INTRODUCTION TO PROGRAMMING FOR PERSISTENT MEMORY

Speaker: Zhiming Li, Peifeng Si (Intel Data Center Group)

<Zhiming.li/Peifeng.si@intel.com>

July, 2019

Agenda

Persistent Memory Overview

Persistent Memory Programming

Persistent Memory Development Libraries

Agenda

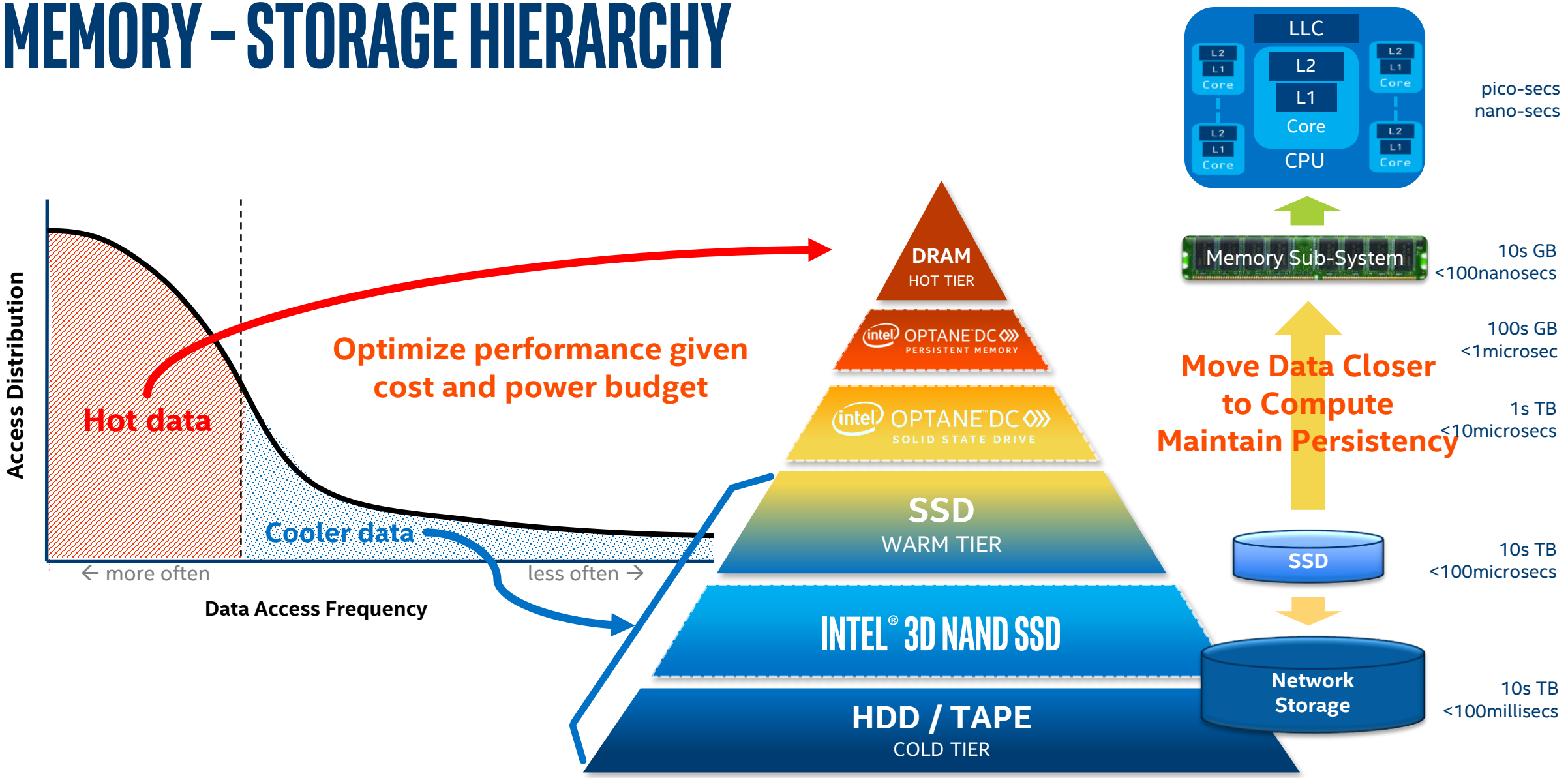
Persistent Memory Overview

- Memory - Storage hierarchy
- What is Persistent Memory?
- Persistent Memory usage modes

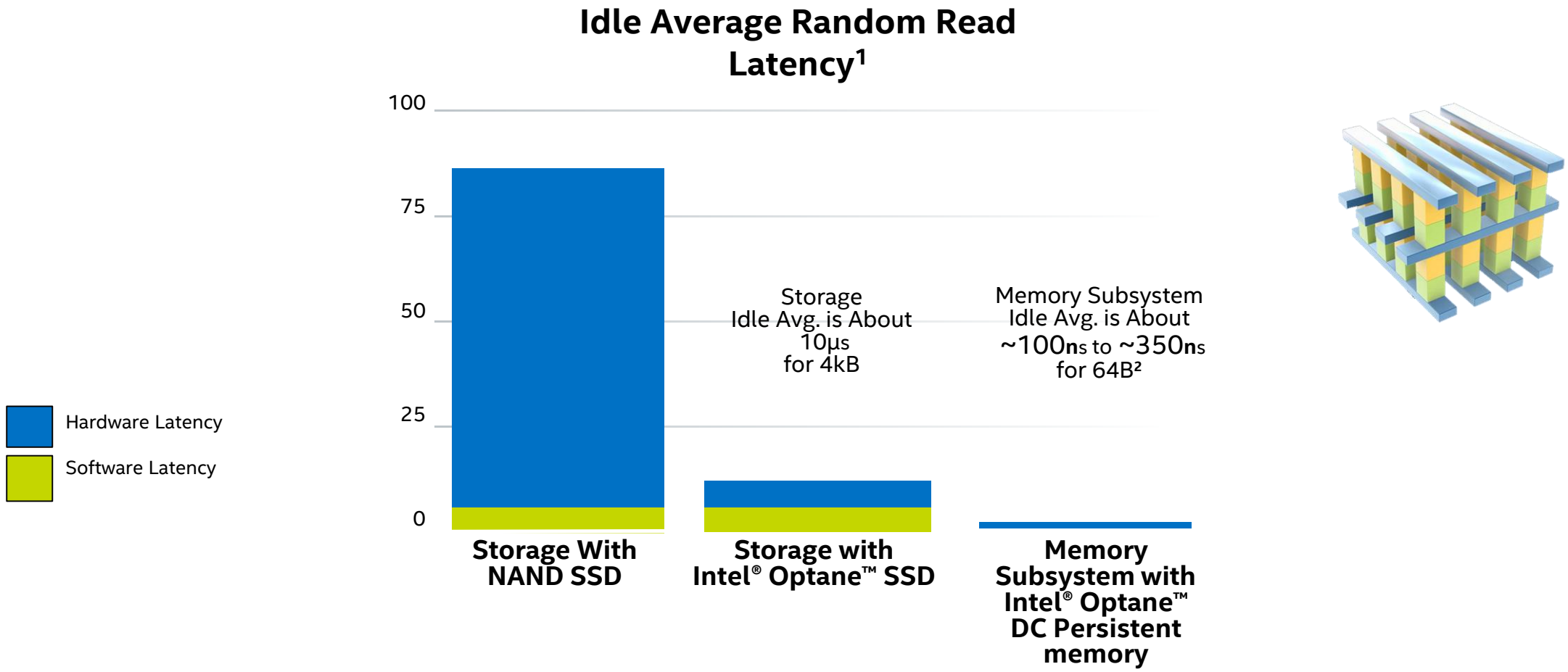
Persistent Memory Programming

Persistent Memory Development Libraries

MEMORY – STORAGE HIERARCHY



MEMORY – STORAGE HIERARCHY



[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)

LATENCY AT HUMAN SCALE

System Event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns (1 cycle)	**1 s**
Level 1 cache access	2 ns (5 cycles)	**5 s**
Level 2 cache access	4.8 ns (12 cycles)	**12 s**
Level 3 cache access	26 ns (65 cycles)	**1 min 5sec**
Main memory access (DDR DIMM)	<100 ns	**4 min 10sec**
NVDIMM-N memory access	<100 ns	**4 min 10sec**
Intel Optane DC Persistent Memory access	<100-300 ns	**4 min 10sec - 12 min**
Intel Optane DC SSD I/O P4800X NVMe	~10 µs	**~7hrs**
NVMe SSD I/O	~25 µs	**17 hrs 21min**
SSD I/O	50–150 µs	**1 day 11hrs – 4 days, 8hrs**
Rotational disk I/O	1–10 ms	**28 days 22hrs – 289 days**
Tape	~100ms	**7 yrs 11 months**

From “Systems Performance: Enterprise and the Cloud”, Brendan Gregg

WHAT IS PERSISTENT MEMORY?

- Byte or block addressable
- load/store memory access
- persistence properties of storage

JEDEC NVDIMM Standards			
	NVDIMM-F	NVDIMM-N	NVDIMM-P
IO Access Methods	Block	Block or Byte	Block or Byte
Capacity	100’s GB – 1's TB	1's - 10's GB	100’s GB – 1's TB
Latency	<50us	<100ns	<300ns
First Availability	2014	2016	2019
Operating System Support	Linux Kernel x.x Windows?	Linux Kernel >4.0 Windows Server 2016	Linux Kernel >4.2 Windows Server 2019

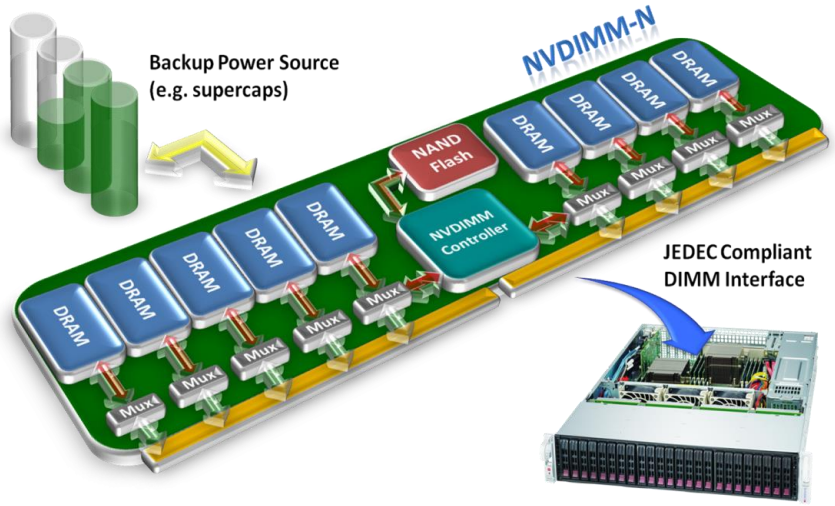
WHAT IS PERSISTENT MEMORY?

- byte-addressable
- load/store memory access
- persistence properties of storage

JEDEC NVDIMM Standards			
	NVDIMM-F	NVDIMM-N	NVDIMM-P
IO Access Methods	Block	Block or Byte	Block or Byte
Capacity	100’s GB – 1's TB	1's - 10's GB	100’s GB – 1's TB
Latency	<50us	<100ns	<300ns
First Availability	2014	2016	2019
Operating System Support	Linux Kernel x.x Windows?	Linux Kernel >4.0 Windows Server 2016	Linux Kernel >4.2 Windows Server 2019

WHAT IS PERSISTENT MEMORY?

- byte-addressable
- load/store memory access
- persistence properties of storage

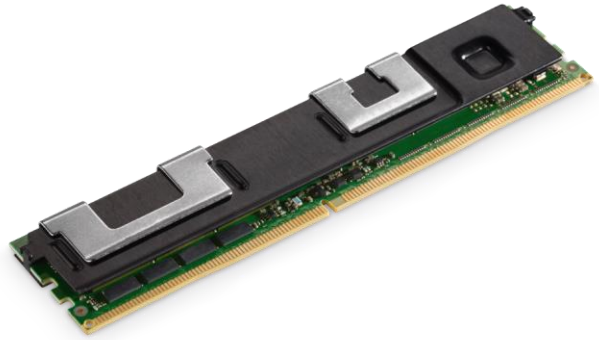


JEDEC NVDIMM Standards			
	NVDIMM-F	NVDIMM-N	NVDIMM-P
IO Access Methods	Block	Block or Byte	Block or Byte
Capacity	100's GB – 1's TB	1's - 10's GB	100's GB – 1's TB
Latency	<50us	<100ns	<300ns
First Availability	2014	2016	2019
Operating System Support	Linux Kernel x.x Windows?	Linux Kernel >4.0 Windows Server 2016	Linux Kernel >4.2 Windows Server 2019

WHAT IS PERSISTENT MEMORY?

intel OPTANE DC PERSISTENT MEMORY

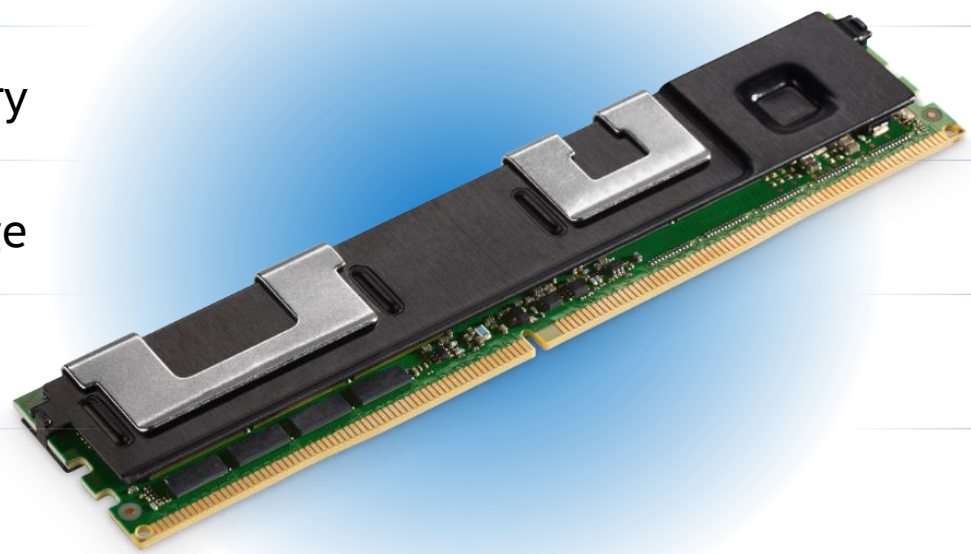
- byte-addressable
- load/store memory access
- persistence properties of storage



JEDEC NVDIMM Standards			
	NVDIMM-F	NVDIMM-N	NVDIMM-P
IO Access Methods	Block	Block or Byte	Block or Byte
Capacity	100's GB – 1's TB	1's - 10's GB	100's GB – 1's TB
Latency	<50us	<100ns	<300ns
First Availability	2014	2016	2019
Operating System Support	Linux Kernel x.x Windows?	Linux Kernel >4.0 Windows Server 2016	Linux Kernel >=4.15 Windows Server 2019

intel OPTANE™ DC PERSISTENT MEMORY

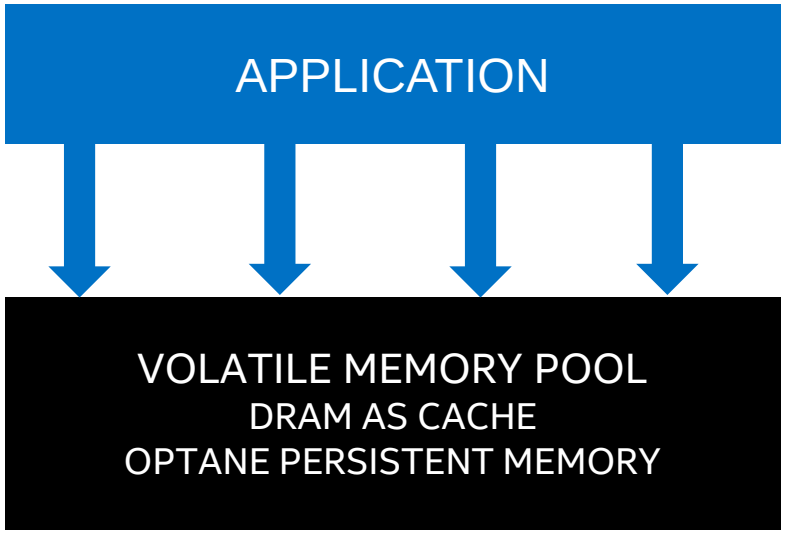
Big and Affordable Memory	128, 256, 512GB
Highest Performance Storage	DDR4 Pin Compatible
Direct Load/Store Access	Hardware Encryption
Native Persistence	High Reliability



PERSISTENT MEMORY USAGE MODES

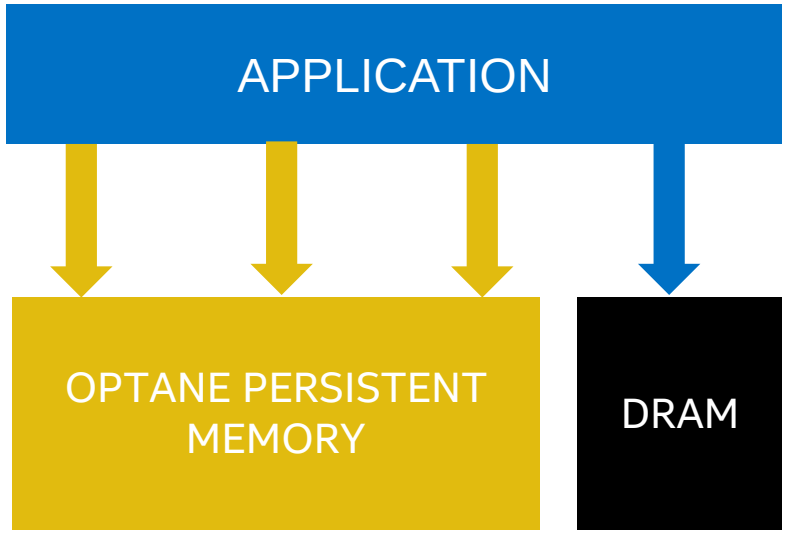
Memory Mode

Affordable memory capacity for many applications



APP Direct Mode

Persistent & maximum capacity



Agenda

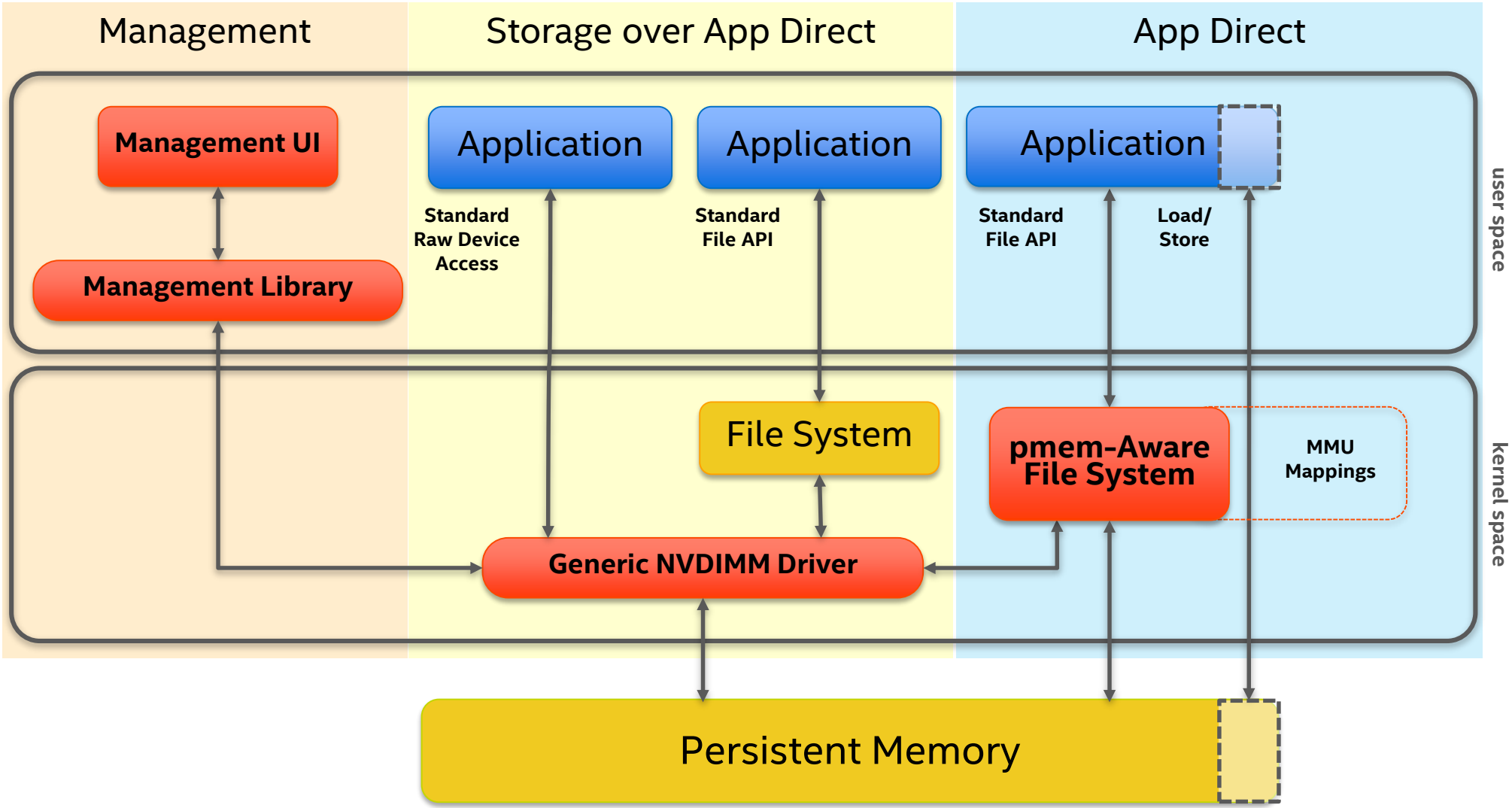
Persistent Memory Overview

Persistent Memory Programming

- SNIA NVM Programming Model
- Application responsibilities:
  - Understanding power-failure atomicity
  - Persistence domain
  - Visibility versus Power Fail Atomicity

Persistent Memory Development Libraries

SNIA NVM PROGRAMMING MODEL

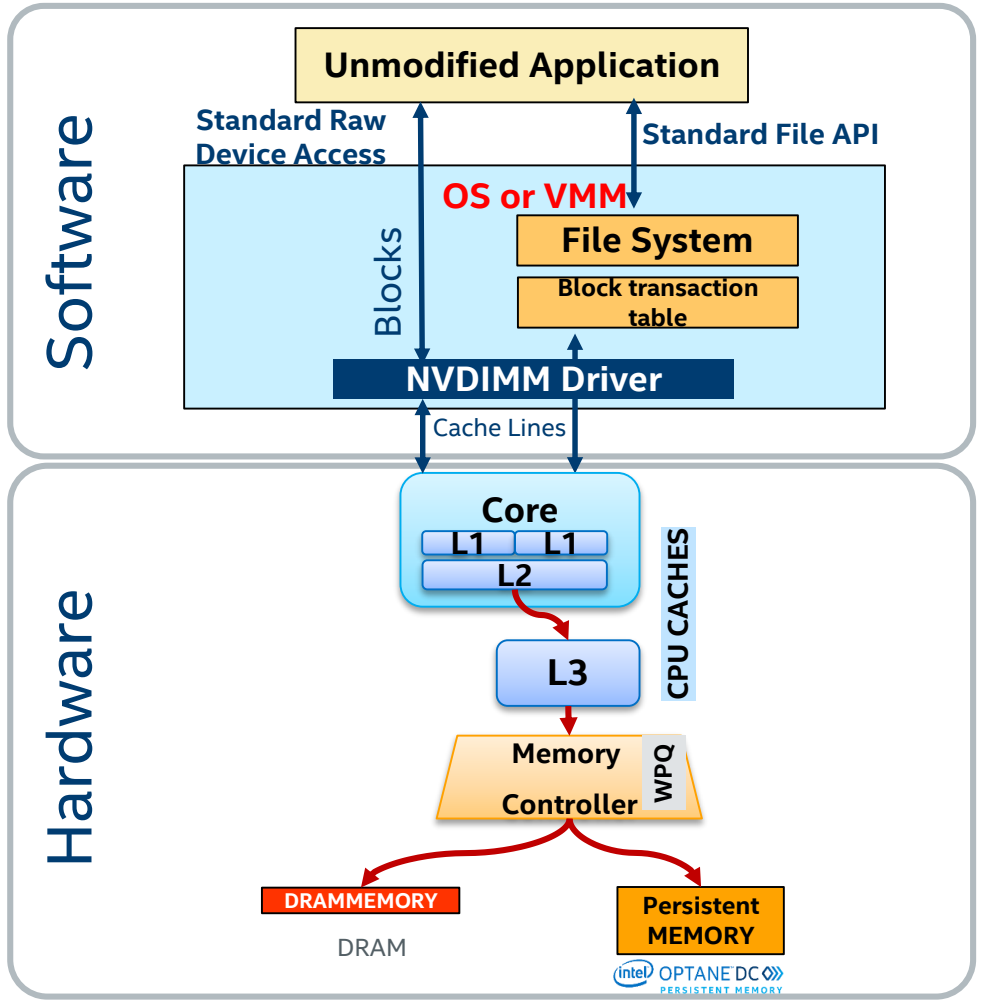


PERSISTENT MEMORY USAGE MODES

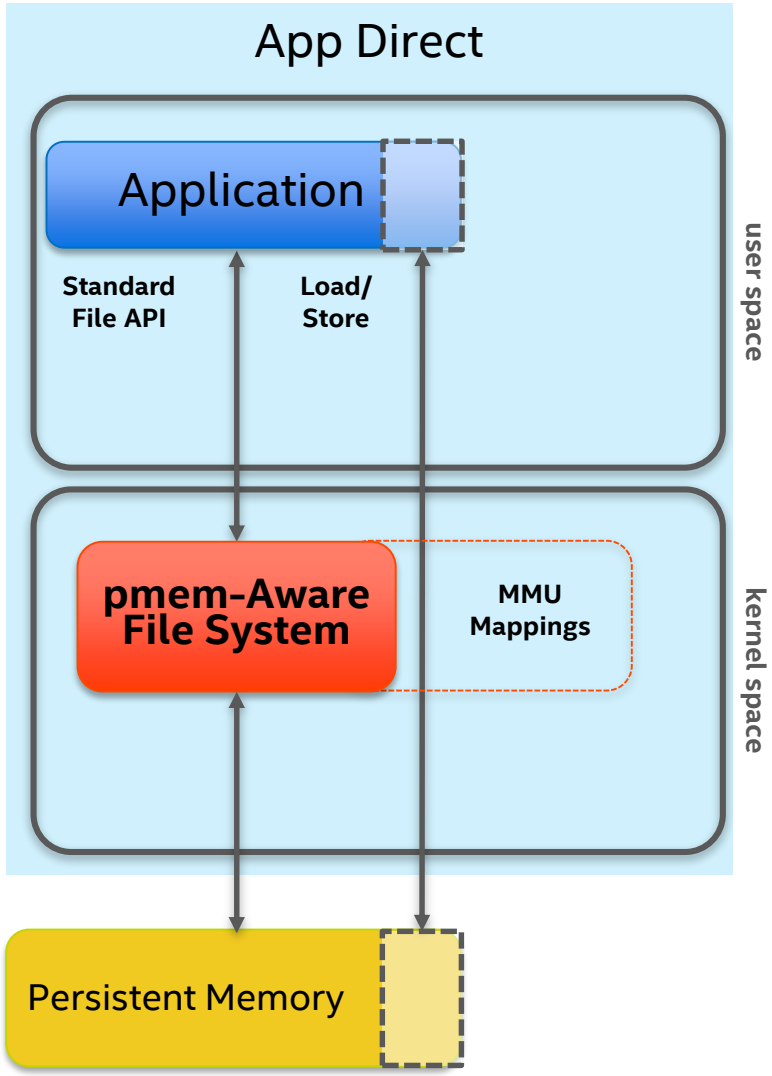
Storage Over App Direct

- Operates in blocks like SSD/HDD
  - Traditional read/write instructions
  - Works with existing file systems
  - Atomicity at block level
  - Block size configurable (4K, 512B)
- NVDIMM driver required
  - Support starting kernel 4.2
- Scalable capacity
- Higher endurance than enterprise class SSDs
- High performance block storage
  - Low latency, higher bandwidth, high IOPs

Linux kernel and driver changes: https://www.youtube.com/watch?v=owmN_lcMK2M



SNIA NVM PROGRAMMING MODEL

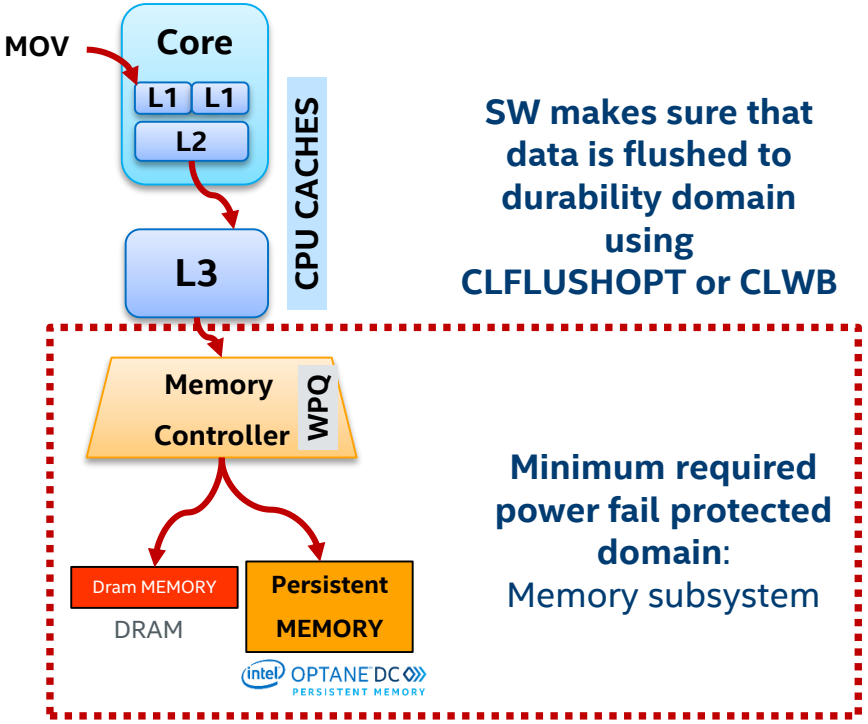



```
fd = open("/my/file", O_RDWR);
…
base = mmap(NULL, filesize,
            PROT_READ|PROT_WRITE,
            MAP_SHARED_VALIDATE|MAP_SYNC, fd, 0);
close(fd);
…
base[100] = 'X';
strcpy(base, "hello there");
msync(…);
…
```

PERSISTENT MEMORY USAGE MODES

App Direct Mode details

- PMEM-aware software/application required
  - Adds a new tier between DRAM and block storage (SSD/HDD)
  - Industry open standard programming model and Intel PMDK
- In-place persistence
  - No paging, context switching, interrupts, nor kernel code executes
- Byte addressable like memory
  - Load/store access, no page caching
- Cache Coherent
- Ability to do DMA & RDMA

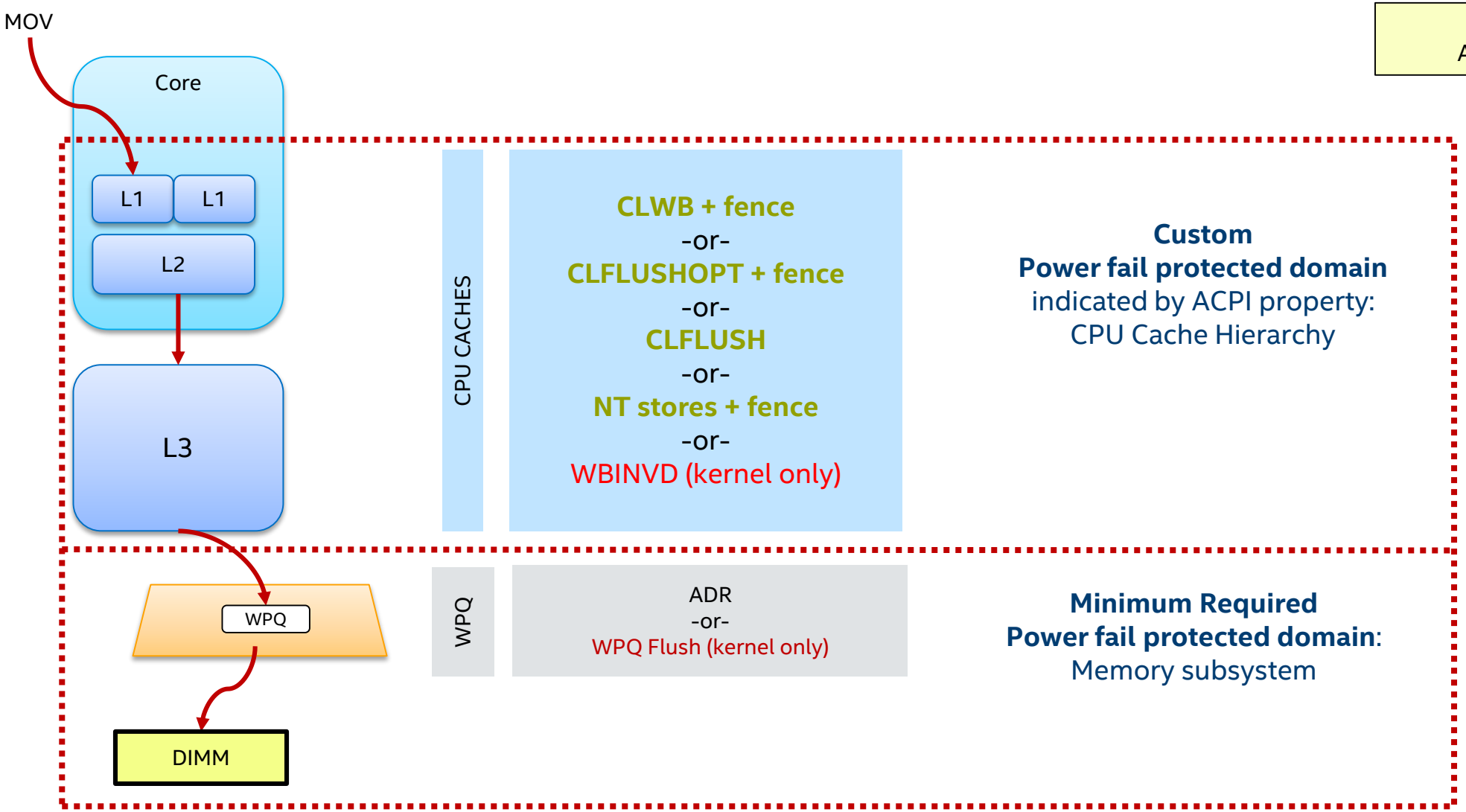


PERSISTENT MEMORY USAGE MODES

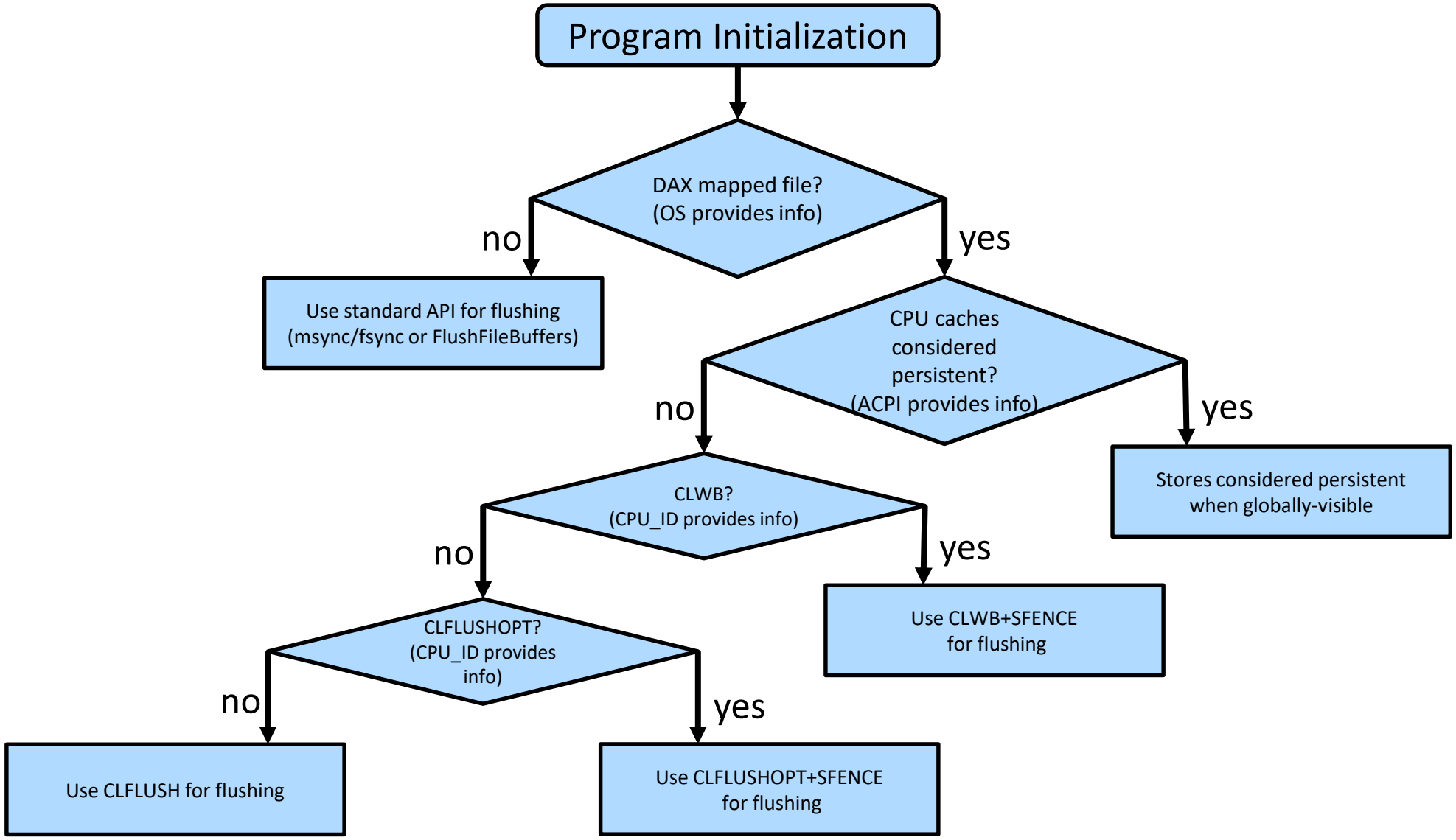
Summary

Volatile (use pmem for its capacity)		Persistent (leverage the fact pmem is persistent)	
Unmodified Apps	Modified Apps	Unmodified Apps	Modified Apps
Lowest impact Transparent for Apps	Low impact App decides on data placement	Lowest impact Apps use Storage API	Highest impact pmem-resident data structures
Memory Mode	*App Direct*	*App direct*	*App Direct*

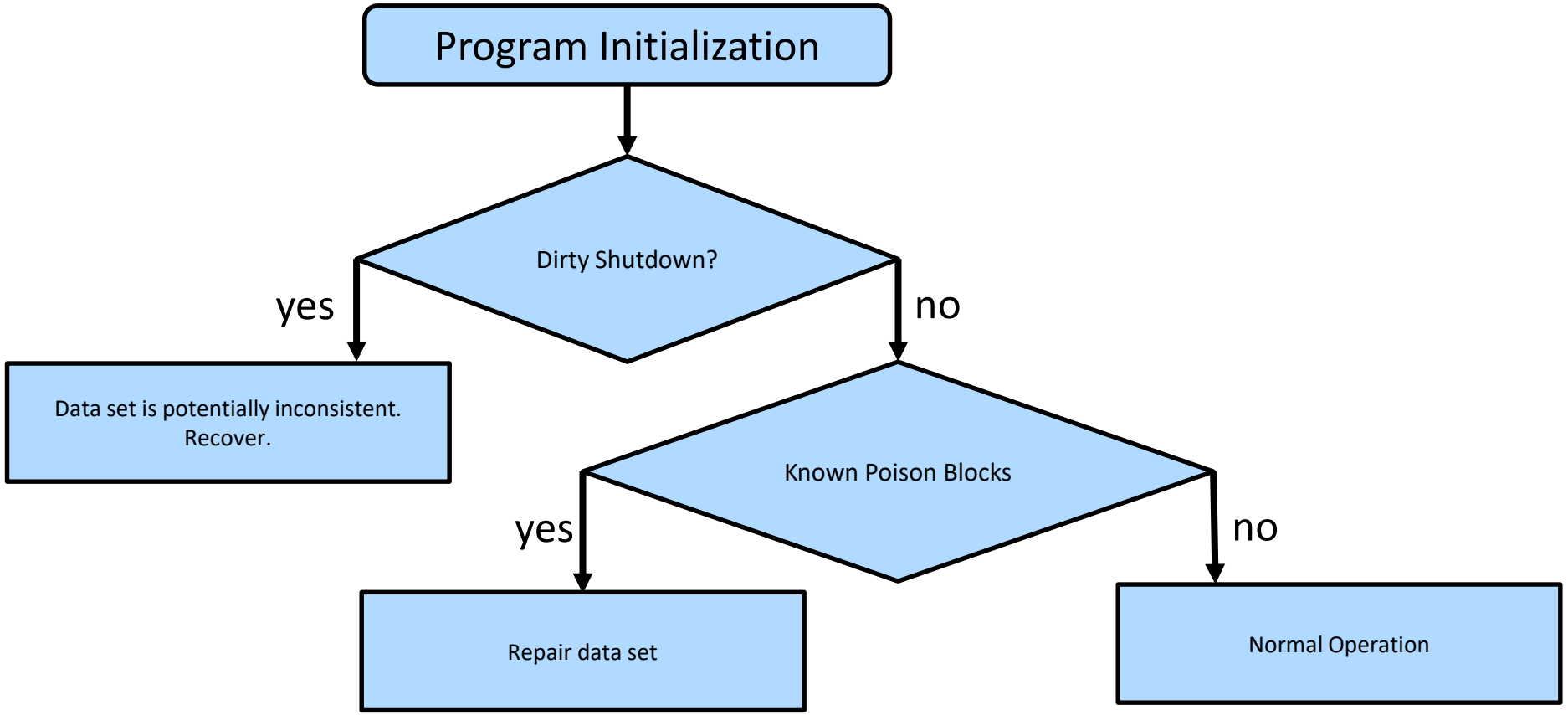
HOW THE HARDWARE WORKS



APPLICATION RESPONSIBILITIES: FLUSHING



APPLICATION RESPONSIBILITIES: RECOVERY



APPLICATION RESPONSIBILITIES: CONSISTENCY

```
open(…);

mmap(…);

strcpy(pmem, "Hello, World!");

msync(…);
```

APPLICATION RESPONSIBILITIES: CONSISTENCY

```
open(…);

mmap(…);

strcpy(pmem, "Hello, World!");

pmem_persist(pmem, 14);
```

Crash

Result

1. "\0\0\0\0\0\0\0\0\0\0..."
2. "Hello, W\0\0\0\0\0\0..."
3. "\0\0\0\0\0\0\0\0orld!\0"
4. "Hello, \0\0\0\0\0\0\0\0"
5. "Hello, World!\0"

APPLICATION RESPONSIBILITIES: CONSISTENCY

```
open(…);

mmap(…);

strcpy(pmem, "Hello, World!");

pmem_persist(pmem, 14);
```

Crash

`pmem_persist()` may be faster,
but is still **not** transactional

Result

1. `"\0\0\0\0\0\0\0\0\0\0..."`
2. `"Hello, W\0\0\0\0\0\0..."`
3. `"\0\0\0\0\0\0\0\0orld!\0"`
4. `"Hello, \0\0\0\0\0\0\0\0"`
5. `"Hello, World!\0"`

VISIBILITY VERSUS POWER FAIL ATOMICITY

Feature	Atomicity
Atomic Store	8 byte power-fail 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

Agenda

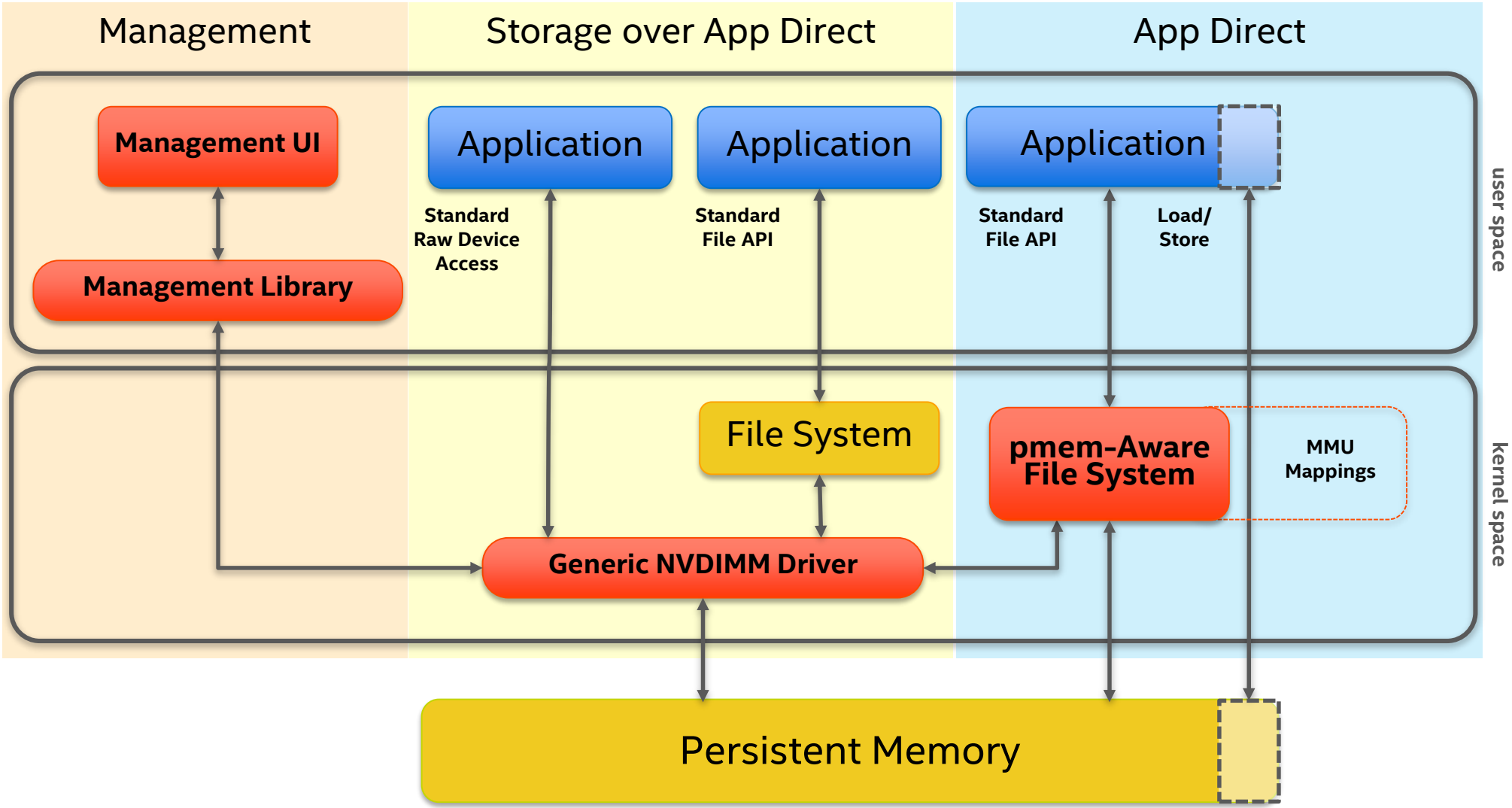
Persistent Memory Overview

Persistent Memory Programming

Persistent Memory Development Libraries

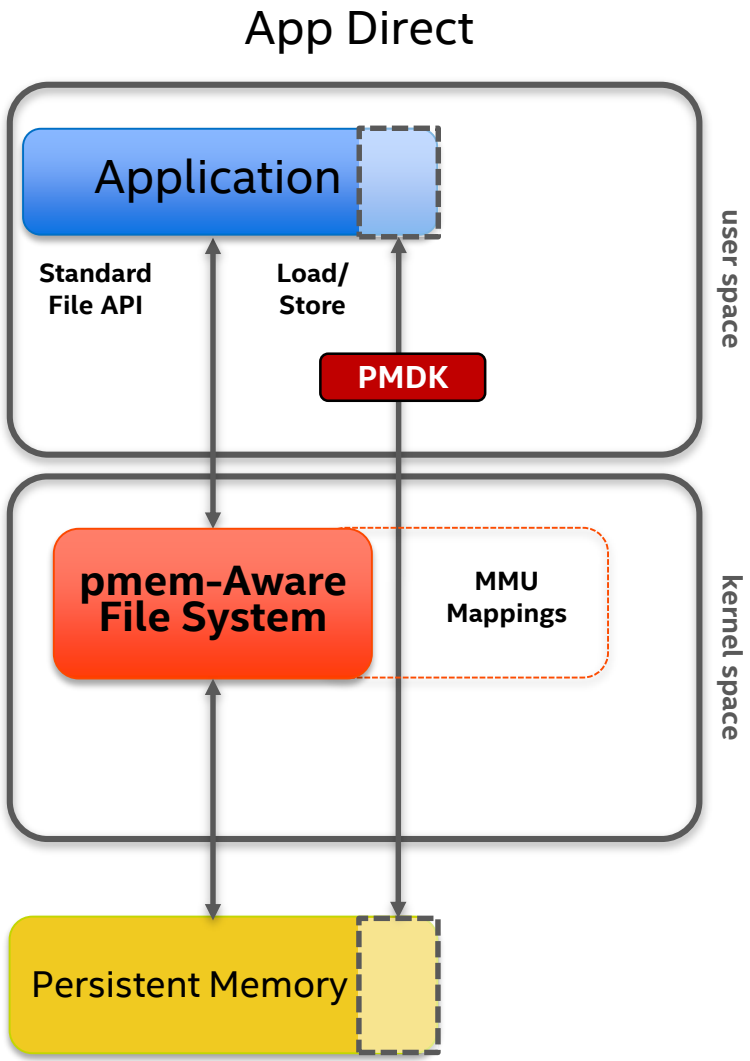
- PMDK Overview
- Introduction to individual library
- Tools

SNIA NVM Programming Model



PMDK overview

- http://pmem.io/
- open-source https://github.com/pmem
- vendor-agnostic
- user-space
- production quality, fully documented
- performance optimized and tuned



PMDK overview

High-level support

- Volatile key-value store optimized for PMEM — **libvmemchache**
- Local/embedded key-value store optimized for PMEM — **pmemkv**

Language bindings

C | C++ | LLPL | PCJ | Python

Transaction support

- Interface to create a persistent memory resident log file, e.g. Write Ahead Logging (WAL) — **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**

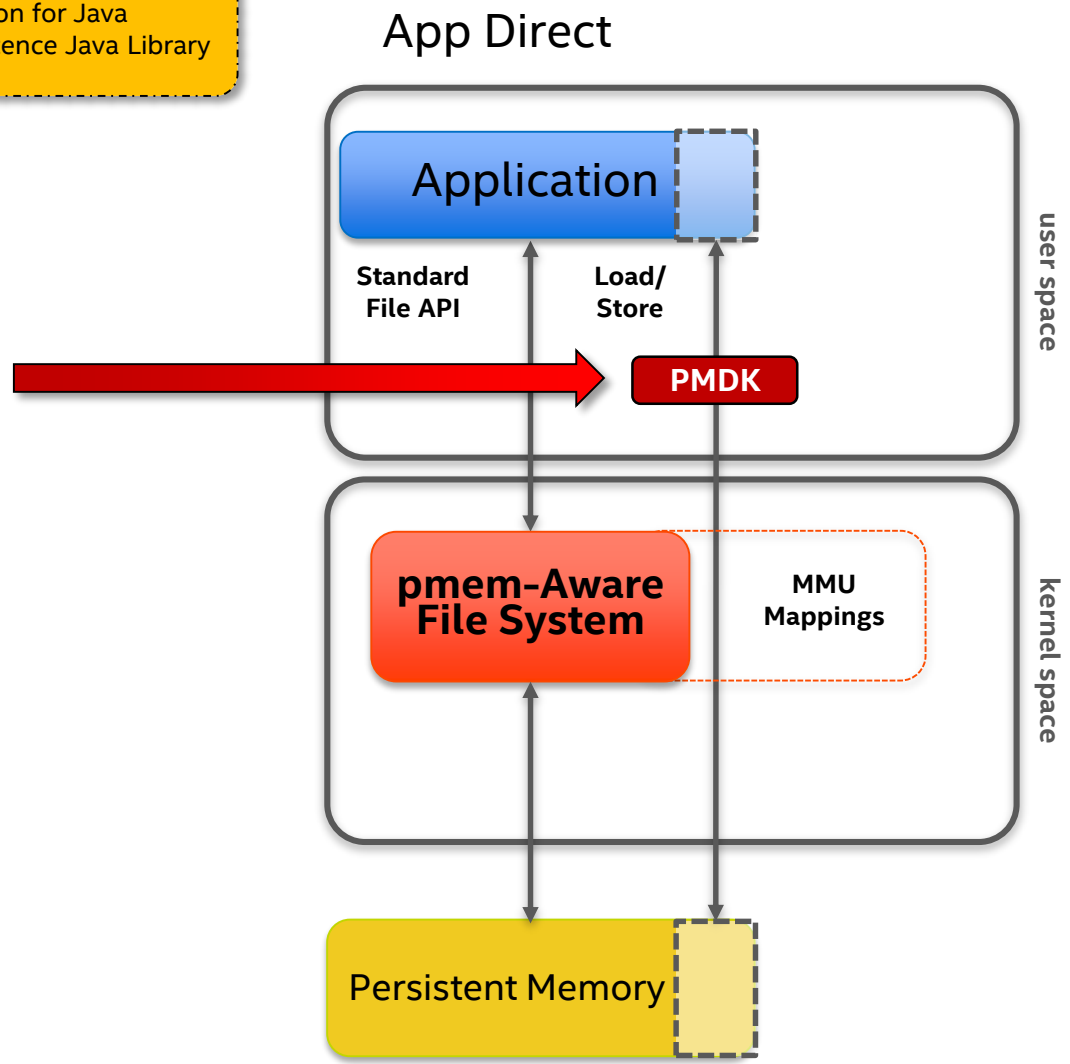
Volatile memory support

- Support for allocations from DRAM and PMEM — **libmemkind**

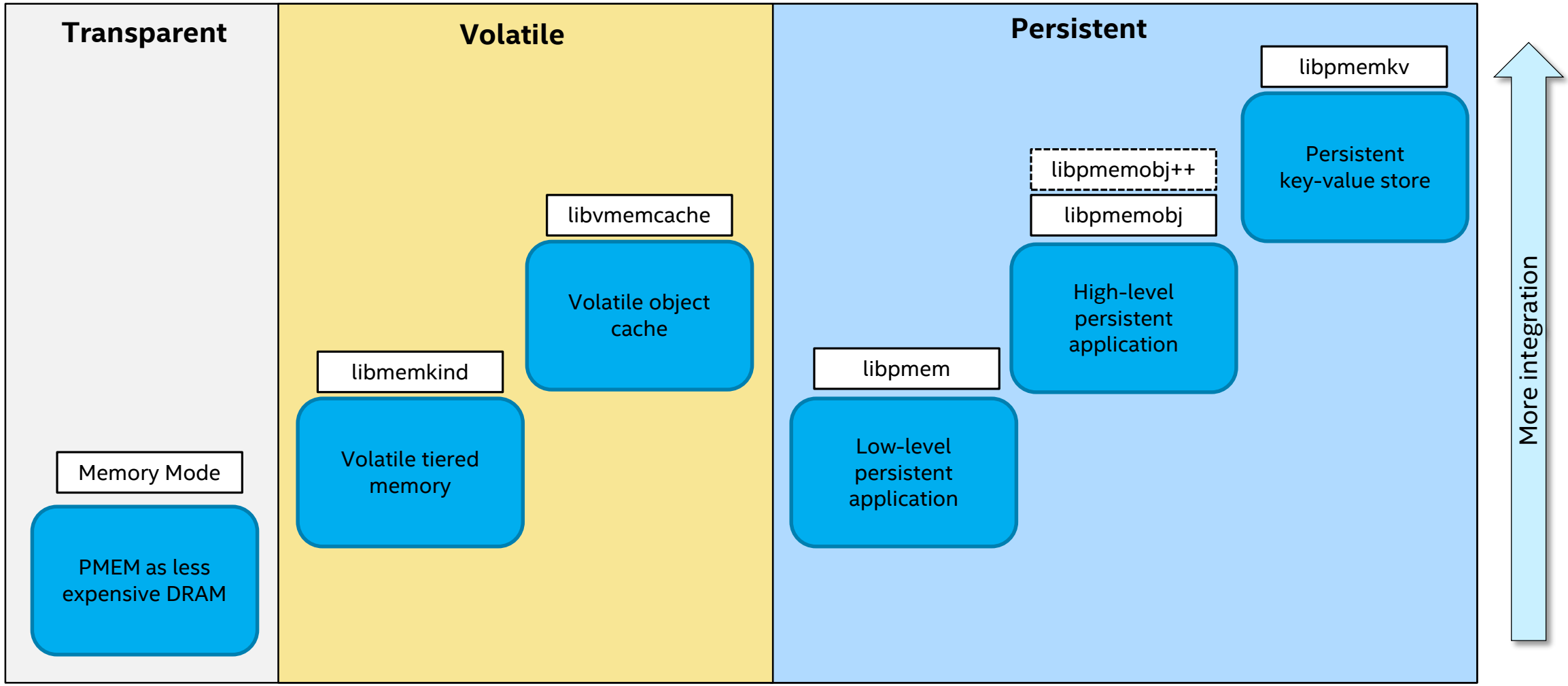
Low-level support

- Low level support for local persistent memory — **libpmem**
- Low level support for remote access to persistent memory — **librpmem**

In Development:
PCJ – Persistent Collection for Java
LLPL – Low-Level Persistence Java Library
Python bindings



Different ways to use persistent memory



Memory Mode

Use when:
- modifying applications is not feasible
- massive amounts of memory is required (more TB)
- CPU utilization is low in shared environment (more VMs)

- Not really a part of PMDK…
- … but it's the easiest way to take advantage of Persistent Memory

```
char *memory = malloc(sizeof(struct my_object));
strcpy(memory, "Hello World");
```

- Memory is automatically placed in PMEM, with caching in DRAM

libmemkind

Use when:
- application can be modified
- different tiers of objects (hot, warm) can be identified
- persistence is not required

- Explicitly manage allocations from App Direct, allowing for fine-grained control of DRAM/PMEM

```
struct memkind *pmem_kind = NULL;
size_t max_size = 1 << 30; /* gigabyte */

/* Create PMEM partition with specific size */
memkind_create_pmem(PMEM_DIR, max_size, &pmem_kind);

/* allocate 512 bytes from 1 GB available */
char *pmem_string = (char *)memkind_malloc(pmem_kind, 512);

memkind_free(pmem_kind, pmem_string);  /* deallocate the pmem object */

memkind_destroy_kind(pmem_kind); /* destroy PMEM partition */
```

- The application can decide what type of memory to use for objects

libvmemcache

Use when:
- caching large quantities of data
- low latency of operations is needed
- persistence is not required

- Seamless and easy-to-use LRU caching solution for persistent memory
  Keys reside in DRAM, values reside in PMEM

```
VMEMcache *cache = vmemcache_new("/tmp", VMEMCACHE_MIN_POOL,
VMEMCACHE_MIN_EXTENT, VMEMCACHE_REPLACEMENT_LRU);

const char *key = "foo";
vmemcache_put(cache, key, strlen(key), "bar", sizeof("bar"));

char buf[128];
ssize_t len = vmemcache_get(cache, key, strlen(key),
    buf, sizeof(buf), 0, NULL);

vmemcache_delete(cache);
```

- Designed for easy integration with existing systems

libpmem

Use when:
- modifying application that already uses memory mapped I/O
- other libraries are too high-level
- only need low-level PMEM-optimized primitives (memcpy etc)

- Low-level library that provides basic primitives needed for persistent memory programming and optimized memcpy/memmove/memset

```
void *pmemaddr = pmem_map_file("/mnt/pmem/data", BUF_LEN,
            PMEM_FILE_CREATE|PMEM_FILE_EXCL,
            0666, &mapped_len, &is_pmem));
const char *data = "foo";
if (is_pmem) {
    pmem_memcpy_persist(pmemaddr, data, strlen(data));
} else {
    memcpy(pmemaddr, data, strlen(data));
    pmem_msync(pmemaddr, strlen(data));
}
close(srcfd);
pmem_unmap(pmemaddr, mapped_len);
```

- The very basics needed for PMEM programming

libpmemobj

Use when:
- direct byte-level access to objects is needed
- using custom storage-layer algorithms
- persistence is required

- Transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming.

```
typedef struct foo {
    PMEMoid bar; // persistent pointer
    int value;
} foo;

int main() {
    PMEMobjpool *pop = pmemobj_open (...);
    TX_BEGIN(pop) {
        TOID(foo) root = POBJ_ROOT(foo);
        D_RW(root)->value = 5;
    } TX_END;
}
```

- Flexible and relatively easy way to leverage PMEM

libpmemkv

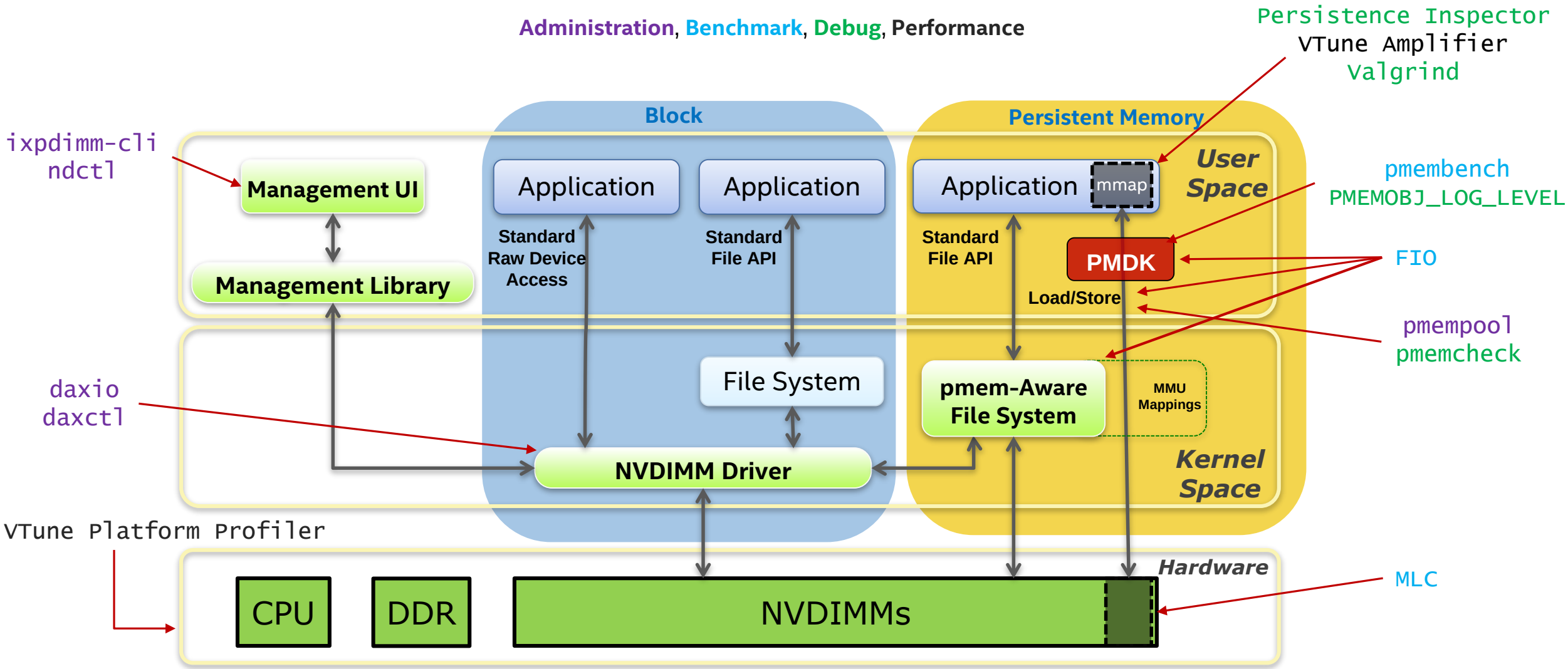
Use when:
- storing large quantities of data
- low latency of operations is needed
- persistence is required

- Local/embedded key-value datastore optimized for persistent memory. Provides different language bindings and storage engines.

```
const pmemkv = require('pmemkv');

const kv = new KVEngine('vsmap', '{"path":"/dev/shm/"}');

kv.put('key1', 'value1');
assert(kv.count === 1);
assert(kv.get('key1') === 'value1');

kv.all((k) => console.log(`  visited: ${k}`));

kv.remove('key1');
kv.stop();
```

- High-level storage layer optimized for PMEM

Programming Model Tools



Summary

PMDK is a comprehensive collection of solutions

- Developers pull only what they need
  - Low level programming support
  - Transaction APIs
- Fully validated
- Performance tuned.

Open Source & Product neutral

More developer resources

Find the PMDK (Persistent Memory Development Kit) at http://pmem.io/pmdk/

Getting Started

- Intel IDZ persistent memory- https://software.intel.com/en-us/persistent-memory
- Entry into overall architecture - http://pmem.io/2014/08/27/crawl-walk-run.html
- Emulate persistent memory - http://pmem.io/2016/02/22/pm-emulation.html

Linux Resources

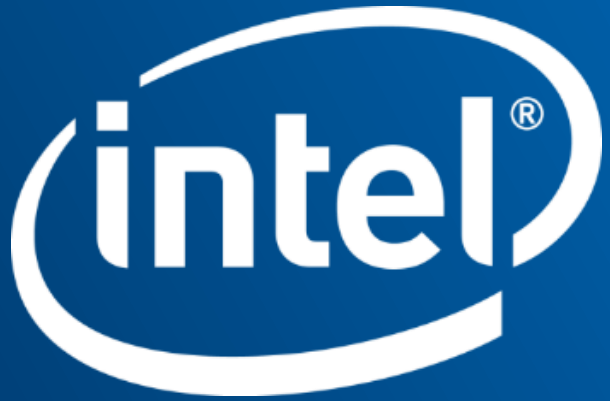
- Linux Community Pmem Wiki - https://nvdimm.wiki.kernel.org/
- Pmem enabling in SUSE Linux Enterprise 12 SP2 - https://www.suse.com/communities/blog/nvdimm-enabling-suse-linux-enterprise-12-service-pack-2/

Windows Resources

- Using Byte-Addressable Storage in Windows Server 2016 -https://channel9.msdn.com/Events/Build/2016/P470
- Accelerating SQL Server 2016 using Pmem - https://channel9.msdn.com/Shows/Data-Exposed/SQL-Server-2016-and-Windows-Server-2016-SCM--FAST

Other Resources

- SNIA Persistent Memory Summit 2018 - https://www.snia.org/pm-summit
- Intel manageability tools for Pmem - https://01.org/ixpdimm-sw/


intel