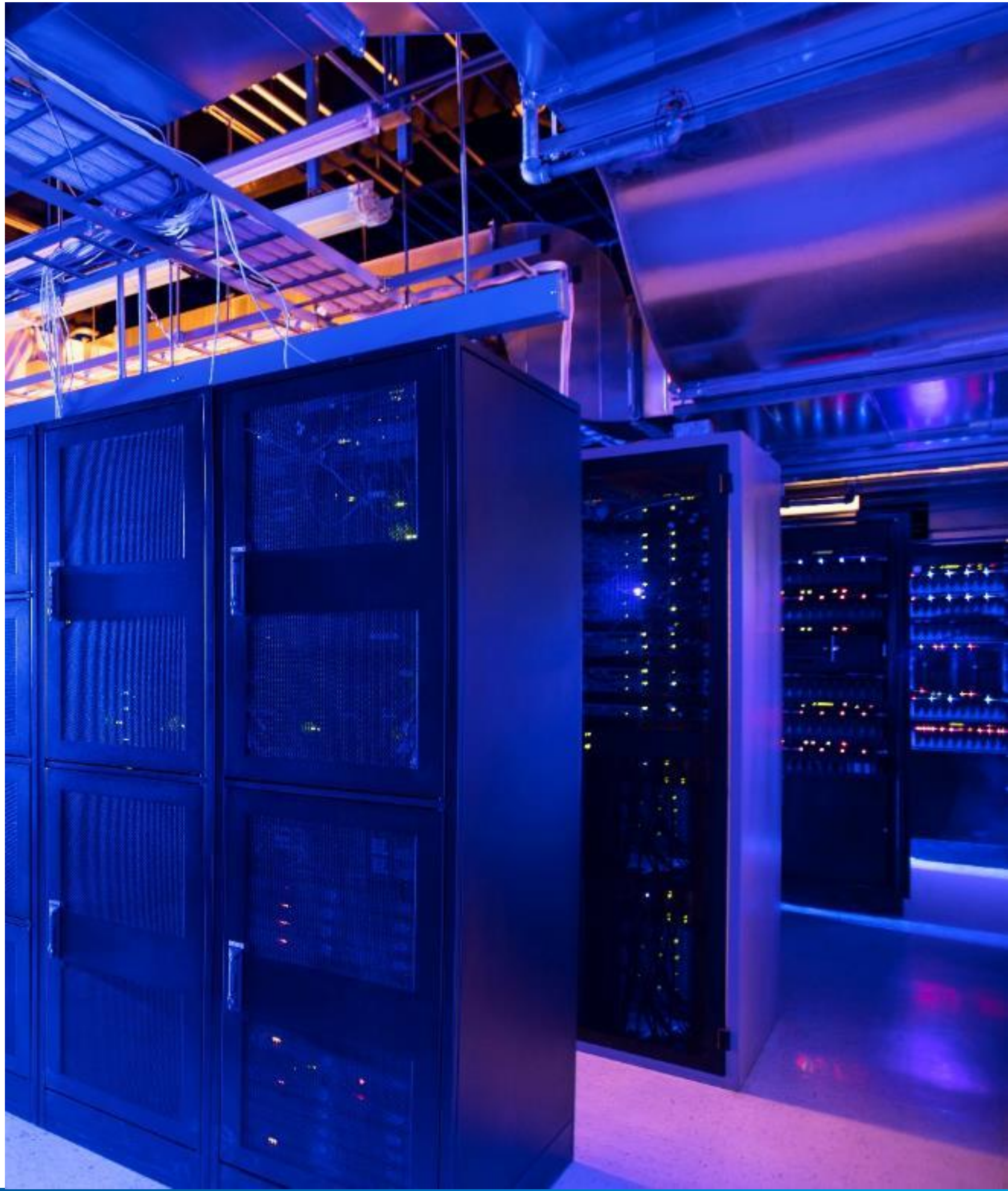
PMDK ESSENTIALS

Andy Rudoff (Intel Data Center Group)

September 5th, 2019

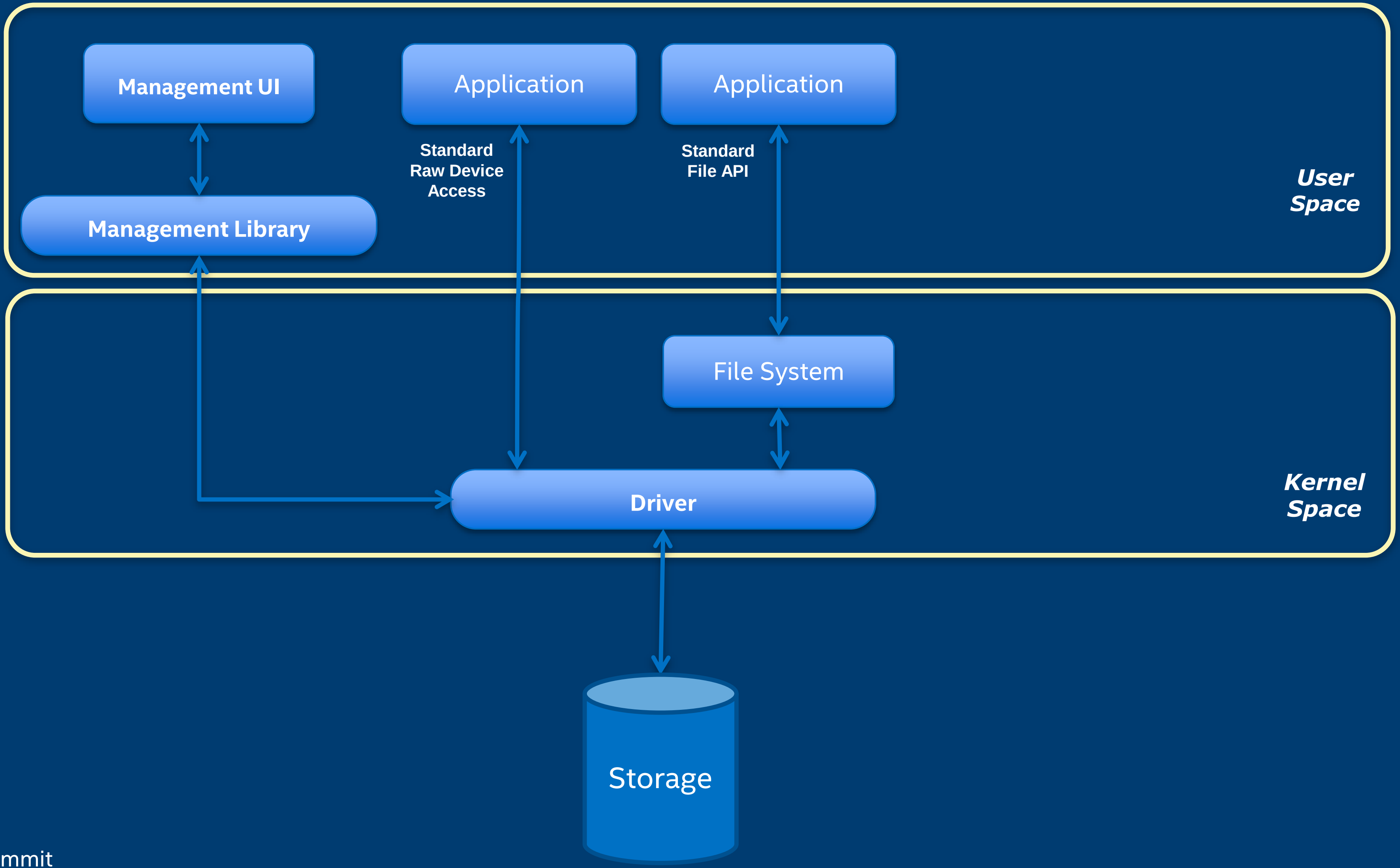
AGENDA

- Persistent Memory Concepts
- Operating System Essentials
- The PMDK Libraries
- Flushing, Transactions, Allocation
- Language Support
- Comparing High and Low Level Languages



PERSISTENT MEMORY CONCEPTS

THE STORAGE STACK (50,000FT 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?

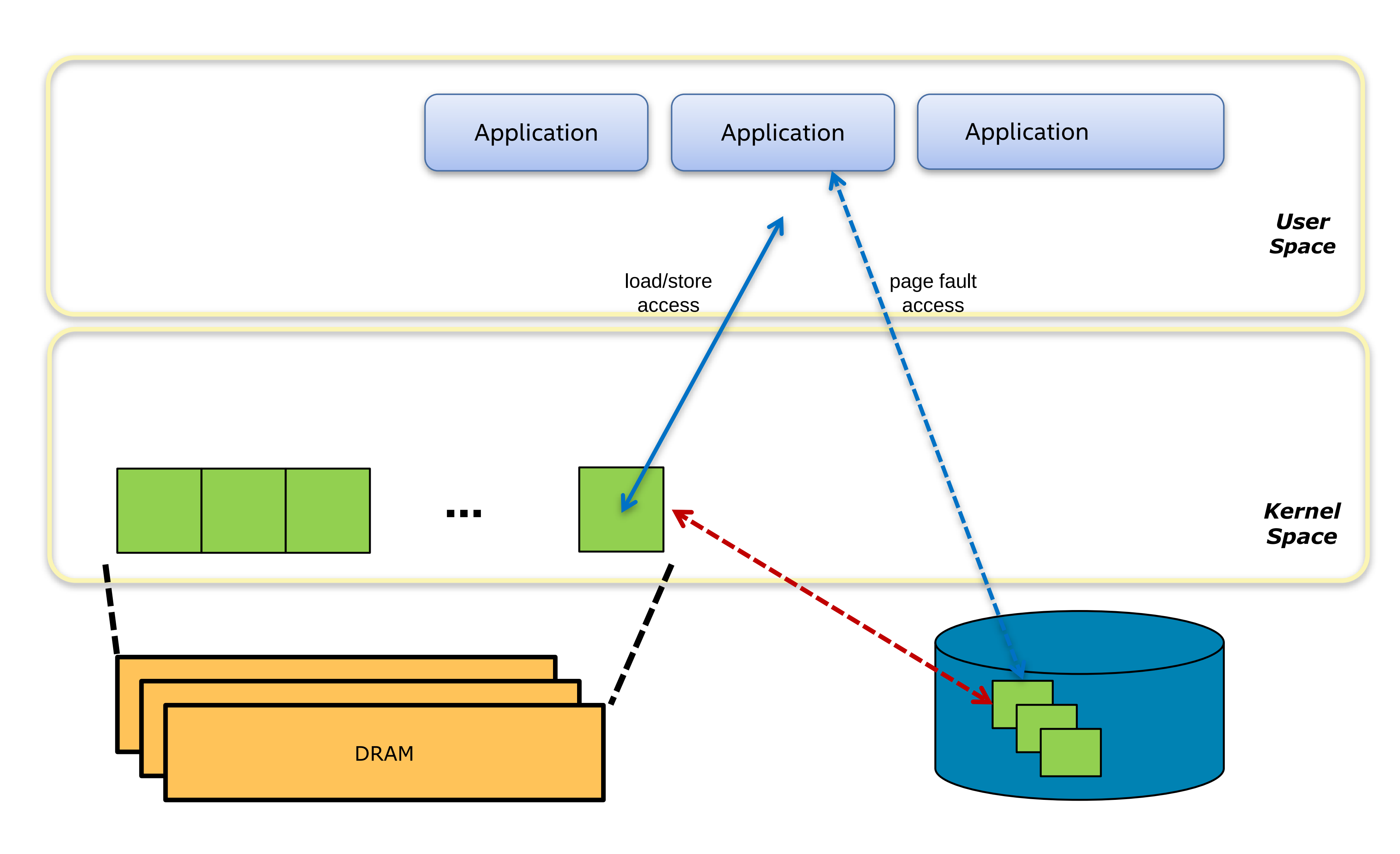
- Read faults/Write faults
- Flush to persistence

Not that commonly used or understood

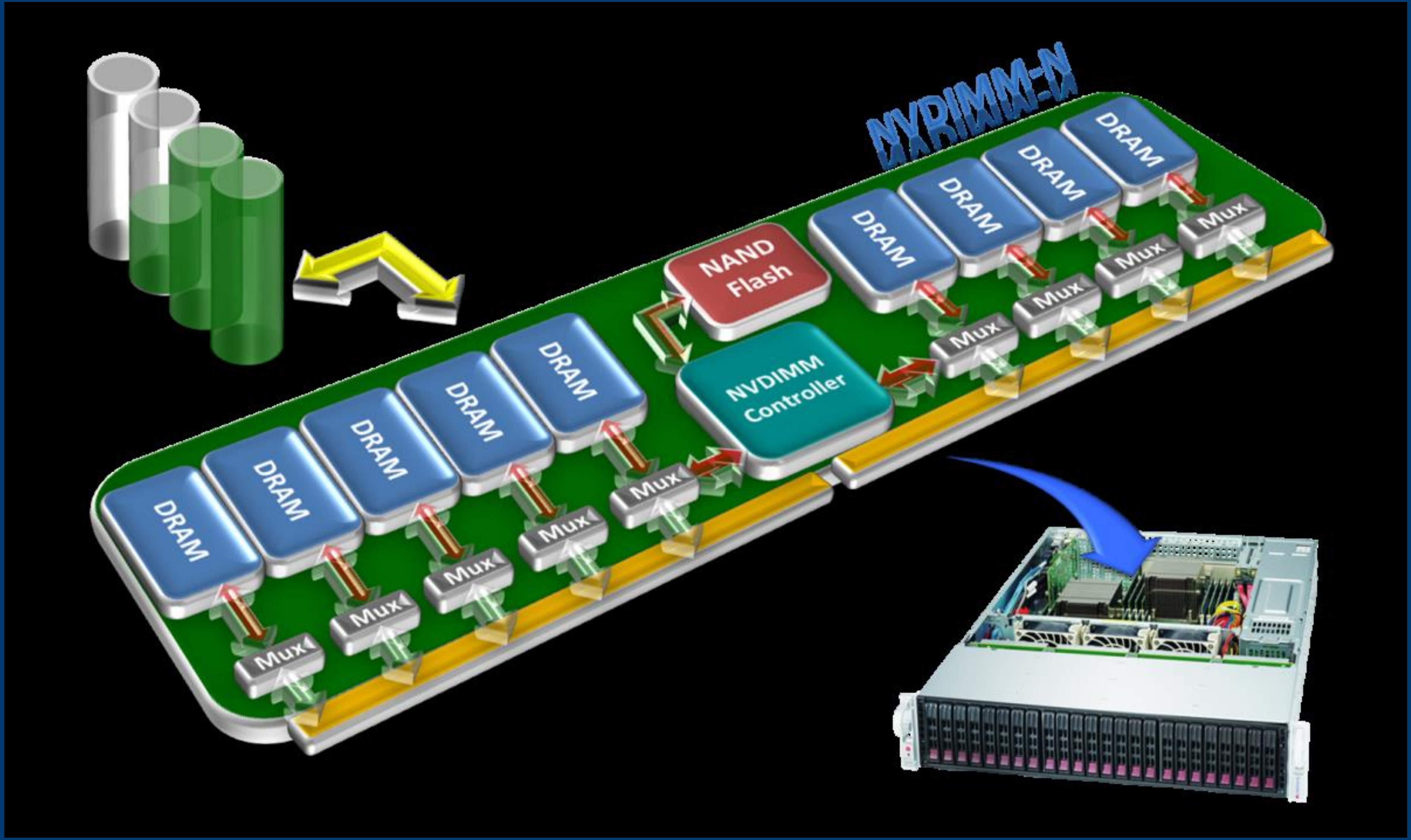
- Quite powerful
- Sometimes used without realizing it

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

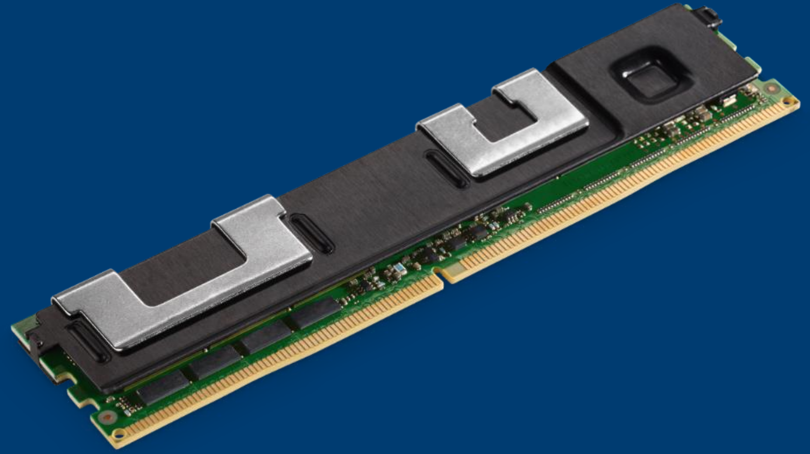
OS PAGING


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

NVDIMM-N


NVDIMM-N
NAND Flash
NVDIMM Controller
DRAM
DRAM
DRAM
DRAM
DRAM
DRAM
DRAM
DRAM
DRAM
Mux
Mux
Mux
Mux
Mux
Mux
Mux
Mux
Mux

intel OPTANE™ DC PERSISTENT MEMORY

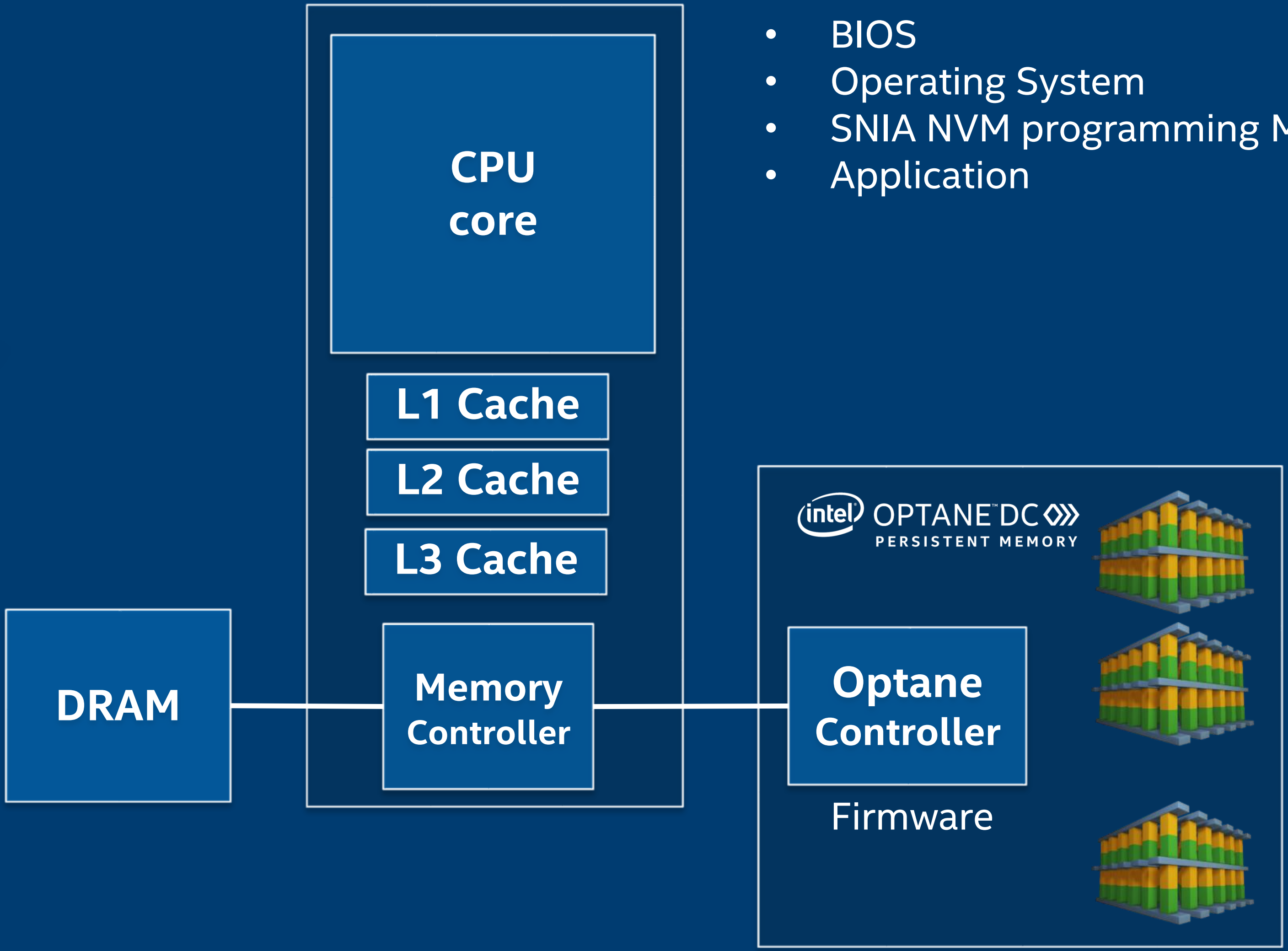


Direct Load/Store Access

Native Persistence

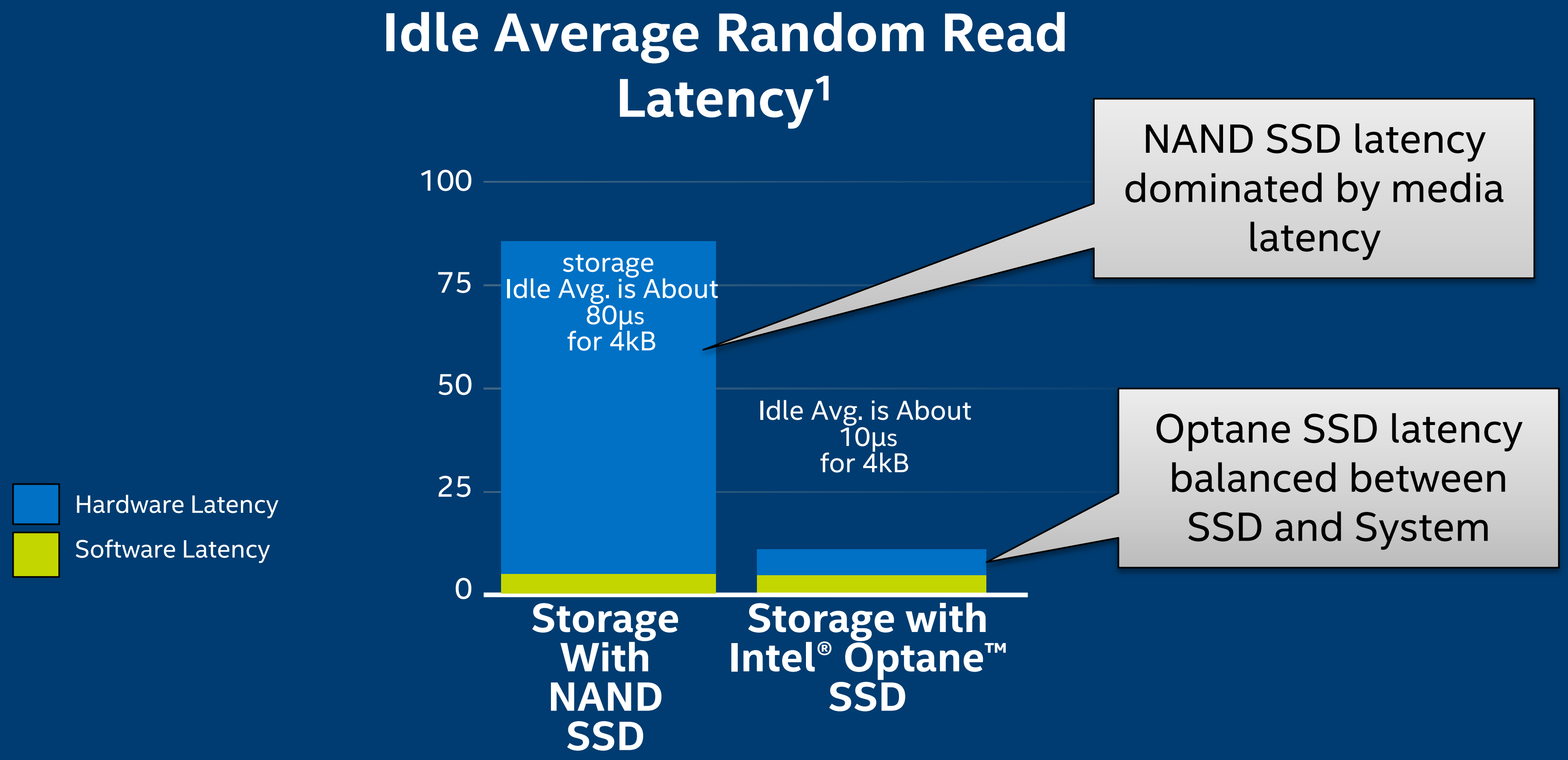
128, 256, 512GB

DDR4 Pin Compatible



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

MOTIVATION FOR THE PM PROGRAMMING MODEL?



Idle Average Random Read Latency[1]

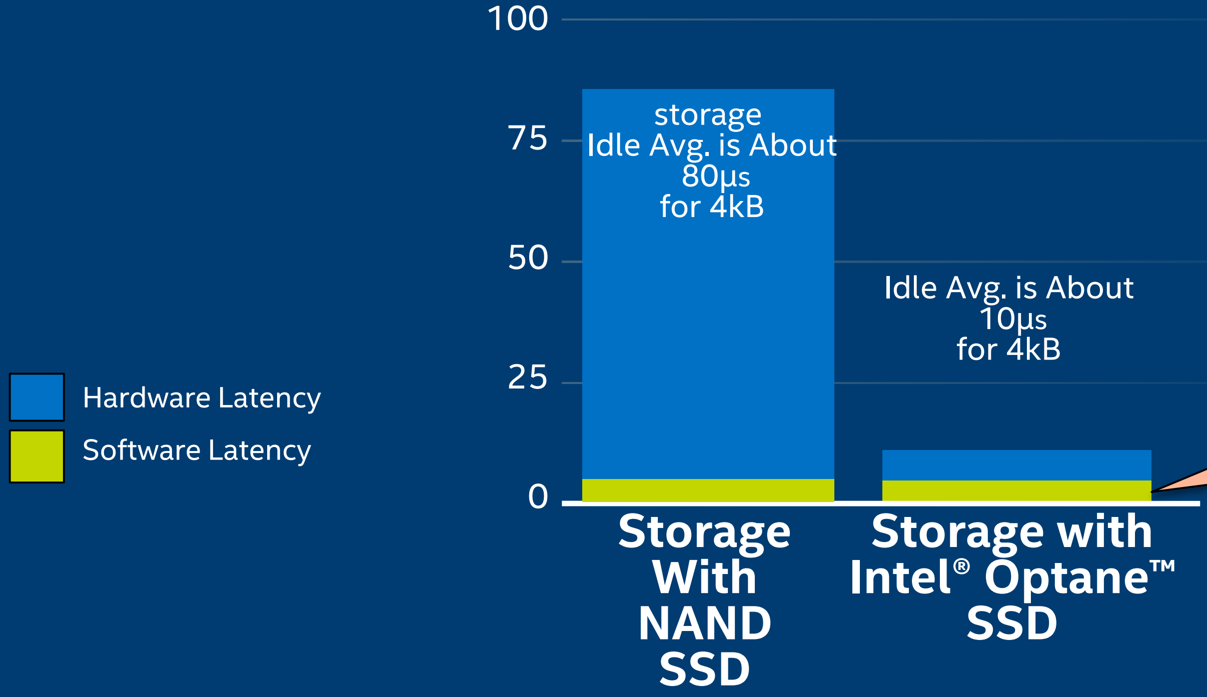
NAND SSD latency dominated by media latency

Optane SSD latency balanced between SSD and System

[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?

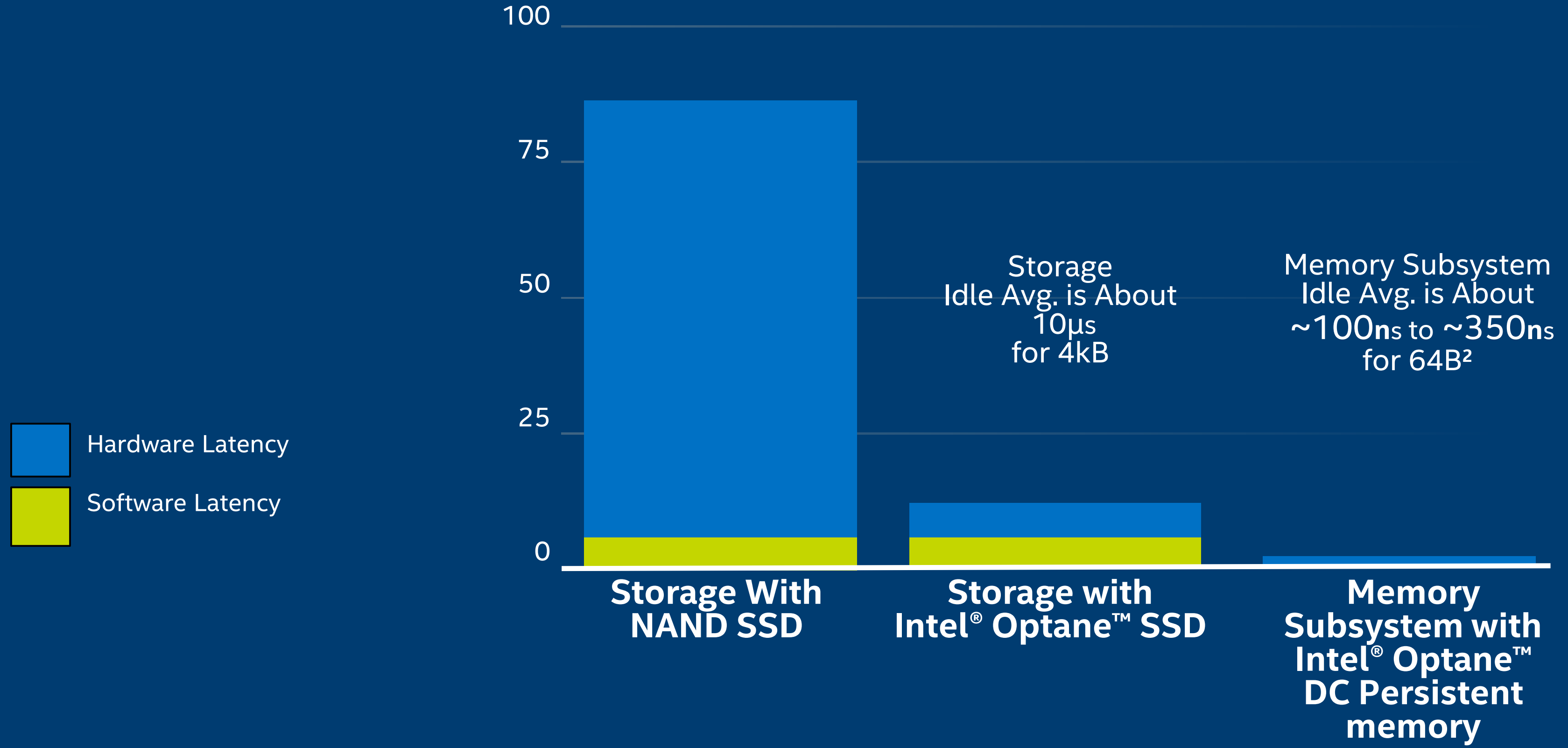
Idle Average Random Read Latency[1]



Next logical improvement: remove the SW stack.

[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]



Storage
Idle Avg. is About
10µs
for 4kB

Memory Subsystem
Idle Avg. is About
~100ns to ~350ns
for 64B[2]

[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)

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

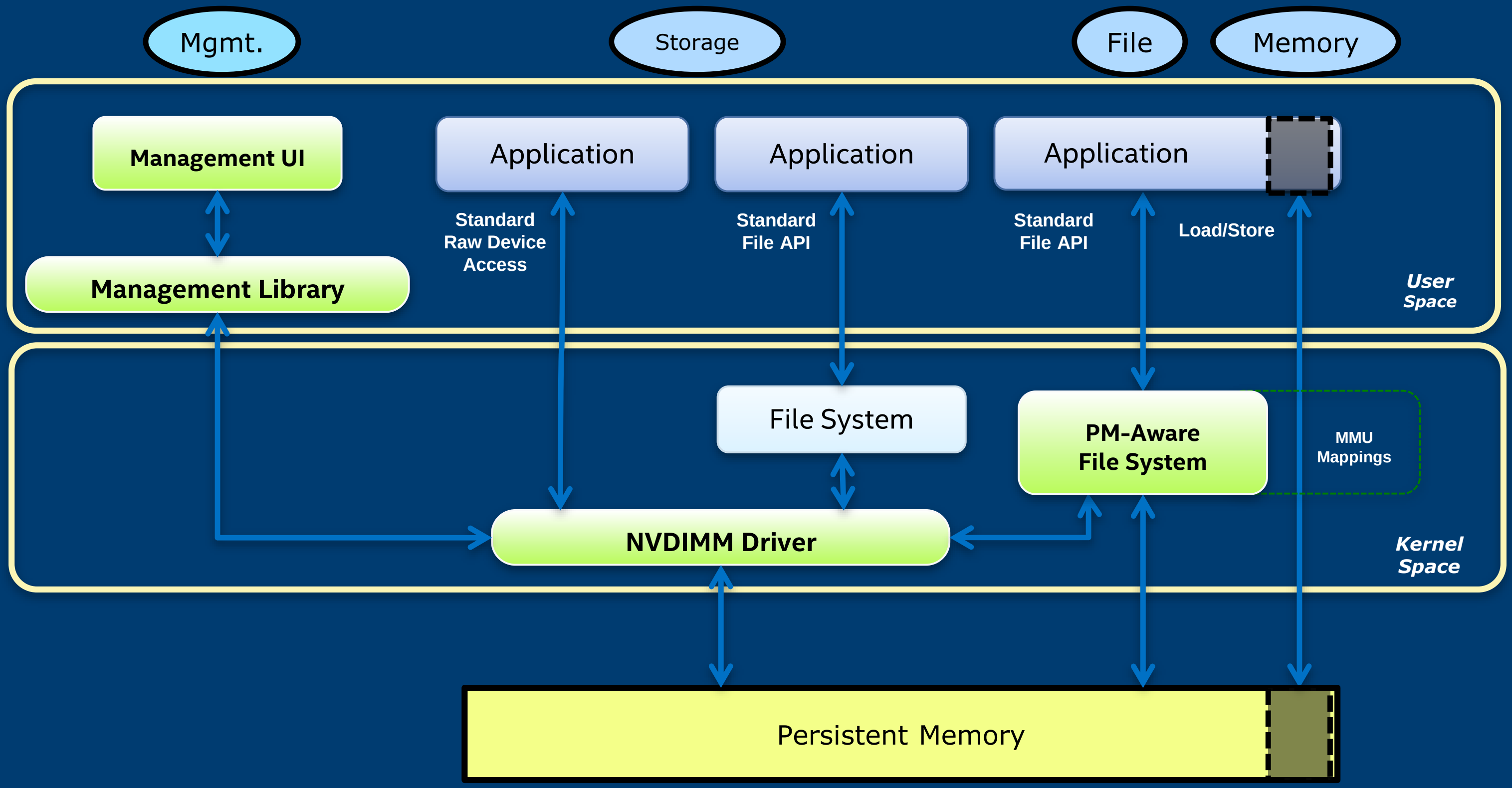
- No time spend loading up memory

Byte addressable

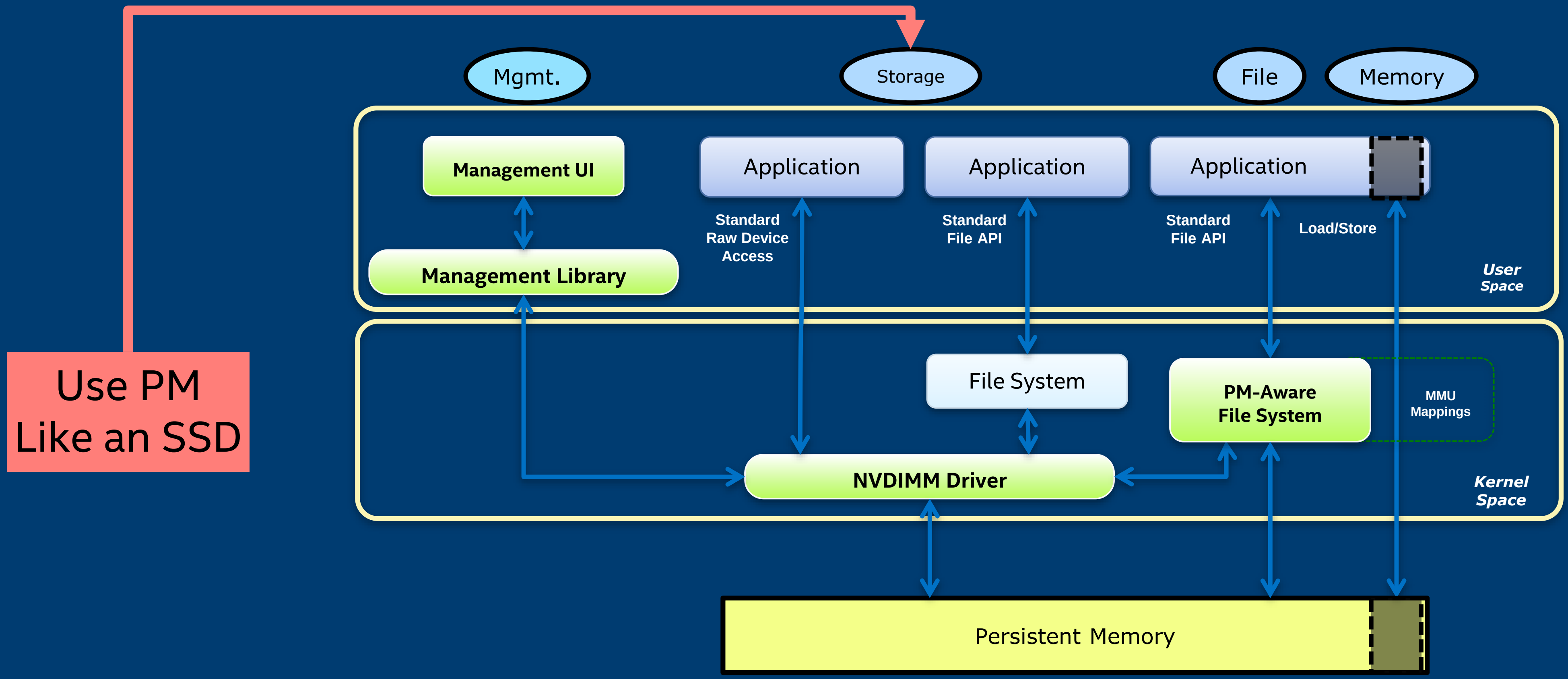
Direct user-mode access

- No kernel code in data path

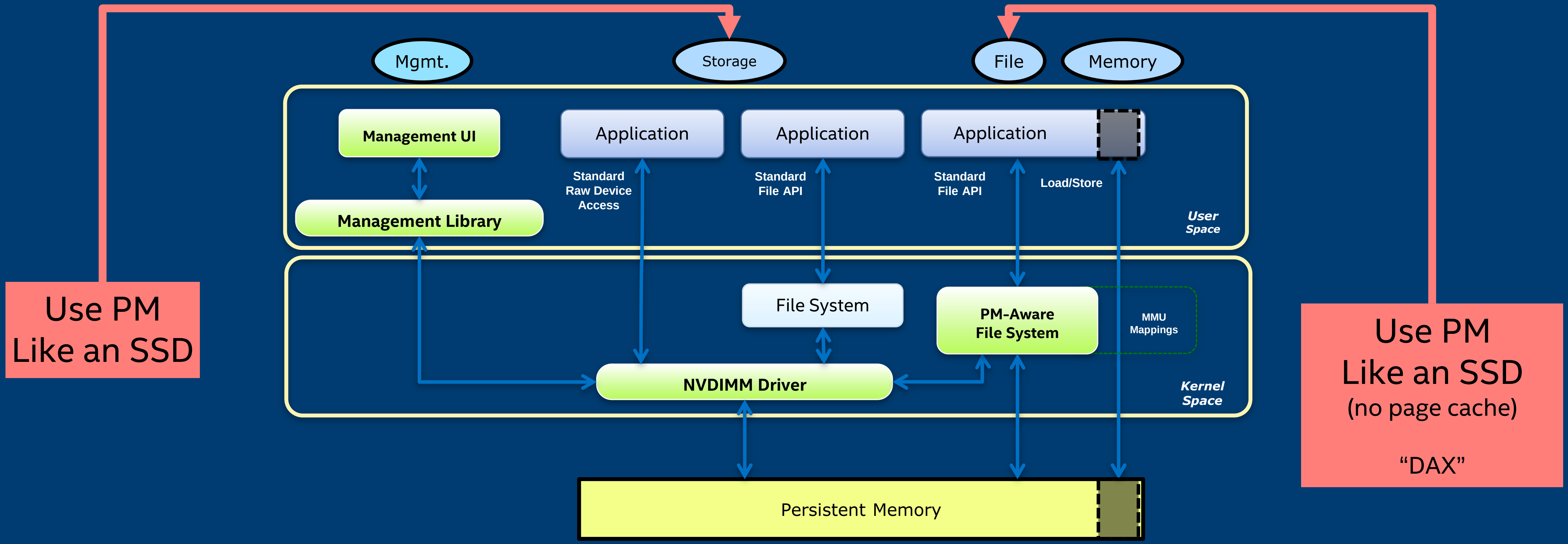
THE SNIA NVM PROGRAMMING MODEL



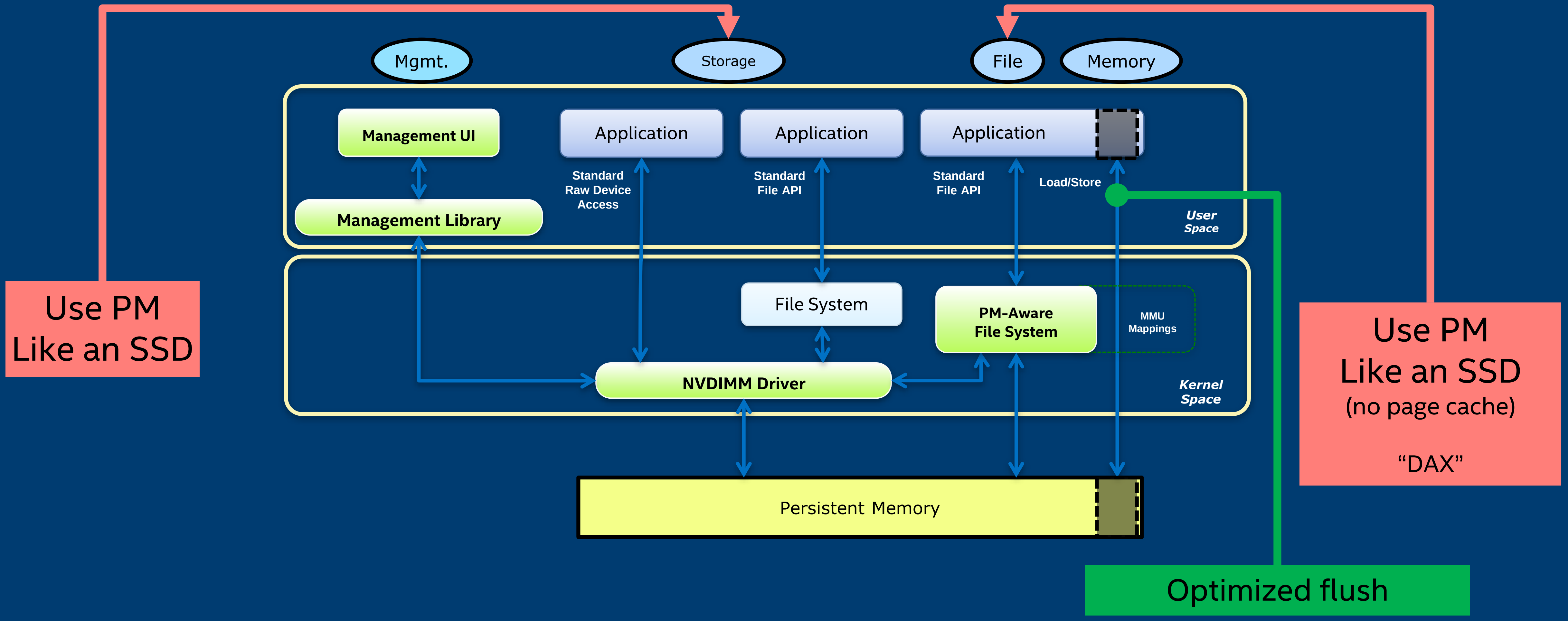
THE PROGRAMMING MODEL BUILDS ON THE STORAGE APIS



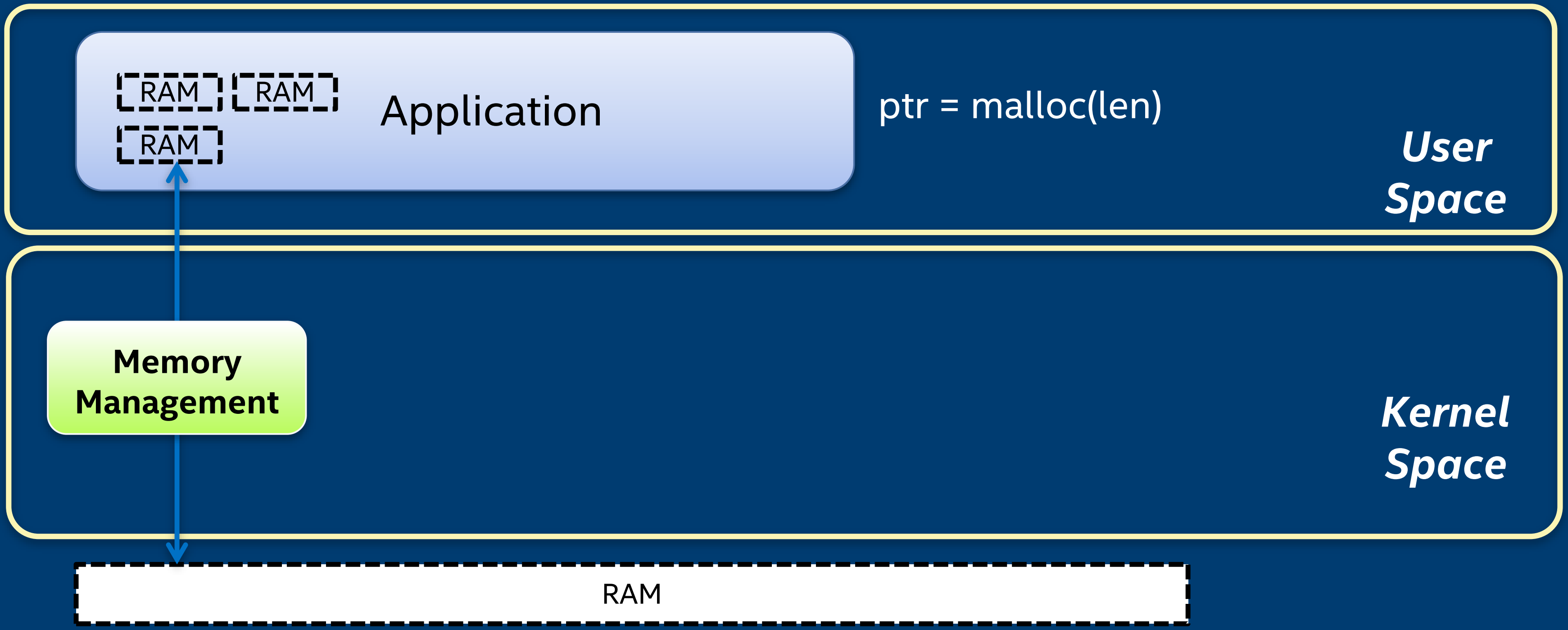
THE PROGRAMMING MODEL BUILDS ON THE STORAGE APIS



OPTIMIZED FLUSH IS THE PRIMARY NEW API

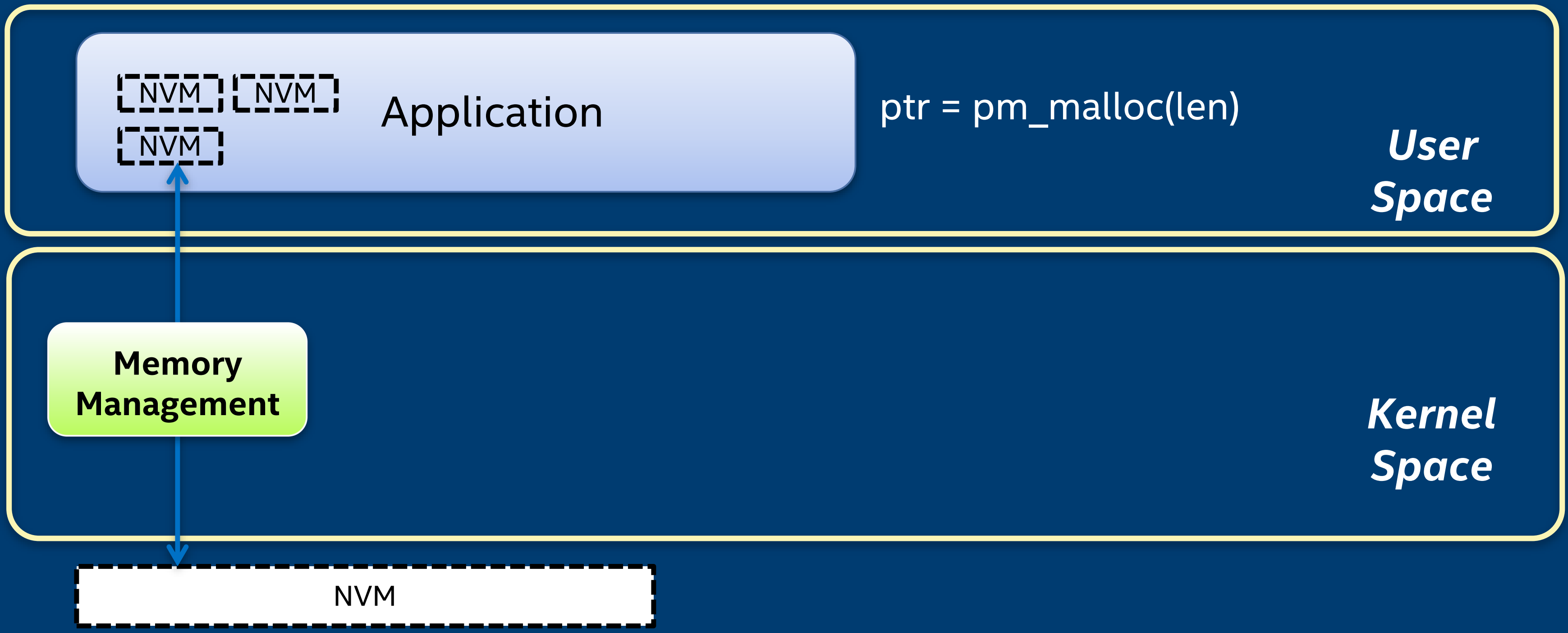


APPLICATION MEMORY ALLOCATION



- 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 PERSISTENCE

It has always been thus:

- open()
- mmap()
- store...

visible

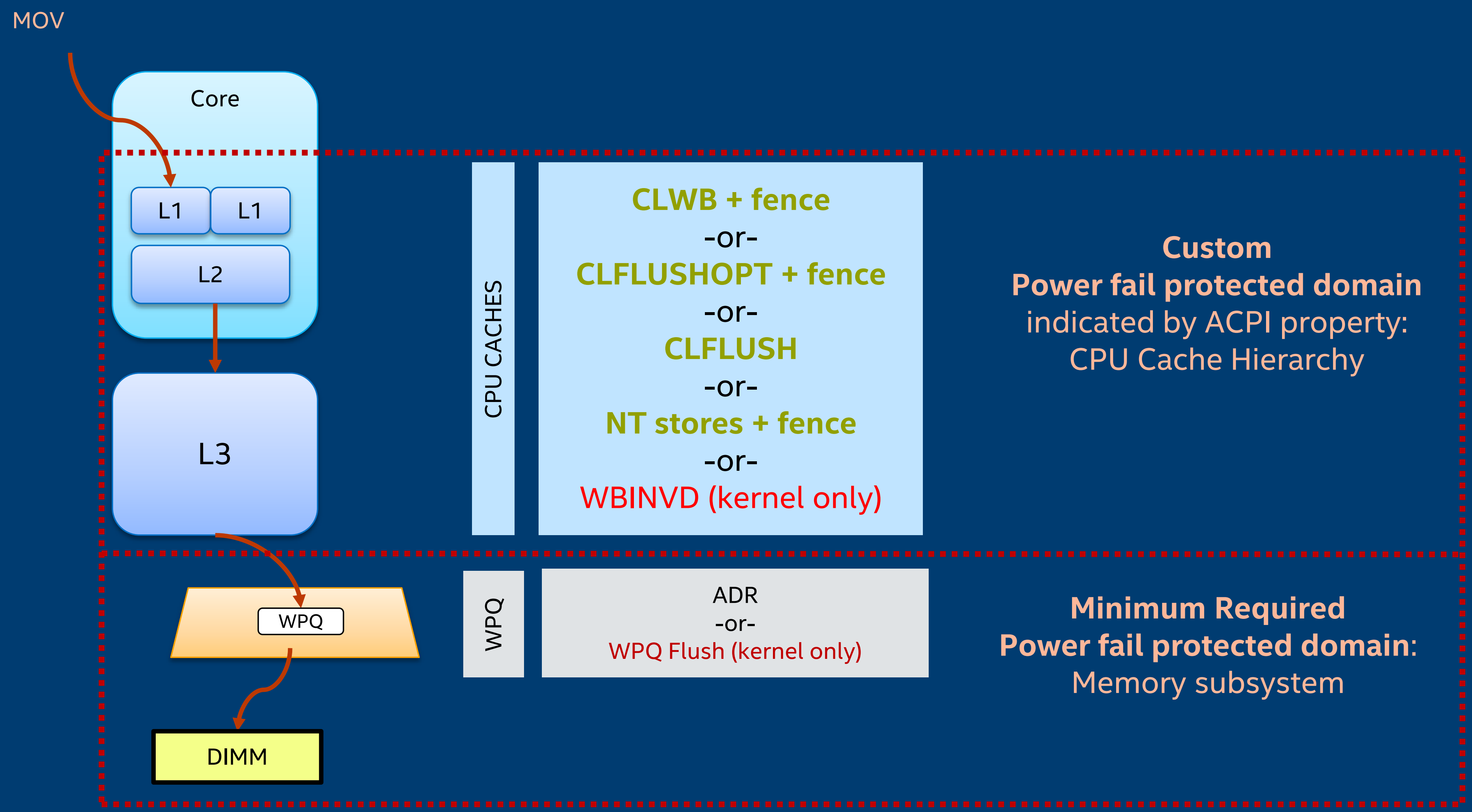
- msync()

persistent

pmem just follows this decades-old model

- But the stores are cached in a different spot

HOW THE HW WORKS



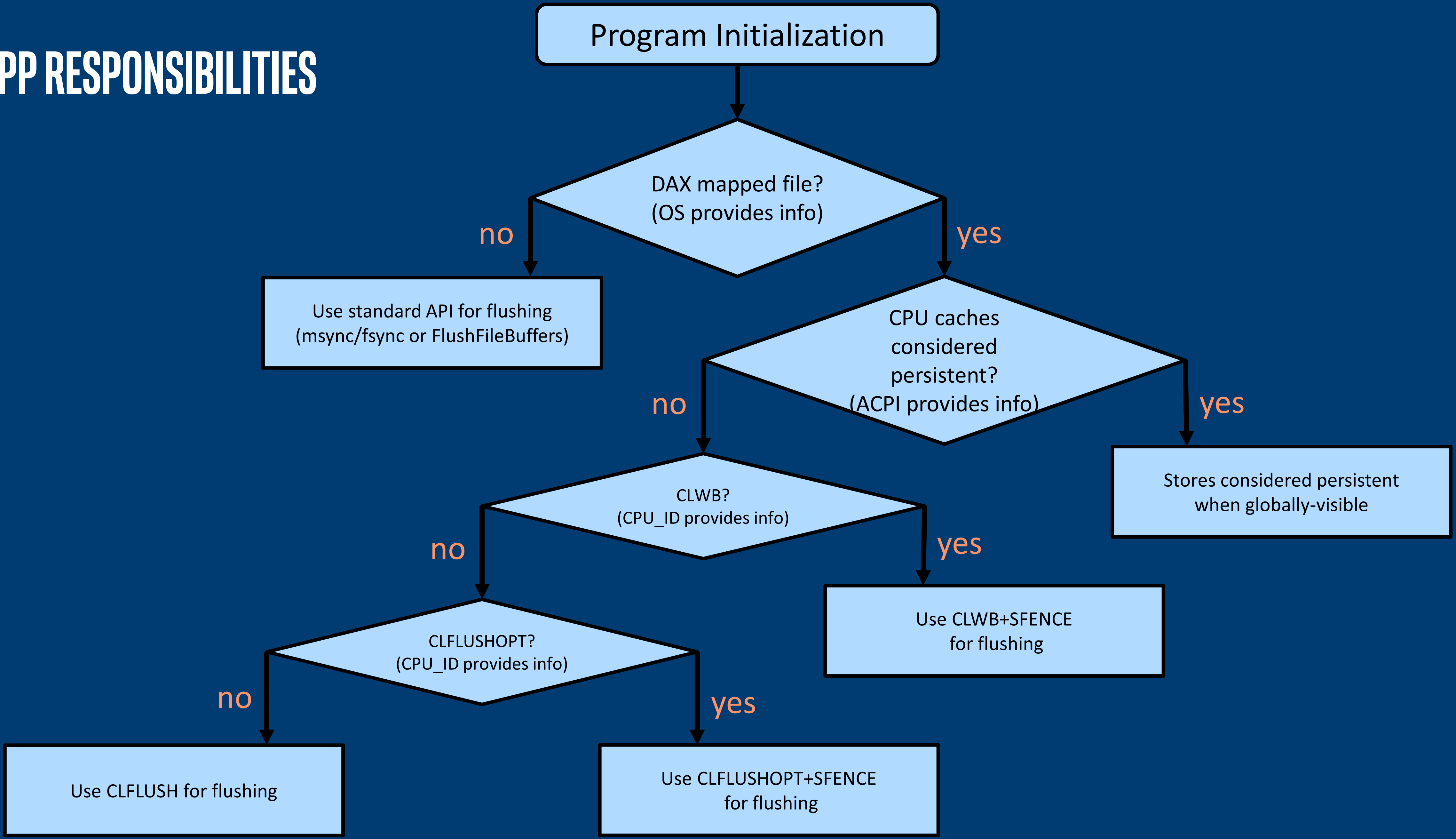
CLWB + fence
-or-
CLFLUSHOPT + fence
-or-
CLFLUSH
-or-
NT stores + fence
-or-
WBINVD (kernel only)

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

ADR
-or-
WPQ Flush (kernel only)

Minimum Required
Power fail protected domain:
Memory subsystem

APP RESPONSIBILITIES



Program Initialization

DAX mapped file?
(OS provides info)

no

Use standard API for flushing
(msync/fsync or FlushFileBuffers)

yes

CPU caches
considered
persistent?
(ACPI provides info)

no

CLWB?
(CPU_ID provides info)

yes

Stores considered persistent
when globally-visible

no

CLFLUSHOPT?
(CPU_ID provides info)

yes

Use CLWB+SFENCE
for flushing

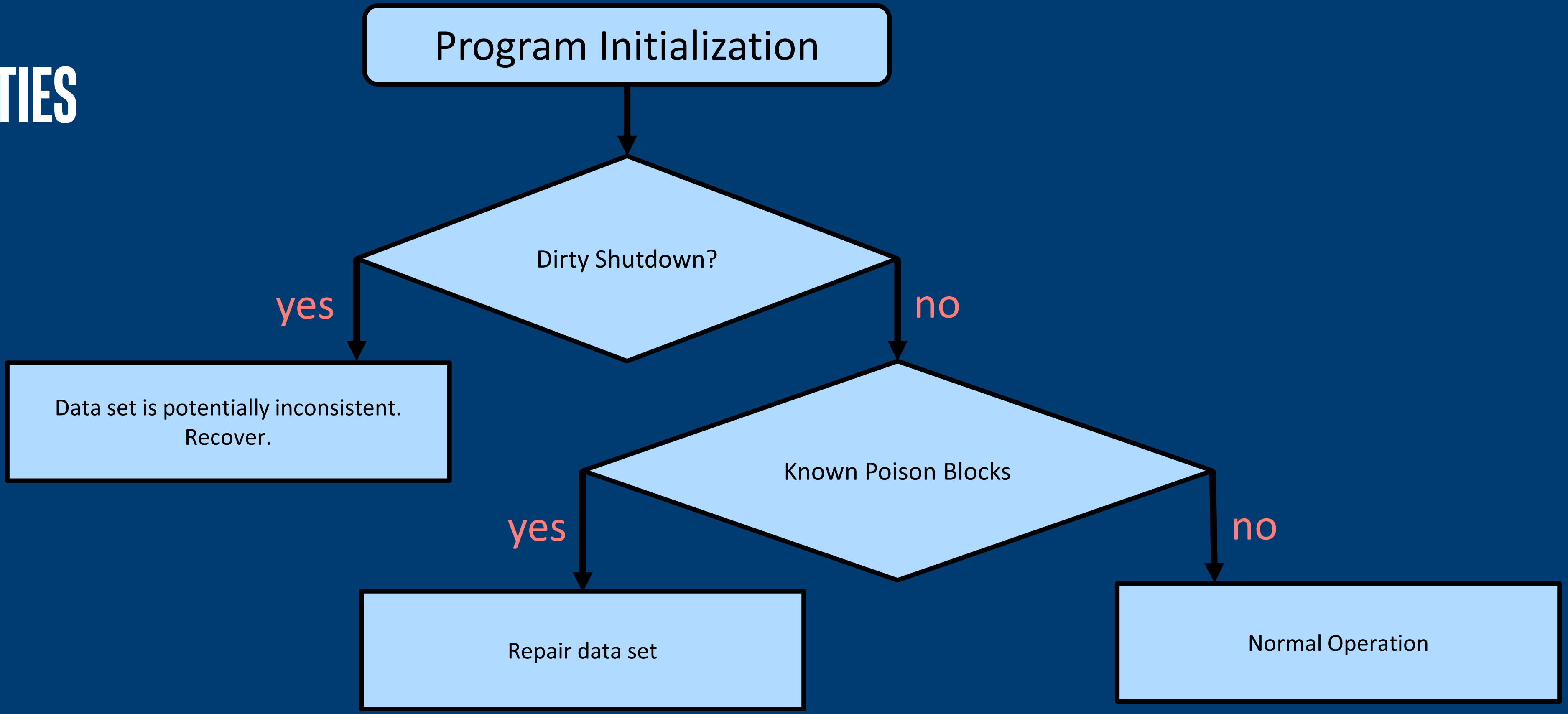
no

Use CLFLUSH for flushing

yes

Use CLFLUSHOPT+SFENCE
for flushing

APP RESPONSIBILITIES (RECOVERY)



Program Initialization

Dirty Shutdown?

yes

Data set is potentially inconsistent. Recover.

no

Known Poison Blocks

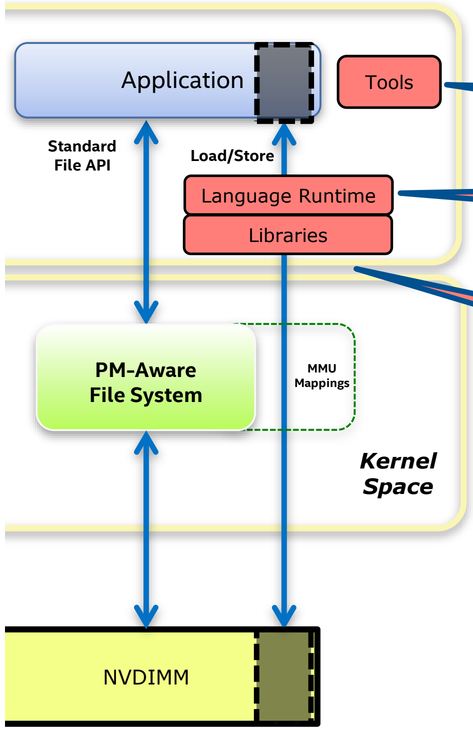
yes

Repair data set

no

Normal Operation

CREATING A PROGRAMMING ENVIRONMENT



Tools for correctness and performance

Language support

Optimized allocators, transactions

Result:
Safer, less error-prone

OPERATING SYSTEM ESSENTIALS

ENABLING IN THE ECOSYSTEM

- Linux kernel version 4.19 (ext4, xfs)
- Windows Server 2019 (NTFS)
- VMware vSphere 6.7
- RHEL 7.5
- SLES 15 and SLES 12 SP4
- Ubuntu 18.*
- Java JDK 12
- Kubernetes 1.13
- OpenStack ‘Stein’

See Steve Scargall’s Webinar on how to provision Optane DC Persistent Memory:
https://software.intel.com/en-us/videos/provisioning-intel-optane-dc-persistent-memory-modules-in-linux

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

THE PMDK LIBRARIES

PMDK LIBRARIES

http://pmem.io
https://github.com/pmem/pmdk

High Level Interfaces
(in development)

- Experimental C++ Persistent Containers
- PCJ – Persistent Collection for Java
- pmemkv

Language bindings

- C++
- C
- PCJ / LLPL
- Python

Transaction Support

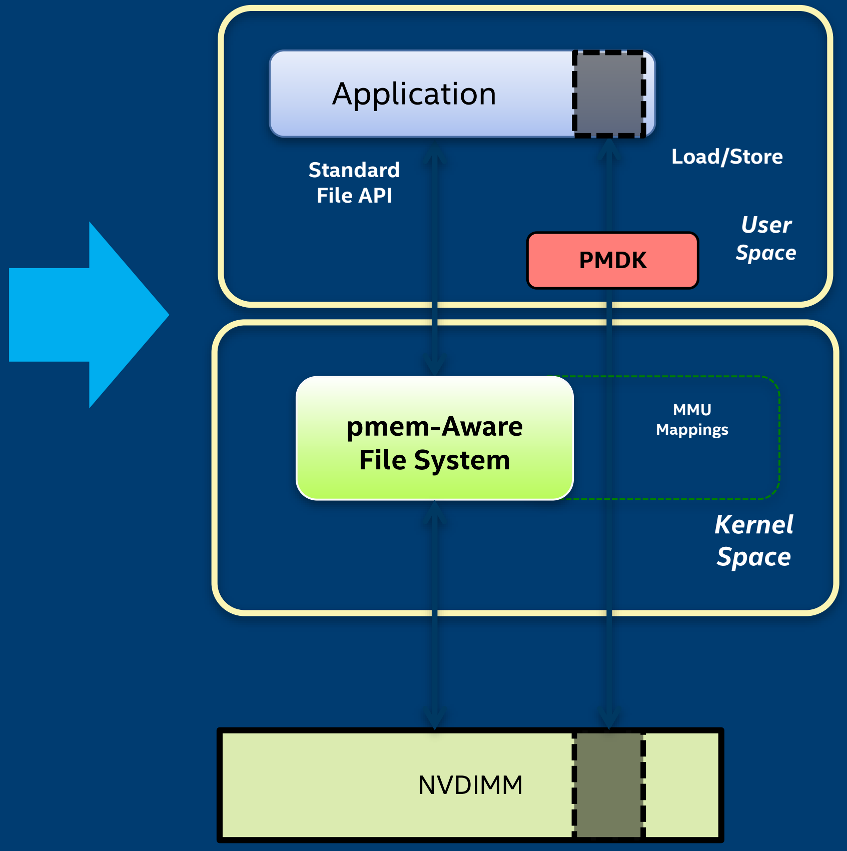
- Interface to create a persistent memory resident log file – **libpmemlog**
- Interface for persistent memory allocation, transactions and general facilities – **libpmemobj**
- Interface to create arrays of pmem-resident blocks, of same size, atomically updated – **libpmemblk**

Support for **volatile** memory usage

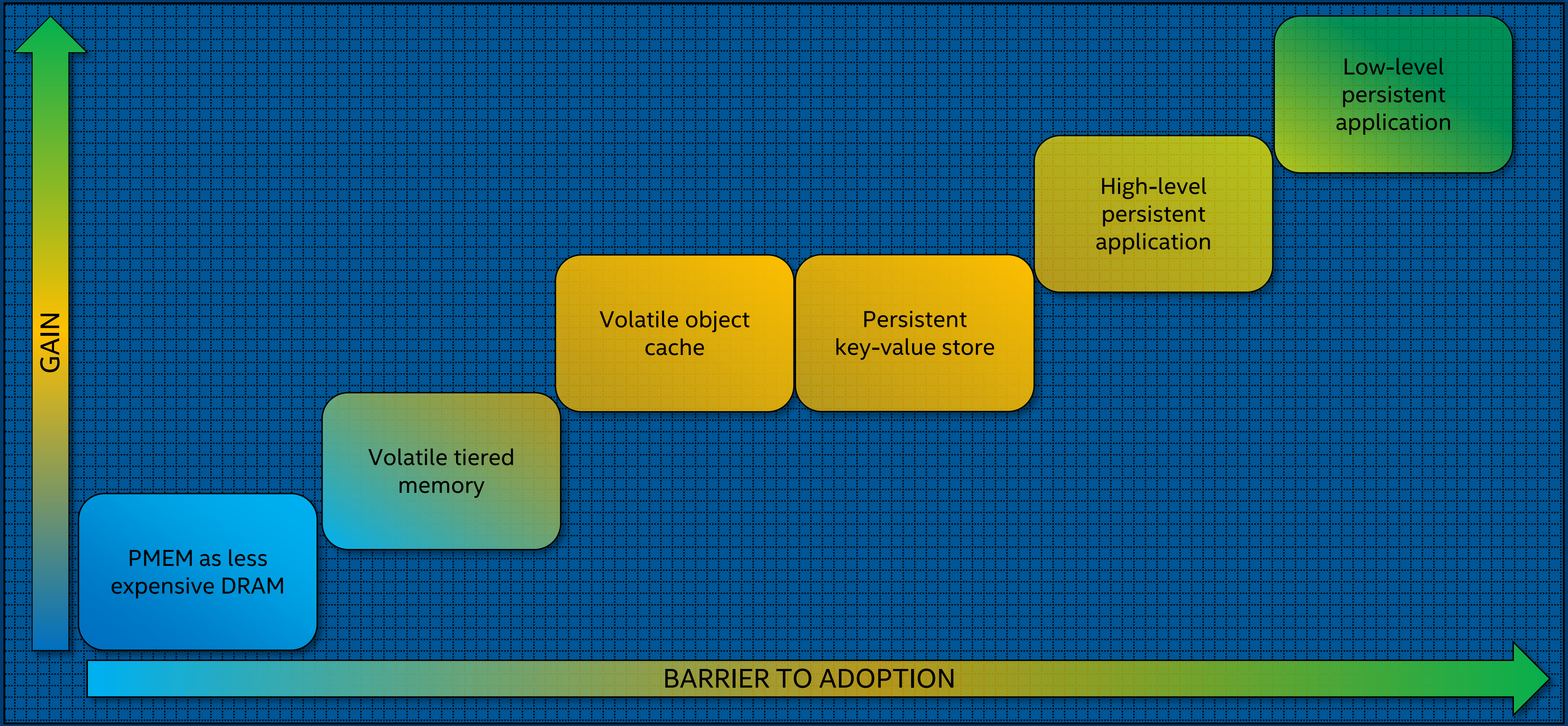
- **memkind**
- **vmemcache**

Low-level support

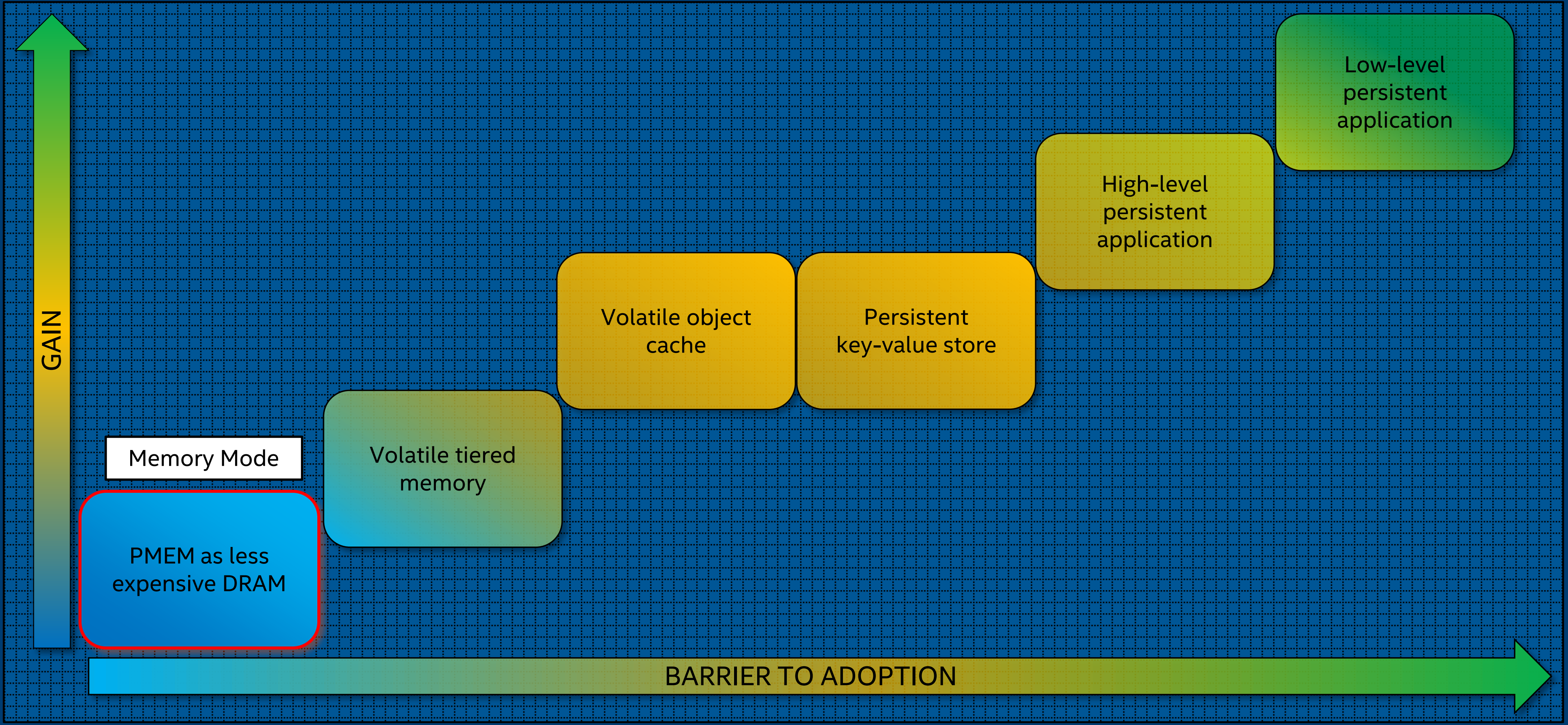
- Low level support for local persistent memory – **libpmem**
- Low level support for remote access to persistent memory – **librpmem**



DIFFERENT WAYS TO USE PERSISTENT MEMORY


GAIN
PMEM as less expensive DRAM
Volatile tiered memory
Volatile object cache
Persistent key-value store
High-level persistent application
Low-level persistent application
BARRIER TO ADOPTION

DIFFERENT WAYS TO USE PERSISTENT MEMORY


GAIN
Memory Mode
PMEM as less expensive DRAM
Volatile tiered memory
Volatile object cache
Persistent key-value store
High-level persistent application
Low-level persistent application
BARRIER TO ADOPTION

MEMORY MODE

When To Use

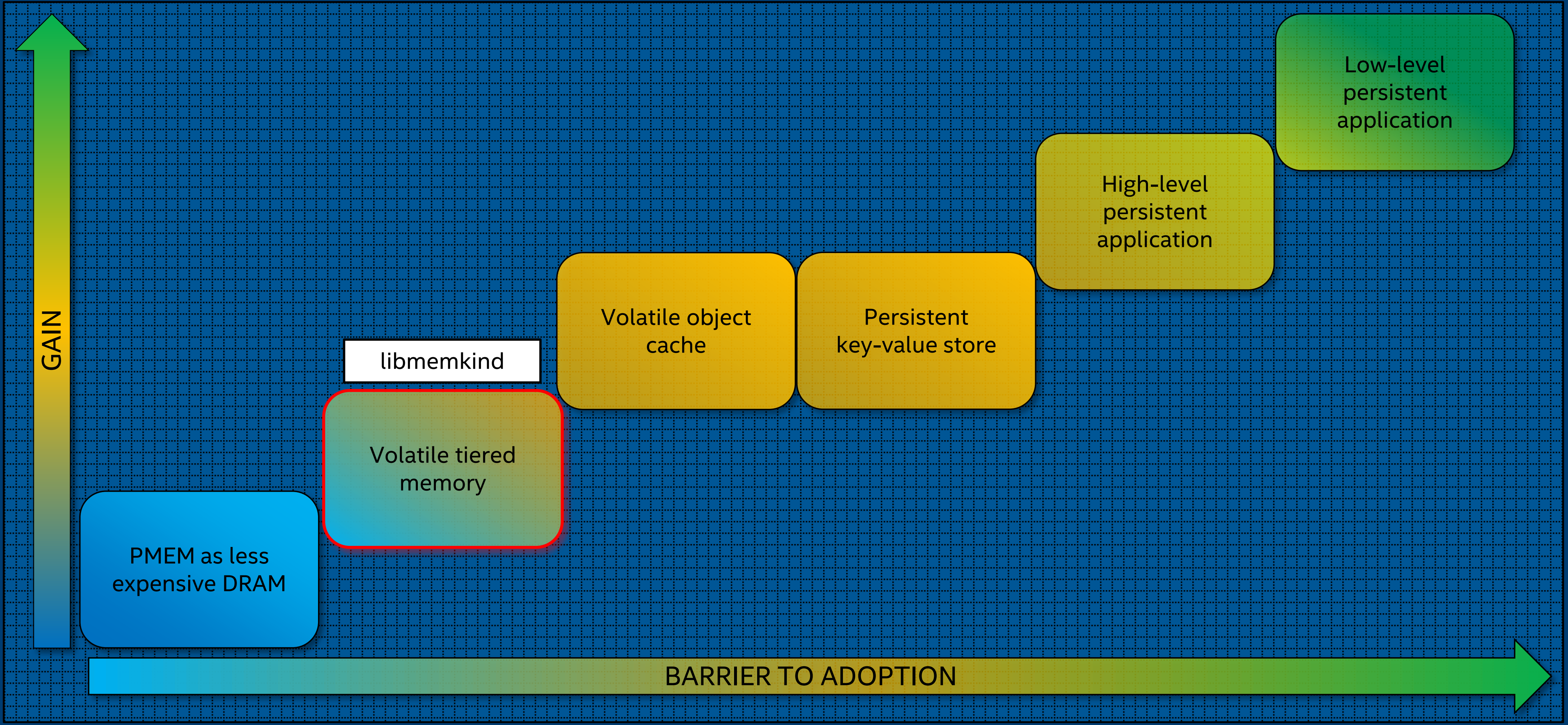
- modifying applications is not feasible
- massive amounts of memory is required (more TB)
- CPU utilization is low in shared environment (more VMs)

- 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

DIFFERENT WAYS TO USE PERSISTENT MEMORY


GAIN
PMEM as less expensive DRAM
libmemkind
Volatile tiered memory
Volatile object cache
Persistent key-value store
High-level persistent application
Low-level persistent application
BARRIER TO ADOPTION

LIBMEMKIND

When To Use

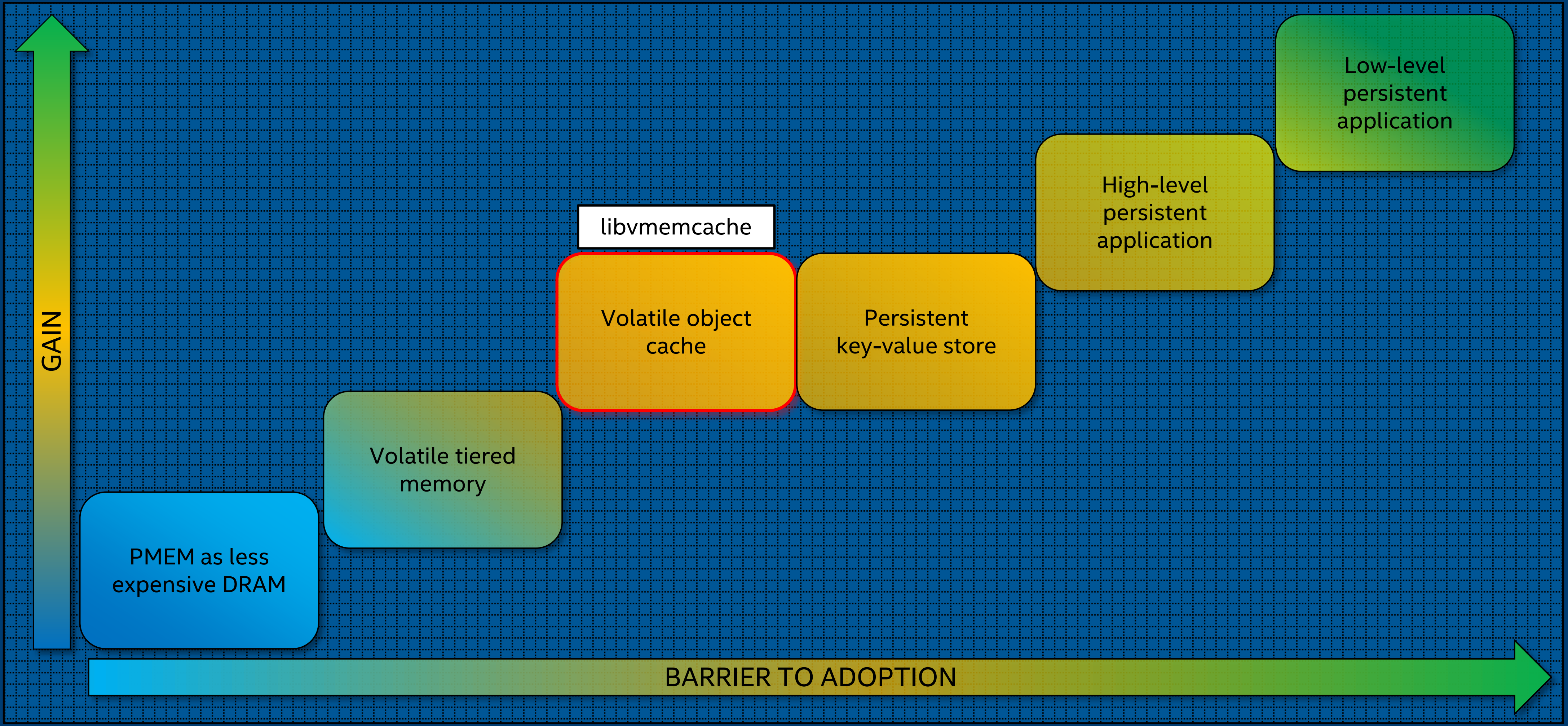
- ➢ application can be modified
- ➢ different tiers of objects (hot, warm) can be identified
- ➢ persistence is not required

- ➢ 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);

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

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

DIFFERENT WAYS TO USE PERSISTENT MEMORY


GAIN
BARRIER TO ADOPTION
PMEM as less expensive DRAM
Volatile tiered memory
libvmemcache
Volatile object cache
Persistent key-value store
High-level persistent application
Low-level persistent application

LIBVMEMCACHE

When To Use

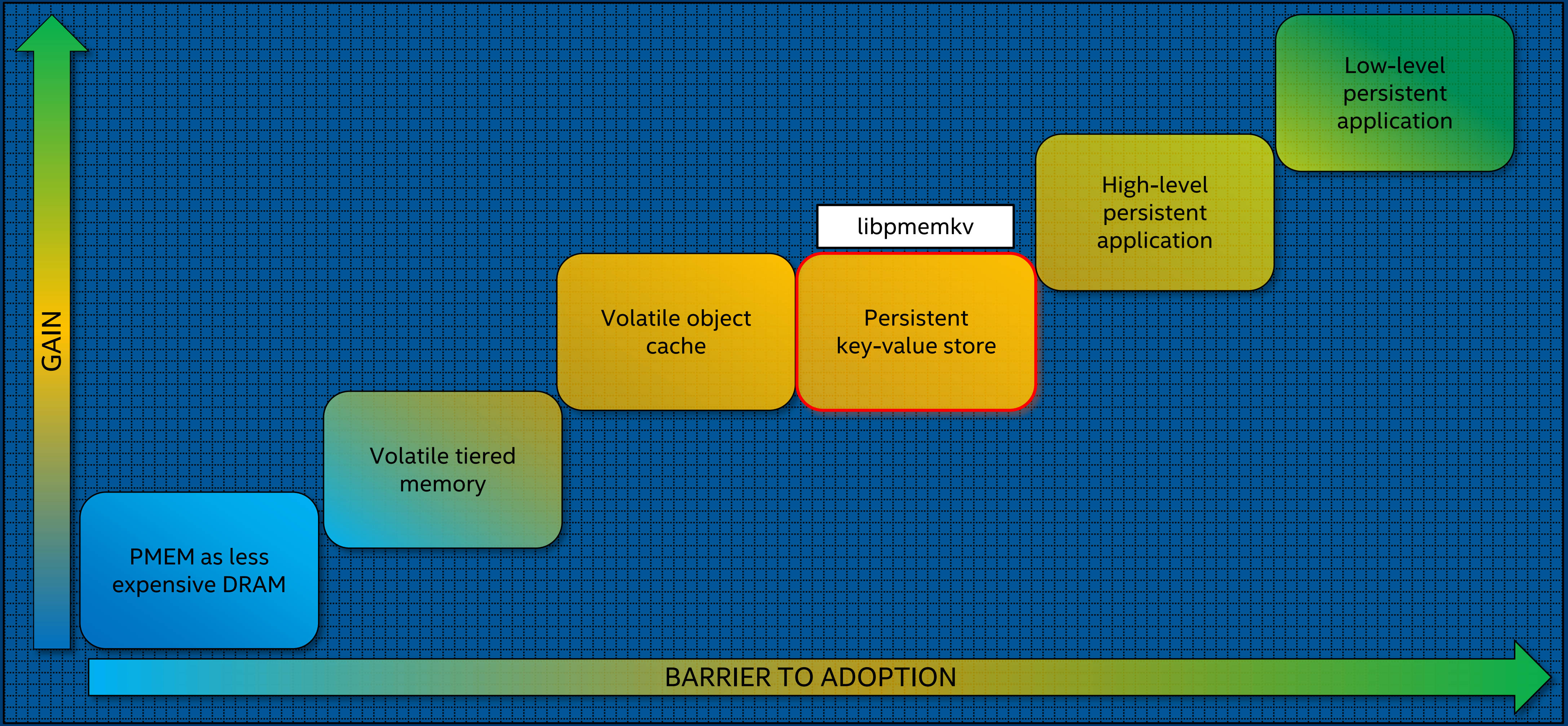
- caching large quantities of data
- low latency of operations is needed
- persistence is not required

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

```c
VMEMcache *cache = vmemcache_new();
vmemcache_add(cache, "/tmp");

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

DIFFERENT WAYS TO USE PERSISTENT MEMORY


GAIN
Low-level persistent application
High-level persistent application
libpmemkv
Persistent key-value store
Volatile object cache
Volatile tiered memory
PMEM as less expensive DRAM
BARRIER TO ADOPTION

LIBPMEMKV

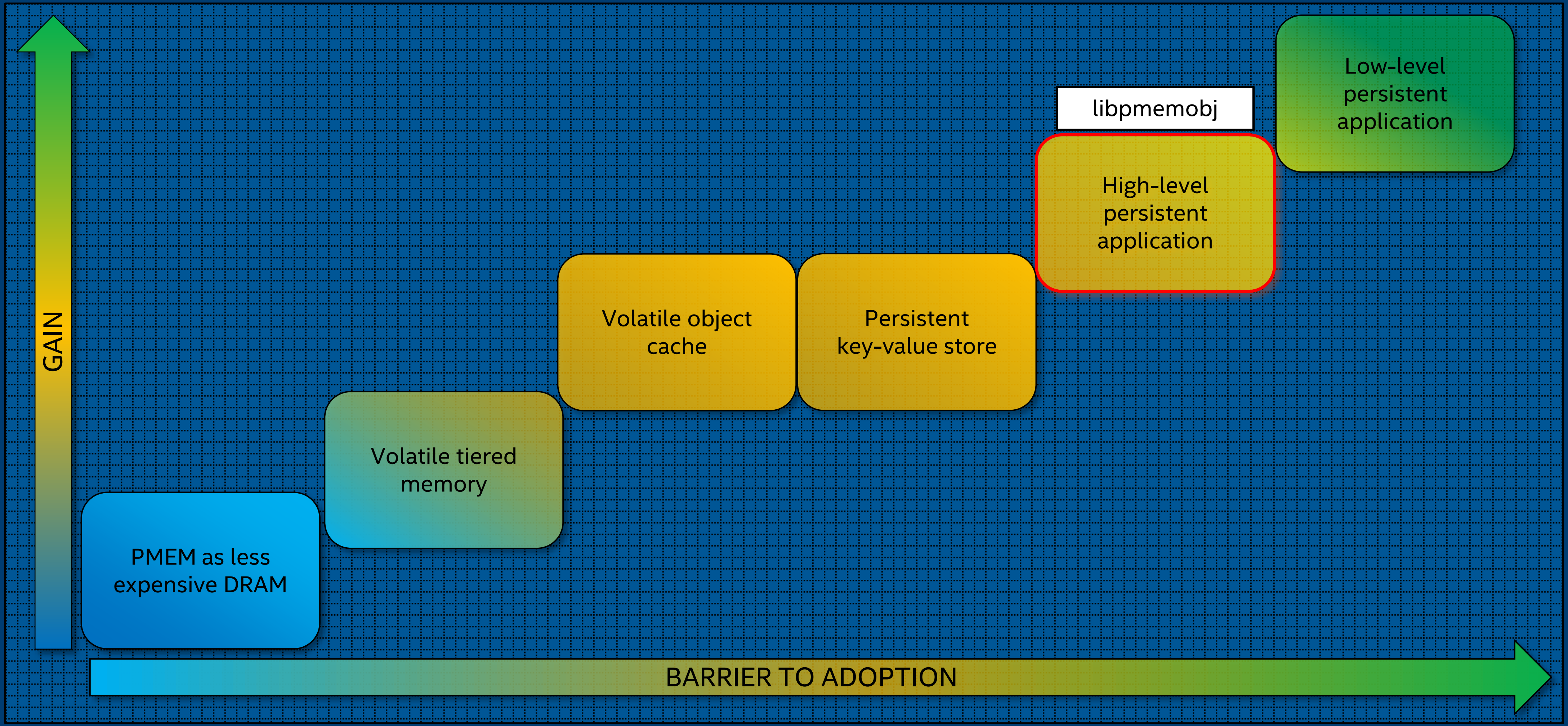
When To Use

- ➢ 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.

```
// add the given key-value pair
if (kv->put(argv[2], argv[3]) != status::OK) {
    cerr << db::errormsg() << endl;
    exit(1);
}
// lookup the given key and print the value
auto ret = kv->get(argv[2], [&](string_view value) {
    cout << argv[2] << "=\"" << value.data() << "\"" << endl;
});
if (ret != status::OK) {
    cerr << db::errormsg() << endl;
    exit(1);
}
```

DIFFERENT WAYS TO USE PERSISTENT MEMORY



GAIN
PMEM as less expensive DRAM
Volatile tiered memory
Volatile object cache
Persistent key-value store
libpmemobj
High-level persistent application
Low-level persistent application
BARRIER TO ADOPTION

LIBPMEMOBJ

When To Use

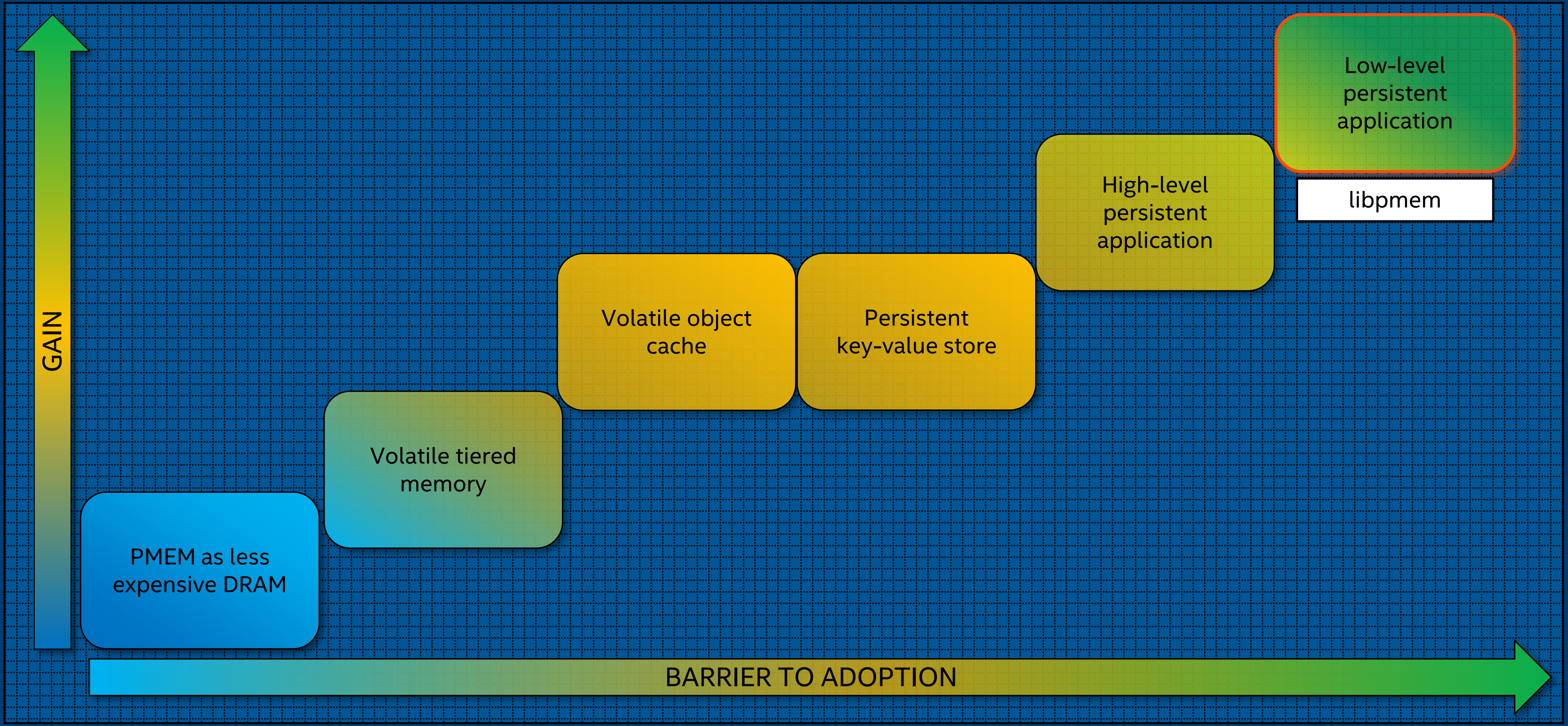
- direct byte-level access to objects is needed
- using custom storage-layer algorithms
- persistence is required

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

```c
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

DIFFERENT WAYS TO USE PERSISTENT MEMORY



GAIN
PMEM as less expensive DRAM
Volatile tiered memory
Volatile object cache
Persistent key-value store
High-level persistent application
Low-level persistent application
libpmem
BARRIER TO ADOPTION

LIBPMEM

When To Use

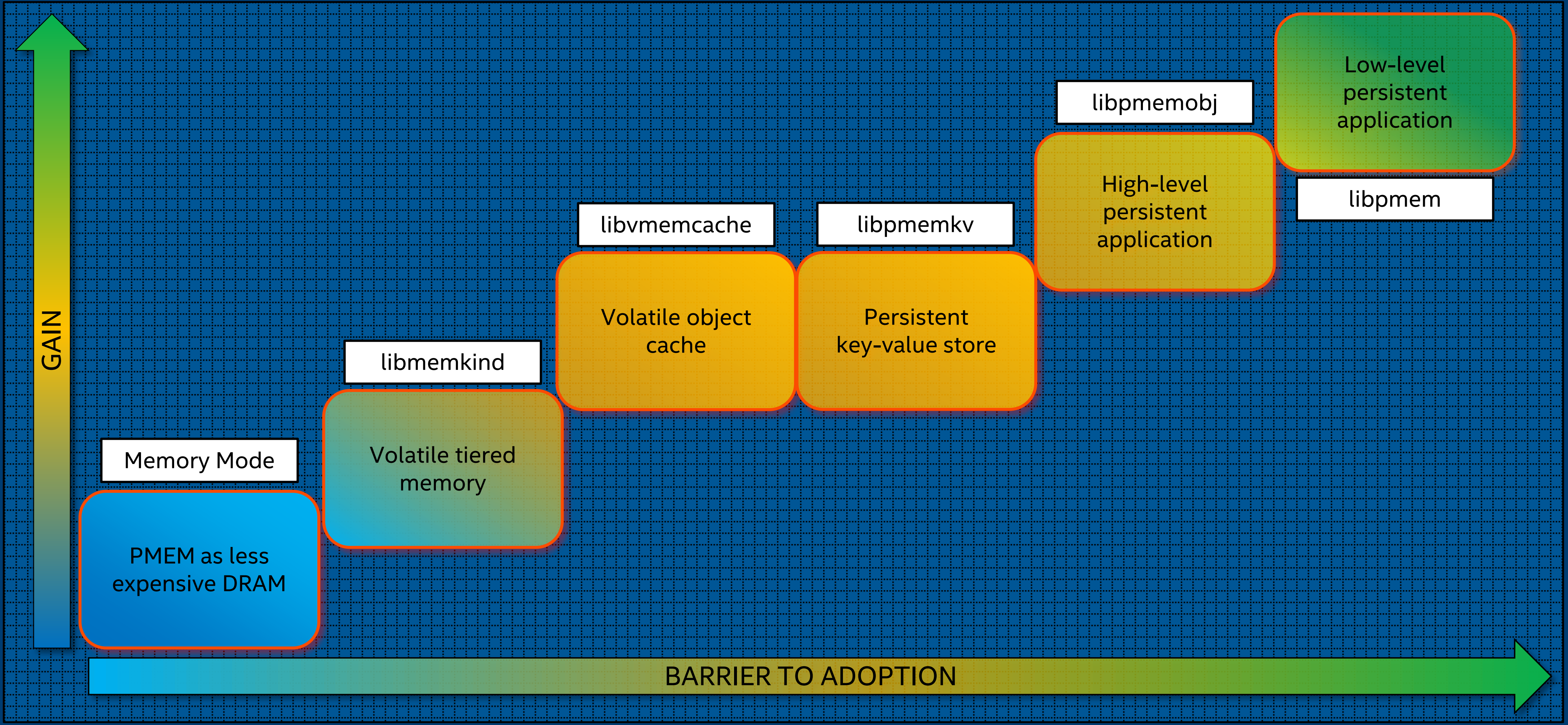
- modifying application that already uses memory mapped I/O
- other libraries are too high-level
- only need low-level PMEM-optimized primitives (memcpy etc)

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

```c
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);
```

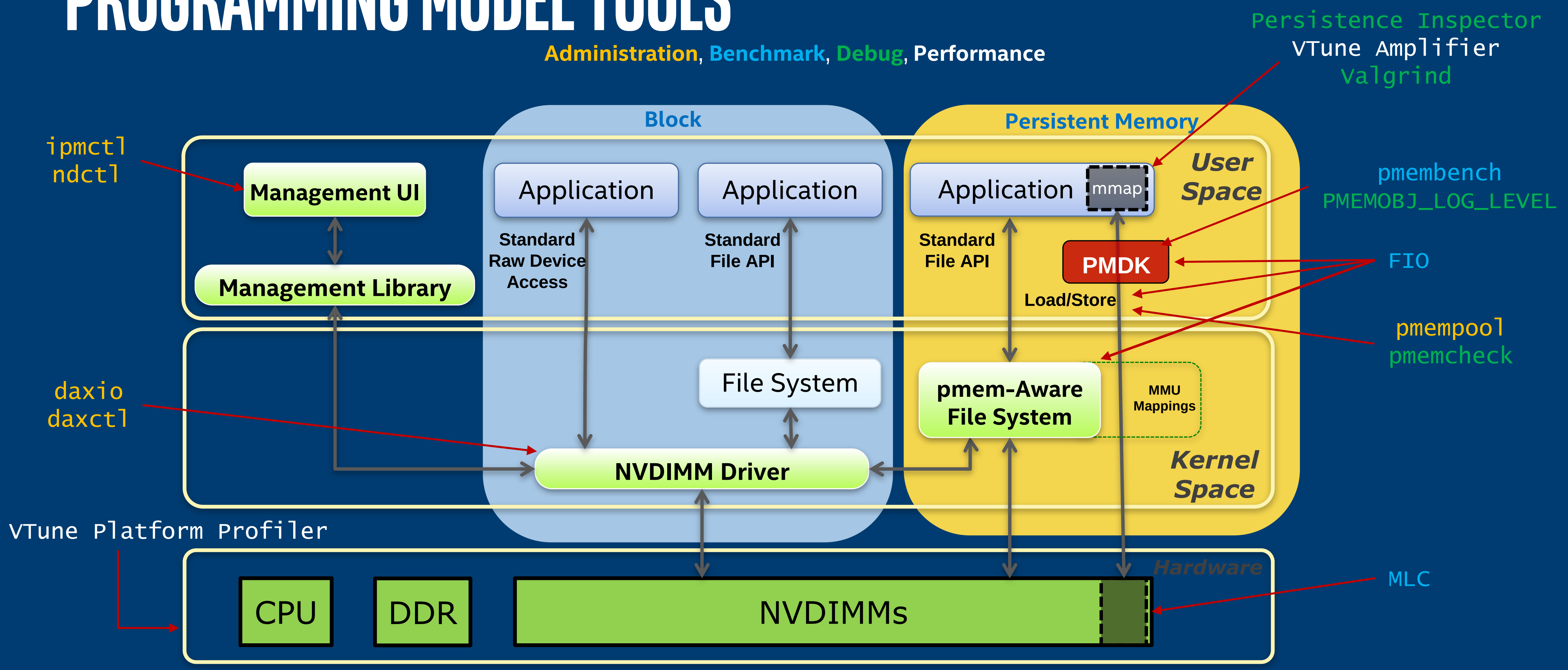
- The very basics needed for PMEM programming

DIFFERENT WAYS TO USE PERSISTENT MEMORY



PROGRAMMING MODEL TOOLS

Administration, Benchmark, Debug, Performance

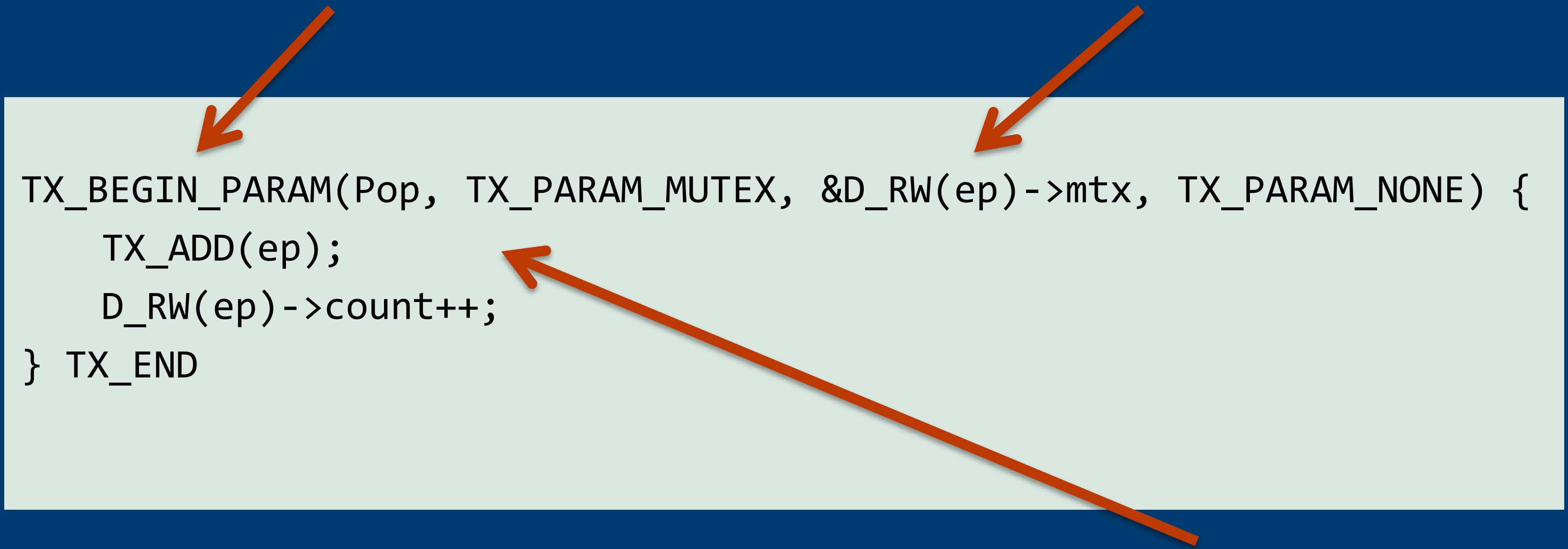


C PROGRAMMING WITH LIBPMEMOBJ

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



Powerfail Atomicity

Multi-Thread Atomicity

```
TX_BEGIN_PARAM(Pop, TX_PARAM_MUTEX, &D_RW(ep)->mtx, TX_PARAM_NONE) {
    TX_ADD(ep);
    D_RW(ep)->count++;
} TX_END
```

Caller must instrument code for undo logging

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

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)

Q&A

LINKS TO MORE INFORMATION

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®