in Windows

DAX Volume Creation

→`Command prompt: Format n: /dax /q`

→`Powershell: Format-Volume –DriveLetter n –IsDAX $true`

DAX Volume Identification

Is it a DAX volume?

→call `GetVolumeInformation("C:\", ...)`

→check `lpFileSystemFlags` for `FILE_DAX_VOLUME (0x20000000)`

Is the file on a DAX volume?

→call `GetVolumeInformationByHandleW(hFile, ...)`

→check `lpFileSystemFlags` for `FILE_DAX_VOLUME (0x20000000)`

Implicit Flush Code Example

Memory Mapping:

```
HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
LPVOID baseAddress = MapViewOfFile(hMapping, FILE_MAP_WRITE, 0, 0, size);
memcpy(baseAddress + writeOffset, dataBuffer, ioSize);
FlushViewOfFile(baseAddress, 0);
```

OR … use non-temporal instructions for NVDIMM-N devices for better performance:

```
HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
LPVOID baseAddress = MapViewOfFile(hMapping, FILE_MAP_WRITE, 0, 0, size);
RtlCopyMemoryNonTemporal(baseAddress + writeOffset, dataBuffer, ioSize );
```

Windows Native PMEM APIs

- Note! PMDK is layered on these native methods, programmer is free to choose
- Available from both user and kernel modes
- Flush to durability in optimal fashion for each hardware architecture
- Supported in Windows 10 1703 (Spring 2017) and Windows Server 2019

- **RtlGetNonVolatileToken**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlgetnonvolatiletoken
- **RtlWriteNonVolatileMemory**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlwritenonvolatilememory
- **RtlFlushNonVolatileMemory / RtlFlushNonVolatileMemoryRanges**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlflushnonvolatilememory
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlflushnonvolatilememoryranges
- **RtlDrainNonVolatileFlush**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtldrainnonvolatileflush
- **RtlFreeNonVolatileToken**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlfreenonvolatiletoken

Context (“token”)

```
_IRQL_requires_max_(APC_LEVEL)
NTSTATUS
RtlGetNonVolatileToken (
    _In_reads_bytes_(Size) PVOID NvBuffer,
    _In_ SIZE_T Size,
    _Outptr_ PVOID *NvToken);
```

- Allocates NvToken object to store properties of given mapped DAX region
- If given region is not DAX mapped, usermode callers will fail with STATUS_INVALID_PARAMETER

```
_IRQL_requires_max_(DPC_LEVEL)
NTSTATUS
RtlFreeNonVolatileToken (
    _In_ PVOID NvToken);

#define FLUSH_NV_MEMORY_DEFAULT_TOKEN        (ULONG_PTR)(-1)
```

- Frees NvToken object allocated by RtlGetNonVolatileToken

Write

```
_IRQL_requires_max_(DPC_LEVEL)
NTSTATUS
RtlWriteNonVolatileMemory (
    _In_ PVOID NvToken,
    _Out_writes_bytes_(Size) VOID UNALIGNED *NvDestination,
    _In_reads_bytes_(Size) VOID UNALIGNED *Source,
    _In_ SIZE_T Size,
    _In_ ULONG Flags);
```

- Functionally equivalent to memcpy(), with token validation and accounting
- This routine does **not** flush the copied memory

Flush

```
_IRQL_requires_max_(DPC_LEVEL)
NTSTATUS
RtlFlushNonVolatileMemory (
    _In_ PVOID NvToken,
    _In_reads_bytes_(Size) PVOID NvBuffer,
    _In_ SIZE_T Size,
    _In_ ULONG Flags);

#define FLUSH_NV_MEMORY_IN_FLAG_NO_DRAIN    (0x00000001)
```

- Optimally flushes the given DAX region
- Optional flag tells routine to not wait for the flush to drain (defers SFENCE)

Flush multiple

```
_IRQL_requires_max_(DPC_LEVEL)
NTSTATUS
RtlFlushNonVolatileMemoryRanges (
    _In_ PVOID NvToken,
    _In_reads_(TotalRanges) PNV_MEMORY_RANGE NvRanges,
    _In_ SIZE_T TotalRanges,
    _In_ ULONG Flags);

typedef struct _NV_MEMORY_RANGE {
    VOID *BaseAddress;
    SIZE_T Length;
} NV_MEMORY_RANGE, *PNV_MEMORY_RANGE;

#define FLUSH_NV_MEMORY_IN_FLAG_NO_DRAIN    (0x00000001)
```

- Only one drain operation issued across all given ranges
- Optional flag to defer waiting for flushes to drain

Drain

```
_IRQL_requires_max_(DPC_LEVEL)
NTSTATUS
RtlDrainNonVolatileFlush (
    _In_ PVOID NvToken);
```

- Waits for flush requests associated with given token to complete (SFENCE)
- Supports efficient flushing of multiple regions, and overlapped processing

NUMA information FSCTL

```
FSCTL_QUERY_VOLUME_NUMA_INFO

typedef struct _FSCTL_QUERY_VOLUME_NUMA_INFO_OUTPUT {

    ULONG NumaNode;

} FSCTL_QUERY_VOLUME_NUMA_INFO_OUTPUT,
 *PFSCTL_QUERY_VOLUME_NUMA_INFO_OUTPUT;
```

- A DAX volume can not span NUMA nodes
- Returns the NUMA node the given DAX volume resides on
- This FSCTL takes a volume handle or a handle to any file or directory on the volume
- Commandline: fsutil volume querynumainfo x:

Code Example

```
HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
LPVOID baseAddress = MapViewOfFile(hMapping, FILE_MAP_WRITE, 0, 0, size);
PVOID token;
RtlGetNonVolatileToken(baseAddress, size, &token);
for (;;)
{
   *(int *)baseAddress = random();            // Write to PMEM
   RtlFlushNonVolatileMemory(token, baseAddress, sizeof(int), 0);
   baseAddress += sizeof(int);
}
RtlFreeNonVolatileToken(token);
```

Asynchronous Code Example

```
HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
LPVOID baseAddress = MapViewOfFile(hMapping, FILE_MAP_WRITE, 0, 0, size);
PVOID token;
RtlGetNonVolatileToken(baseAddress, size, &token);
for (;;)
{
   int nextrec = random();
   RtlWriteNonVolatileMemory(token, baseAddress, &nextrec, sizeof (int), 0);
   RtlFlushNonVolatileMemory(token, baseAddress, sizeof(int),
       FLUSH_NV_MEMORY_IN_FLAG_NO_DRAIN );
   baseAddress += sizeof(int);
   // do more stuff
   RtlDrainNonVolatileFlush(token);     // Wait for prior Flush
}
RtlFreeNonVolatileToken(token);
```

Demo:
Persistent Memory

BREAK

15 minutes

Libraries

Allocation, Transactions, Replication

Starting from a mapped file…

raw.c

```
if ((fd = open(argv[1], O_RDWR)) < 0)
        err(1, "open: %s", argv[2]);

if ((pmaddr = (char *)mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                                MAP_SHARED, fd, 0)) == MAP_FAILED)
        err(1, "mmap: %s", argv[2]);

close(fd);

/* write to persistent memory... */
strcpy(pmaddr, "Hello, persistent memory!");

/* flush the changes... */
if (msync((void *)pmaddr, 4096, MS_SYNC) < 0)
        err(1, "msync: %s", argv[2]);
```

PMDK Libraries: pmem.io

C/C++ (Java early access, Python, js experimental) on Linux and Windows

PCJ – Persistent Collection for Java

pmemkv

C++

C

PCJ/ LLPL

Python

Transaction Support

Interface to create a persistent memory resident log file

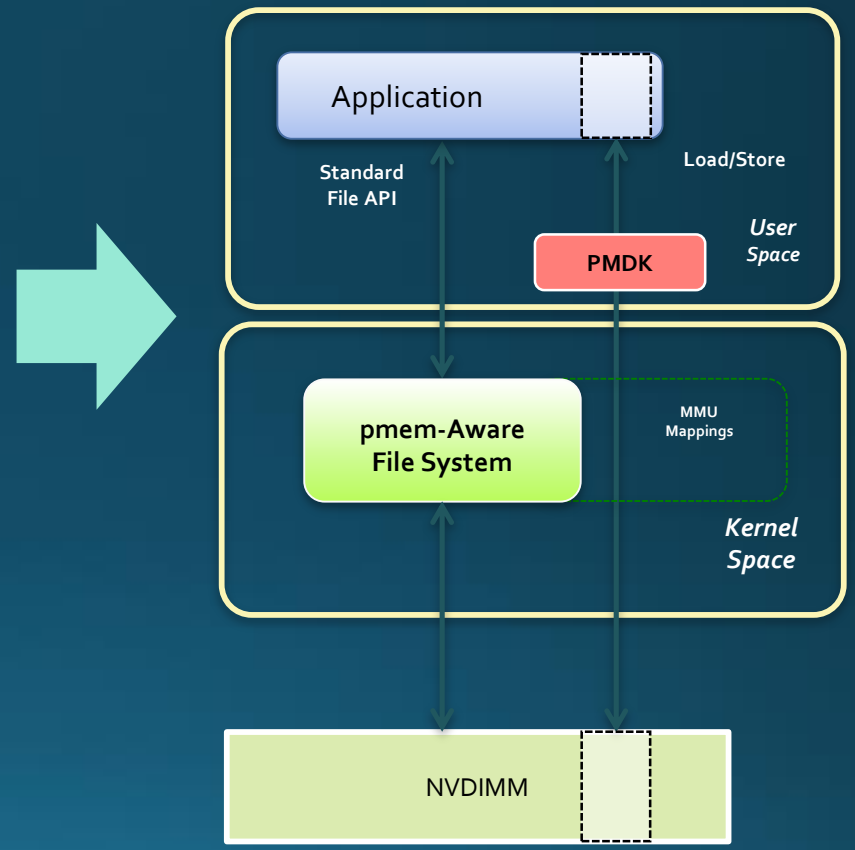
libpmemlog

Interface for persistent memory allocation, transactions and general facilities

libpmemobj

Interface to create arrays of pmem-resident blocks, of same size, atomically updated

libpmemblk



Support for **volatile** memory usage

memkind

Low level support for local persistent memory

libpmem

Low level support for remote access to persistent memory

librpmem

Low-level support

In Development

High Level Language Support

- What's the opposite of dealing with a raw range from mmap?
  - Simple key-value APIs
- High Level languages…
  - Java
  - JavaScript (really node.js)
  - Ruby
  - C++ (but using a very simple API)
  - C (but using a very simple API)

- https://github.com/pmem/pmemkv
- https://github.com/pmem/pmemkv-tools

libpmemkv Basics

- Experimental
  - So why am I telling you about it? "Productization" is underway
- Basic key-value operations:
  - Get
  - Put
  - Remove
  - All/Each
- Multiple "engines"
  - Most performant: highly-scalable, based on TBB's concurrent hashmap

What Does the API Look Like?

```
// add some keys and values
kv->put("key1", "value1");

// …

// iterate through the key-value store
kv->get_all(kvprint);
```

What Does the API Look Like?

```
// add some keys and values
kv->put("key1", "value1");

// …

// iterate through the key-value store
kv->get_all(kvprint);
```

- Transactional
- Immediately persistent

What Does the API Look Like?

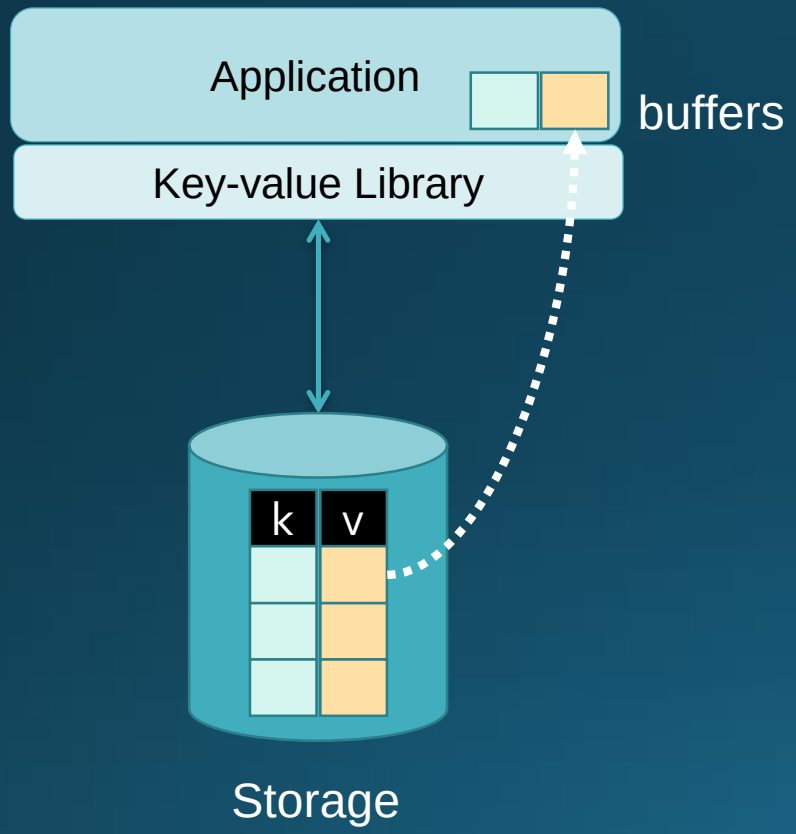
```
/*
 * kvprint -- print a single key-value pair
 */
int kvprint(string_view k, string_view v) {
    cout << "key: " << k.data() << ", value: " << v.data() << endl;
    return 0;
}
```

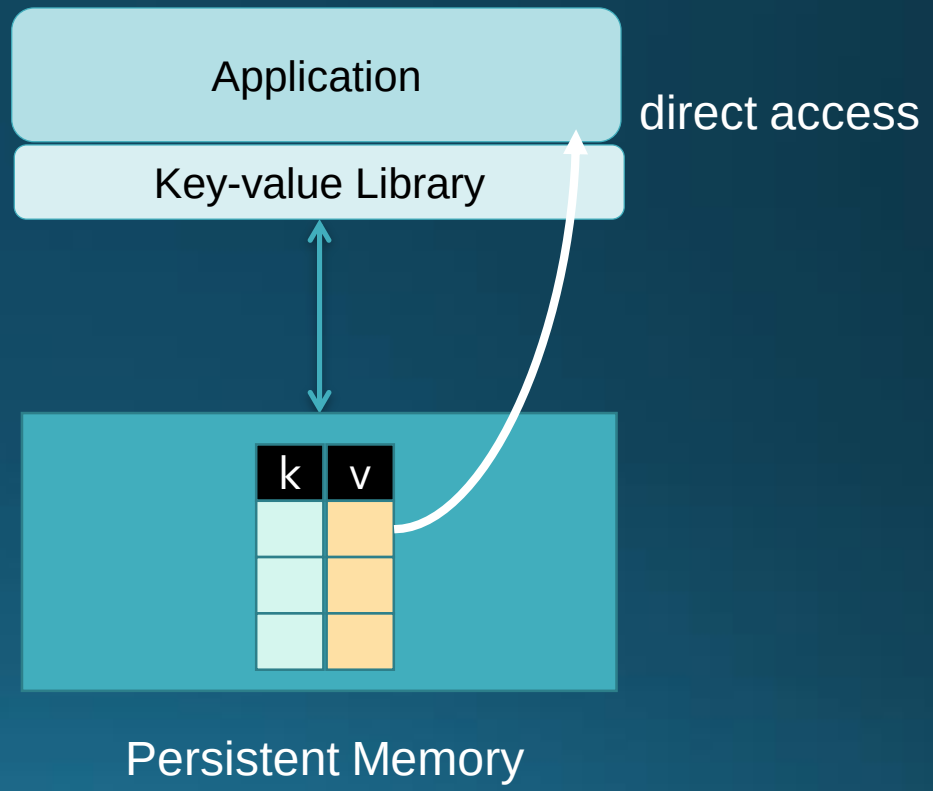
What Does the API Look Like?

```
/*
 * kvprint -- print a single key-value pair
 */
int kvprint(string_view k, string_view v) {
    cout << "key: " << k.data() << ", value: " << v.data() << endl;
    return 0;
}
```

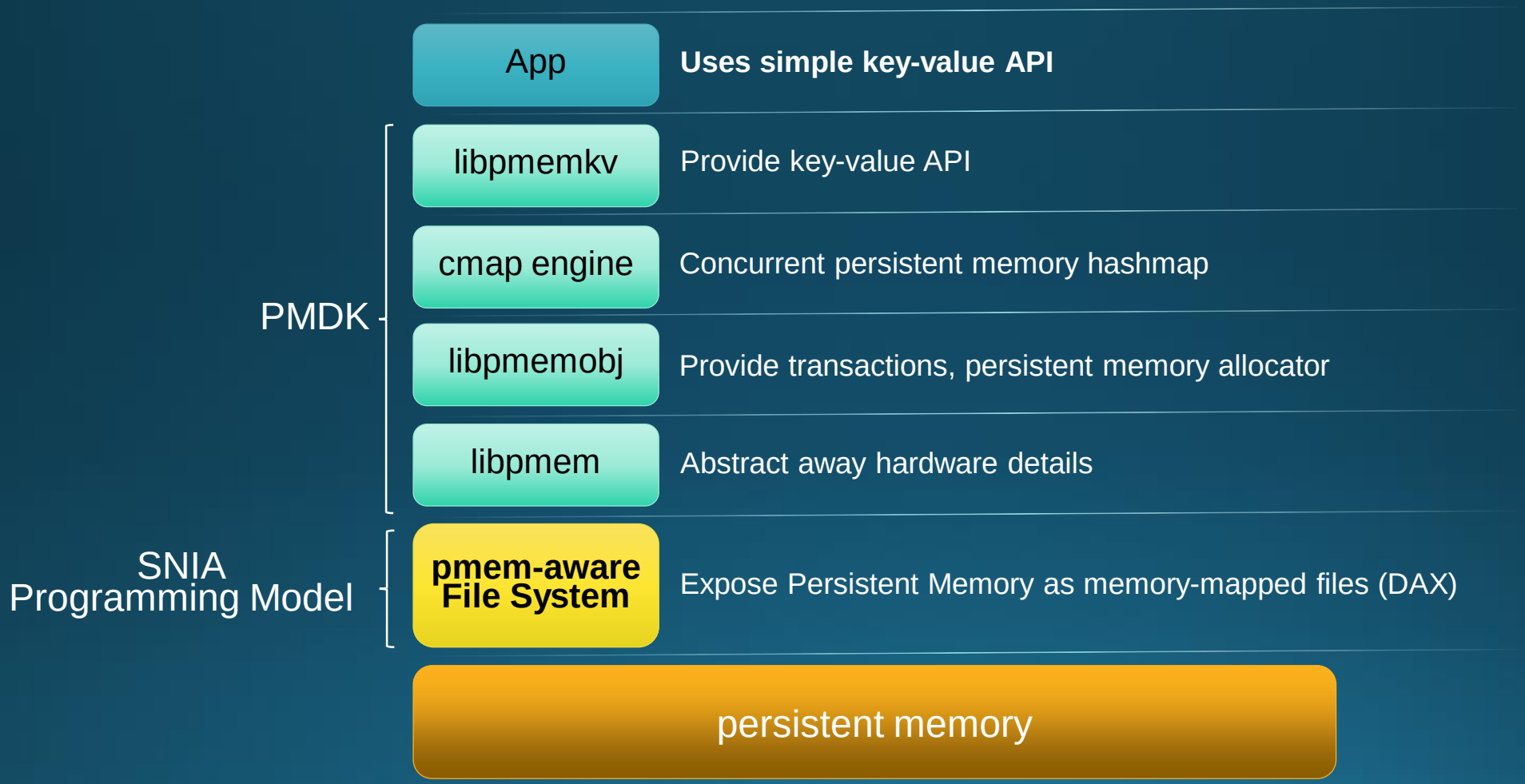
- Access in-place

How is This K-V Store Different?





The Full Stack

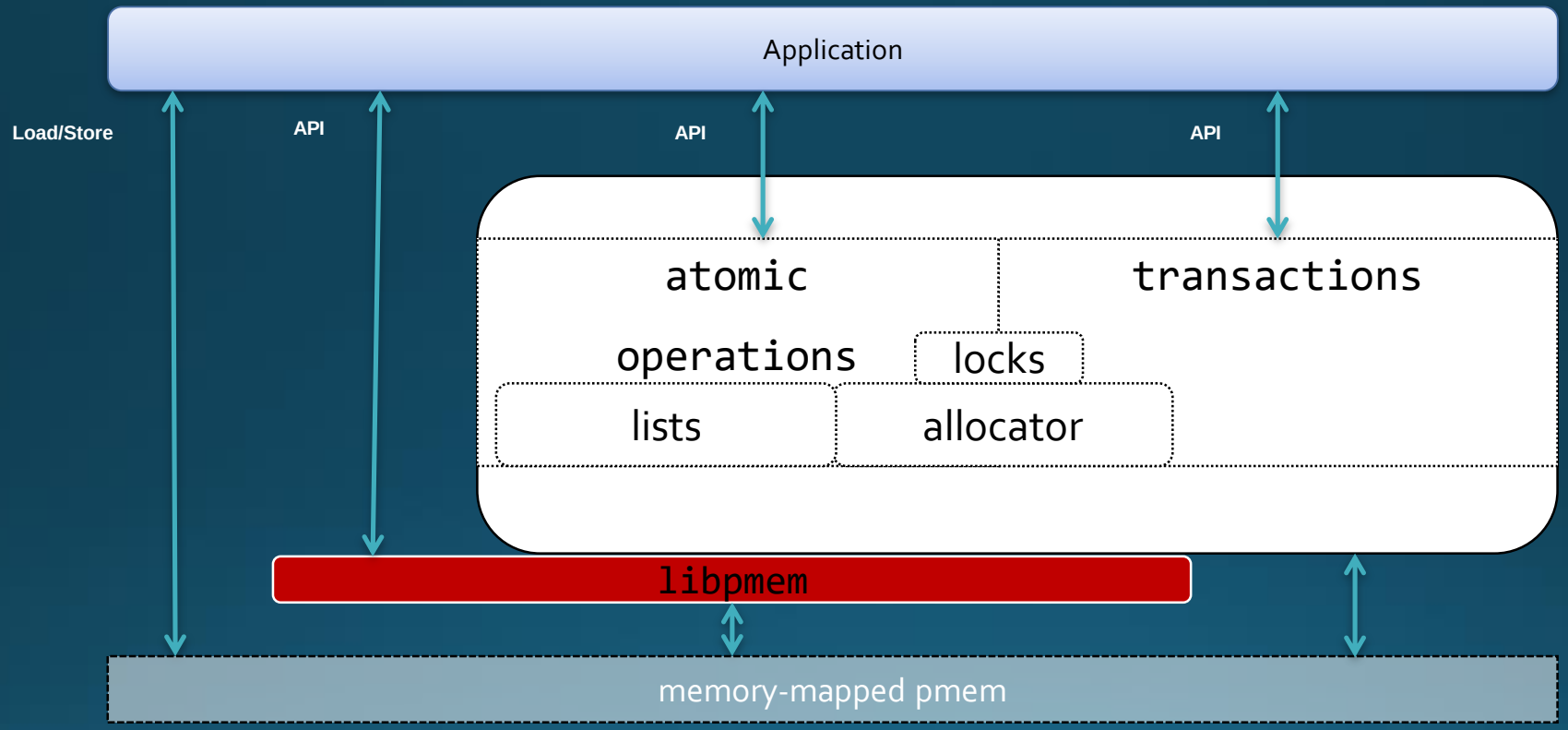


Additional High-Level Examples…

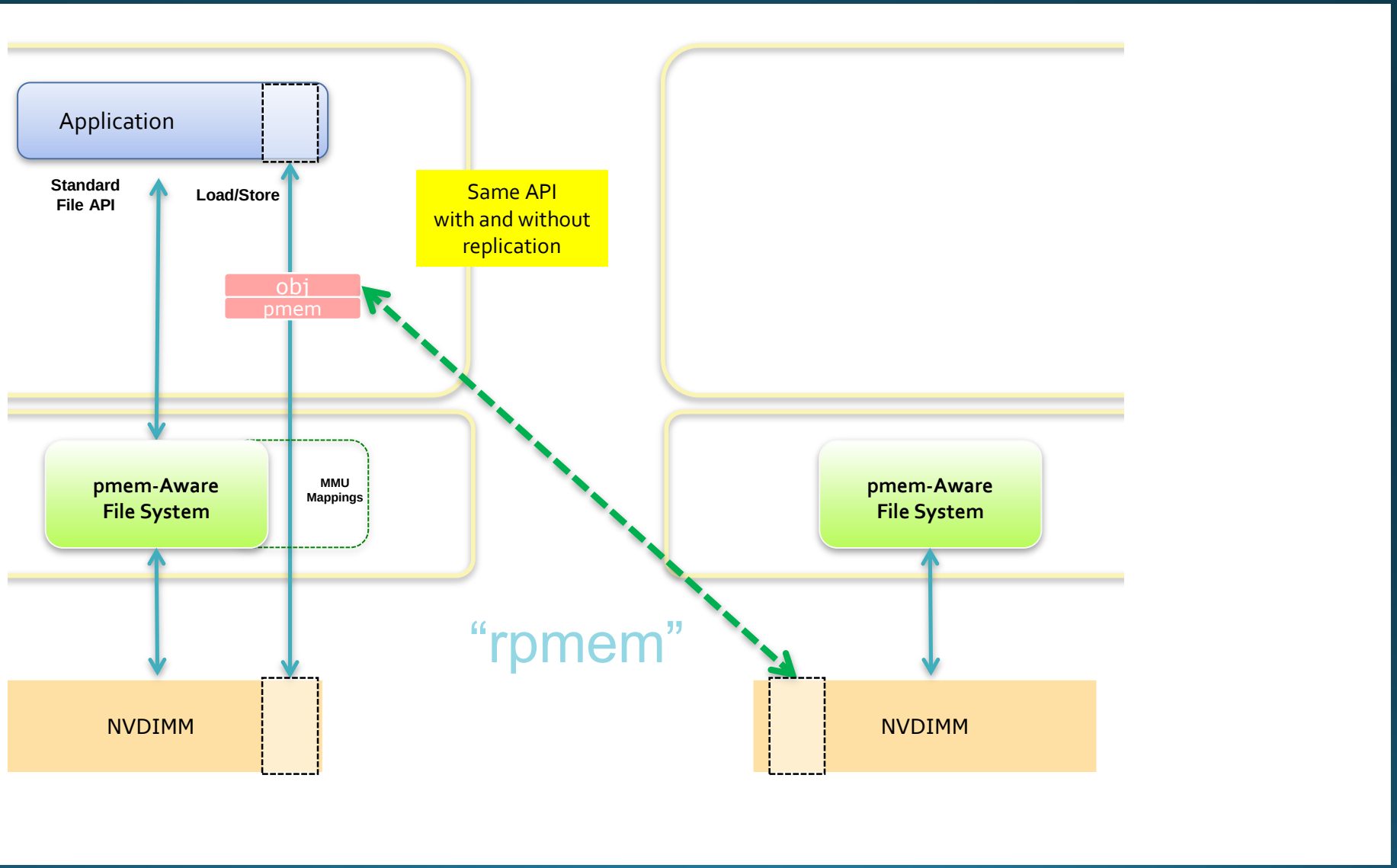
- On GitHub:
  - node.js
    - https://github.com/pmem/pmemkv-nodejs
  - C & C++
    - https://github.com/pmem/pmemkv

libpmemobj

"transactional object store"



Libpmemobj Replication: Application Transparent
(except for performance overhead)



Building on **libpmemobj**

- From C
  - Fully validated, delivered on Linux, early access on Windows
  - Can stick to pure API calls, but macros add:
    - Compile-time type safety
    - Transaction syntax, similar to try/catch
- From C++
  - Fully validated, delivered on Linux, early access on Windows
  - Use C++ type system & syntax: much cleaner, less error-prone
- From Java
  - Persistent Containers for Java (Experimental)
- From Python
  - PyNVM (Experimental)
- Other work
  - valgrind (and a similar tool coming from Intel)
  - JavaScript (Pre-release)

The pmempool command

pmempool-info(1)
Prints information and statistics in human-readable format about specified pool.

pmempool-check(1)
Checks pool's consistency and repairs pool if it is not consistent.

pmempool-create(1)
Creates a pool of specified type with additional properties specific for this type of pool.

pmempool-dump(1)
Dumps usable data from pool in hexadecimal or binary format.

pmempool-rm(1)
Removes pool file or all pool files listed in poolset configuration file.

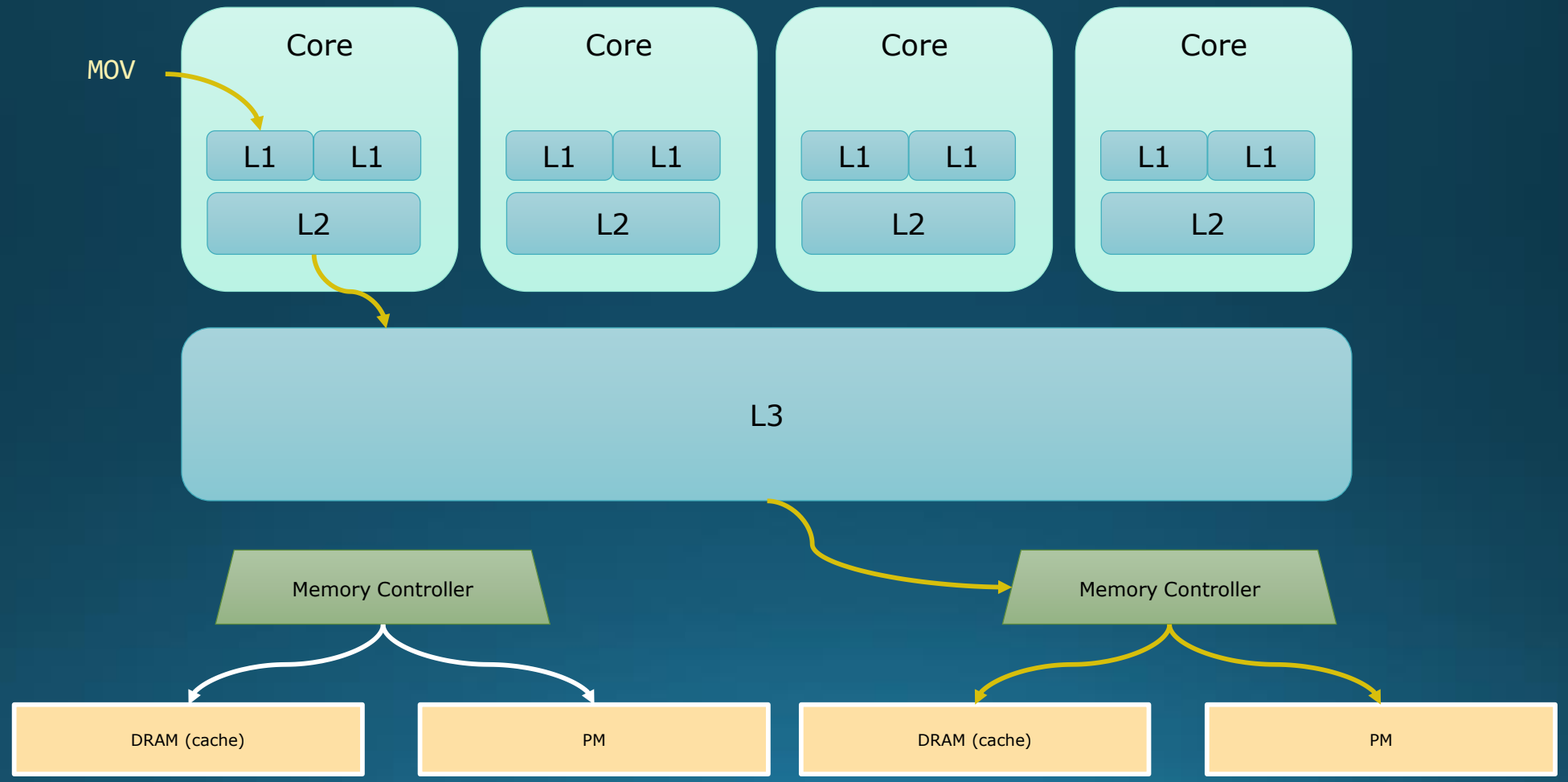
pmempool-convert(1)
Updates the pool to the latest available layout version.

Brief Digression:

Do you even need persistence?

Many volatile use cases are coming up.
Most use pmem for capacity/cost.
There are (at least) two ways to do this…

Memory Mode


MOV
Core
L1
L1
L2
Core
L1
L1
L2
Core
L1
L1
L2
Core
L1
L1
L2
L3
Memory Controller
Memory Controller
DRAM (cache)
PM
DRAM (cache)
PM

libmemkind

- http://memkind.github.io/memkind/
- C malloc/free library
- Other libraries can build on it
  - example: Java off-heap storage
- Really just the popular jemalloc library
  - Modified to support multiple, independent pools of memory
- Allocate memory by "kind"
  - pmem mapped file is one of the kinds

- Application chooses placement of volatile data
  - DRAM or pmem

Essential libpmem Knowledge

libpmem examples

Source: https://github.com/pmem/nvml/tree/master/src/examples/libpmem

```
/*
 * simple_copy.c -- show how to use pmem_memcpy_persist()
 *
 * usage: simple_copy src-file dst-file
 *
 * Reads 4k from src-file and writes it to dst-file.
 */
```

```
    /* create a pmem file and memory map it */
    if ((pmemaddr = pmem_map_file(argv[2], BUF_LEN,
                                  PMEM_FILE_CREATE|PMEM_FILE_EXCL,
                                  0666, &mapped_len, &is_pmem)) == NULL) {
            perror("pmem_map_file");
            exit(1);
    }
```

Using `is_pmem`

```
if (is_pmem) {
        pmem_memcpy_persist(pmemaddr, buf, cc);
} else {
        memcpy(pmemaddr, buf, cc);
        pmem_msync(pmemaddr, cc);
}
```

POSIX Load/Store Persistence

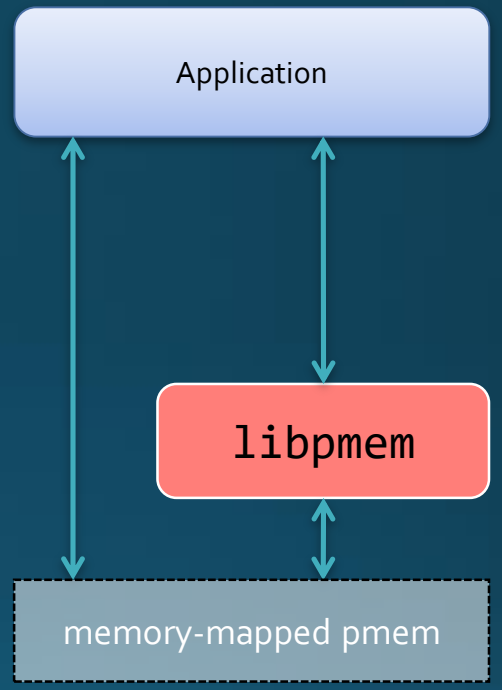
```
open(…);
mmap(…);

strcpy(pmem, "andy");

msync(pmem, 5, MS_SYNC);
```

Optimized Flush (only use when safe!)

```
open(…);
mmap(…);

strcpy(pmem, "andy");

pmem_persist(pmem, 5);
```

libpmem Load/Store Persistence

```
open(…);
mmap(…);

strcpy(pmem, "andy");

pmem_persist(pmem, 5);
```



Crossing the 8-byte Store

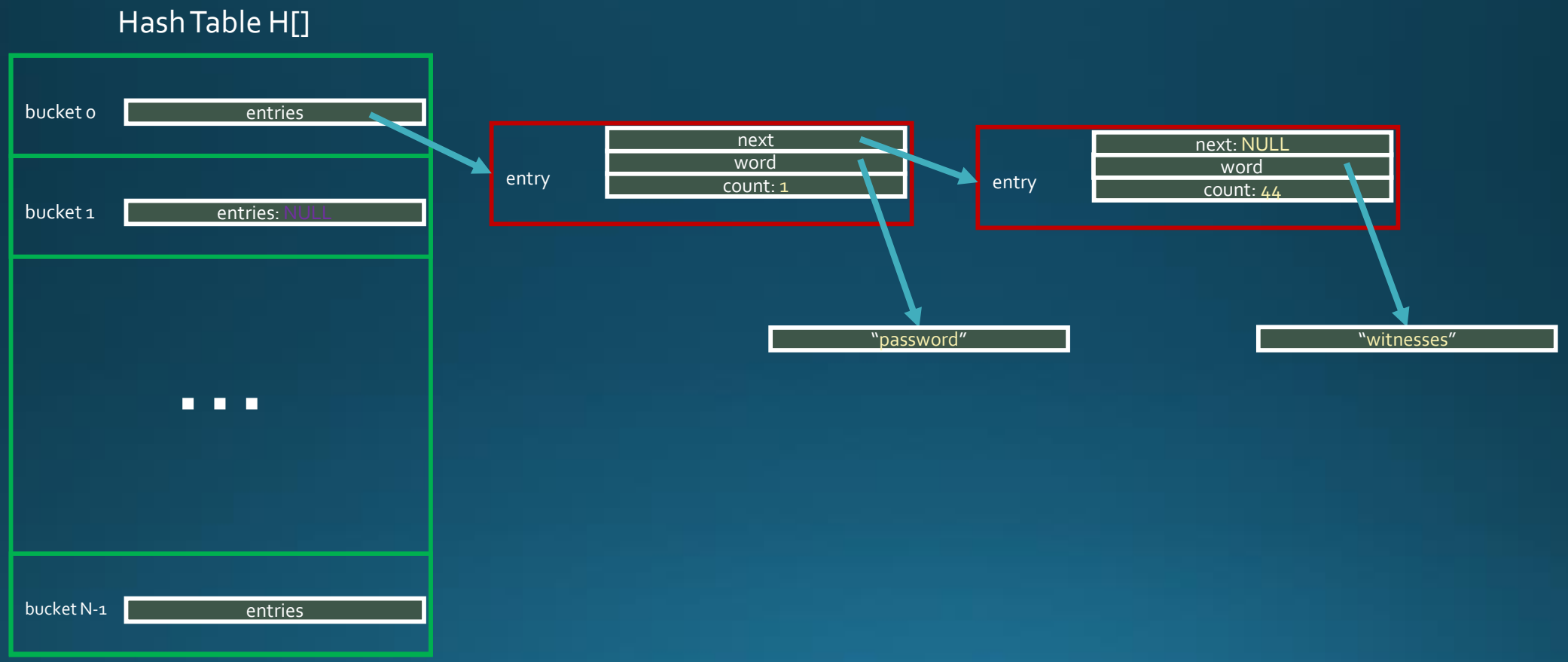
```
open(…);
mmap(…);

strcpy(pmem, "andy rudoff");

pmem_persist(pmem, 12);
```

Which Result?

1. "\0\0\0\0\0\0\0\0…"
2. "andy\0\0\0\0\0\0…"
3. "andy rud\0\0\0\0\0…"
4. "\0\0\0\0\0\0\0off\0\0..."
5. "andy rudoff\0"

libpmemobj Examples

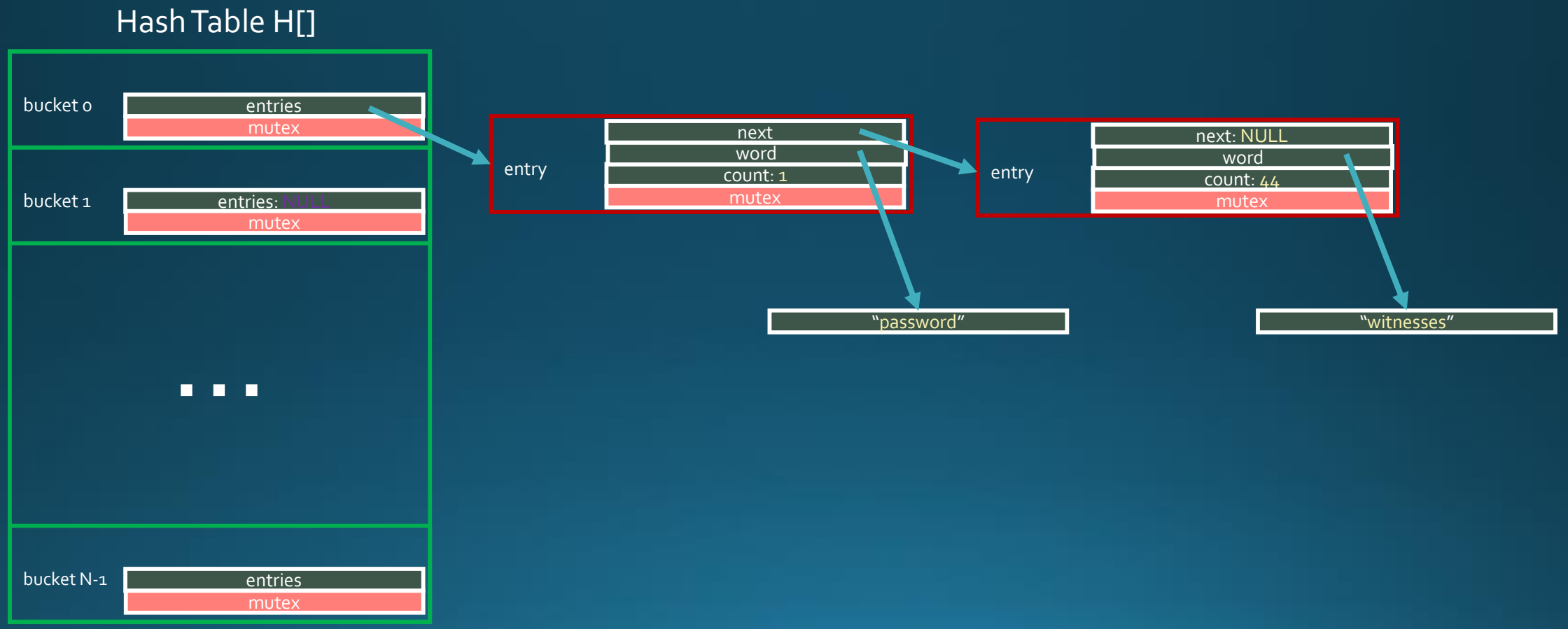
Simple C program to build example on (nothing related to pmem yet)



freq.c

```
$ freq -p words.txt
1 is
1 all
1 for
2 to
1 men
1 good
2 the
1 come
1 their
1 Now
1 time
1 country
1 aid
1 of
```

Adding multi-threading support (nothing related to pmem yet)

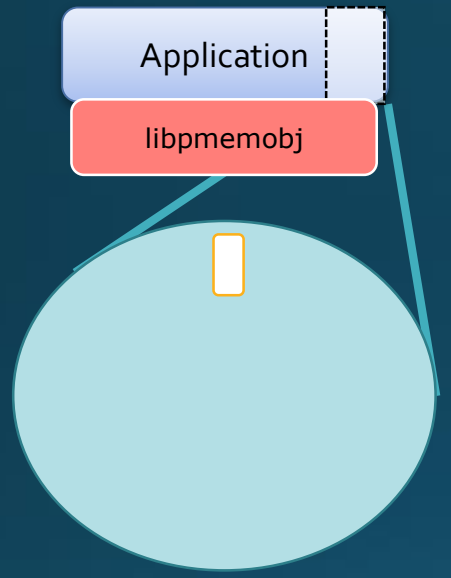


freq_mt.c

```
$ freq_mt -p words.txt words.txt words.txt
3 is
3 all
3 for
6 to
3 men
3 good
6 the
3 come
3 their
3 Now
3 time
3 country
3 aid
3 of
```

The *Root Object*:

At well-known location, used to find all other persistent data structures

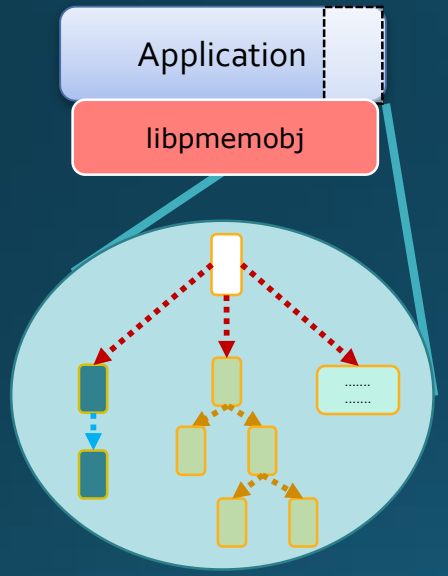


pmem pool "myfile"

root object:

- `assume it is always there`
- `created first time accessed`
- `initially zeroed`

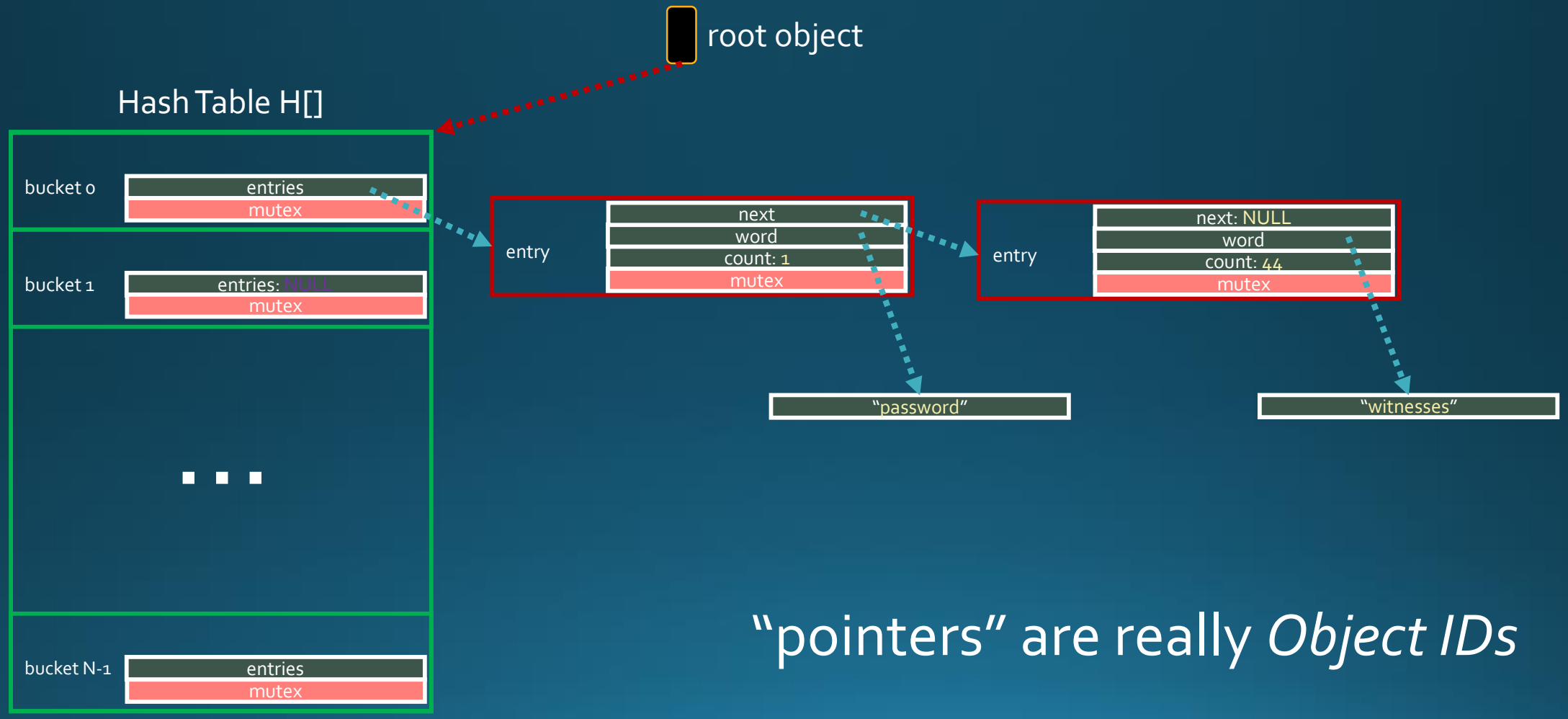
Using the Root Object



Link pmem data structures in pool off the root object to find them on each program run

"pointers" are really *Object IDs*

Moving data the example to pmem



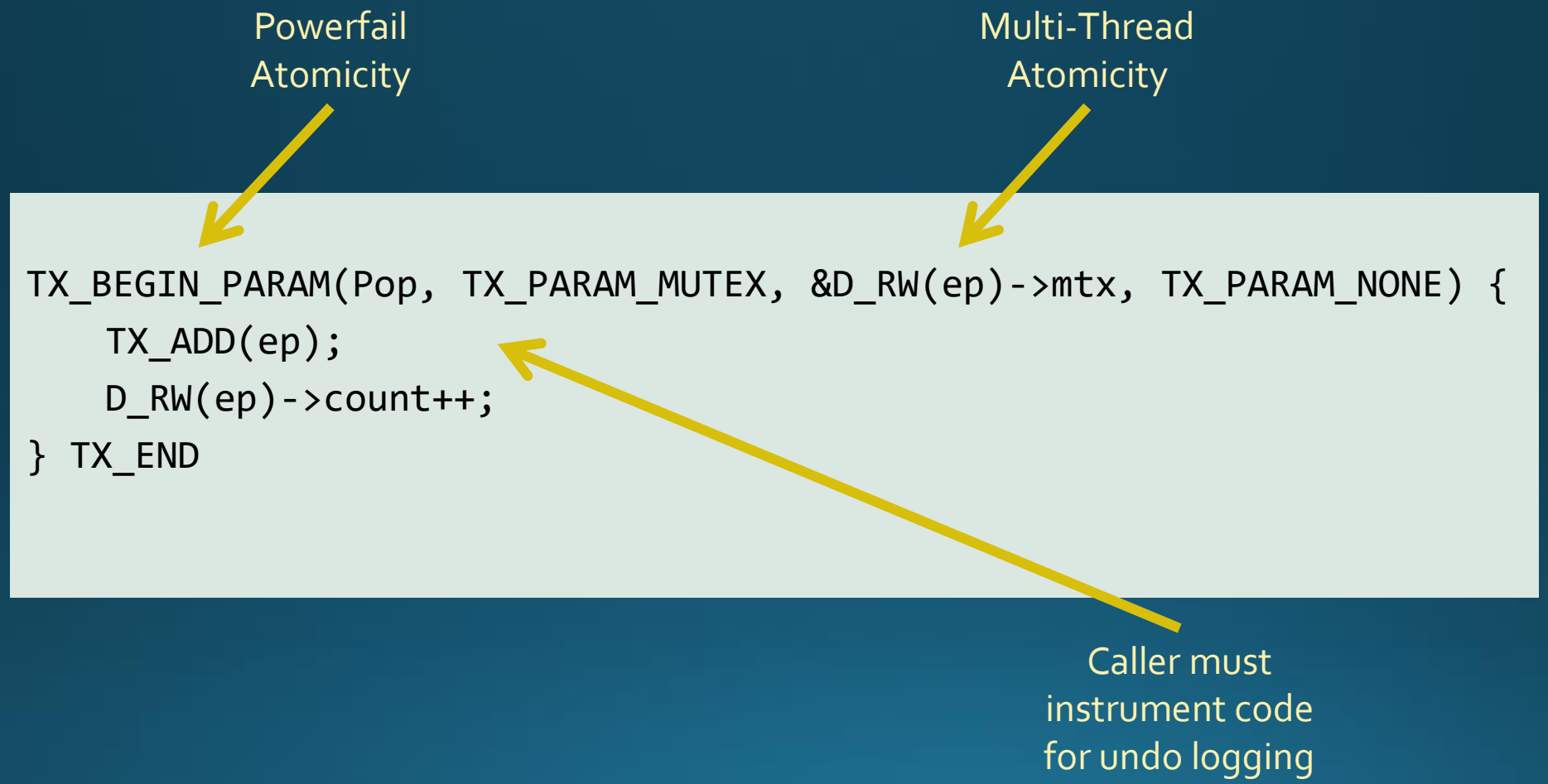
"pointers" are really *Object IDs*

C Programming with libpmemobj

Transaction Syntax

```
TX_BEGIN(Pop) {
                    /* the actual transaction code goes here... */
} TX_ONCOMMIT {
                    /*
                     * optional - executed only if the above block
                     * successfully completes
                     */
} TX_ONABORT {
                    /*
                     * optional - executed if starting the transaction fails
                     * or if transaction is aborted by an error or a call to
                     * pmemobj_tx_abort()
                     */
} TX_FINALLY {
                    /*
                     * optional - if exists, it is executed after
                     * TX_ONCOMMIT or TX_ONABORT block
                     */
} TX_END /* mandatory */
```

Properties of Transactions



Persistent Memory Locks

- Want locks to live near the data they protect (i.e. inside structs)
- Does the state of locks get stored persistently?
  - Would have to flush to persistence when used
  - Would have to recover locked locks on start-up
    - Might be a different program accessing the file
  - Would run at pmem speeds

- PMEMmutex
  - Runs at DRAM speeds
  - Automatically initialized on pool open

freq_pmem.c

```
$ pmempool create obj --layout=freq -s 1G freqcount

$ freq_pmem_print freqcount

$ freq_pmem freqcount words.txt words.txt words.txt

$ freq_pmem_print freqcount
3 is
3 all
3 for
6 to
3 men
3 good
6 the
…
```

C++ Programming with libpmemobj

C++ Queue Example: Declarations

```
/* entry in the queue */
struct pmem_entry {
   persistent_ptr<pmem_entry> next;
   p<uint64_t> value;
};
```

`persistent_ptr<T>`	Pointer is really a position-independent Object ID in pmem. Gets rid of need to use C macros like D_RW()
`p<T>`	Field is pmem-resident and needs to be maintained persistently. Gets rid of need to use C macros like TX_ADD()

C++ Queue Example: Transaction

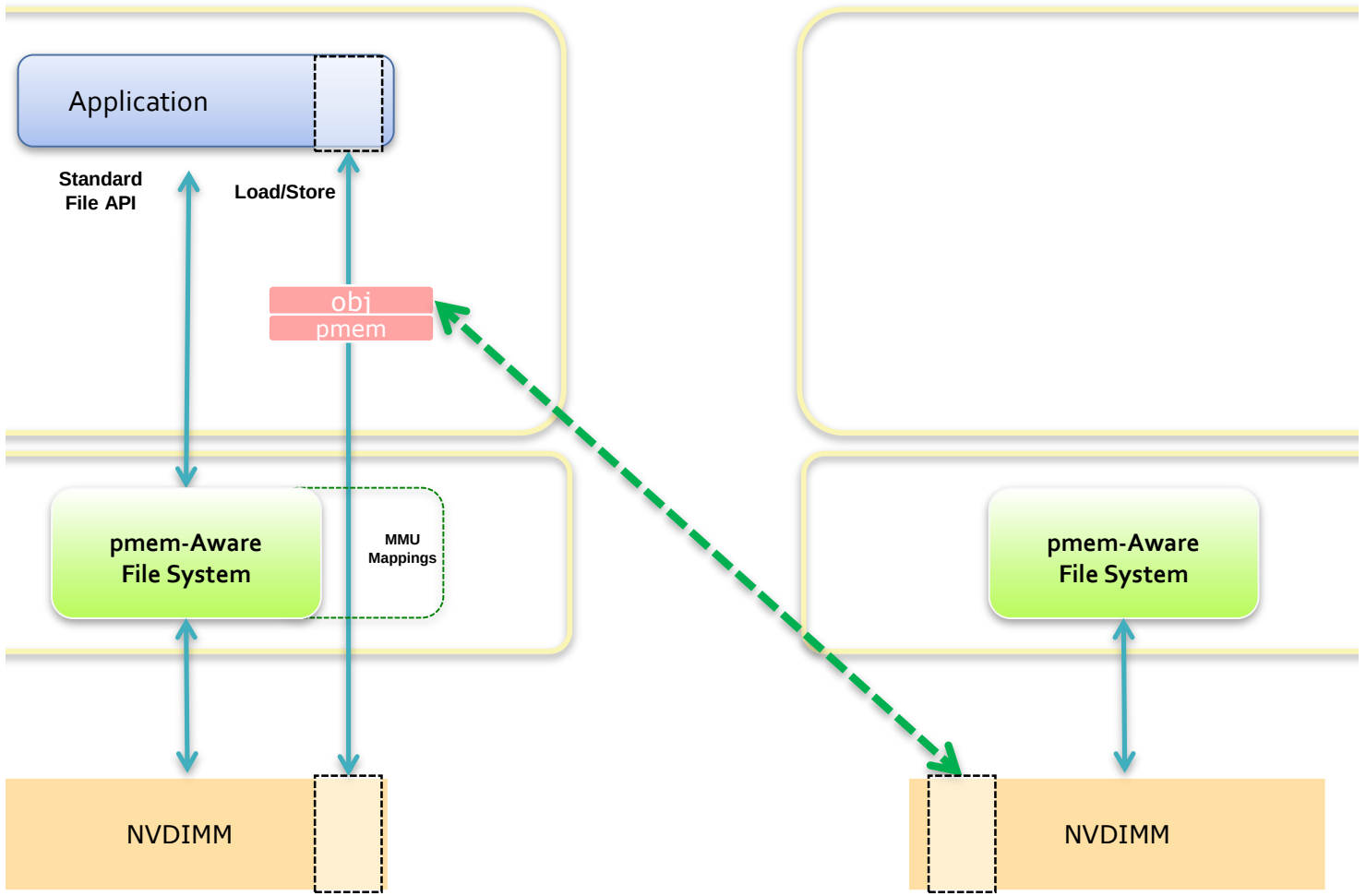
```
void push(pool_base &pop, uint64_t value) {
  transaction::run(pop, [&] {
    auto n = make_persistent<pmem_entry>();

    n->value = value;
    n->next = nullptr;
    if (head == nullptr) {
      head = tail = n;
    } else {
      tail->next = n;
      tail = n;
    }
  });
}
```

Transactional
(including allocations & frees)

freq_pmem_cpp.c

```
$ freq_pmem_cpp freqcount words.txt words.txt words.txt

$ freq_pmem_print freqcount
6 is
6 all
6 for
12 to
6 men
6 good
12 the
6 come
6 their
6 Now
6 time
6 country
6 aid
6 of
```

Libpmemobj Replication: Application Transparent
(except for performance overhead)



Persistent Memory

Q&A

More Information

Q&A

Links Used in This Deck

- http://pmem.io
  - Website for pmem programming, blogs, tutorials, examples
- https://github.com/pmem/pmdk
  - Source for PMDK supporting Windows, Linux in C and C++
- http://pmem.io/pmdk/manpages/master/libpmemobj.3.html
  - libpmemobj man page (for C programming)
- http://pmem.io/pmdk/cpp_obj/master/cpp_html/index.html
  - libpmemobj C++ interface documentation
- https://github.com/pmem/pmdk/tree/master/src/examples
  - PMDK examples, all buildable and runnable
- https://github.com/andyrudoff/pearc19
  - Code examples used in this deck

Windows Native PMEM APIs

- Note! PMDK is layered on these native methods, programmer is free to choose
- Available from both user and kernel modes
- Flush to durability in optimal fashion for each hardware architecture
- Supported in Windows 10 1703 (Spring 2017) and Windows Server 2019

- **RtlGetNonVolatileToken**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlgetnonvolatiletoken
- **RtlWriteNonVolatileMemory**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlwritenonvolatilememory
- **RtlFlushNonVolatileMemory / RtlFlushNonVolatileMemoryRanges**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlflushnonvolatilememory
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlflushnonvolatilememoryranges
- **RtlDrainNonVolatileFlush**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtldrainnonvolatileflush
- **RtlFreeNonVolatileToken**
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-rtlfreenonvolatiletoken

Links to Additional Information

- http://pmem.io/documents
  - Getting Started Guide…. Lots of additional links
- https://nvdimm.wiki.kernel.org/
  - Linux kernel information
- https://software.intel.com/pmem
  - Intel Developer Zone, tutorials, videos, links to more info

SNIA PMEM Activity

http://www.snia.org/PM

- SNIA Standards Portfolio
  - NVM Programming Model v1.2 – SNIA Technical Position
  - NVM Programming Model v1.1- SNIA Technical Position
  - NVM Programming Model v1.0 - SNIA Technical Position
- SNIA Technical White Papers
  - NVM PM Remote Access for High Availability
  - Persistent Memory Atomics and Transactions
- SNIA Videos and Presentations
  - The SNIA NVM Programming Model – Latest Developments and Challenges
  - Persistent Memory Summit