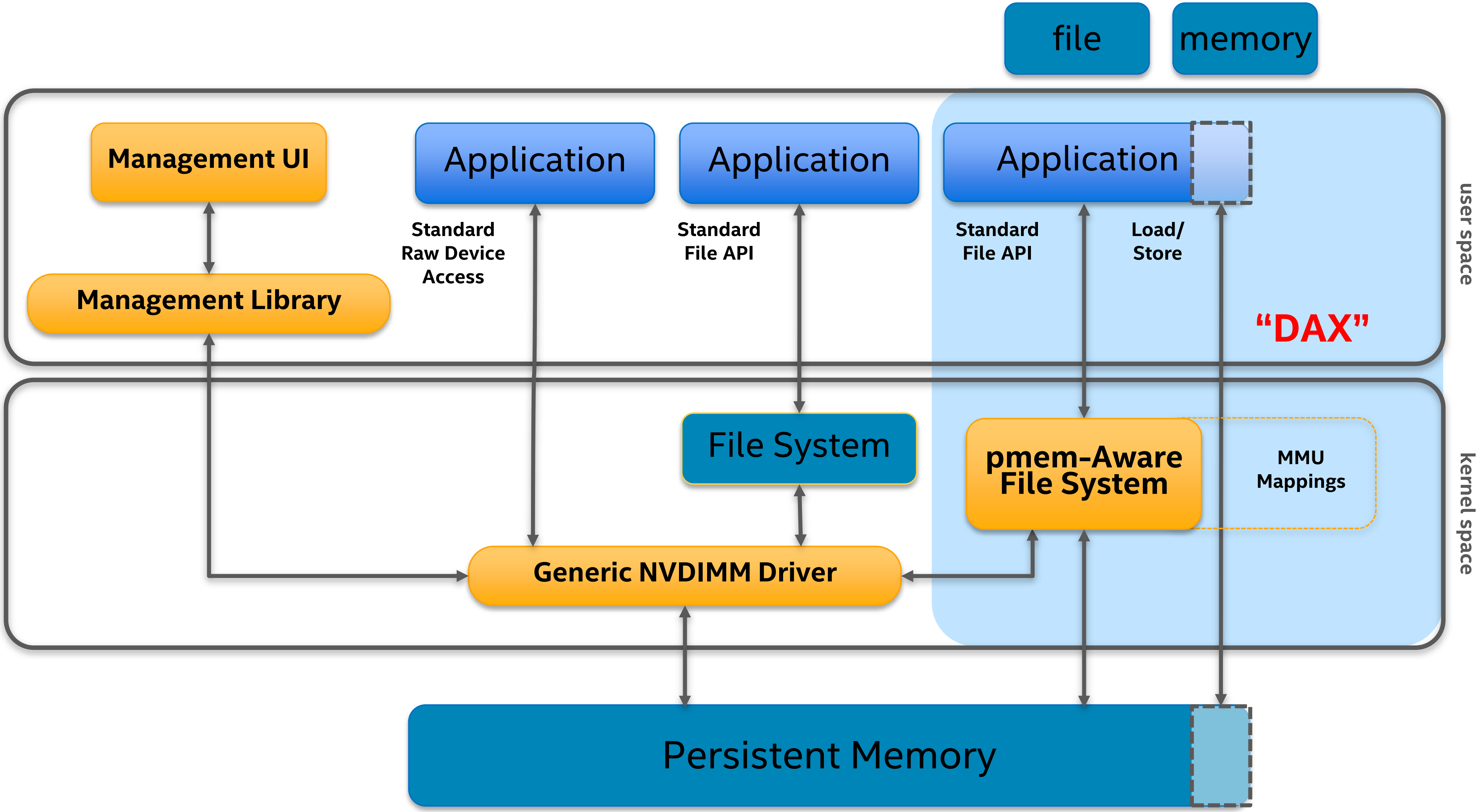
# PROGRAMMING FOR PERSISTENT MEMORY

Piotr Balcer
<piotr.balcer@intel.com>
Intel® Data Center Group

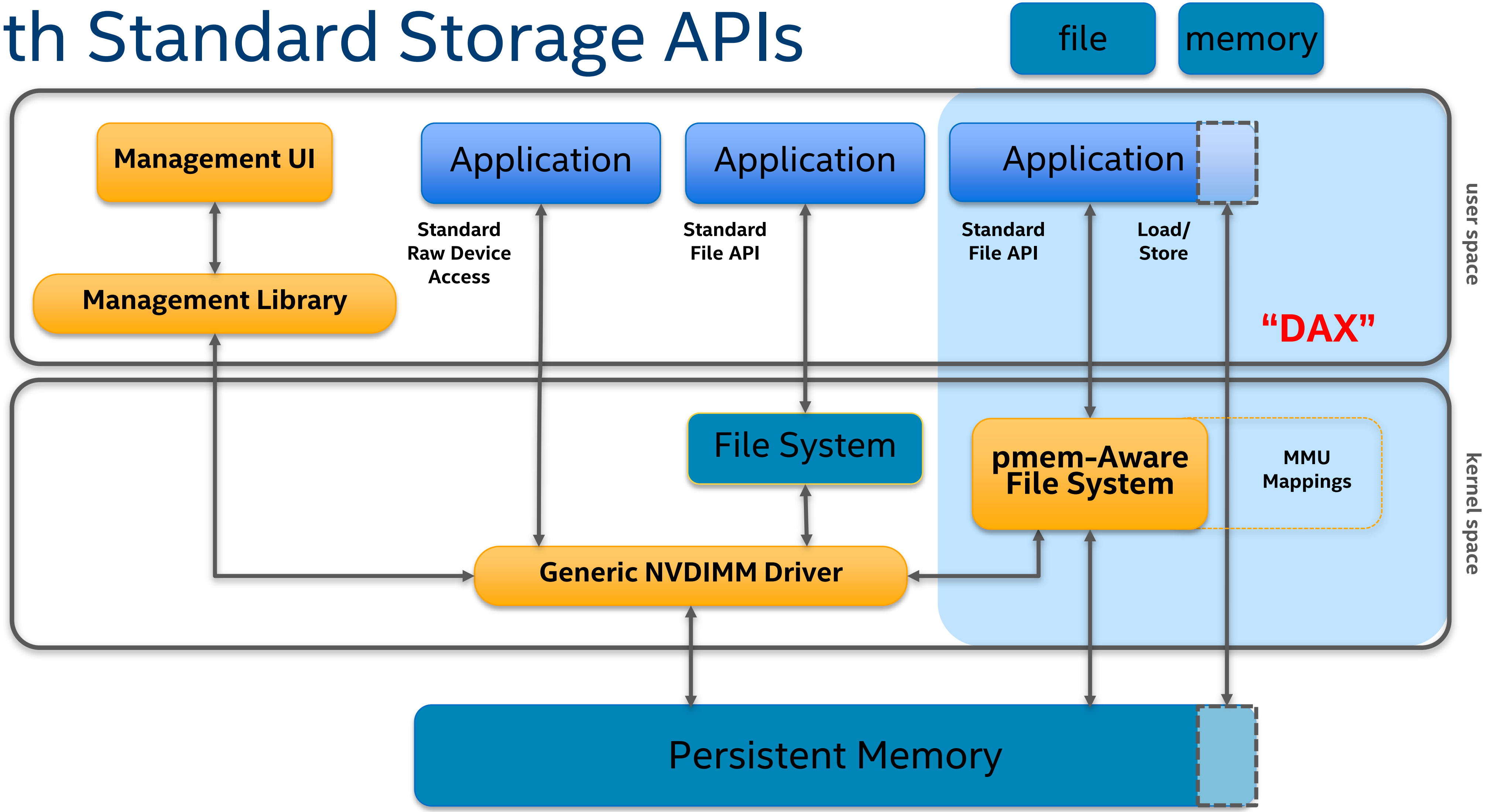
# Agenda

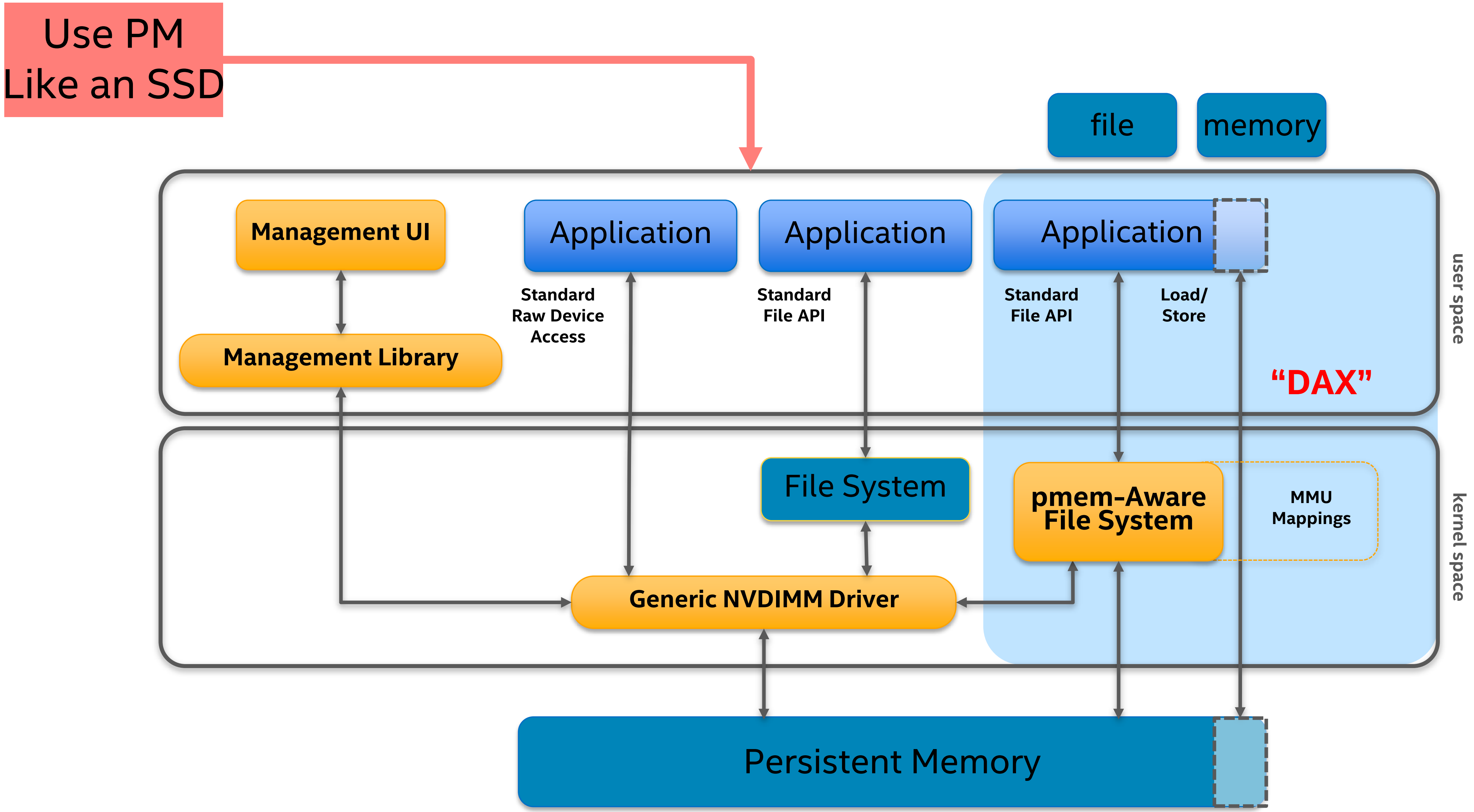
- SNIA NVM Programming Model
  - Block based I/O
  - Memory Mapped I/O
- Understanding power-failure atomicity
- Persistence domain
- Visibility versus Power Fail Atomicity

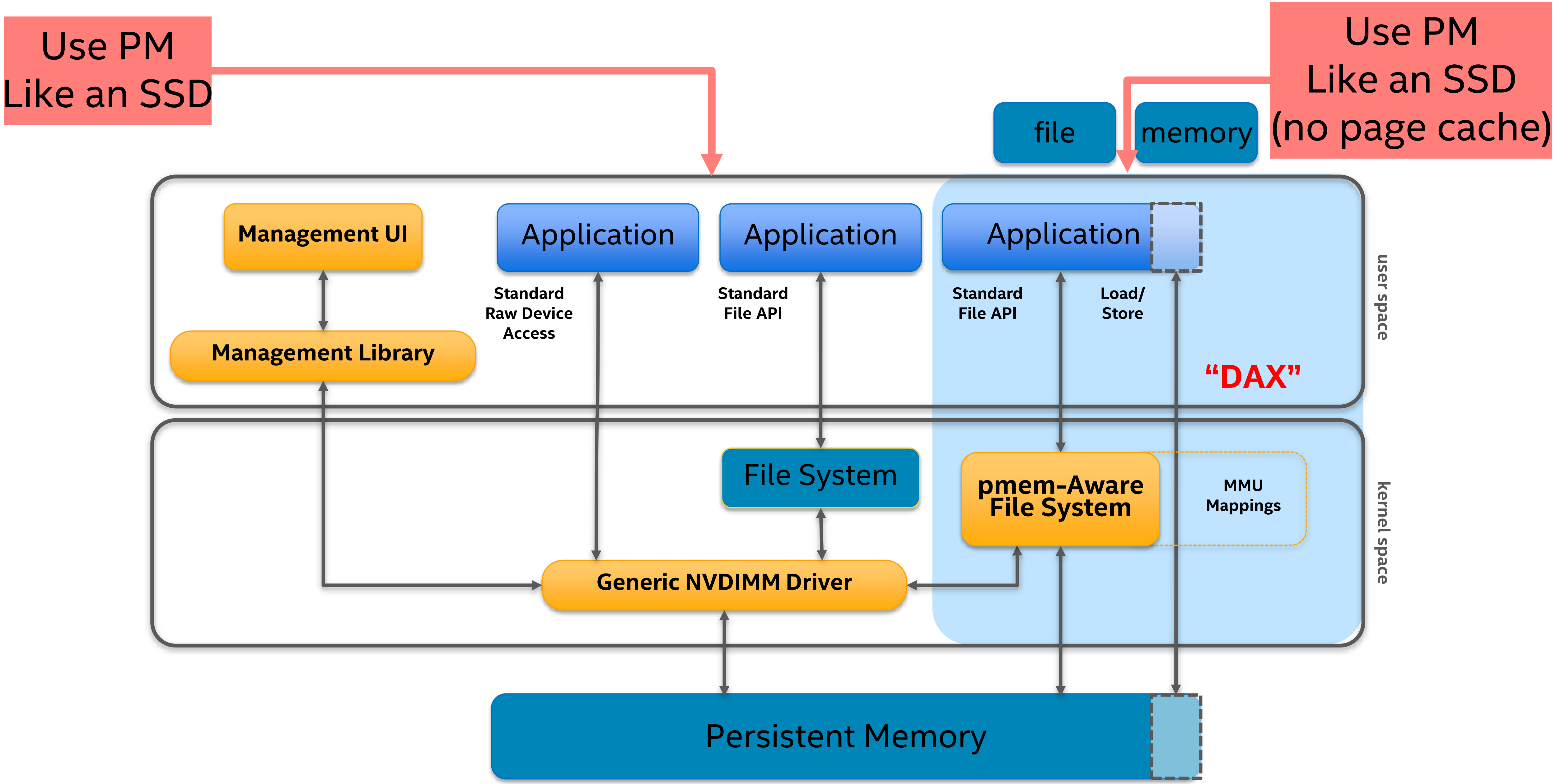
# The SNIA NVM Programming Model

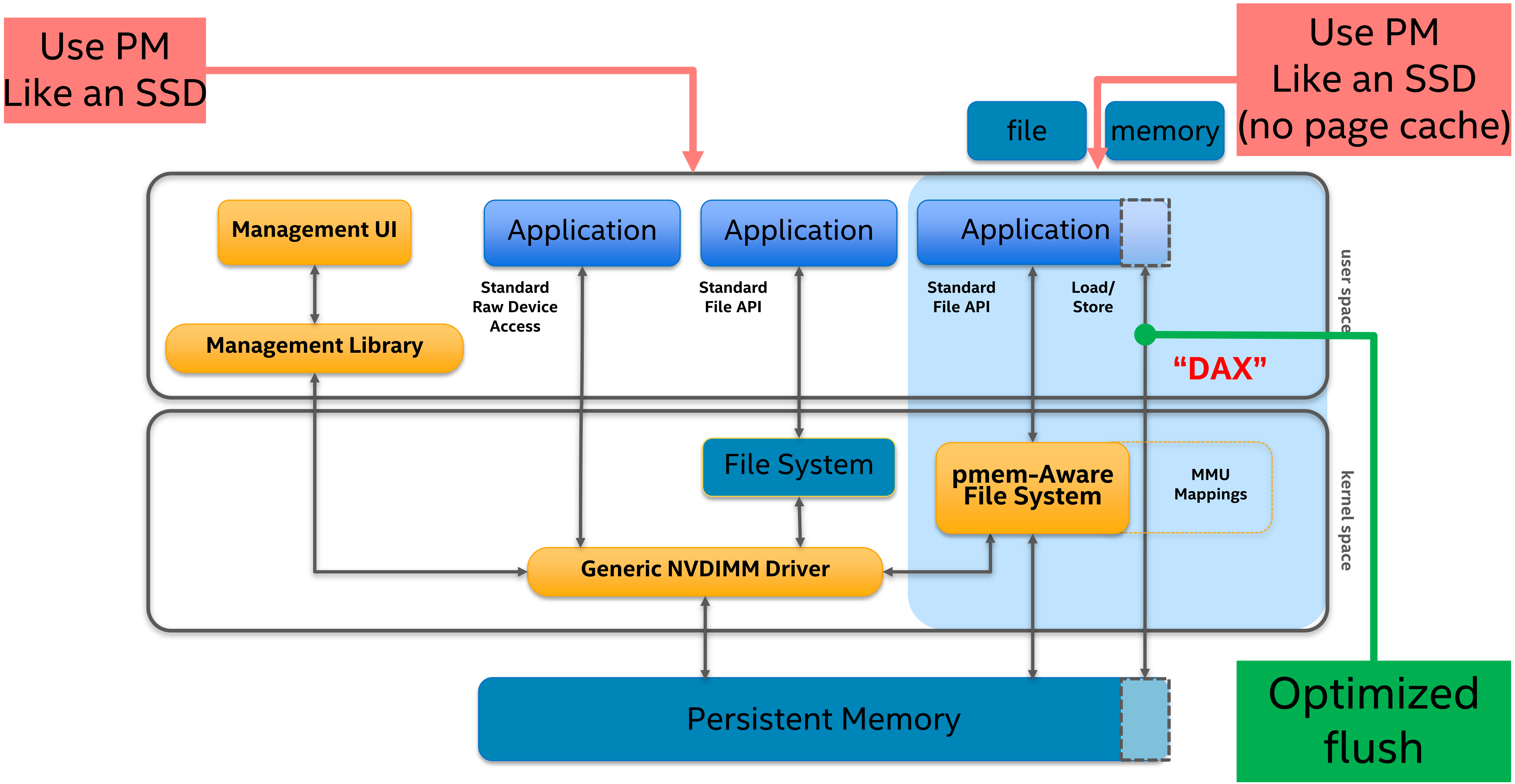


# Don't Forget: The NVM Programming Model Starts With Standard Storage APIs




Use PM
Like an SSD
file
memory
Management UI
Application
Application
Application
Standard
Raw Device
Access
Standard
File API
Standard
File API
Load/
Store
Management Library
"DAX"
user space
File System
pmem-Aware
File System
MMU
Mappings
kernel space
Generic NVDIMM Driver
Persistent Memory


Use PM
Like an SSD
Use PM
Like an SSD
(no page cache)
file
memory
Management UI
Application
Application
Application
Standard
Raw Device
Access
Standard
File API
Standard
File API
Load/
Store
Management Library
"DAX"
user space
File System
pmem-Aware
File System
MMU
Mappings
kernel space
Generic NVDIMM Driver
Persistent Memory


Use PM
Like an SSD
Use PM
Like an SSD
(no page cache)
file
memory
Management UI
Application
Application
Application
Standard
Raw Device
Access
Standard
File API
Standard
File API
Load/
Store
Management Library
"DAX"
user space
File System
pmem-Aware
File System
MMU
Mappings
kernel space
Generic NVDIMM Driver
Persistent Memory
Optimized
flush

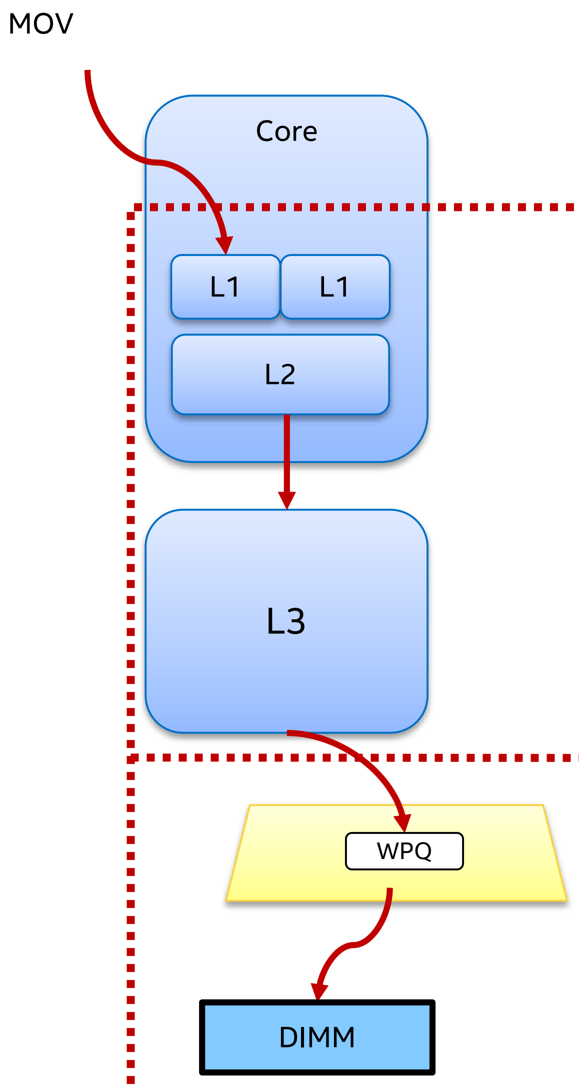
# A Programmer's View (mapped files)

```
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"

# How the Hardware Works

Not shown:
MCA
ADR Failure Detection




CPU CACHES

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

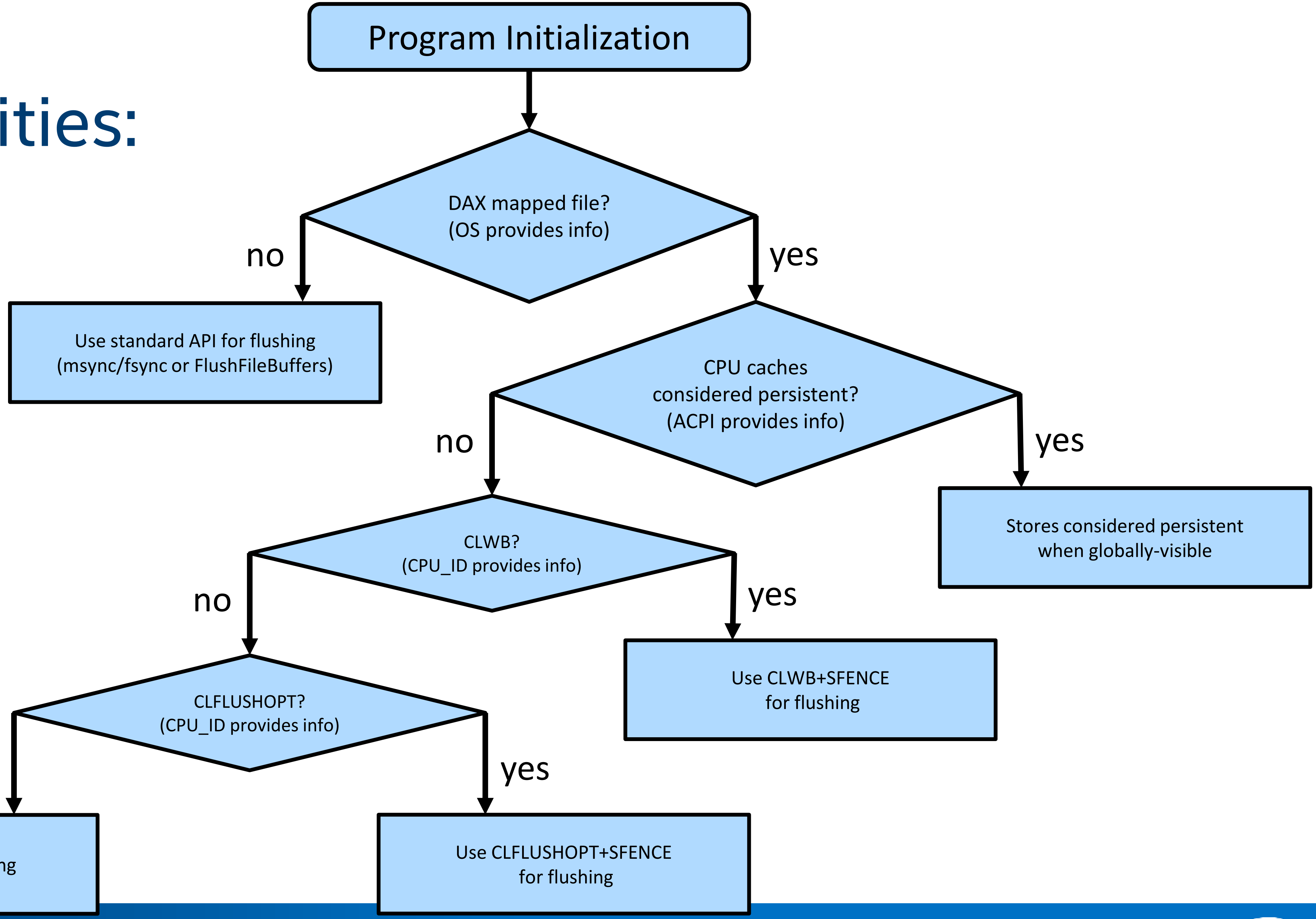
**Custom**
**Power fail protected domain**
indicated by ACPI property:
CPU Cache Hierarchy

WPQ

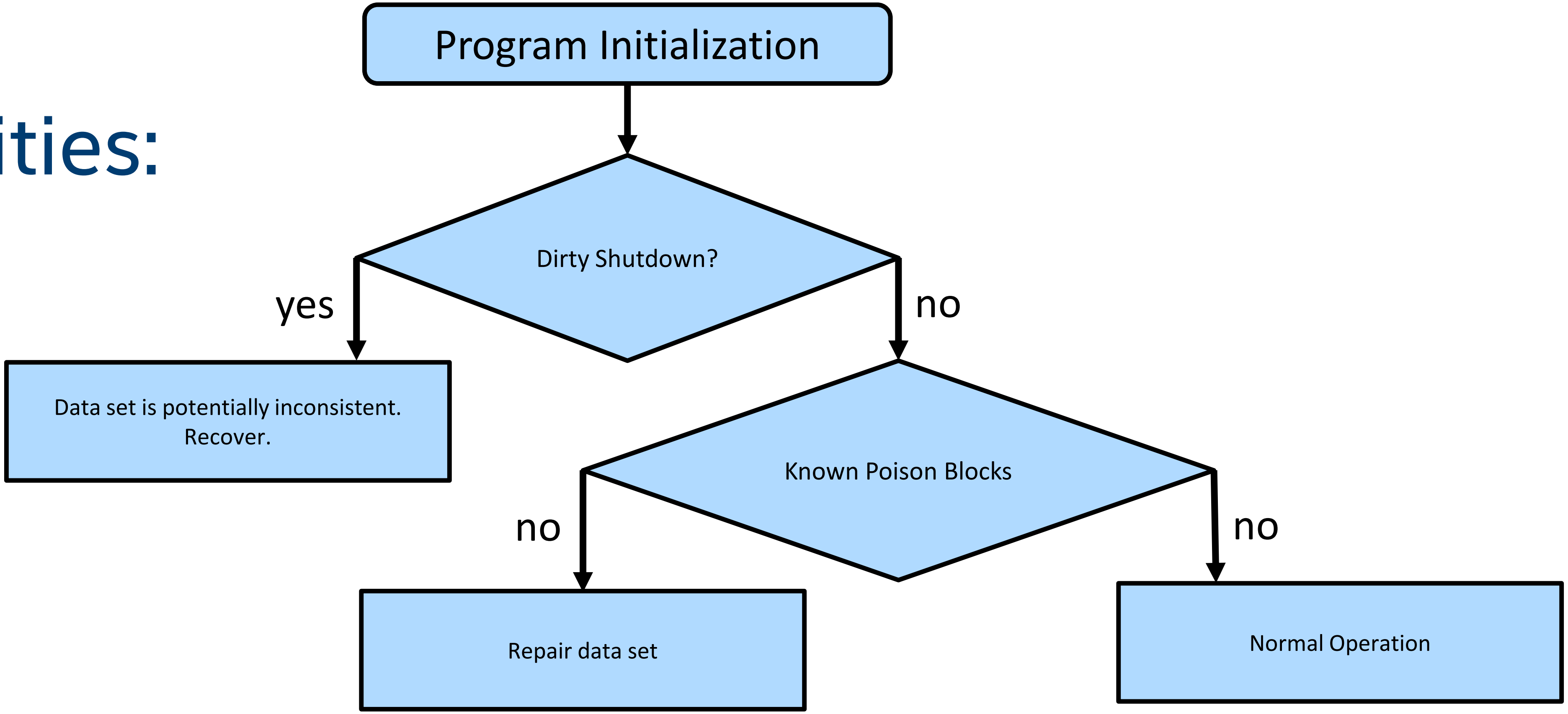
ADR
-or-
WPQ Flush (kernel only)

**Minimum Required**
**Power fail protected domain**:
Memory subsystem

# Application Responsibilities: Flushing



# Application Responsibilities: Recovery



# Application Responsibilities: Consistency

```
open(…);

mmap(…);

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

msync(…);
```

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

Result

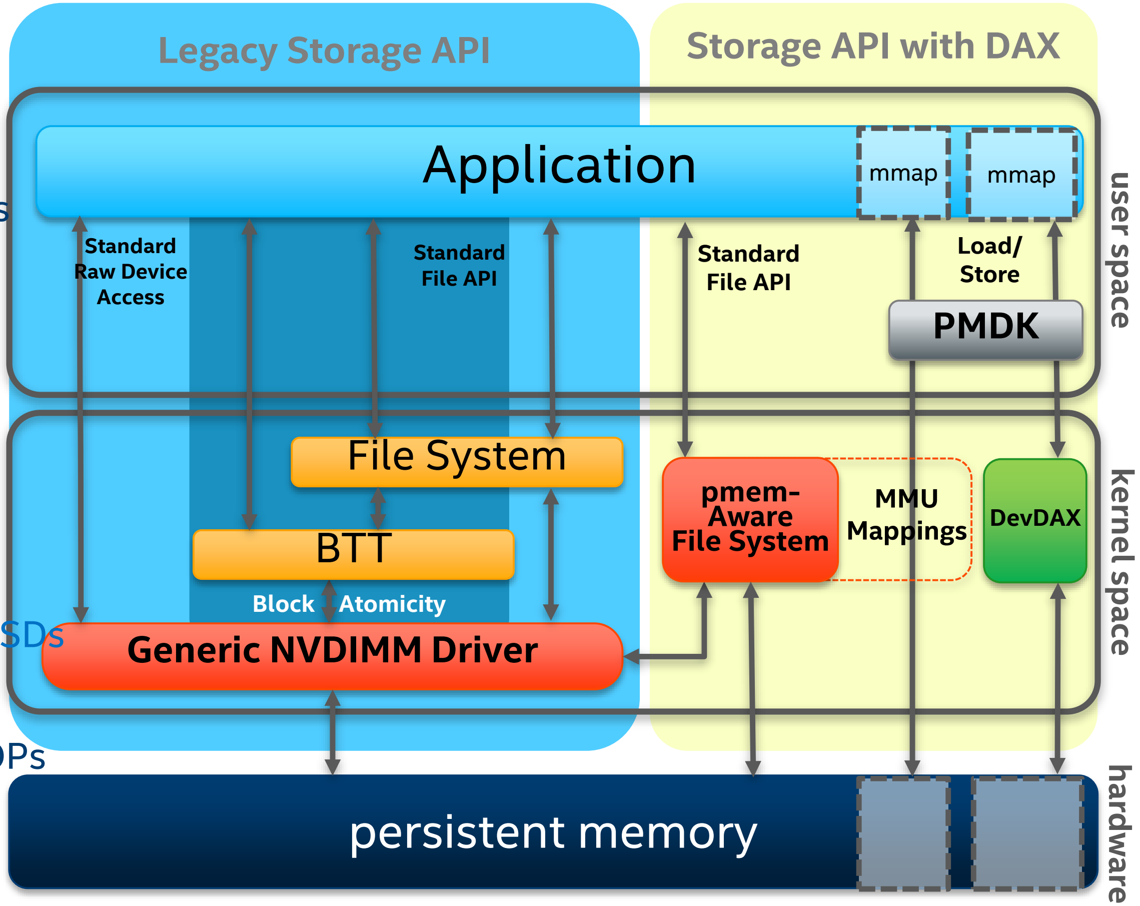
```
1. "\0\0\0\0\0\0\0\0\0\0..."

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"
```

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

# Possible ways to access persistent memory

- No Code Changes Required
- Operates in Blocks like SSD/HDD
  - Traditional read/write
  - Works with Existing File Systems
  - Atomicity at block level
  - Block size configurable
    - 4K, 512B*
- NVDIMM Driver required
  - Support starting Kernel 4.2
- Configured as Boot Device
- Higher Endurance than Enterprise SSDs
- High Performance Block Storage
  - Low Latency, higher BW, High IOPs

*Requires Linux




- Code changes may be required*
- Bypasses file system page cache
- Requires DAX enabled file system
  - XFS, EXT4, NTFS
- No Kernel Code or interrupts
- No interrupts
- Fastest IO path possible

* Code changes required for load/store direct access if the application does not already support this.

# Visibility versus Power Fail Atomicity

| Feature | Atomicity |
|---|---|
| Atomic Store | 8 byte powerfail atomicity<br>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

# If caches are not flush on failure…

- Can’t easily use compare_and_swap / fetch_and_add on Persistent Memory resident variables
- Can’t use Hardware Transactional Memory (TSX) on Persistent Memory
- Must manually flush all data after writing

# If caches are flush on failure…

- No need to flush data
- But applications still need do their own transactions
  - Can use HTM/TSX for that

# 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     durable = 0 */
        __sync_fetch_and_add(&object->refcount, 1); /*            visible = 1     durable = ? */
        persist(&object->refcount, sizeof(object->refcount)); /*  visible = 1     durable = 1 */
}
```

Decision is made based on visible but not durable value

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

Decision is based on a known durable value

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

experience
what’s inside™