INTRODUCTION TO PROGRAMMING FOR PERSISTENT MEMORY

Speaker: Szymon Romik (Intel Data Center Group)

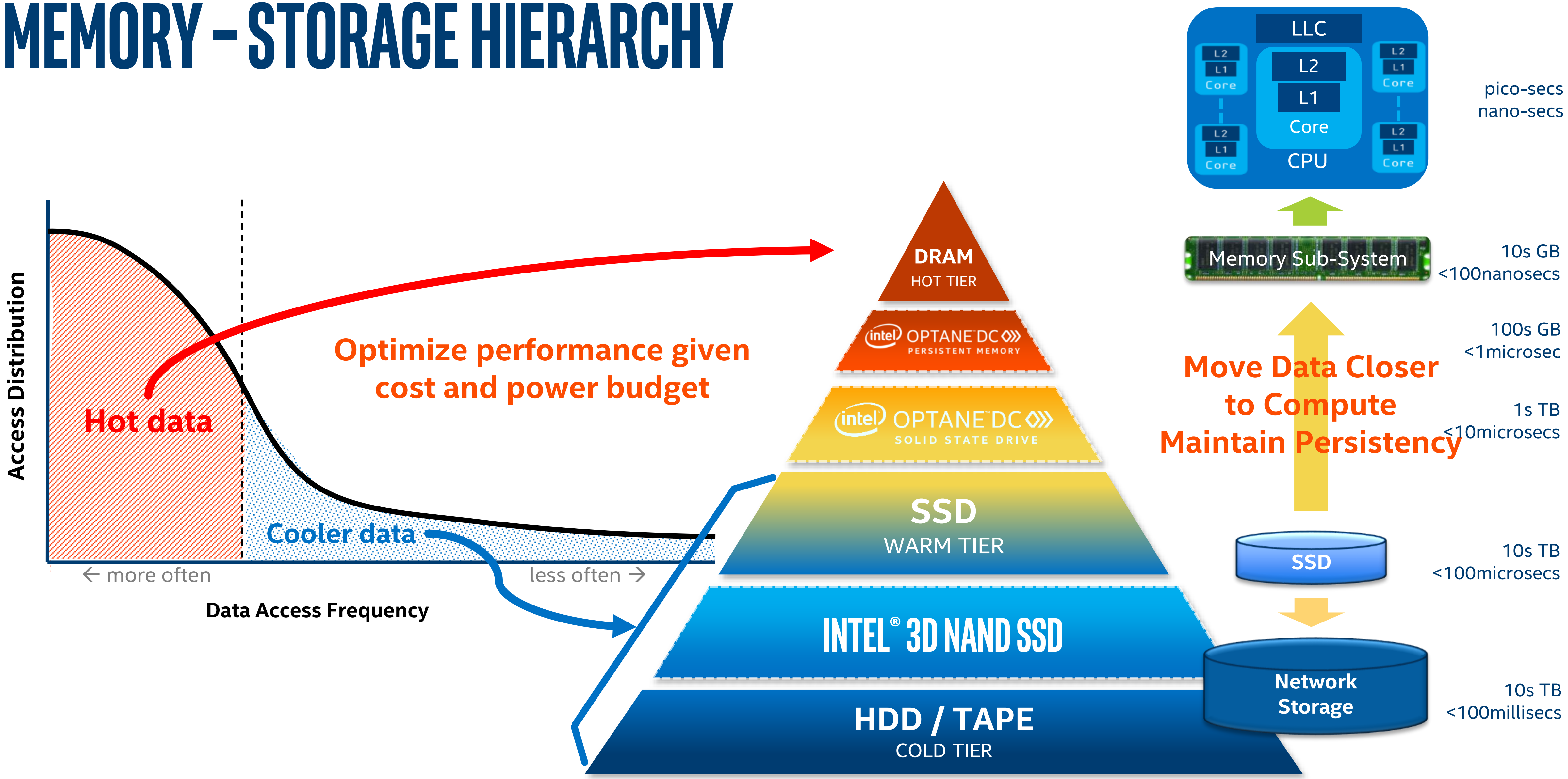
<szymon.romik@intel.com>

May, 2019

AGENDA

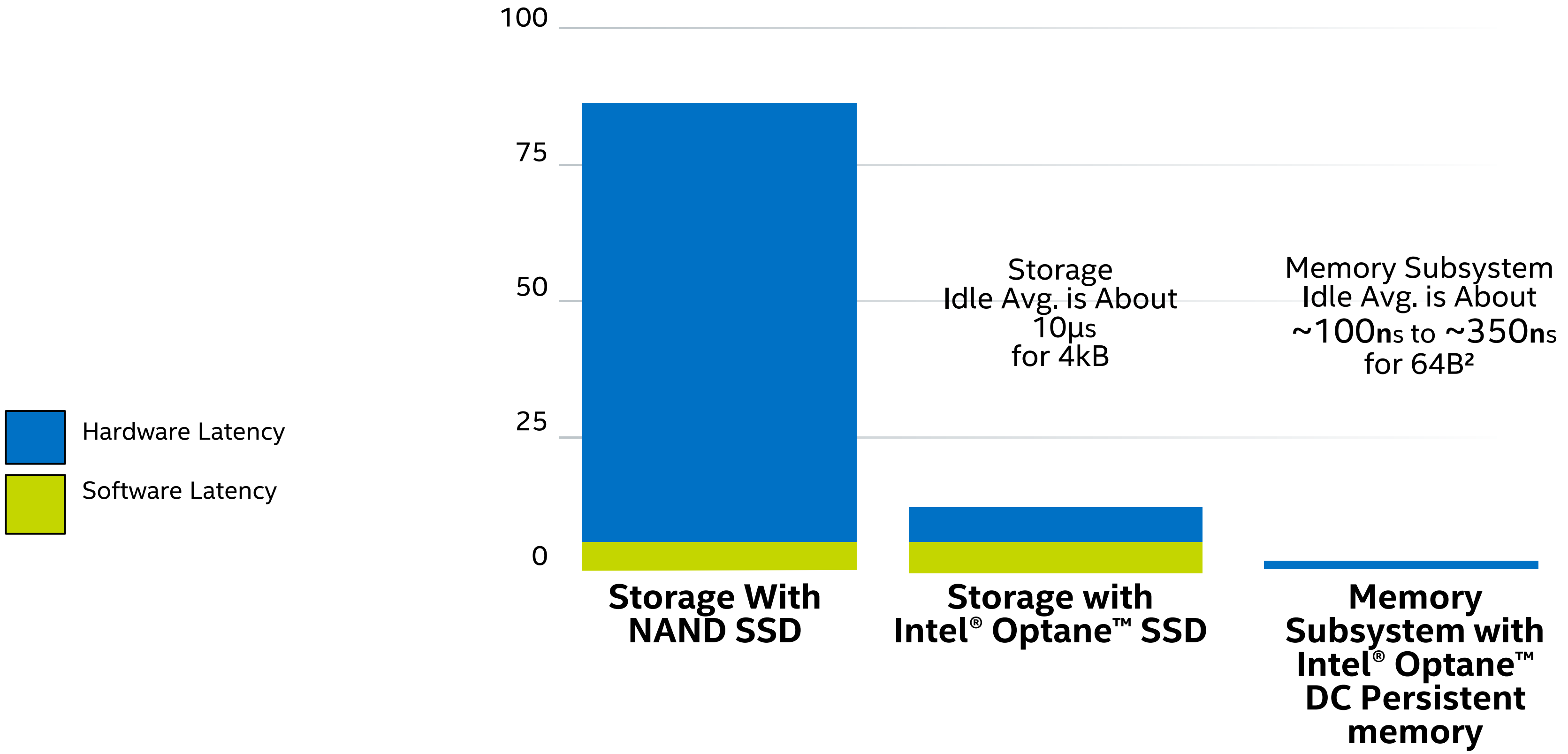
- Memory - Storage hierarchy
- What is Persistent Memory?
- Persistent Memory usage modes
- SNIA NVM Programming Model
- Application responsibilities:
  - Understanding power-failure atomicity
  - Persistence domain
  - Visibility versus Power Fail Atomicity

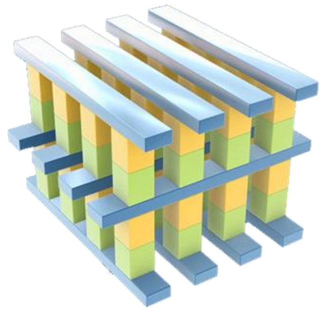
MEMORY – STORAGE HIERARCHY



MEMORY – STORAGE HIERARCHY

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)

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-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

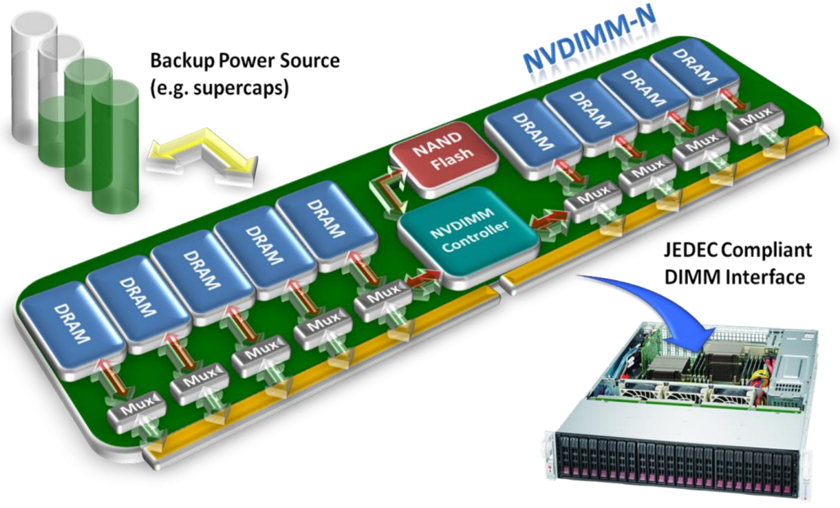
WHAT IS PERSISTENT MEMORY?

- 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?

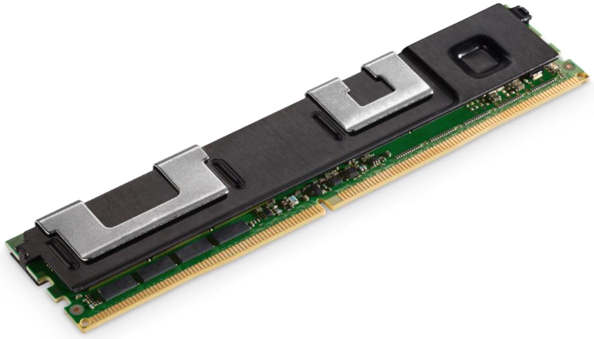
- byte-addressable
- load/store memory access
- persistence properties of storage



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?

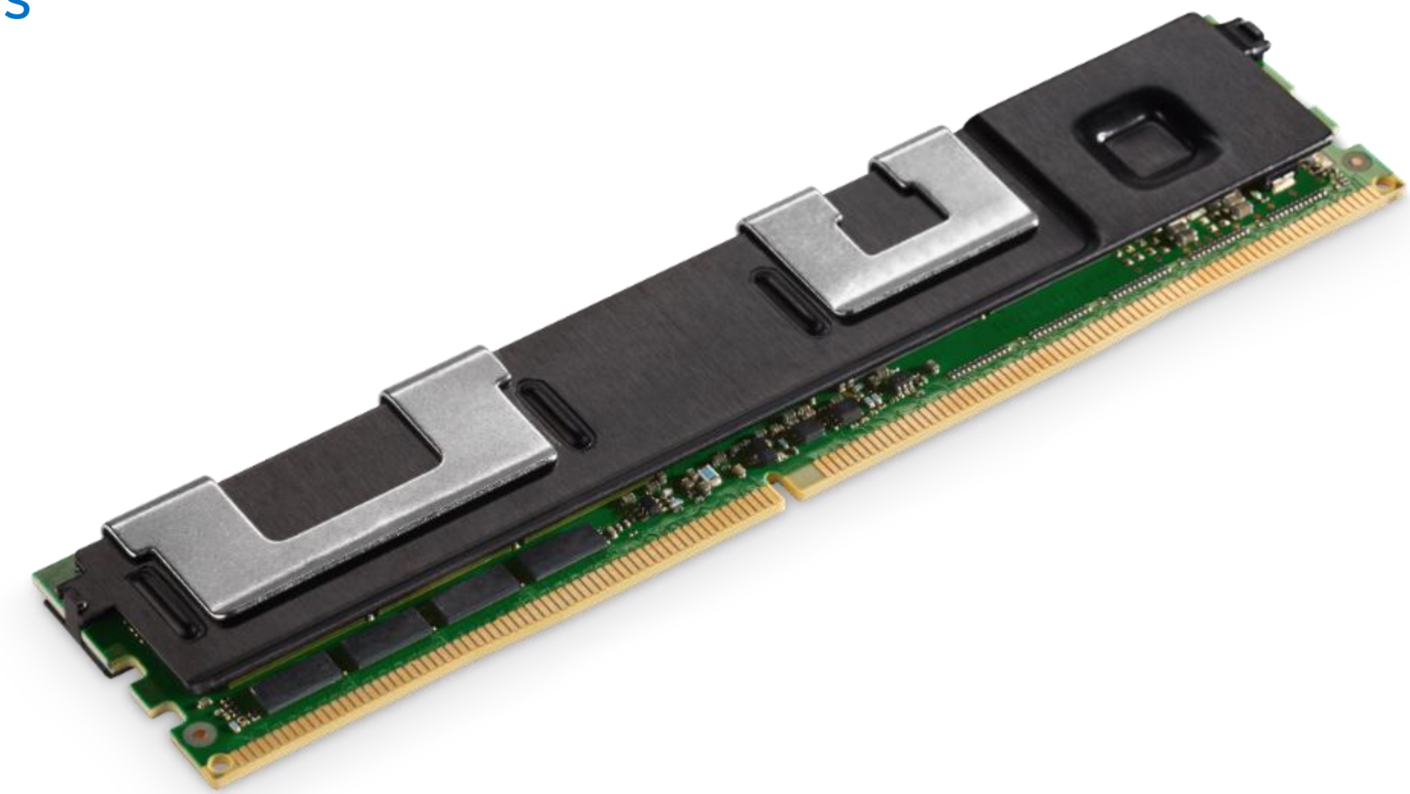
- byte-addressable
- load/store memory access
- persistence properties of storage



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

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

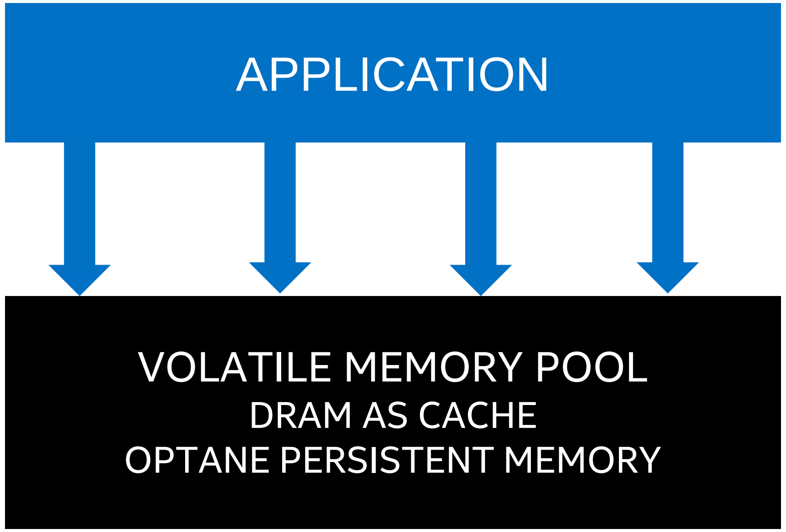
intel OPTANE DC
PERSISTENT MEMORY



PERSISTENT MEMORY USAGE MODES

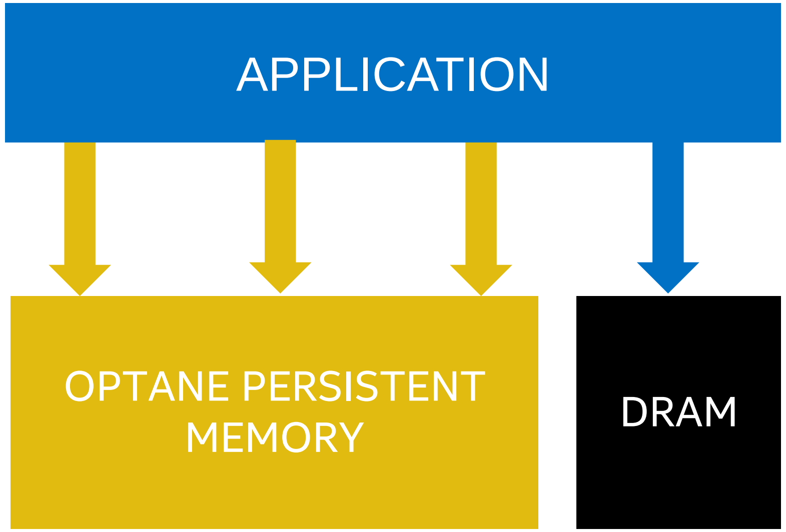
Memory Mode

Affordable memory capacity for many applications



APP Direct Mode

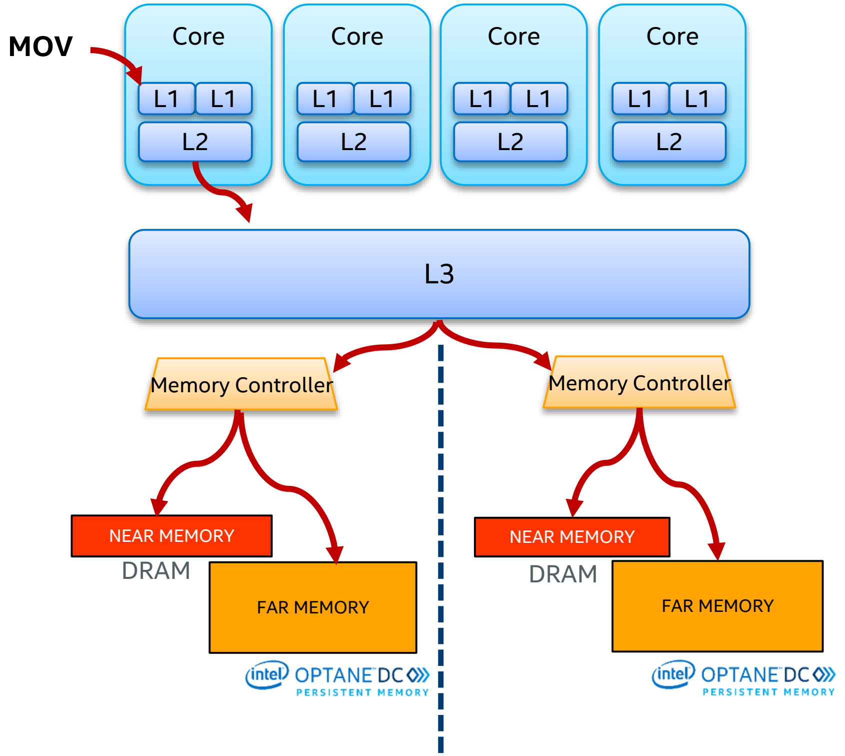
Persistent & maximum capacity



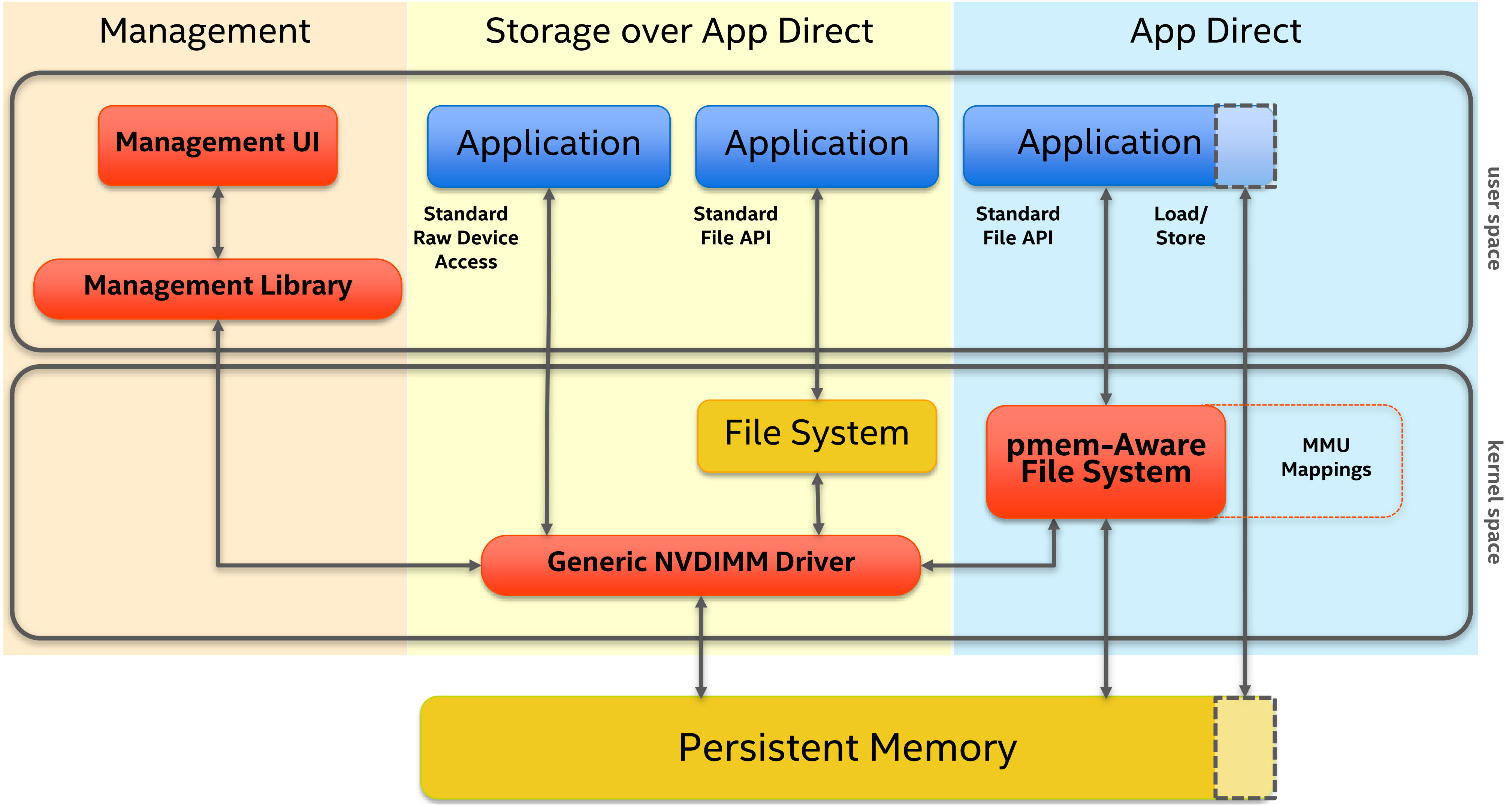
PERSISTENT MEMORY USAGE MODES

Memory Mode details

- No software/application changes required
- To mimic traditional memory, data is “volatile”
  - Volatile mode key cleared and regenerated every power cycle
- DRAM is “near memory”
- Used as a write-back cache
- Managed by host memory controller
- Within the same host memory controller, not across
- Ratio of far/near memory (PMEM/DRAM) can vary
- Overall latency
- Same as DRAM for cache hit
- Intel® Optane™ DC persistent memory + DRAM for cache miss



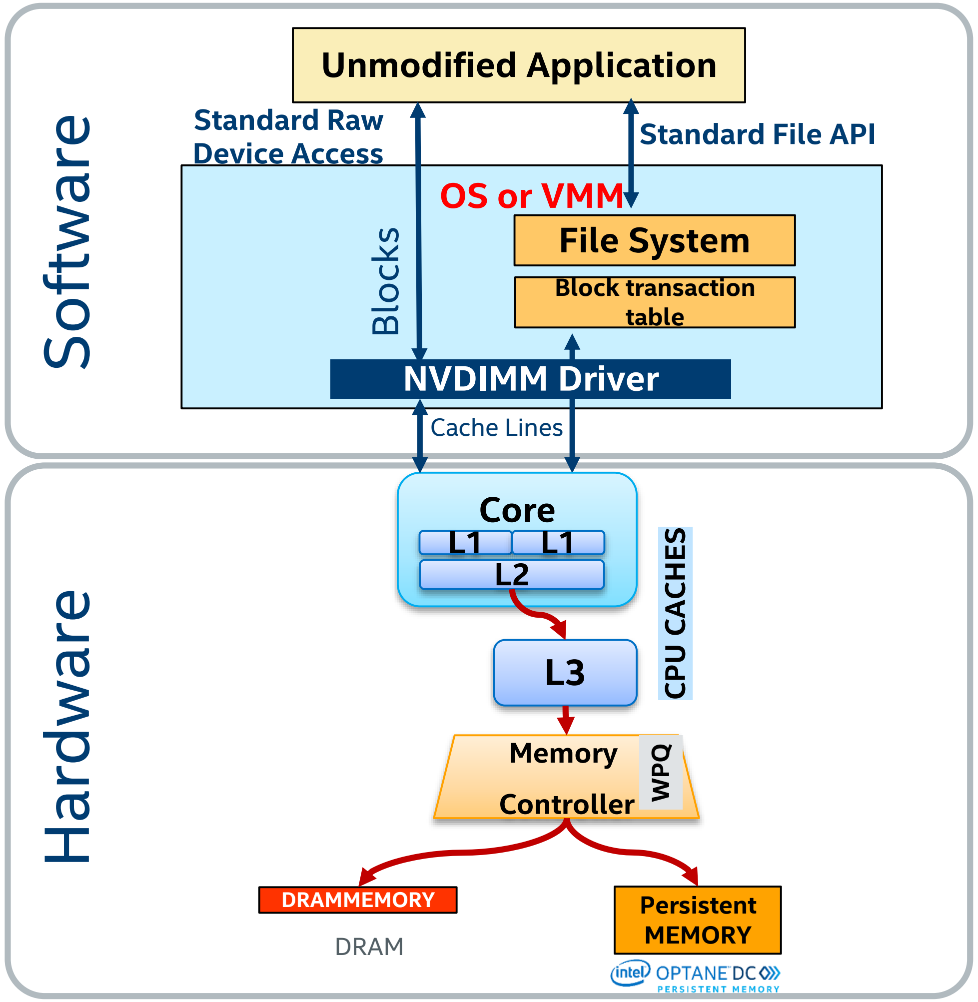
SNIA NVM PROGRAMMING MODEL



PERSISTENT MEMORY USAGE MODES

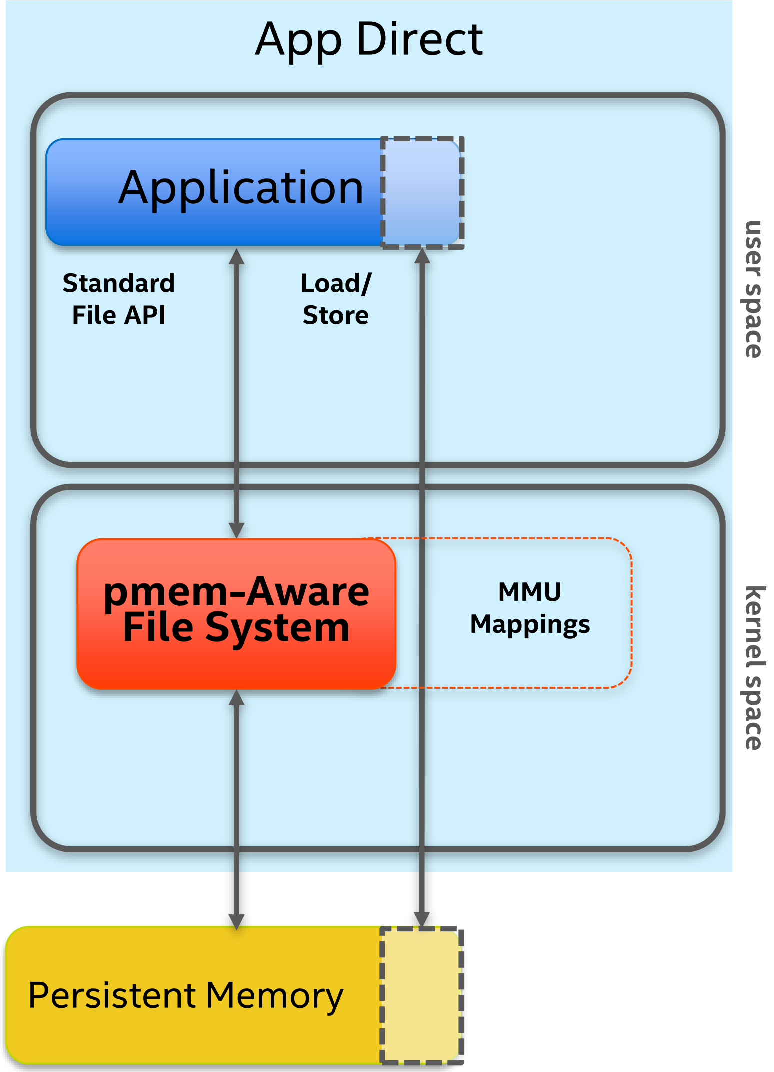
Storage Over App Direct

- 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

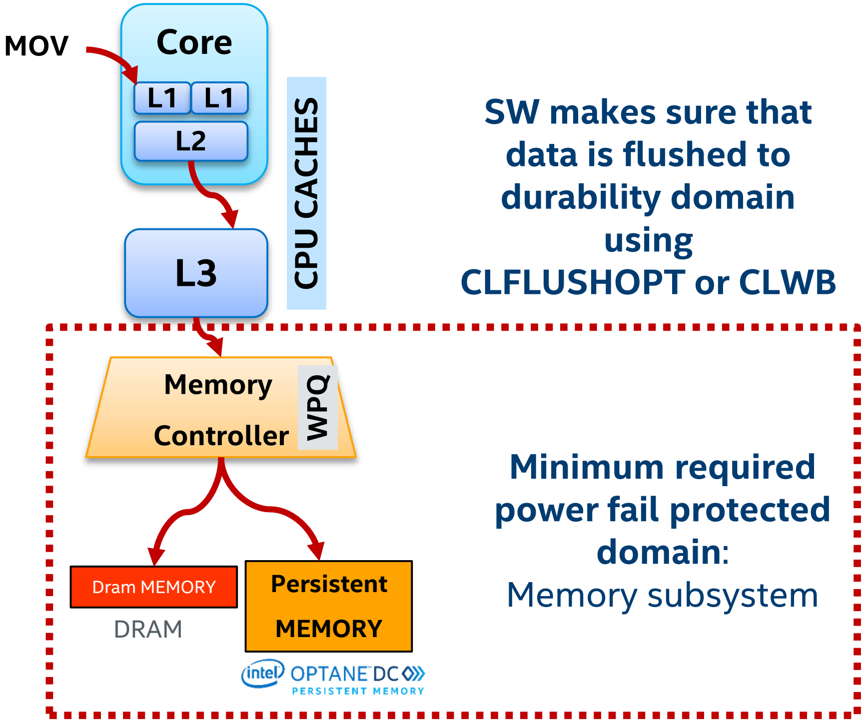



```
fd = open("/my/file", O_RDWR);
…
base = mmap(NULL, filesize,
         PROT_READ|PROT_WRITE,
         MAP_SHARED_VALIDATE|MAP_SYNC, fd, 0);
close(fd);
…
base[100] = 'X';
strcpy(base, "hello there");
*structp = *base_structp;
…
```

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

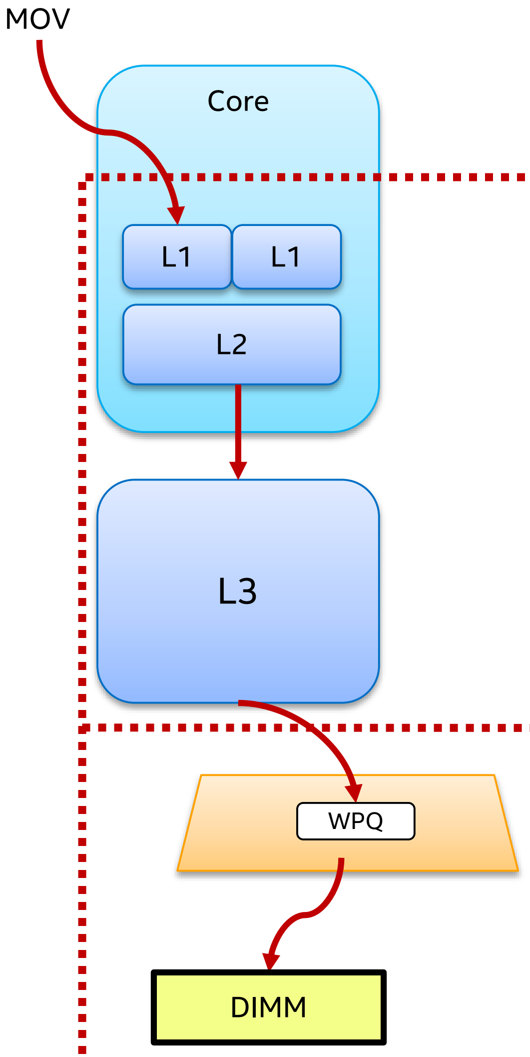


PERSISTENT MEMORY USAGE MODES

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



Not shown:
ADR Failure Detection

CPU CACHES

CLWB + fence
-or-
CLFLUSHOPT + fence
-or-
CLFLUSH
-or-
NT stores + fence
-or-
WBINVD (kernel only)

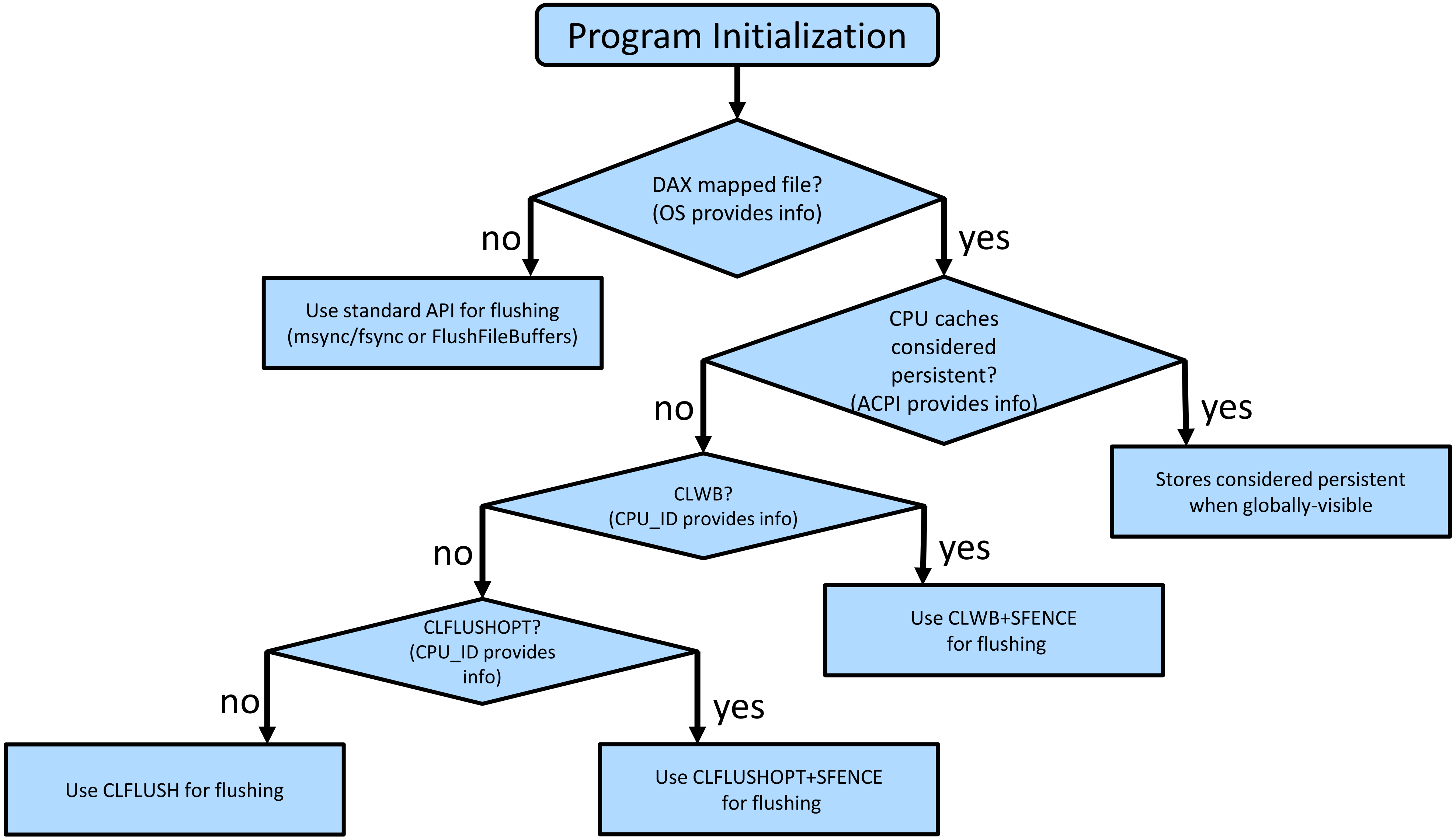
Custom
Power fail protected domain
indicated by ACPI property:
CPU Cache Hierarchy

WPQ

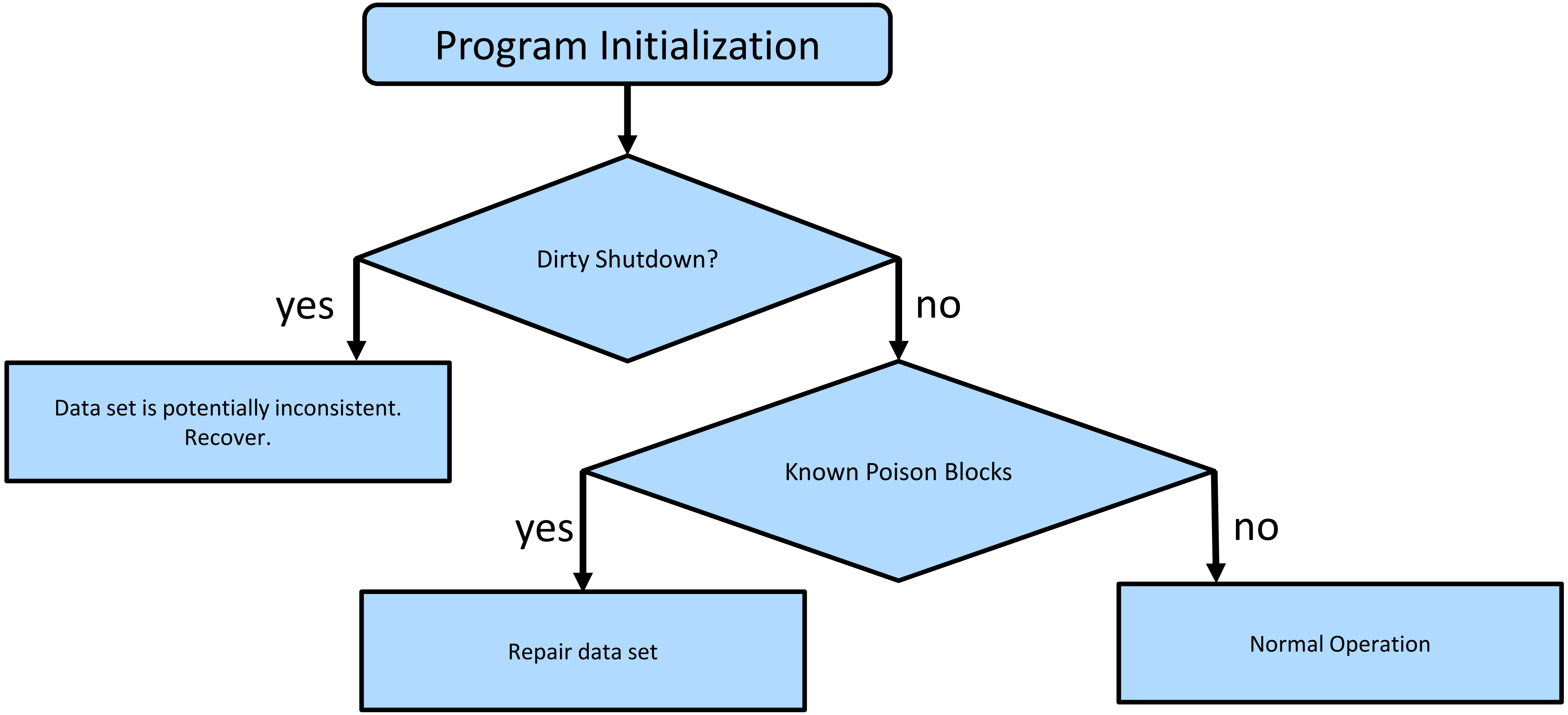
ADR
-or-
WPQ Flush (kernel only)

Minimum Required
Power fail protected domain:
Memory subsystem

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

PMEM reference counter – BAD example

```
struct my_object {
    uint64_t refcount;
    type some_resource;
};
                                                   No decision based on this value in this thread...

static void object_ref(struct my_object *object) {  /*  refcount  visible = 0 persistent = 0 */
    __sync_fetch_and_add(&object->refcount, 1); /*         visible = 1 persistent = ? */
    persist(&object->refcount, sizeof(object->refcount)); /*  visible = 1 persistent = 1 */
}

                                                   Decision is made based on visible but not persistent value

static void object_deref(struct my_object *object) {  /*  visible = 1 persistent = 1 */
    if (__sync_sub_and_fetch(&object->refcount, 1) == 0) {/*  visible = 0 persistent = ? */
        delete_some_resource(object->some_resource);   /* visible = 0 persistent = ? */
    }
    persist(&object->refcount, sizeof(object->refcount)); /*  visible = 0 persistent = 0 */
}
```

PMEM reference counter – GOOD example

```
struct my_object {
    uint64_t refcount;
    type some_resource;
};
                                                                   No decision based on this value in this thread...
static void object_ref(struct my_object *object) {  /*  refcount  visible = 0 persistent = 0 */
    __sync_fetch_and_add(&object->refcount, 1); /*       visible = 1 persistent = ? */
    persist(&object->refcount, sizeof(object->refcount)); /*  visible = 1 persistent = 1 */
}

                                                                   Decision is based on a known persistent value

static void object_deref(struct my_object *object) {  /*  visible = 1 persistent = 1 */
  if (__sync_sub_and_fetch(&object->refcount, 1) == 0) {  /*  visible = 0 persistent = ? */
    persist(&object->refcount, sizeof(object->refcount)); /*  visible = 0 persistent = 0 */
    delete_some_resource(object->some_resource);  /*  visible = 0 persistent = 0 */
  }
}
```

Atomic variables need to be read and flushed before making any decisions/calculations with them to ensure that the action is taken on a value that is known to have been persistent at some point.

intel®