Storage Systems (StoSys) XM_0092

Lecture 6: Byte-Addressable Persistent Memories

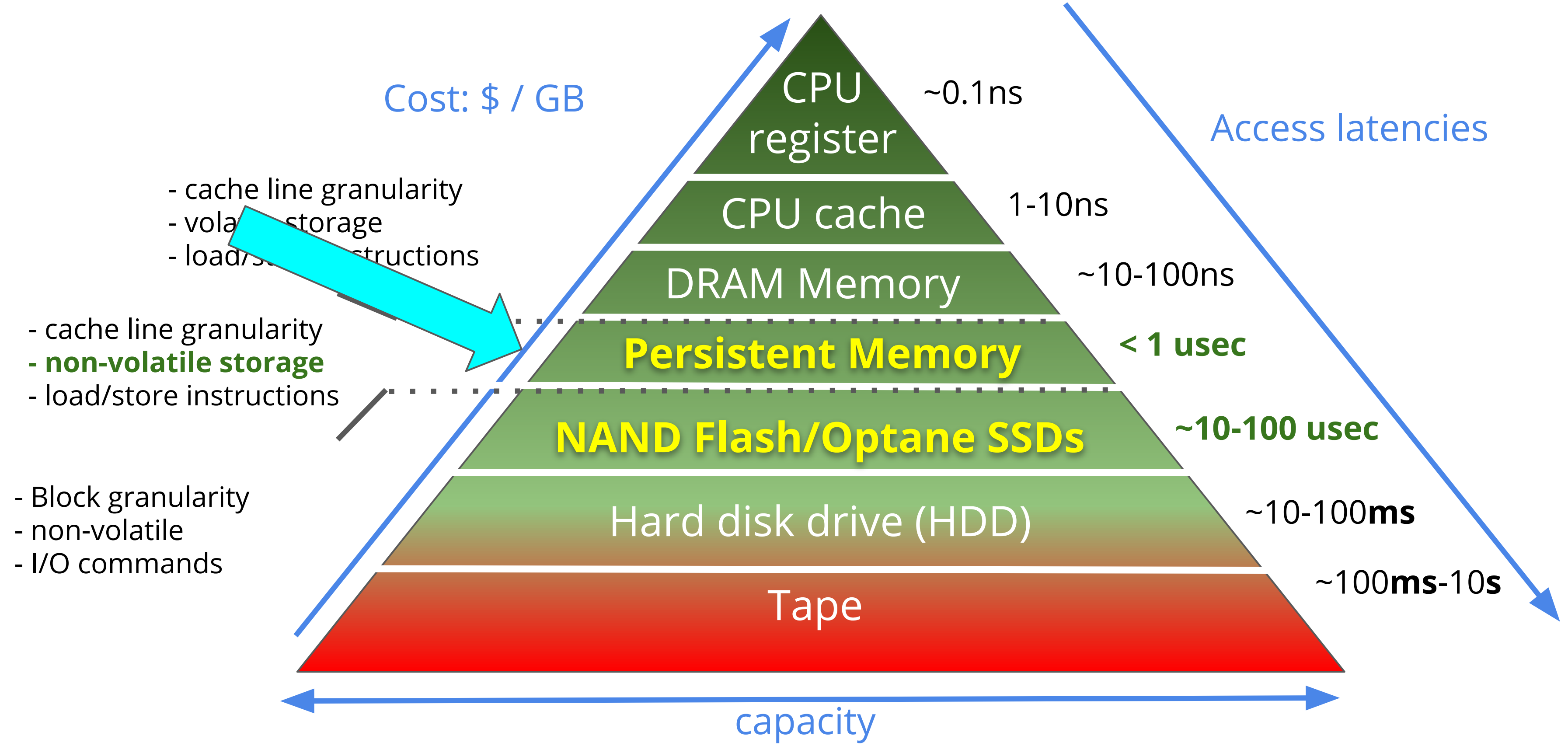
Animesh Trivedi
Autumn 2020, Period 2

VRIJE UNIVERSITEIT AMSTERDAM

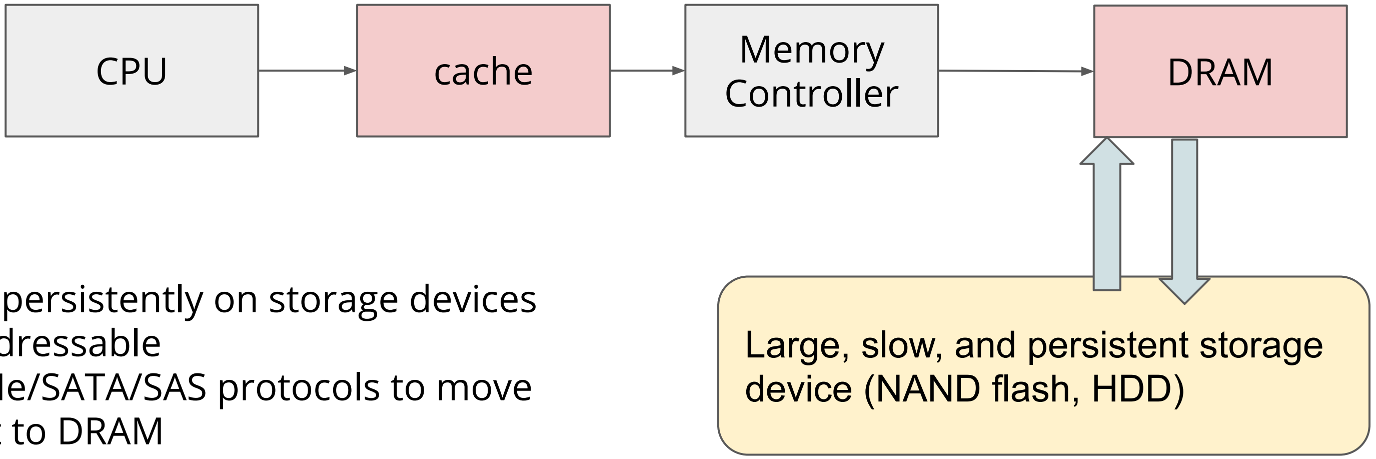
Syllabus outline

1. ~~Welcome and introduction to NVM (today)~~
2. ~~Host interfacing and software implications~~
3. ~~Flash Translation Layer (FTL) and Garbage Collection (GC)~~
4. ~~NVM Block Storage File systems~~
5. ~~NVM Block Storage Key-Value Stores~~
6. Emerging Byte-addressable Storage ⇐
7. Networked NVM Storage
8. Trends: Specialization and Programmability
9. Distributed Storage / Systems - I
10. Distributed Storage / Systems - II

The (new) triangle of storage hierarchy



Basic Model

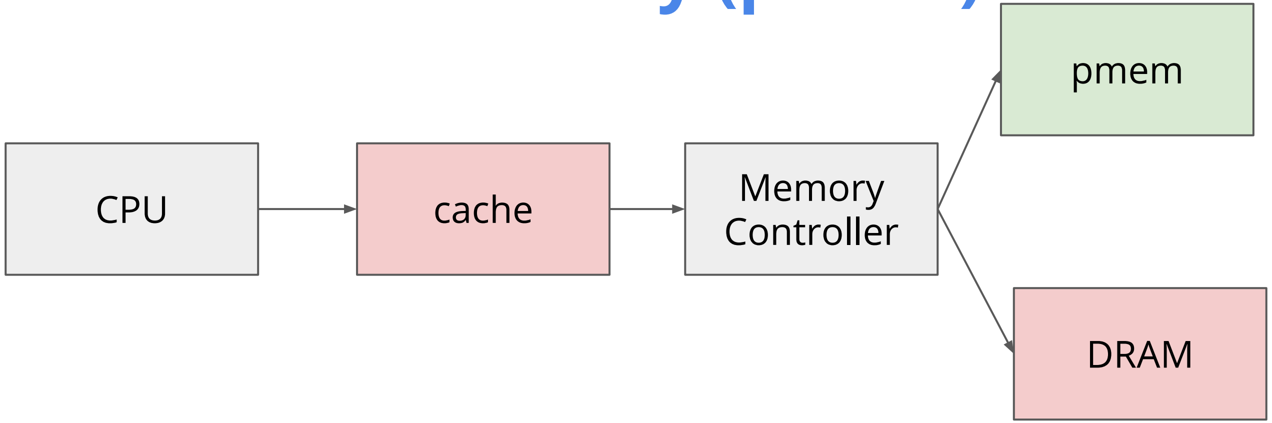

CPU
cache
Memory Controller
DRAM
Large, slow, and persistent storage device (NAND flash, HDD)

Data is stored persistently on storage devices

- Block addressable
- Use NVMe/SATA/SAS protocols to move data first to DRAM
- CPU can _only_ access data from DRAM
- To make data persistent write out again to the storage

2-level of storage: **Memory** (fast, byte-addressable, small, volatile) and
Storage (slower, block-addressed, large, and non-volatile)

NVM as Persistent Memory (pmem)



Persistent Memories (pmem) have been the holy grail of memory hierarchies

Ideally: performance close to DRAM, but persistent

We have been anticipating these memories from many years, and hence, continued to do research in “software” architectures before they arrived
→ *However, in this lecture we try to cover the latest research on these devices*

Today: Intel Optane

Released in **2019** (latest and greatest piece of storage technology today)

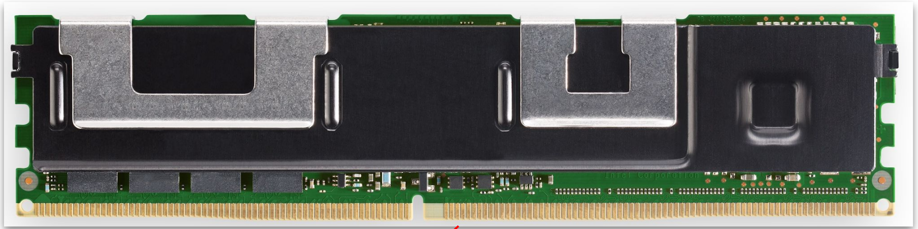
It is a byte-addressable, load-store accessible (from the CPU) storage that can be put in a DDR4 DIMM slot (uses the same mechanical and electrical protocols)

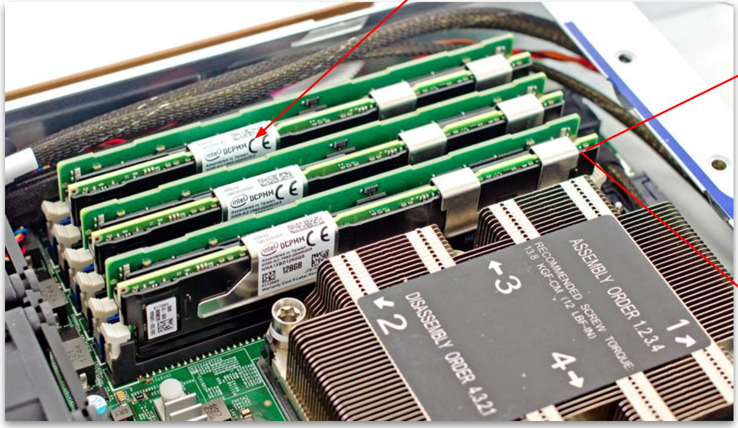
In comparison to DRAM

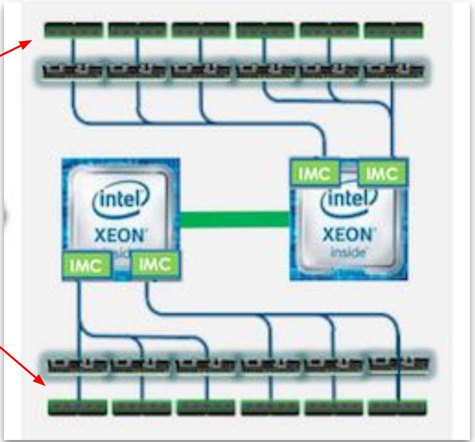
- **More capacity :** 128, 256, and 512 GB DIMMs (DRAMs are usually at 32-64GB then they get super expensive)
- **Cheaper :** than the DRAM (2-4x) times, but more expensive than Flash (10-100x)
- **Energy Efficient :** Unlike DRAM, no need to constantly refresh

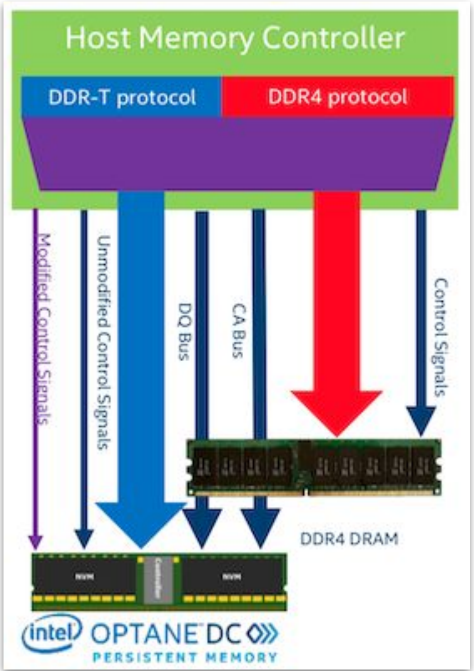
Btw - be ready to refresh basic ideas in computer architecture now :)

Today: Intel Optane







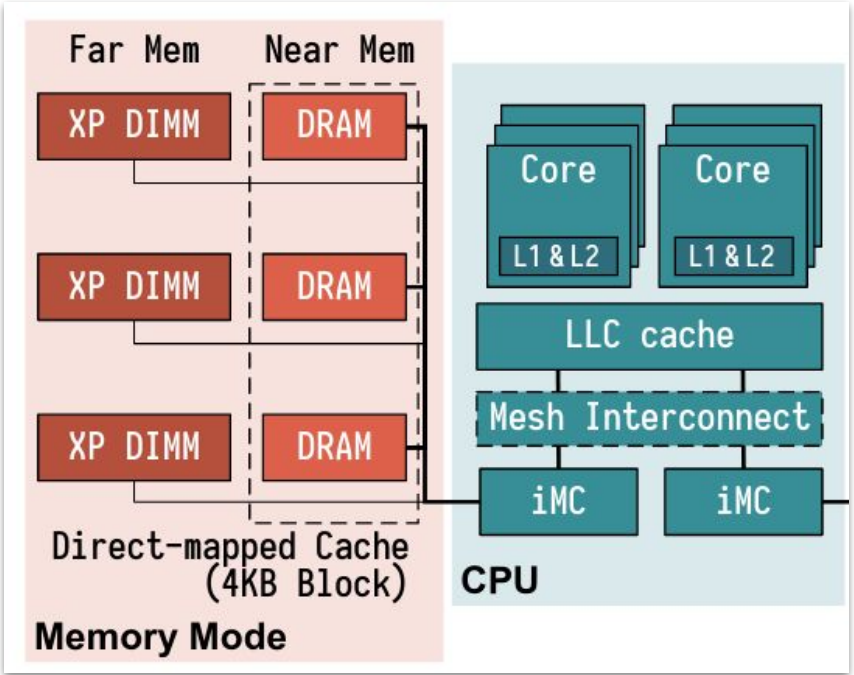


Intel Optane DC Persistent Memory Module (PMM), https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm

Optane Memory Layout and Operation Modes

Memory Mode: Optane behaves as a large (slower) DRAM, thus not leveraging its persistent qualities

- DRAM is used as a cache in front of XP DIMM
- Good for applications with needs for large DRAM



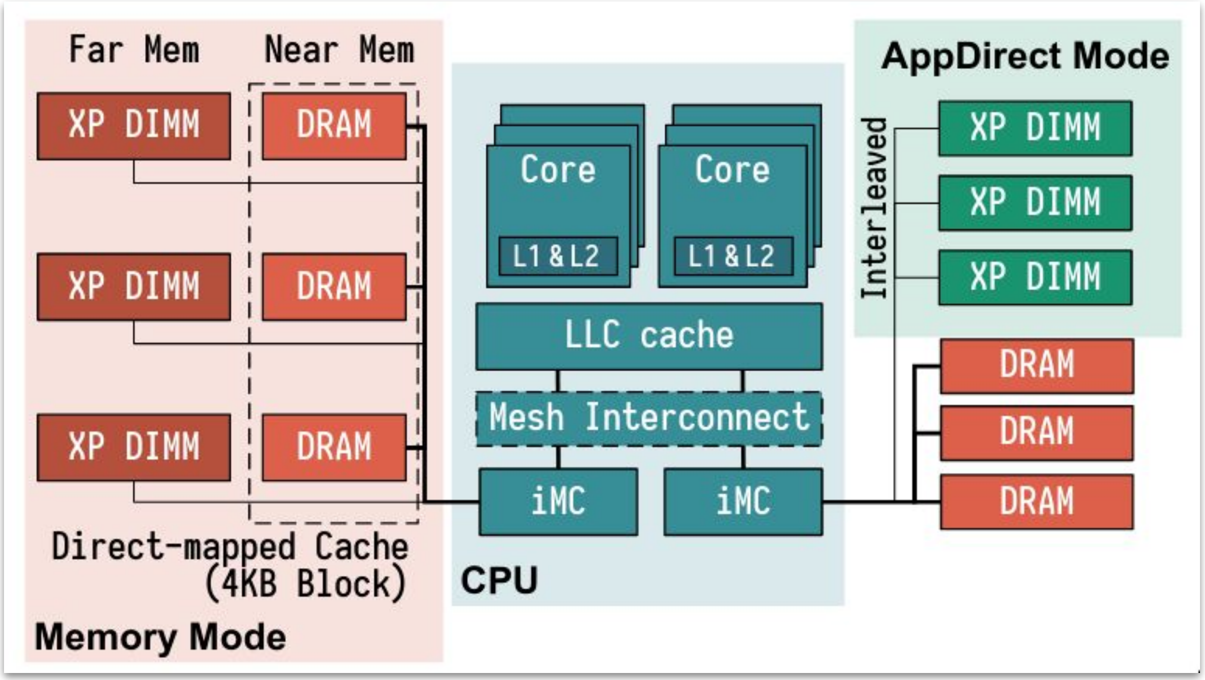
Optane Memory Layout and Operation Modes

Memory Mode: Optane behaves as a large (slower) DRAM, thus not leveraging its persistent qualities

- DRAM is used as a cache in front of XP DIMM
- Good for applications with needs for large DRAM

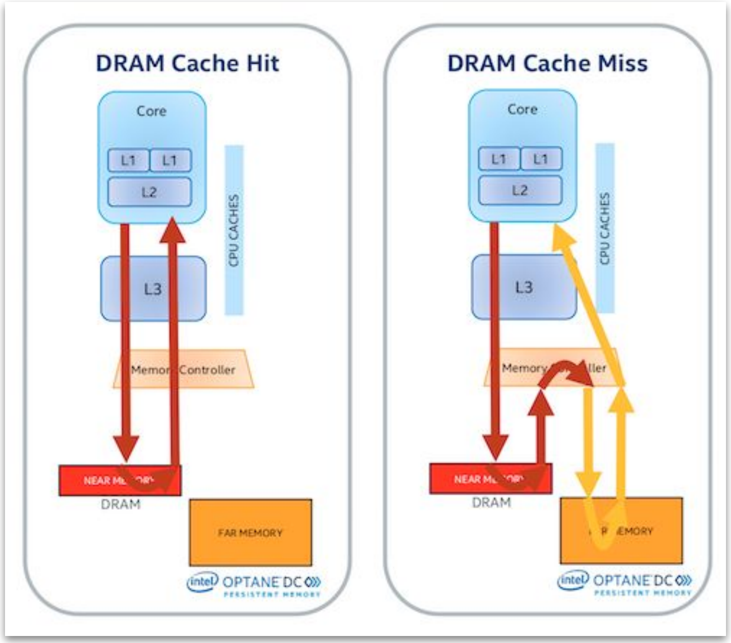
AppDirect Mode: Optane is used as a persistent memory and exposed to the OS/application

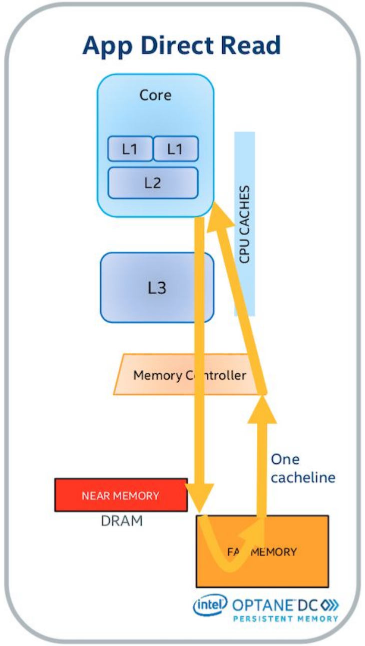
- Applications should be aware of its performance and persistence properties



Optane Memory Layout and Operation Modes

Two modes: **Memory Mode** and **App Direct Mode**

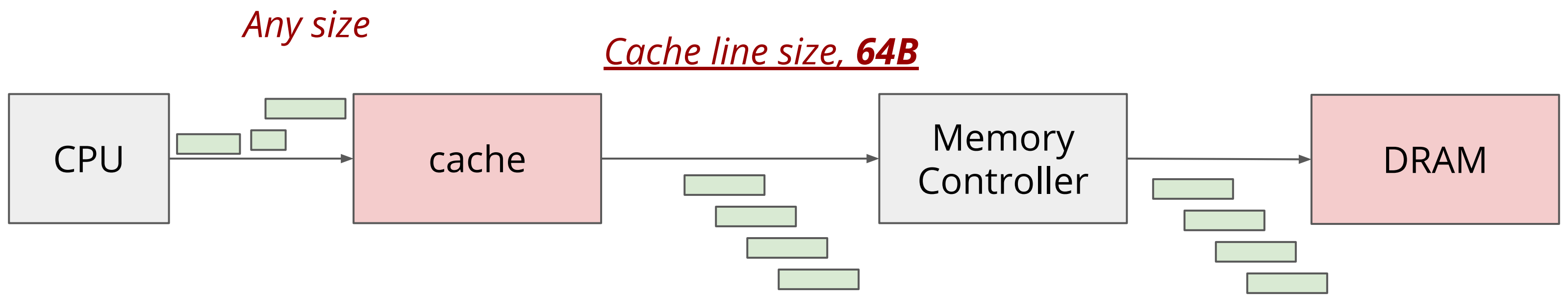




Potential challenges?

https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm

How Does the Current CPU Work? (simplified)



All CPU load and store accesses go to the cache

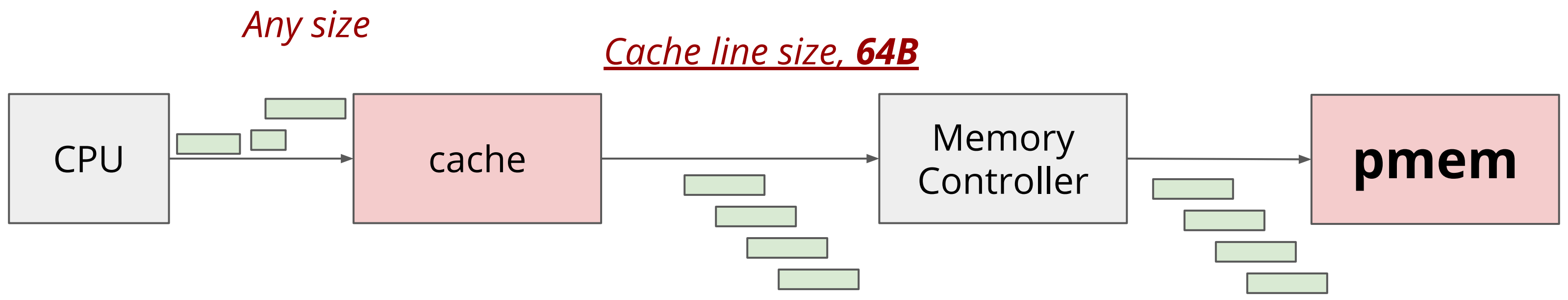
- **Cache Hit**: data is immediately transferred to the CPU
- **Cache Miss**: data is fetched from DRAM into the cache, and then transferred to the CPU

Caches are always managed in the cache line granularity (64B), and this is also the unit of DRAM access (*so, is DRAM truly a byte-addressable memory?*)

Memory controller can reorder loads and stores (out of order execution), hence, there is no guarantees in which order instruction get to DRAM

***Question 1:** How can a (i) CPU make sure that data is always flushed/pushed to DRAM; (ii) ordering?*
***Question 2:** Why are these concerns important?*

How Does the Current CPU Work? (simplified)

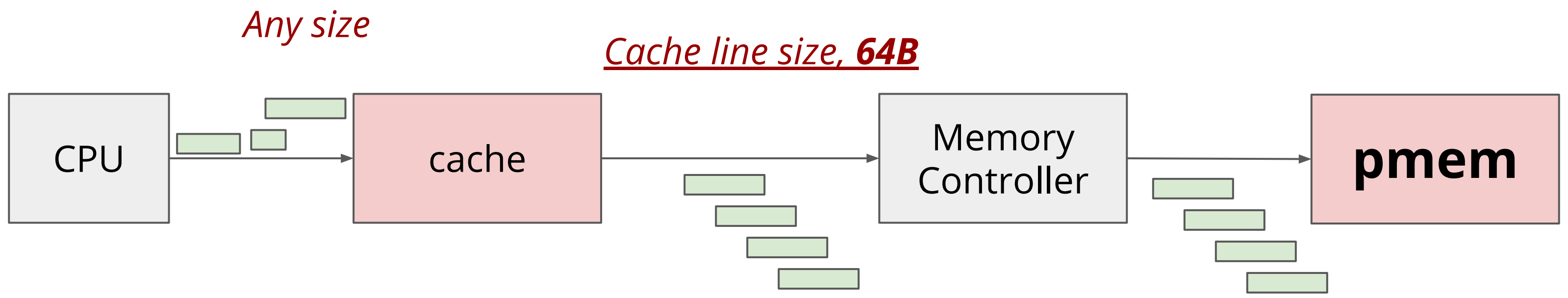


Now instead of DRAM, there is persistent memory

1. What if my writes are only stored in the cache?
2. Do you program cache? Isn't cache suppose to be a micro-architecture, programmer invisible CPU feature
3. What if my writes to pmem are-reordered?

Question 1: *How can a (i) CPU make sure that data is always flushed/pushed to PMEM; (ii) ordering?*
Question 2: *Why are these concerns important?*

How Does the Current CPU Work? (simplified)



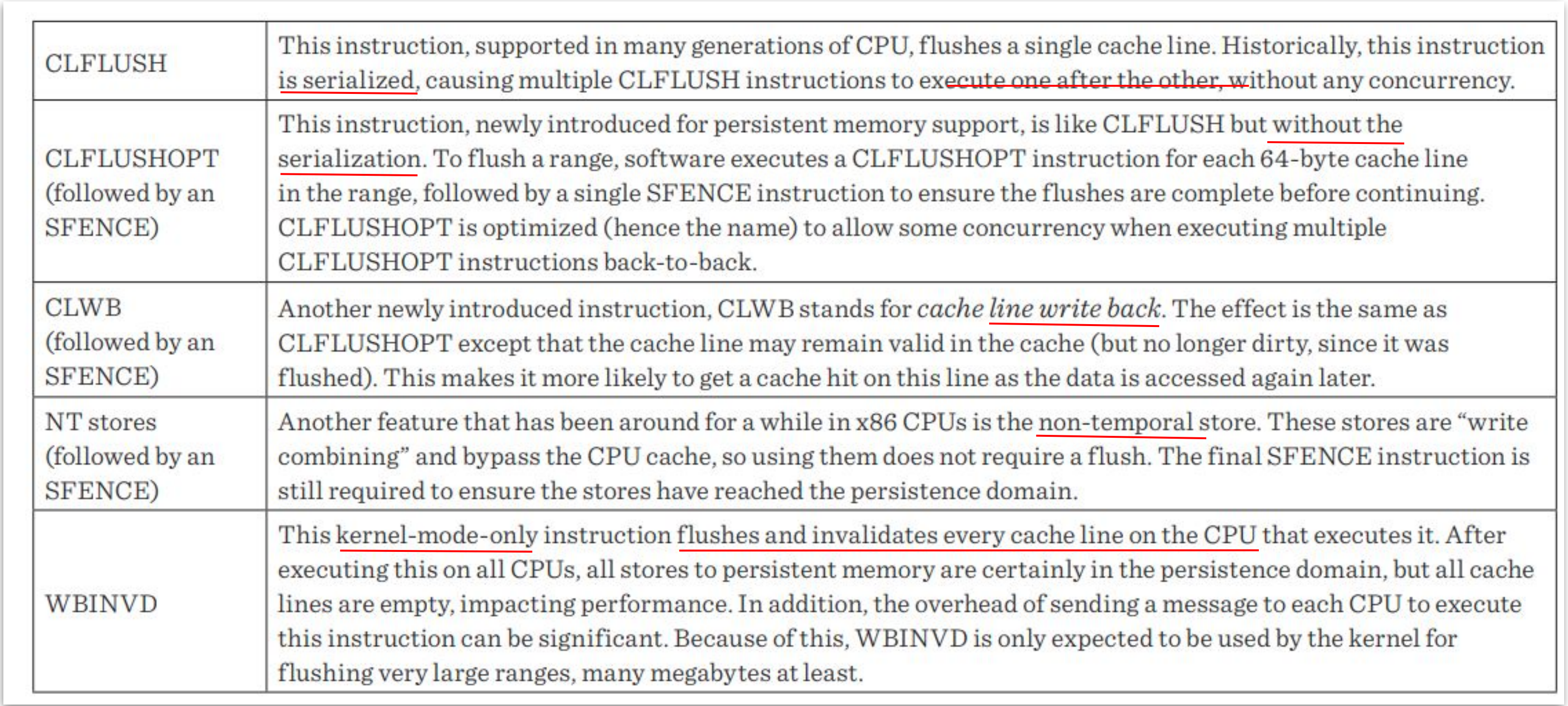
Special instructions available on modern CPUs

1. Non-temporal instructions, e.g., `movnta`,`movntadqa` (bypasses the CPU cache)
2. Explicitly flush cache lines (`clflush, clflushopt, clwb`)

Further use `sfence` to ensure all writes are globally visible and flushed

1. The Significance of the x86 SFENCE Instruction, https://hadibrais.wordpress.com/2019/02/26/the-significance-of-the-x86-sfence-instruction/
2. Memory part 5: What programmers can do https://lwn.net/Articles/255364/
3. https://en.wikipedia.org/wiki/X86_instruction_listings
4. https://stackoverflow.com/questions/40096894/do-current-x86-architectures-support-non-temporal-loads-from-normal-memory

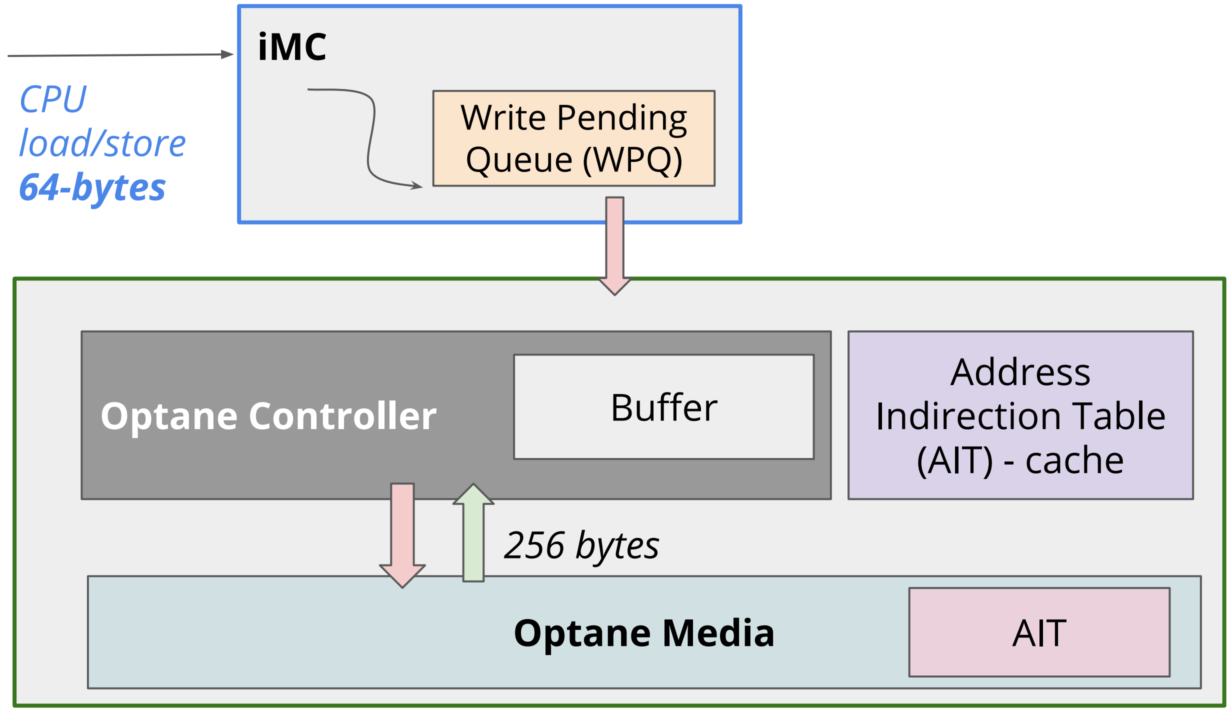
CPU Instructions



CLFLUSH	This instruction, supported in many generations of CPU, flushes a single cache line. Historically, this instruction is serialized, causing multiple CLFLUSH instructions to execute one after the other, without any concurrency.
CLFLUSHOPT (followed by an SFENCE)	This instruction, newly introduced for persistent memory support, is like CLFLUSH but without the serialization. To flush a range, software executes a CLFLUSHOPT instruction for each 64-byte cache line in the range, followed by a single SFENCE instruction to ensure the flushes are complete before continuing. CLFLUSHOPT is optimized (hence the name) to allow some concurrency when executing multiple CLFLUSHOPT instructions back-to-back.
CLWB (followed by an SFENCE)	Another newly introduced instruction, CLWB stands for *cache line write back*. The effect is the same as CLFLUSHOPT except that the cache line may remain valid in the cache (but no longer dirty, since it was flushed). This makes it more likely to get a cache hit on this line as the data is accessed again later.
NT stores (followed by an SFENCE)	Another feature that has been around for a while in x86 CPUs is the non-temporal store. These stores are "write combining" and bypass the CPU cache, so using them does not require a flush. The final SFENCE instruction is still required to ensure the stores have reached the persistence domain.
WBINVD	This kernel-mode-only instruction flushes and invalidates every cache line on the CPU that executes it. After executing this on all CPUs, all stores to persistent memory are certainly in the persistence domain, but all cache lines are empty, impacting performance. In addition, the overhead of sending a message to each CPU to execute this instruction can be significant. Because of this, WBINVD is only expected to be used by the kernel for flushing very large ranges, many megabytes at least.

Once flushed, data will move to the memory controller …

Optane Internals - the Write Path


iMC
CPU load/store **64-bytes**
Write Pending Queue (WPQ)
Optane Controller
Buffer
Address Indirection Table (AIT) - cache
256 bytes
Optane Media
AIT

Writes end up in iMC, at WPQ

Then flushed into Optane DIMM

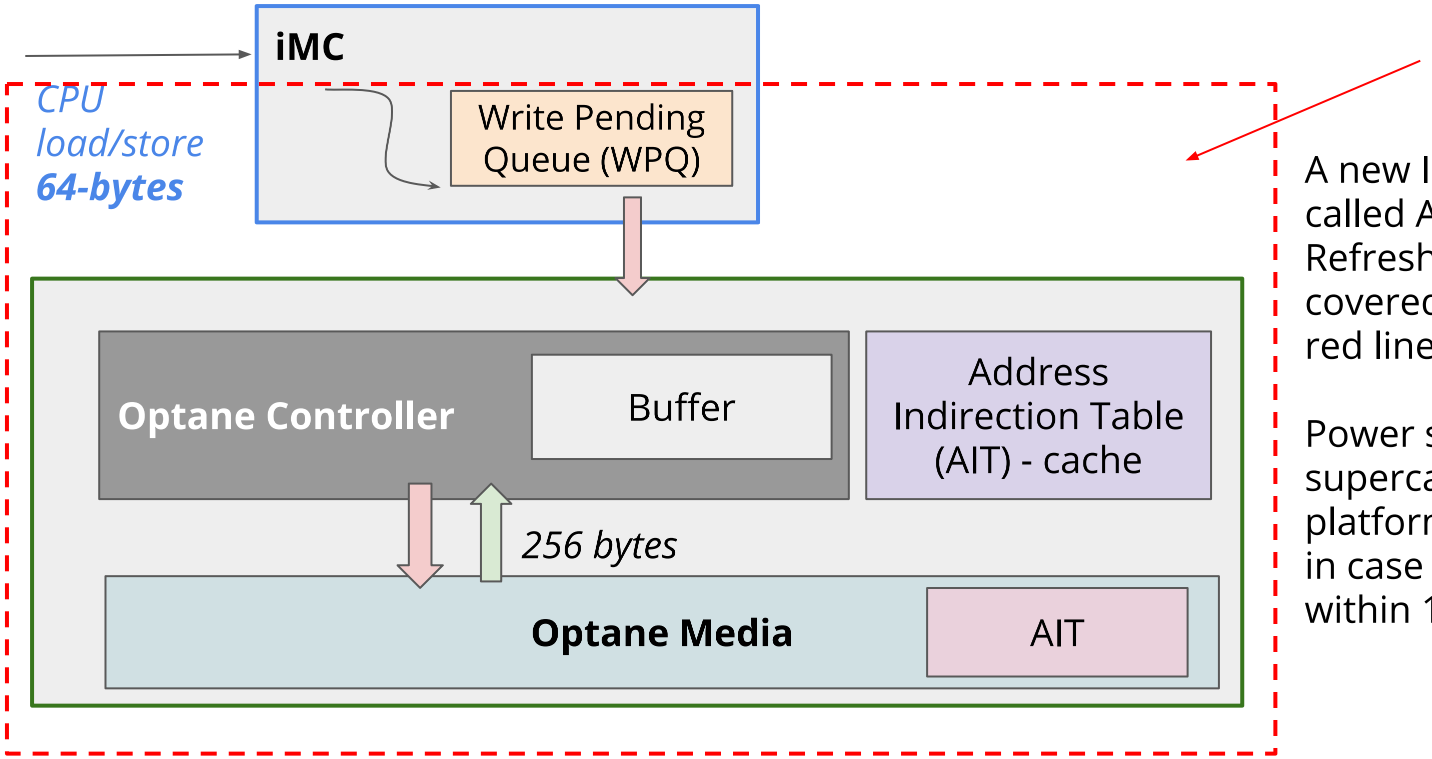
Optane DIMM has a write buffer, where 64 bytes r/w are merged into 256 bytes accesses to Optane

There is indirection table mapping and its cache

The Optane controller runs the logic. *Many of the Optane details are secret*

How do we make sure that data is not lost in the case of a power cut?

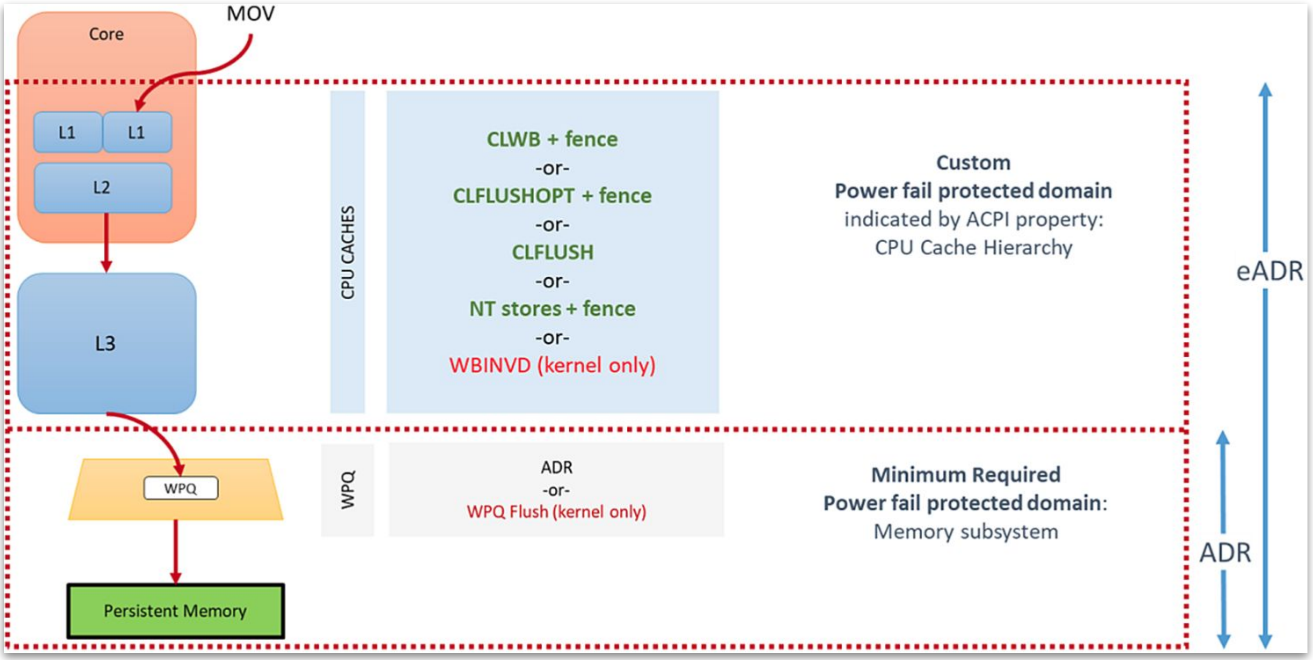
Optane Internals - the Write Path



A new Intel platform feature called Asynchronous DRAM Refresh **(ADR)** domain (area covered inside the dotted red line)

Power storage (battery, supercapacitor) on the platform to ensure writeback in case of a failure (typically within 100 usec)

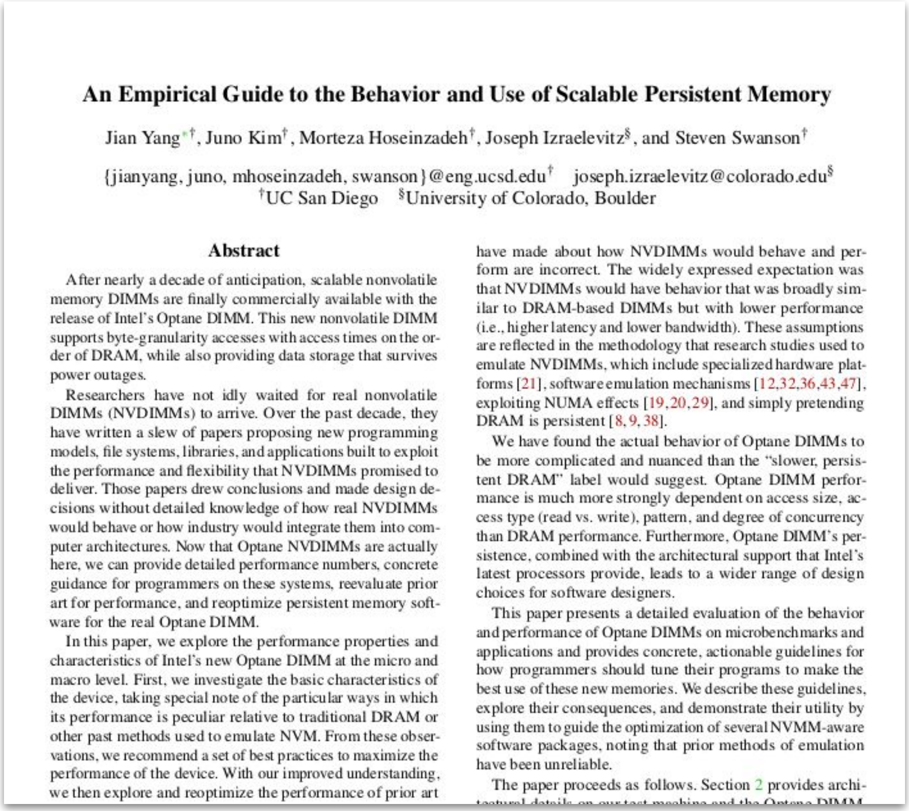
ADR and eADR - Bringing Persistency to the whole CPU



Optionally eADR available with the 3rd generation of Xeon processors (CopperLake, 2020)

Build Persistent Memory Applications with Reliability Availability and Serviceability, https://software.intel.com/content/www/us/en/develop/articles/build-pmem-apps-with-ras.html
Third Generation Intel® Xeon® Processor Scalable Family Technical Overview, https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-overview.html

An Empirical Guide to the Behavior and Use of Scalable Persistent Memory (Feb, 2020)



An Empirical Guide to the Behavior and Use of Scalable Persistent Memory

Jian Yang*†, Juno Kim†, Morteza Hoseinzadeh†, Joseph Izraelevitz§, and Steven Swanson†

{jianyang, juno, mhoseinzadeh, swanson}@eng.ucsd.edu† joseph.izraelevitz@colorado.edu§
†UC San Diego §University of Colorado, Boulder

Abstract

After nearly a decade of anticipation, scalable nonvolatile memory DIMMs are finally commercially available with the release of Intel's Optane DIMM. This new nonvolatile DIMM supports byte-granularity accesses with access times on the order of DRAM, while also providing data storage that survives power outages.

Researchers have not idly waited for real nonvolatile DIMMs (NVDIMMs) to arrive. Over the past decade, they have written a slew of papers proposing new programming models, file systems, libraries, and applications built to exploit the performance and flexibility that NVDIMMs promised to deliver. Those papers drew conclusions and made design decisions without detailed knowledge of how real NVDIMMs would behave or how industry would integrate them into computer architectures. Now that Optane NVDIMMs are actually here, we can provide detailed performance numbers, concrete guidance for programmers on these systems, reevaluate prior art for performance, and reoptimize persistent memory software for the real Optane DIMM.

In this paper, we explore the performance properties and characteristics of Intel's new Optane DIMM at the micro and macro level. First, we investigate the basic characteristics of the device, taking special note of the particular ways in which its performance is peculiar relative to traditional DRAM or other past methods used to emulate NVM. From these observations, we recommend a set of best practices to maximize the performance of the device. With our improved understanding, we then explore and reoptimize the performance of prior art

have made about how NVDIMMs would behave and perform are incorrect. The widely expressed expectation was that NVDIMMs would have behavior that was broadly similar to DRAM-based DIMMs but with lower performance (i.e., higher latency and lower bandwidth). These assumptions are reflected in the methodology that research studies used to emulate NVDIMMs, which include specialized hardware platforms [21], software emulation mechanisms [12,32,36,43,47], exploiting NUMA effects [19,20,29], and simply pretending DRAM is persistent [8, 9, 38].

We have found the actual behavior of Optane DIMMs to be more complicated and nuanced than the "slower, persistent DRAM" label would suggest. Optane DIMM performance is much more strongly dependent on access size, access type (read vs. write), pattern, and degree of concurrency than DRAM performance. Furthermore, Optane DIMM's persistence, combined with the architectural support that Intel's latest processors provide, leads to a wider range of design choices for software designers.

This paper presents a detailed evaluation of the behavior and performance of Optane DIMMs on microbenchmarks and applications and provides concrete, actionable guidelines for how programmers should tune their programs to make the best use of these new memories. We describe these guidelines, explore their consequences, and demonstrate their utility by using them to guide the optimization of several NVMM-aware software packages, noting that prior methods of emulation have been unreliable.

The paper proceeds as follows. Section 2 provides archi-

System Setup

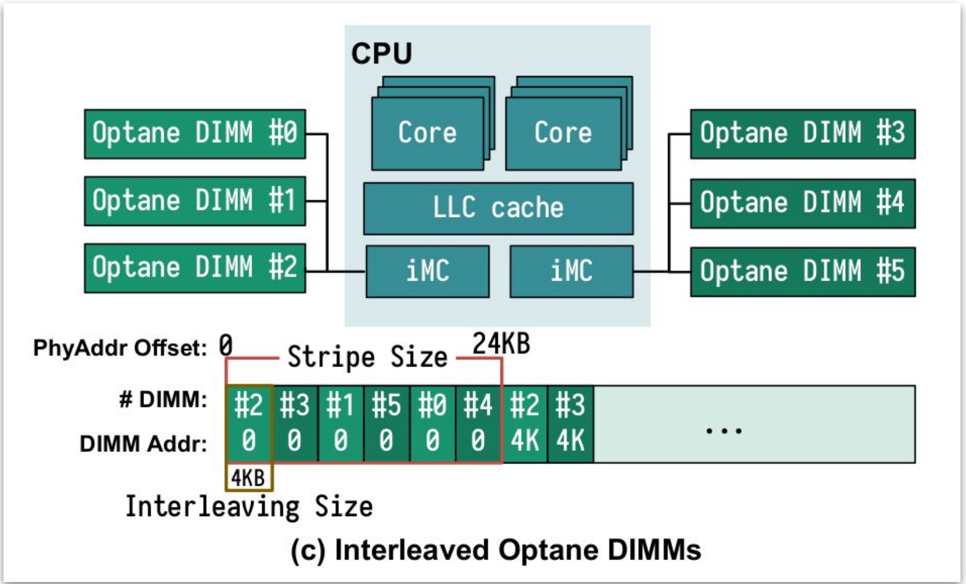
2 x CPU 24 cores Cascade Lake

Each CPU: 2 x iMC with 3 memory channels each

Total 6 channels for DRAM and Optane

2 CPU x 6 Ch. x 32 GB = **192 GB DRAM**

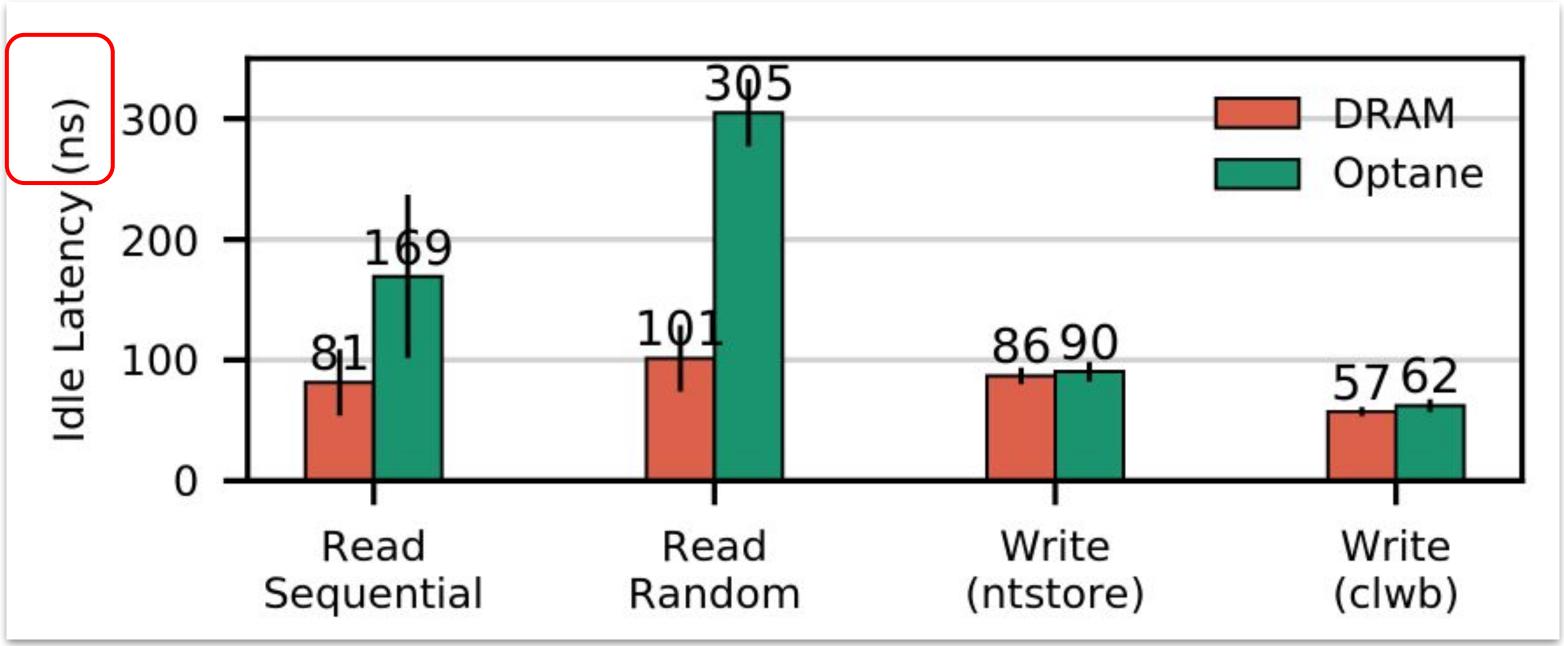
2 CPU x 6 Ch. x 256GB = **3TB Optane**



(c) Interleaved Optane DIMMs

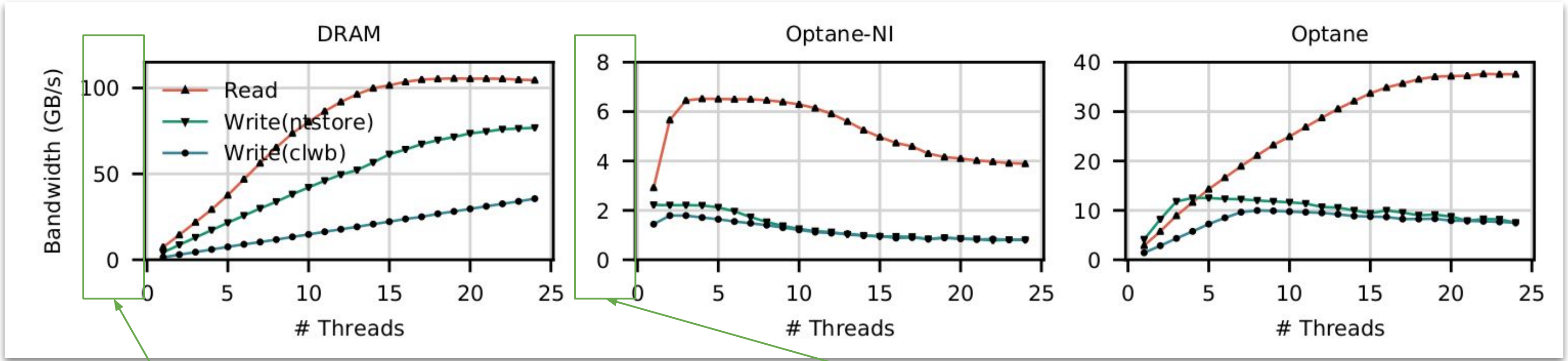
There are 2 of such CPUs

Basic Performance: Latency



- The read latency for Optane is 2-3x higher than DRAM
- The random-vs-sequential gap is 20% for DRAM but 80% for Optane memory
- Write performance measures write reaching the ADR domain (not necessarily Optane)

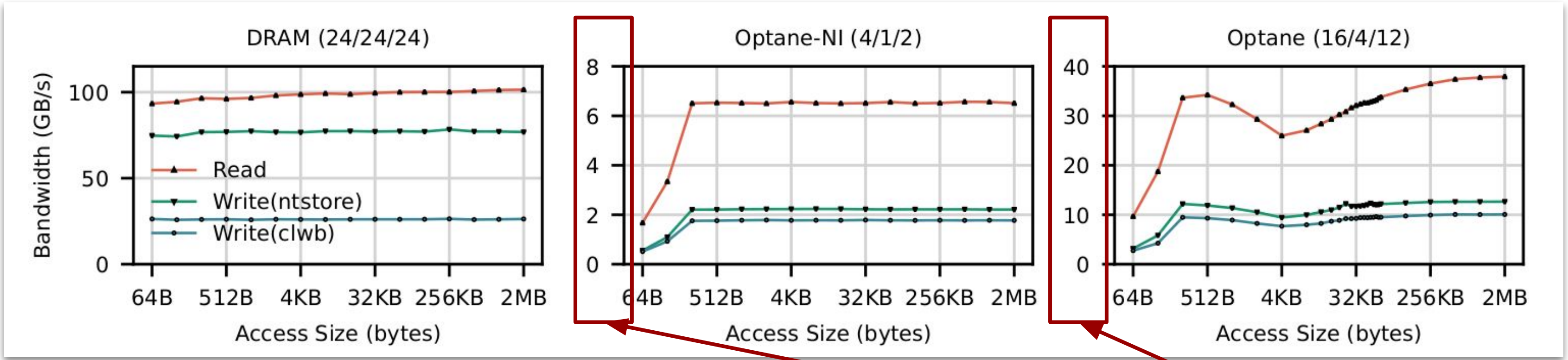
Optane Bandwidth Comparison - Scalability



- Peak DRAM bandwidth can be significantly higher than the Optane bandwidth
  - NI = non-interleaved (single Optane DIMM)

- Both scale nicely with the number of threads. Optane write performance dips as the content on the device increases. Interleaving helps with improved performance.

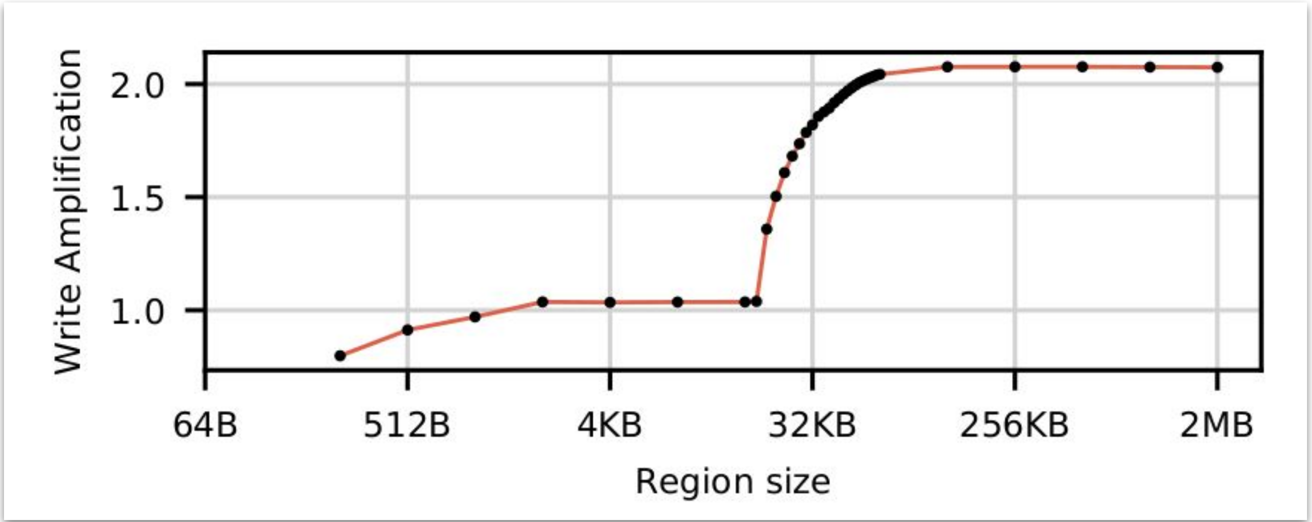
Optane Bandwidth Comparison - Size



- Larger gap between read/write performance in Optane than in DRAM
- Interleaving improves peak read and write bandwidth by ~5x (see y-axis)
- Optane bandwidth for random accesses under 256 B is poor

Recommendation - use 256 bytes aligned data structures and accesses

Detecting Optane Buffer Size

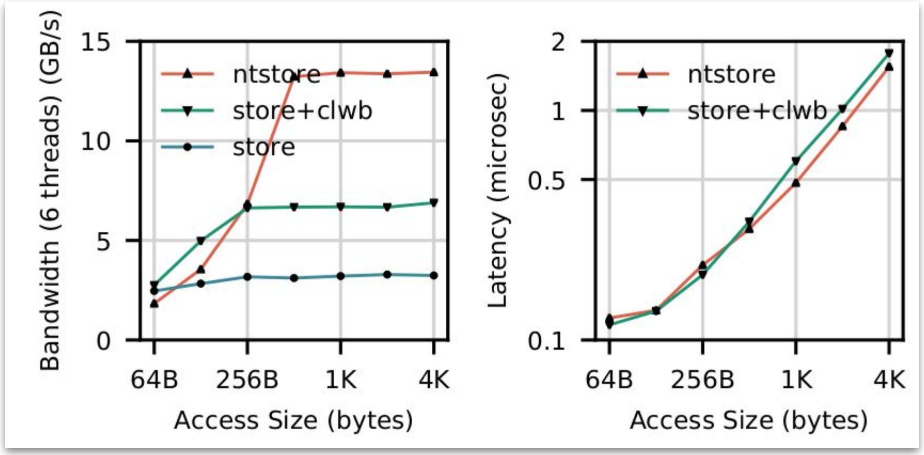


Write addresses repeatedly which are separated by certain XP Line size (256B)

Measure WA (DIMM counter), which shows at 16 lines, 16 x 256 = 16 KB buffer

Recommendation: Try to put related data items together in a buffer of 16kB

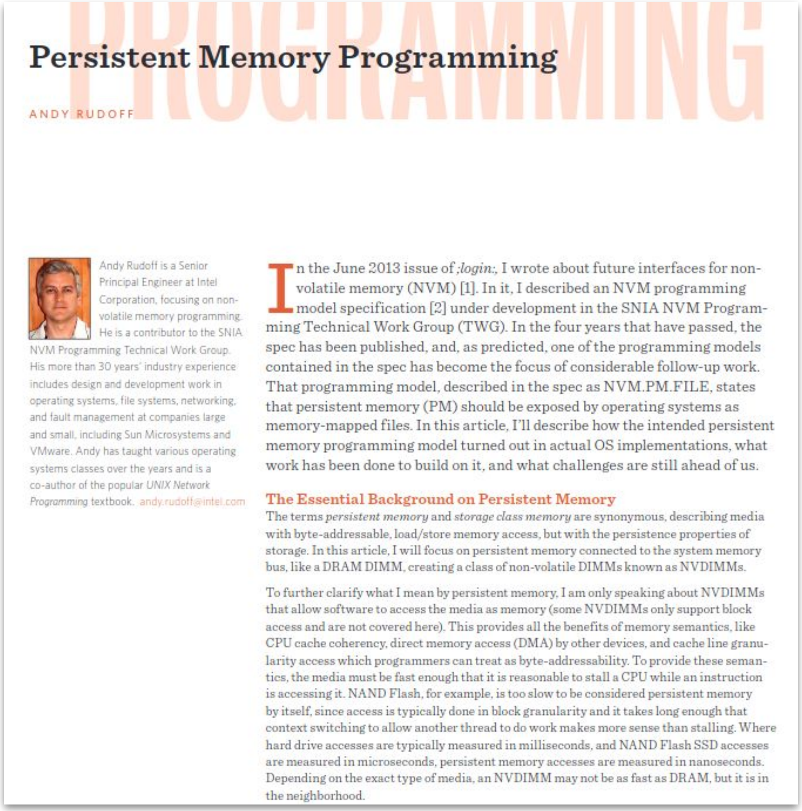
Which Write/Flush Mechanism to Use?



- Non-temporal instruction has better bandwidth (lower latency) for large accesses (because it does not bring cache line)
- For small accesses (<256B), clwb is fast

Recommendation: Based on what you are writing back, pick one - dynamic selection

Persistent Memory Programming (2017)



PROGRAMMING

Persistent Memory Programming

ANDY RUDOFF

Andy Rudoff is a Senior Principal Engineer at Intel Corporation, focusing on non-volatile memory programming. He is a contributor to the SNIA NVM Programming Technical Work Group. His more than 30 years' industry experience includes design and development work in operating systems, file systems, networking, and fault management at companies large and small, including Sun Microsystems and VMware. Andy has taught various operating systems classes over the years and is a co-author of the popular *UNIX Network Programming* textbook. andy.rudoff@intel.com

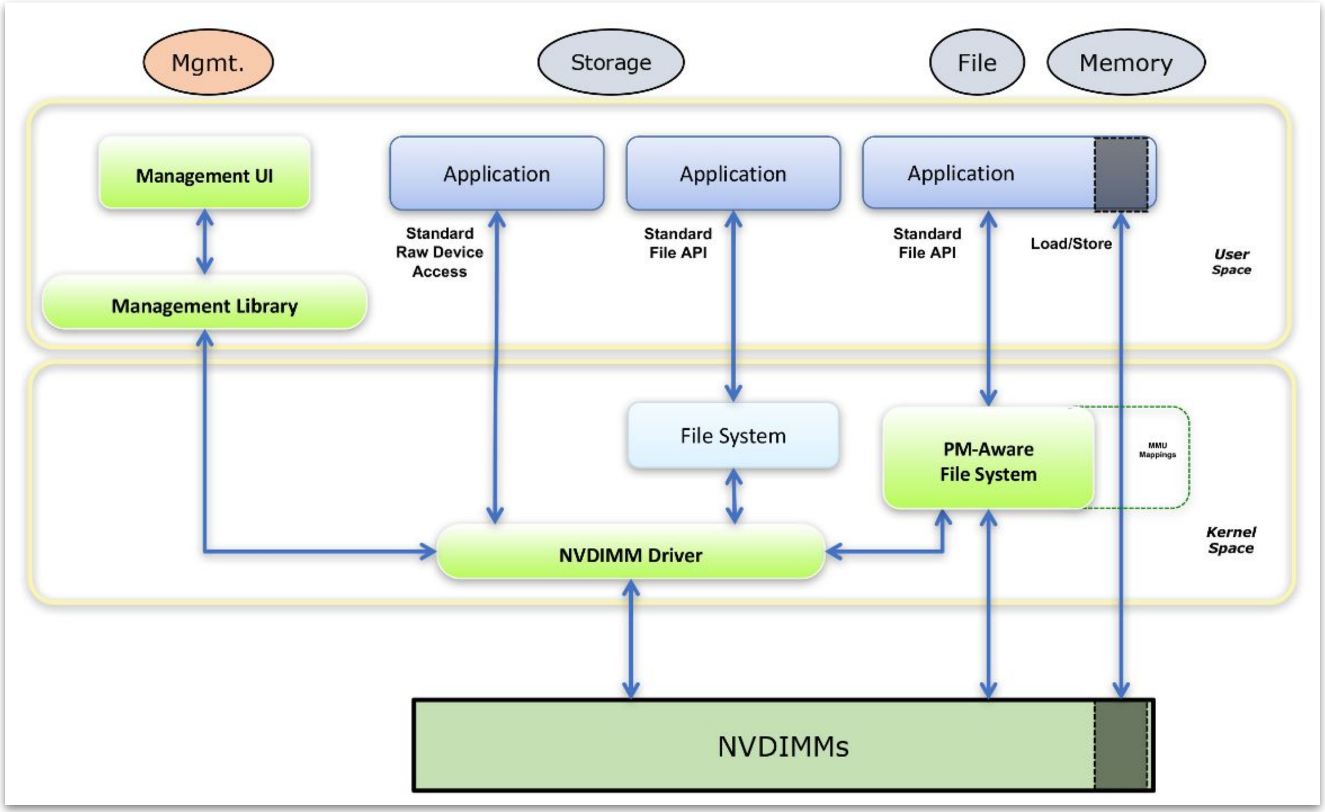
In the June 2013 issue of *;login:*, I wrote about future interfaces for non-volatile memory (NVM) [1]. In it, I described an NVM programming model specification [2] under development in the SNIA NVM Programming Technical Work Group (TWG). In the four years that have passed, the spec has been published, and, as predicted, one of the programming models contained in the spec has become the focus of considerable follow-up work. That programming model, described in the spec as NVM.PM.FILE, states that persistent memory (PM) should be exposed by operating systems as memory-mapped files. In this article, I'll describe how the intended persistent memory programming model turned out in actual OS implementations, what work has been done to build on it, and what challenges are still ahead of us.

The Essential Background on Persistent Memory

The terms *persistent memory* and *storage class memory* are synonymous, describing media with byte-addressable, load/store memory access, but with the persistence properties of storage. In this article, I will focus on persistent memory connected to the system memory bus, like a DRAM DIMM, creating a class of non-volatile DIMMs known as NVDIMMs.

To further clarify what I mean by persistent memory, I am only speaking about NVDIMMs that allow software to access the media as memory (some NVDIMMs only support block access and are not covered here). This provides all the benefits of memory semantics, like CPU cache coherency, direct memory access (DMA) by other devices, and cache line granularity access which programmers can treat as byte-addressability. To provide these semantics, the media must be fast enough that it is reasonable to stall a CPU while an instruction is accessing it. NAND Flash, for example, is too slow to be considered persistent memory by itself, since access is typically done in block granularity and it takes long enough that context switching to allow another thread to do work makes more sense than stalling. Where hard drive accesses are typically measured in milliseconds, and NAND Flash SSD accesses are measured in microseconds, persistent memory accesses are measured in nanoseconds. Depending on the exact type of media, an NVDIMM may not be as fast as DRAM, but it is in the neighborhood.

So, How Do You Program/Manage Your NVDIMMs?



I am going to use the term NVDIMM to refer to a general pmem technology not specifically to Optane

Understand: Storage and Memory

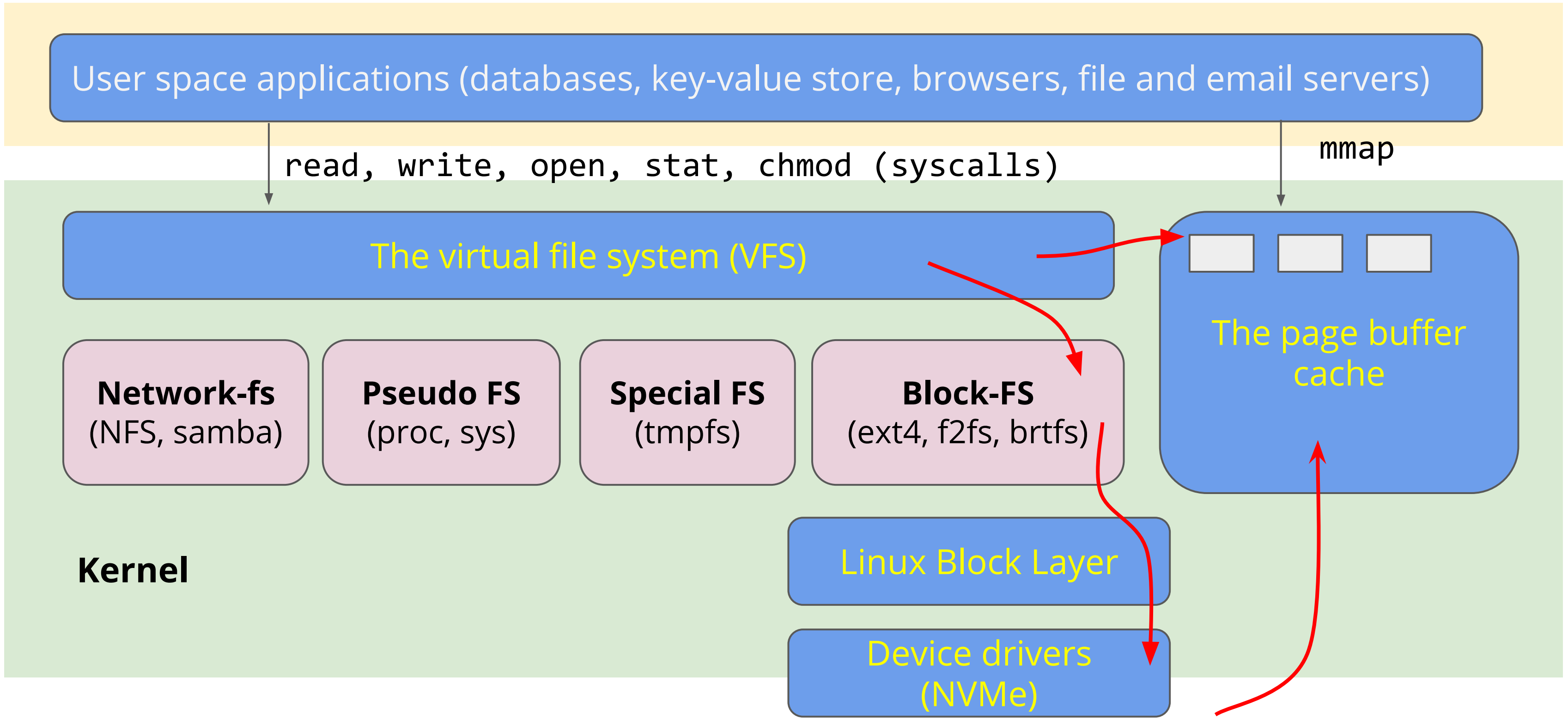
Storage and Memory are what is known as classical two-level storage system

- **Memory (DRAM)** is fast, byte-addressable and keeps data (technically cached) that is being worked on
- **Storage (block storage)** is slower, block-addressable and keeps data persistently
    - Optane and DRAM is also block-addressable, 64B blocks

Why do we want to run a file system on top of a persistent memory?

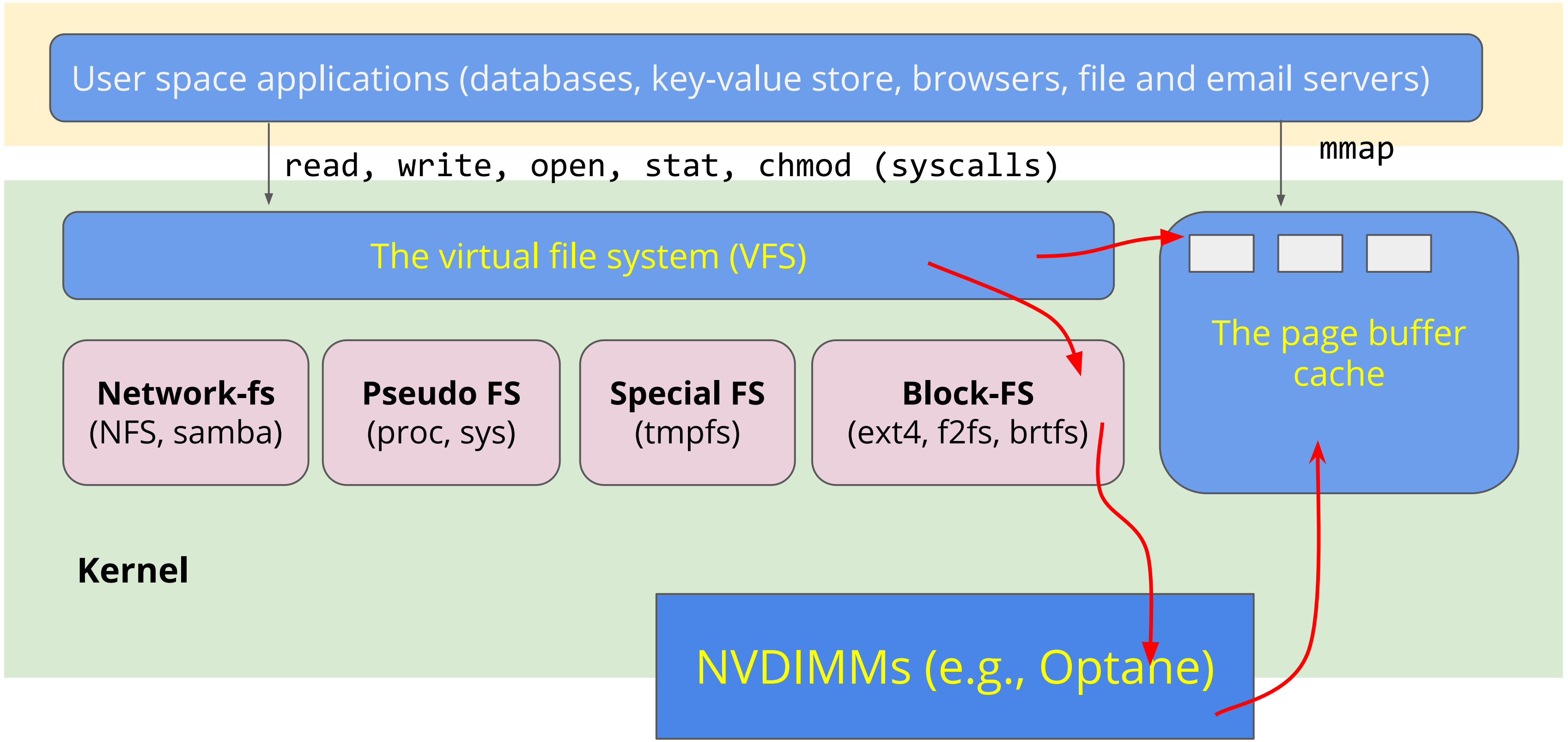
- Because it is known familiar interface which maintains the two-level distinction of storage and memory
- Data must be brought into DRAM from storage before being accessed

Looking at the Storage Stack Again

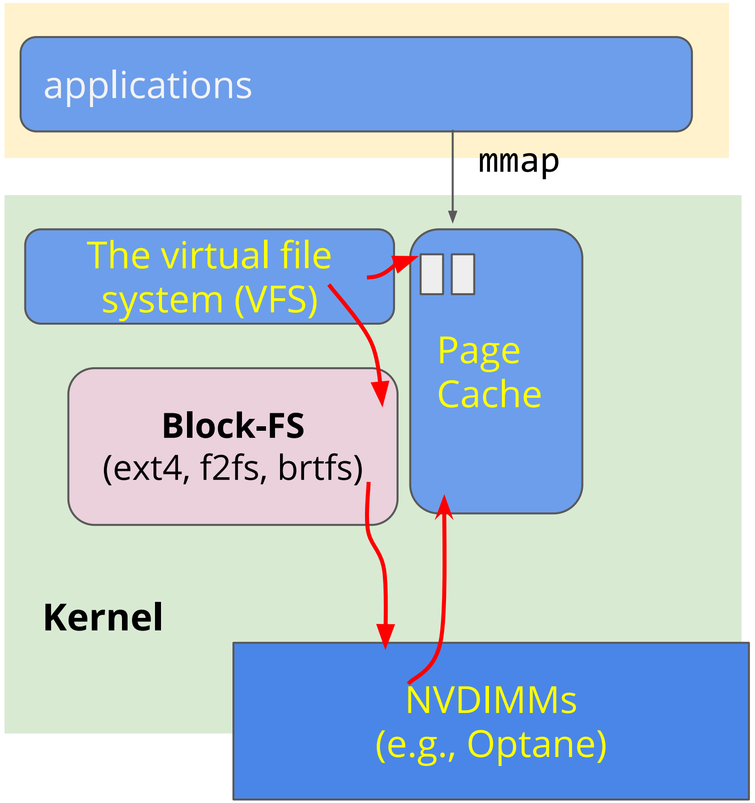


User space applications (databases, key-value store, browsers, file and email servers)
read, write, open, stat, chmod (syscalls)
mmap
The virtual file system (VFS)
The page buffer cache
Network-fs (NFS, samba)
Pseudo FS (proc, sys)
Special FS (tmpfs)
Block-FS (ext4, f2fs, brtfs)
Kernel
Linux Block Layer
Device drivers (NVMe)

Looking at the Storage Stack Again


User space applications (databases, key-value store, browsers, file and email servers)
read, write, open, stat, chmod (syscalls)
mmap
The virtual file system (VFS)
Network-fs (NFS, samba)
Pseudo FS (proc, sys)
Special FS (tmpfs)
Block-FS (ext4, f2fs, brtfs)
The page buffer cache
Kernel
NVDIMMs (e.g., Optane)

What Happens When I `mmap` a Page?



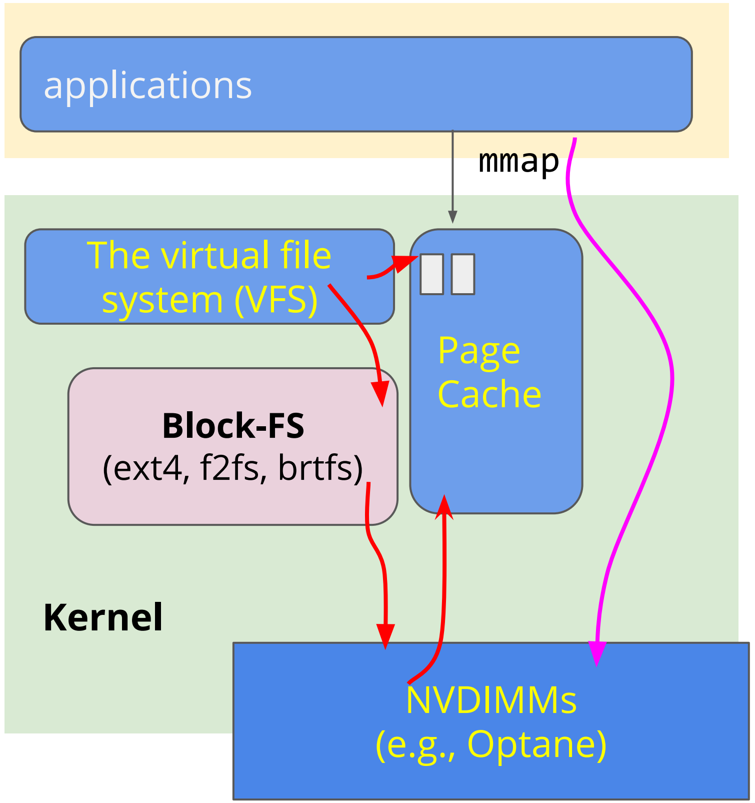
mmap takes pages from the page cache

If no page exist then the FS brings the page in the cache

Once in the cache then those DRAM address is used in the mmap and the pages are shared between the kernel and application

Does this make sense on NVDIMM?

Direct Access (DAX) Extensions for files



New file system support to directly mapped pages from NVDIMMs instead of making copies into the page cache

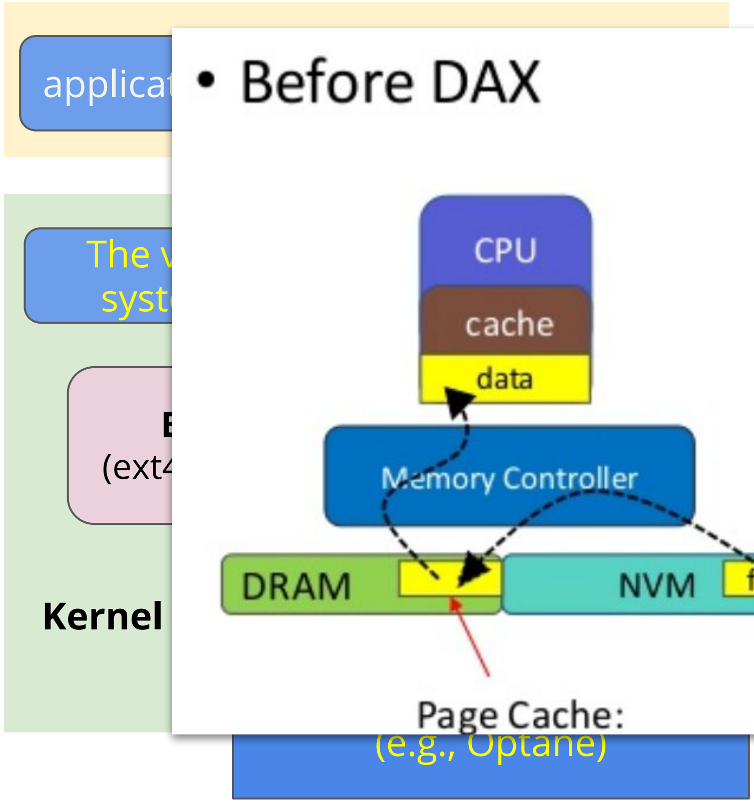
Needs modification into the file system to support this operation

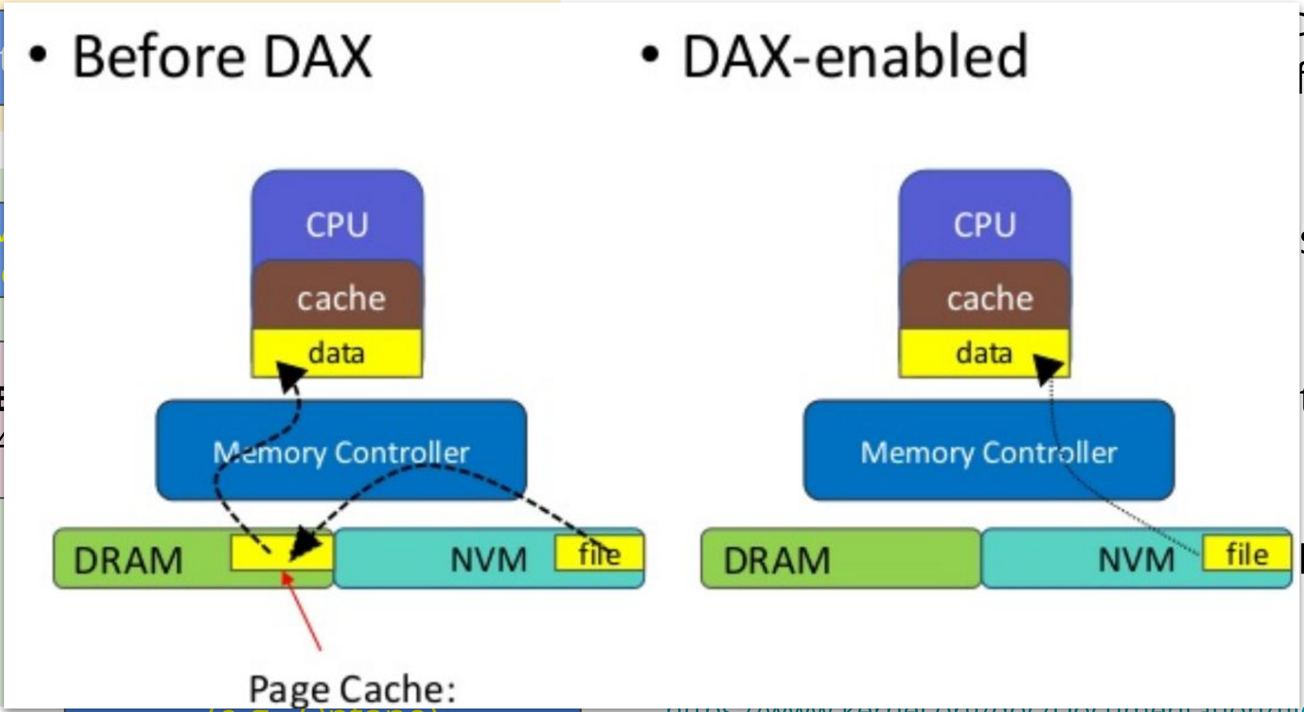
The block size must be equal to the page size

Currently 3 filesystems support DAX: ext2, ext4 and xfs

https://www.kernel.org/doc/Documentation/filesystems/dax.txt

Direct Access (DAX) Extensions for files





tly mapped
f making

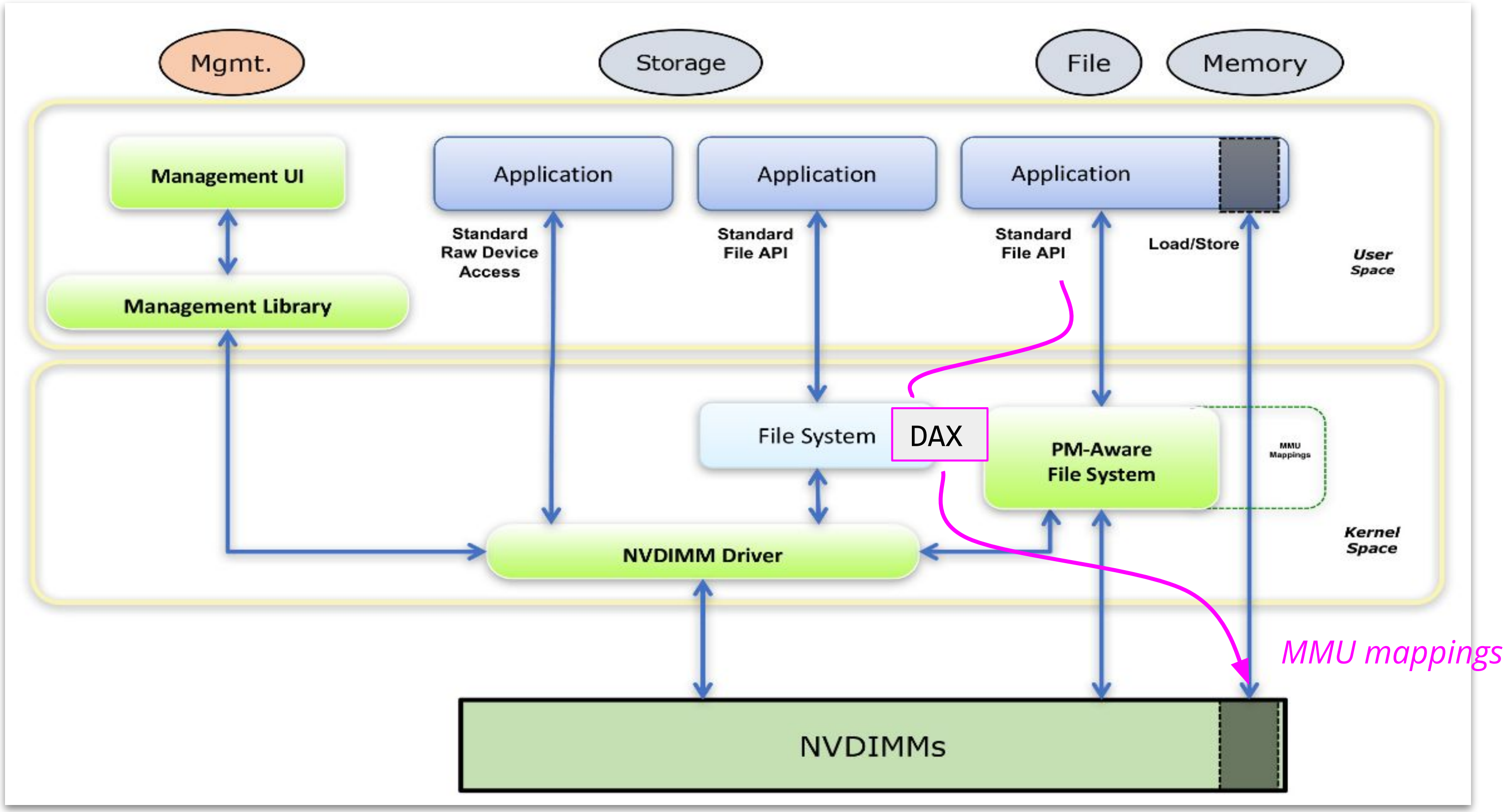
system to

the page

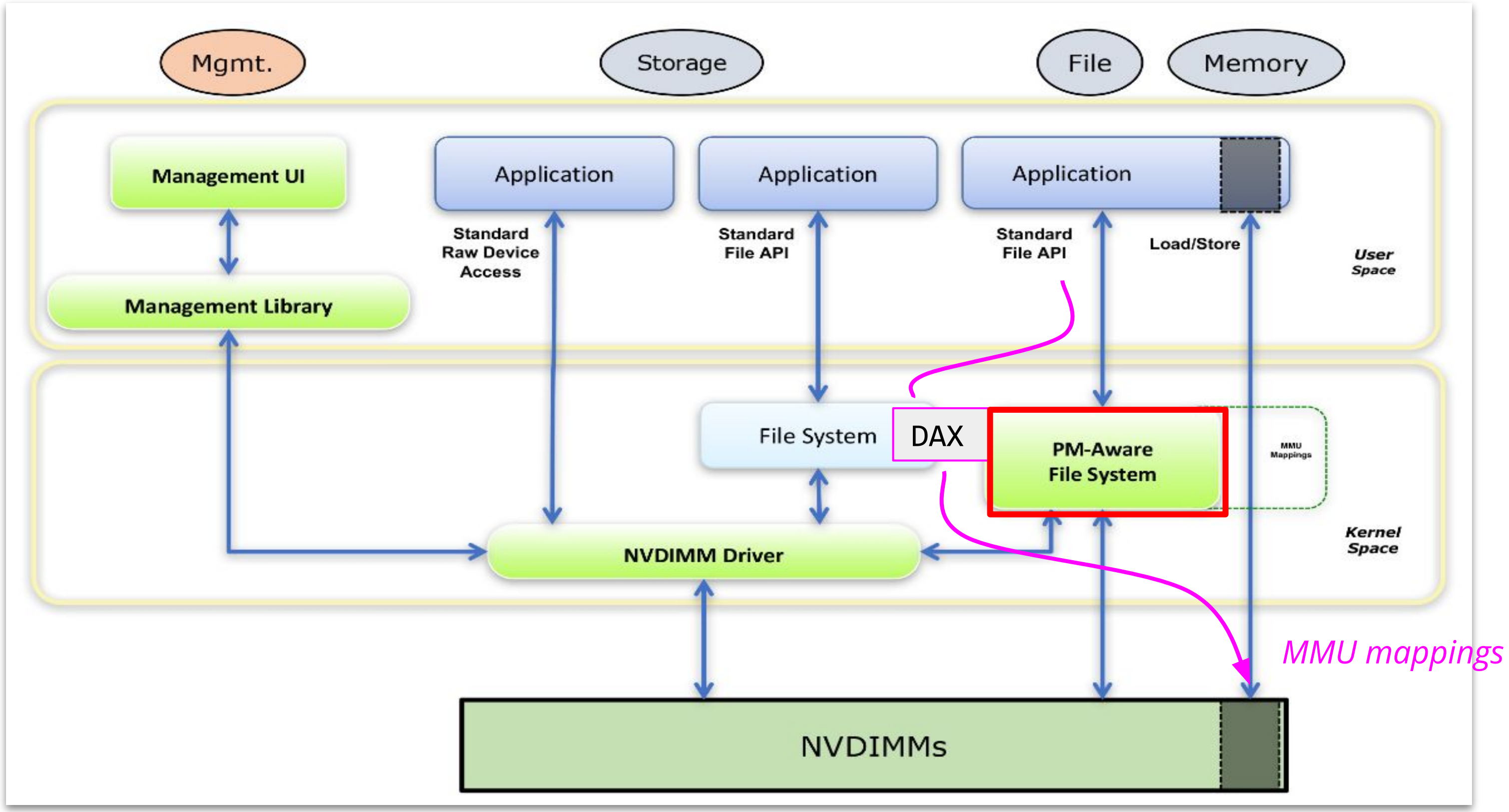
DAX: ext2,

https://www.kernel.org/doc/Documentation/filesystems/dax.txt

Updated Stack Image

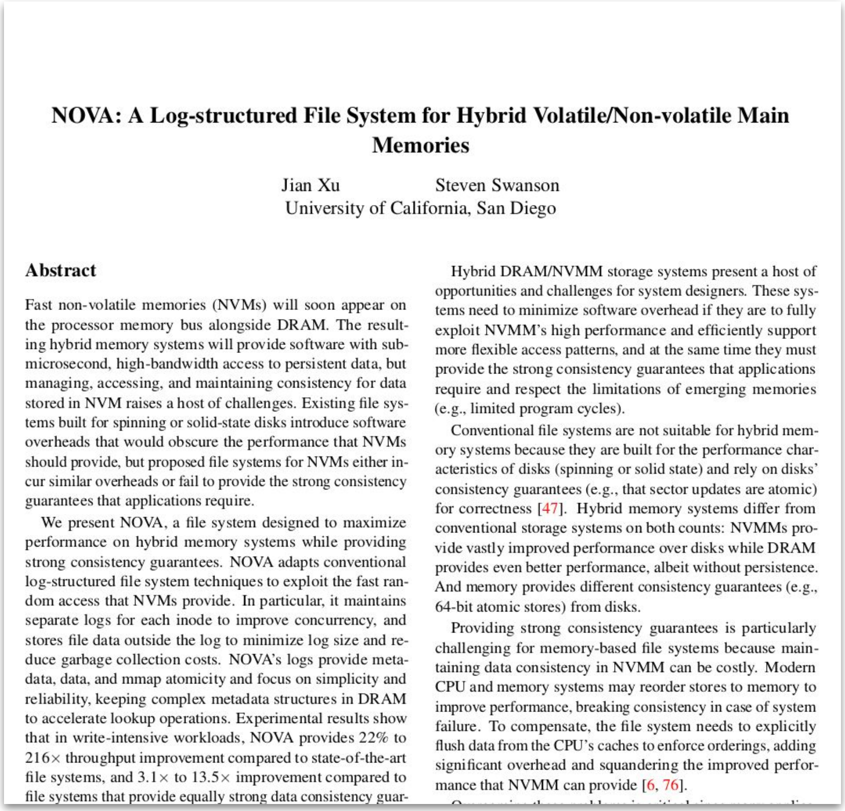


Updated Stack Image



What are pmem fs?

NOVA: A *Log-structured* File System for Hybrid Volatile/Non-volatile Main Memories (2016)



NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories

Jian Xu Steven Swanson
University of California, San Diego

Abstract

Fast non-volatile memories (NVMs) will soon appear on the processor memory bus alongside DRAM. The resulting hybrid memory systems will provide software with sub-microsecond, high-bandwidth access to persistent data, but managing, accessing, and maintaining consistency for data stored in NVM raises a host of challenges. Existing file systems built for spinning or solid-state disks introduce software overheads that would obscure the performance that NVMs should provide, but proposed file systems for NVMs either incur similar overheads or fail to provide the strong consistency guarantees that applications require.

We present NOVA, a file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. NOVA adapts conventional log-structured file system techniques to exploit the fast random access that NVMs provide. In particular, it maintains separate logs for each inode to improve concurrency, and stores file data outside the log to minimize log size and reduce garbage collection costs. NOVA's logs provide metadata, data, and mmap atomicity and focus on simplicity and reliability, keeping complex metadata structures in DRAM to accelerate lookup operations. Experimental results show that in write-intensive workloads, NOVA provides 22% to 216× throughput improvement compared to state-of-the-art file systems, and 3.1× to 13.5× improvement compared to file systems that provide equally strong data consistency guar-

Hybrid DRAM/NVMM storage systems present a host of opportunities and challenges for system designers. These systems need to minimize software overhead if they are to fully exploit NVMM's high performance and efficiently support more flexible access patterns, and at the same time they must provide the strong consistency guarantees that applications require and respect the limitations of emerging memories (e.g., limited program cycles).

Conventional file systems are not suitable for hybrid memory systems because they are built for the performance characteristics of disks (spinning or solid state) and rely on disks' consistency guarantees (e.g., that sector updates are atomic) for correctness [47]. Hybrid memory systems differ from conventional storage systems on both counts: NVMMs provide vastly improved performance over disks while DRAM provides even better performance, albeit without persistence. And memory provides different consistency guarantees (e.g., 64-bit atomic stores) from disks.

Providing strong consistency guarantees is particularly challenging for memory-based file systems because maintaining data consistency in NVMM can be costly. Modern CPU and memory systems may reorder stores to memory to improve performance, breaking consistency in case of system failure. To compensate, the file system needs to explicitly flush data from the CPU's caches to enforce orderings, adding significant overhead and squandering the improved performance that NVMM can provide [6, 76].

We will discuss this briefly (this is homework)

Why do We Need Yet Another File System

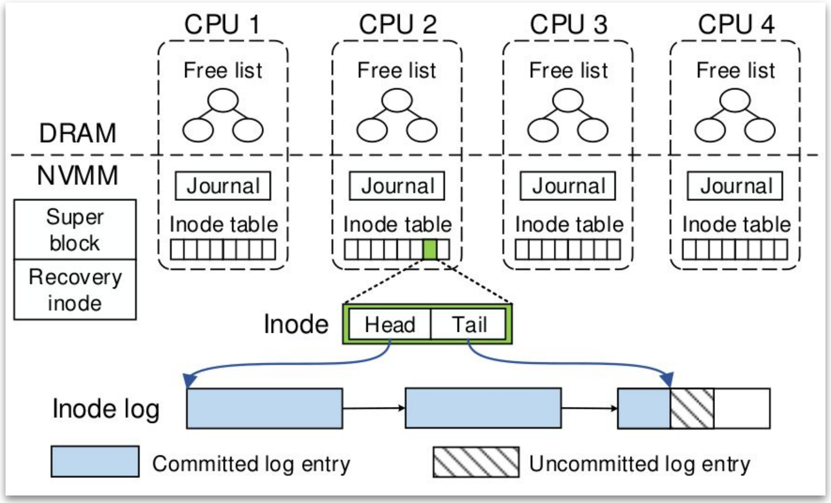
Why do we need a new file system for NVMDIMM?

1. High software overheads
2. CPU may reorder writes : *need to use fence and flush appropriately*
3. Different atomicity guarantees : *page vs 8-bytes or 64-bytes*
4. With directly mapped areas (DAX), *how do you provide data and metadata consistency?*
5. Decrease contention on a shared NVDIMM from multiple cores (cache coherency and locking overheads)
6. Performance: high concurrency of NVDIMMs vs block devices

Developed NOVA file system for hybrid DRAM-NVDIMM memories

https://www.usenix.org/sites/default/files/conference/protected-files/fast16_slides_xu.pdf

NOVA Design and Ideas



A Log-structured file system

Each inode has its own log (concurrency and parallelism)

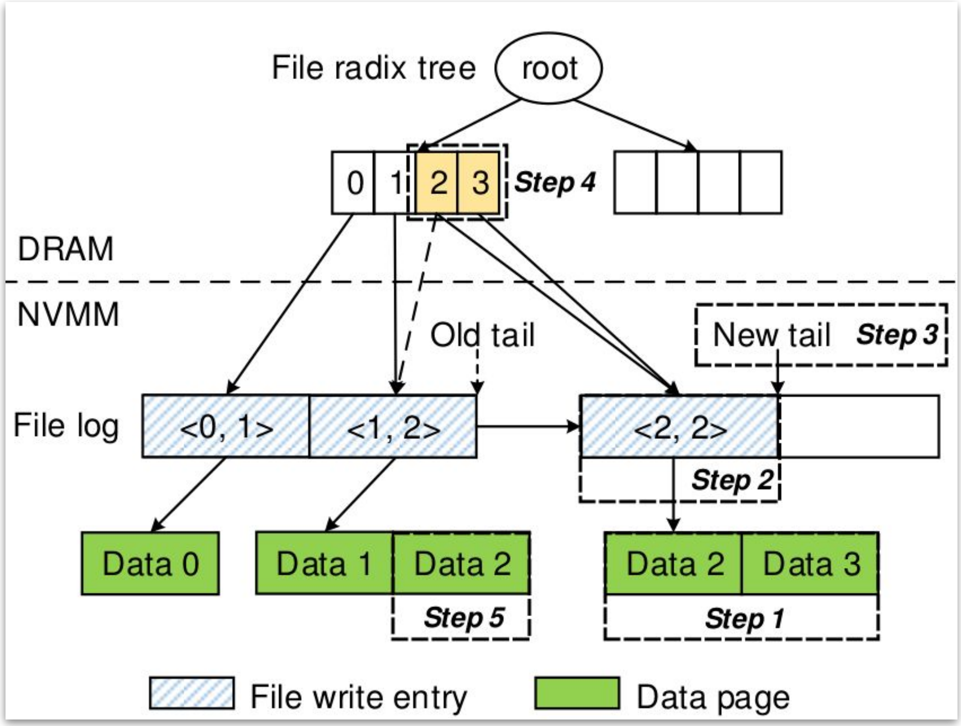
Each CPU has its own set of inodes and free list to manage

Performance: logs in NV, and index in DRAM

Smaller segments (4kB) and implement the log as a link list

Single inode updates (in the inode log), multiple inodes (uses the journal, 64B entries)

Nova : Write Example



We are modifying Data2 and add Data3
`<write order, number of page affected>`

Step 1: find and copy blocks which are to be updated (Copy-on-write)

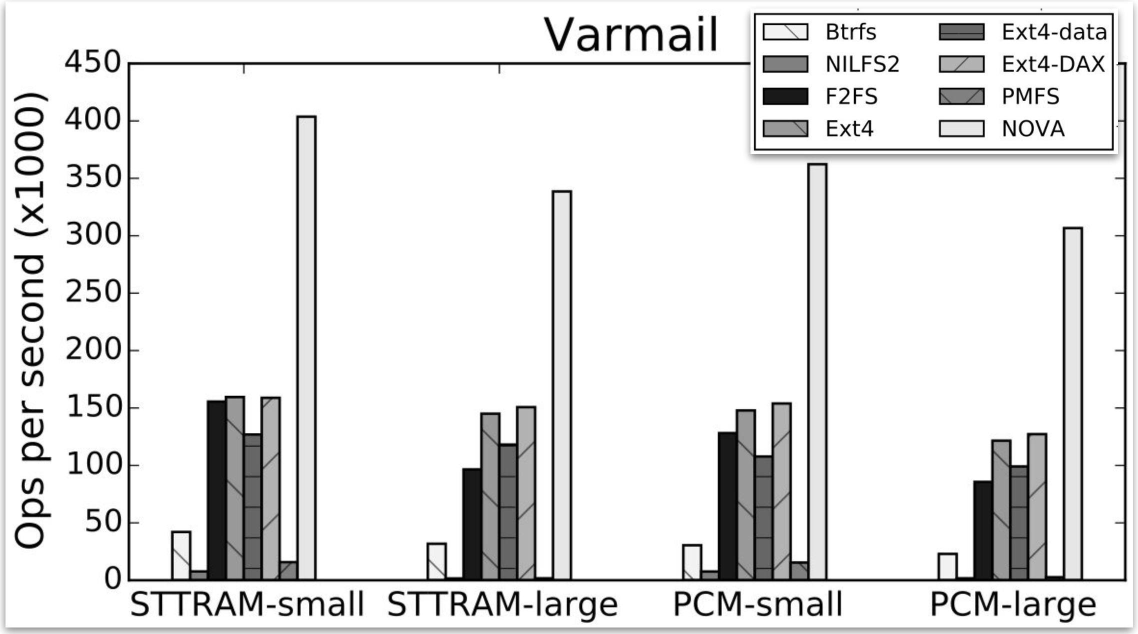
Step 2: Add to the file inode log

Step 3: Update the log tail pointer (after this the write is it durable)

Step 4: Update the DRAM index for fast lookup

Step 5: Garbage collect old pages

Nova Performance Comparison



For varmail (this workload) : 3.1-216x outperforms other file systems

Is File System the Best Way to Use NVDIMM?

We do not use file system with DRAM, do we?

With a file system

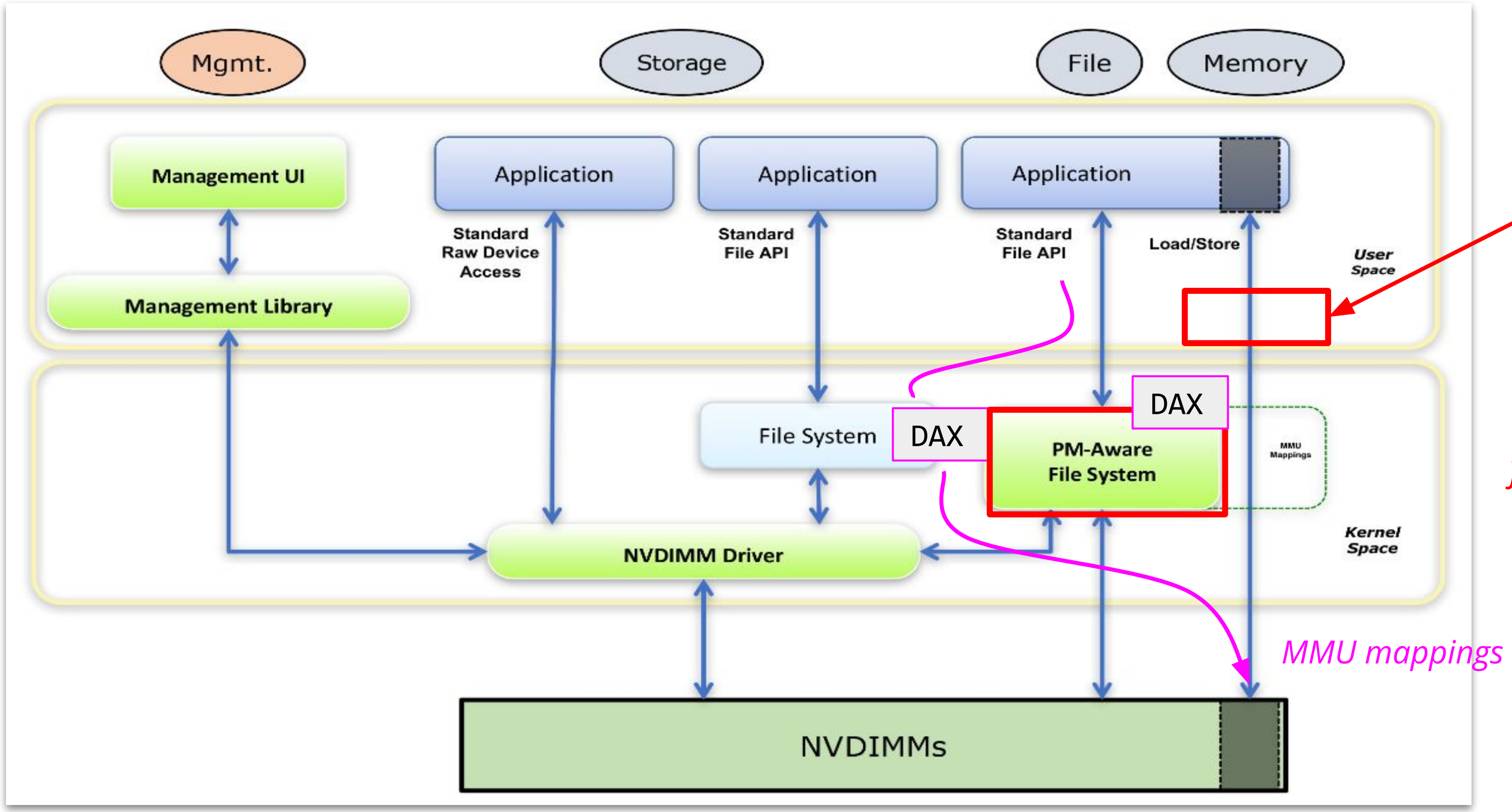
1. Data must first be (de)serialized when reading in
2. When writing out data must be serialized to be written out to a file

Overheads from

1. Complex buffer management (read, write)
2. Serialization, deserialization process
3. File system, block layer, I/O operations etc.

So, coming back to the point -- how do we use DRAM actually?

Updated Stack Image



How do we use DRAM?

What are pmem fs - NOVA

How do We Acquire/Use DRAM?

1. Mmap → page granularity
2. malloc / calloc → small memory, allocated on the process heap

We then build data structures in the allocated heap space (link list, trees, hash table)

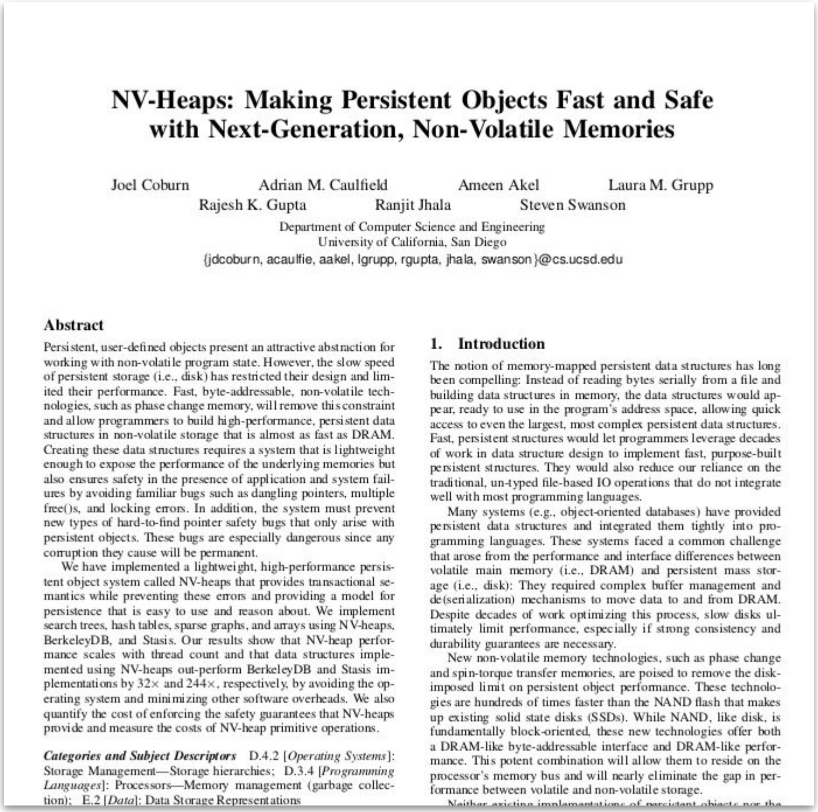
Can we do calloc or malloc on NVDIMM memory area?

How do we build a data structure in NVDIMM memory area?

What are the concerns here?

What does that would mean after a system restart?

NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories (2011)



NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories

Joel Coburn Adrian M. Caulfield Ameen Akel Laura M. Grupp
Rajesh K. Gupta Ranjit Jhala Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego
{jdcoburn, acaulfie, aakel, lgrupp, rgupta, jhala, swanson}@cs.ucsd.edu

Abstract

Persistent, user-defined objects present an attractive abstraction for working with non-volatile program state. However, the slow speed of persistent storage (i.e., disk) has restricted their design and limited their performance. Fast, byte-addressable, non-volatile technologies, such as phase change memory, will remove this constraint and allow programmers to build high-performance, persistent data structures in non-volatile storage that is almost as fast as DRAM. Creating these data structures requires a system that is lightweight enough to expose the performance of the underlying memories but also ensures safety in the presence of application and system failures by avoiding familiar bugs such as dangling pointers, multiple free()s, and locking errors. In addition, the system must prevent new types of hard-to-find pointer safety bugs that only arise with persistent objects. These bugs are especially dangerous since any corruption they cause will be permanent.

We have implemented a lightweight, high-performance persistent object system called NV-heaps that provides transactional semantics while preventing these errors and providing a model for persistence that is easy to use and reason about. We implement search trees, hash tables, sparse graphs, and arrays using NV-heaps, BerkeleyDB, and Stasis. Our results show that NV-heap performance scales with thread count and that data structures implemented using NV-heaps out-perform BerkeleyDB and Stasis implementations by 32× and 244×, respectively, by avoiding the operating system and minimizing other software overheads. We also quantify the cost of enforcing the safety guarantees that NV-heaps provide and measure the costs of NV-heap primitive operations.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Storage hierarchies; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); E.2 [*Data*]: Data Storage Representations

1. Introduction

The notion of memory-mapped persistent data structures has long been compelling: Instead of reading bytes serially from a file and building data structures in memory, the data structures would appear, ready to use in the program's address space, allowing quick access to even the largest, most complex persistent data structures. Fast, persistent structures would let programmers leverage decades of work in data structure design to implement fast, purpose-built persistent structures. They would also reduce our reliance on the traditional, un-typed file-based IO operations that do not integrate well with most programming languages.

Many systems (e.g., object-oriented databases) have provided persistent data structures and integrated them tightly into programming languages. These systems faced a common challenge that arose from the performance and interface differences between volatile main memory (i.e., DRAM) and persistent mass storage (i.e., disk): They required complex buffer management and de(serialization) mechanisms to move data to and from DRAM. Despite decades of work optimizing this process, slow disks ultimately limit performance, especially if strong consistency and durability guarantees are necessary.

New non-volatile memory technologies, such as phase change and spin-torque transfer memories, are poised to remove the disk-imposed limit on persistent object performance. These technologies are hundreds of times faster than the NAND flash that makes up existing solid state disks (SSDs). While NAND, like disk, is fundamentally block-oriented, these new technologies offer both a DRAM-like byte-addressable interface and DRAM-like performance. This potent combination will allow them to reside on the processor's memory bus and will nearly eliminate the gap in performance between volatile and non-volatile storage.

Neither existing implementations of persistent objects nor the

NV Heaps

A more interesting way to use NVDIMM is to make a persistent heap from where various data types can be allocated, tree, link list, hash table, etc.

```
Insert(Object * a, List<Object> * l);
```

NV Heaps

A more interesting way to use NVDIMM is to make a persistent heap from where various data types can be allocated, tree, link list, hash table, etc.

```
Insert(Object * a, List<Object> * l);
```

Is "l" pointer suppose to be non-volatile or volatile?

Is "a" pointer suppose to be non-volatile or volatile?

*Let's say if "a" came from DRAM, and we inserted it into "*l" which came from NV memory, then after a restart, "*l" will contain a bogus pointer*

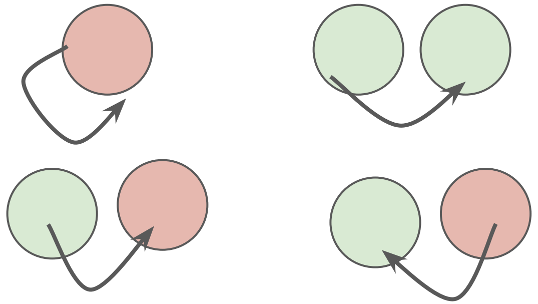
NV Heaps

A more interesting way to use NVDIMM is to make a persistent heap from where various data types can be allocated, tree, link list, hash table, etc.

So, what do we need to consider?

🟥 *Non-Volatile heap*

🟦 *Volatile heap*



1. Pointer management:
   a. Non-Volatile pointers within a NV heap
   b. Non-Volatile across NV heaps
   c. Volatile pointers to a NV heap
   d. NV heap pointer to a volatile pointer
2. Memory management: memory leaks, double free
3. How and when to make structure consistent, and concurrent
   a. Locking, transaction issues ← hard to get it right even with DRAM

Which one of the 4 pointers type should be allowed, or not allowed?

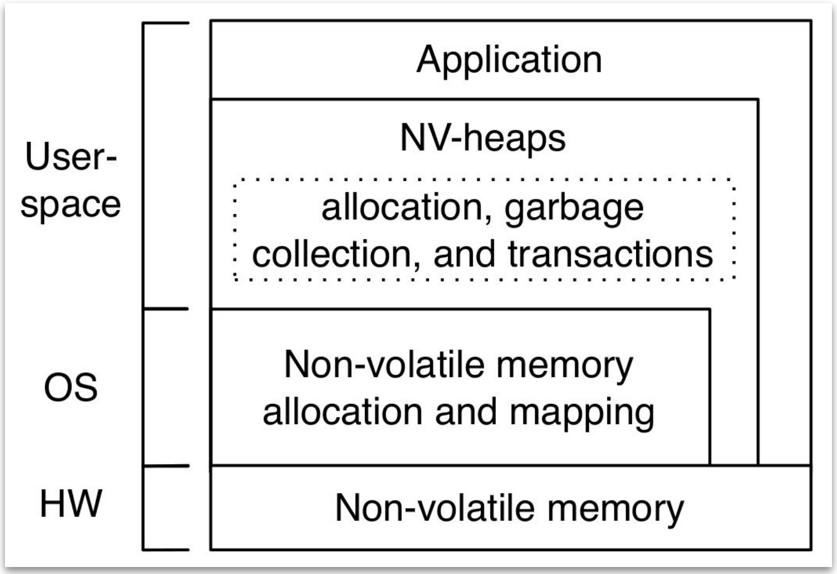
NVHeaps

Simple primitives:

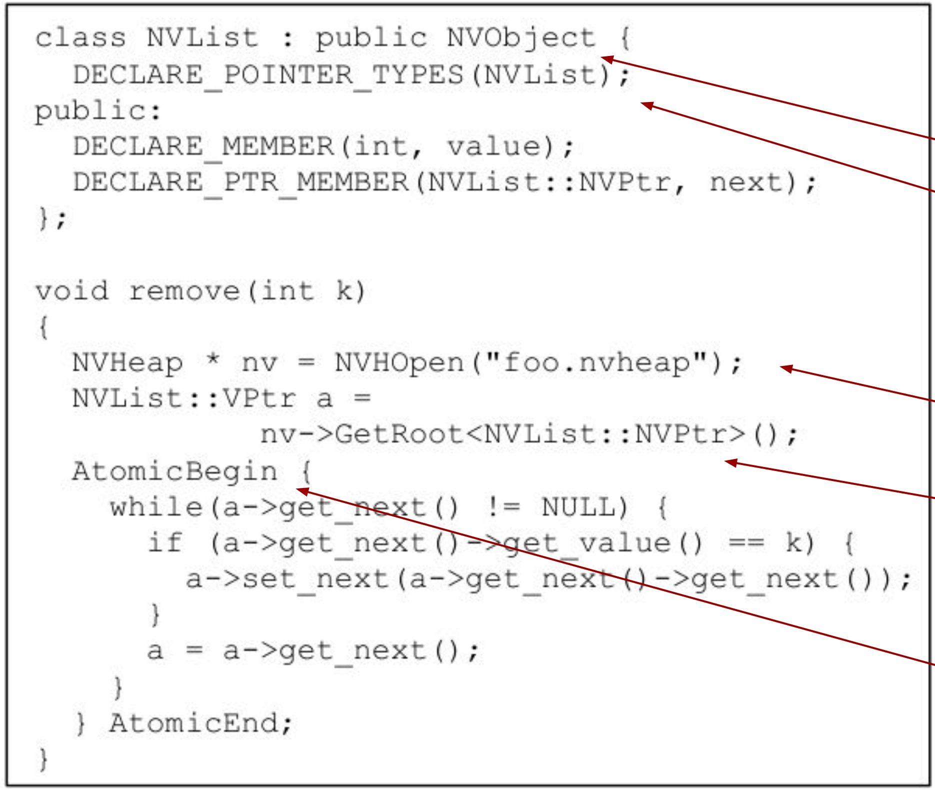
- persistent objects
- specialized pointer types
- a memory allocator
- atomic sections

A heap “root” object (a NV pointer) can be created or opened by passing a file name to HVHeap

Files are self sufficient and contains offset based references from the “root” to maintain integrity


User-space
OS
HW
Application
NV-heaps
allocation, garbage collection, and transactions
Non-volatile memory allocation and mapping
Non-volatile memory

Example



```
class NVList : public NVObject {
  DECLARE_POINTER_TYPES(NVList);
public:
  DECLARE_MEMBER(int, value);
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};

void remove(int k)
{
  NVHeap * nv = NVHOpen("foo.nvheap");
  NVList::VPtr a =
           nv->GetRoot<NVList::NVPtr>();
  AtomicBegin {
    while(a->get_next() != NULL) {
      if (a->get_next()->get_value() == k) {
        a->set_next(a->get_next()->get_next());
      }
      a = a->get_next();
    }
  } AtomicEnd;
}
```

A base class

A special pointer type

Open a heap

Get the “root” of this heap

Atomically iterate and remove the item

NVHeap - Managing Memory

How to implement safe memory management?

Uses operational logging and reference counting together

- Operational logs is kept in NVM and tracks operations (free, alloc, moving, transactions, read, write) → helps to find bad operations

- Reference counting on each object with automatic garbage collection of objects by scanning if their count has hit 1 (1 because they are internally always referenced by the root node in the NVHeap)

- Locks to protect reference counting with concurrent threads, *where should lock be stored?*
  - In DRAM: lose protection in case of a failure
  - In NVDIMM: then you need to scan the whole NVDIMM to find all taken but failed locks

NVHeap - Managing Memory

How to implement safe memory management?

Uses operational logging and reference counting together

- Operational logs is kept in NVM and tracks operations (free, alloc, moving, transactions, read, write) → helps to find bad operations

- Reference counting on each object with automatic garbage collection of objects by scanning if their count has hit 1 (1 because they are internally always referenced by the root node in the NVHeap)

- Locks to protect reference counting with concurrent threads, *where should lock be stored?*

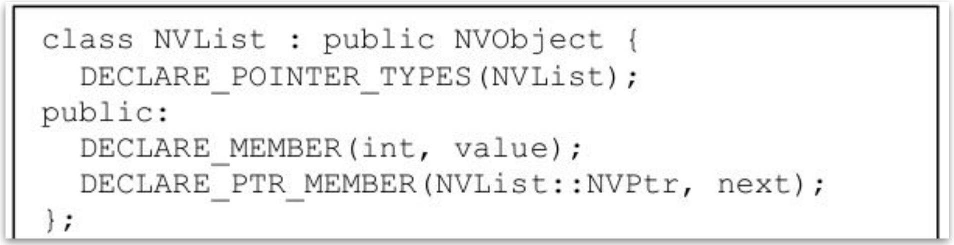
Use generational locks: everytime a NVHeap file is open, increment its generation and discard all old dirty locks

NVHeap - How to do pointer management

How to do pointer management?

Smart pointers and overloading

- Not supported: inter NV heap pointers or NV pointers to volatile structures
- **Operator overloading** : a pointer internally contain offset and heap id to identify which heap they belong to
    - Thus, creation of an inter NVheap pointer can be rejected



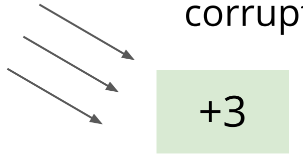
```
class NVList : public NVObject {
  DECLARE_POINTER_TYPES(NVList);
public:
  DECLARE_MEMBER(int, value);
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};
```

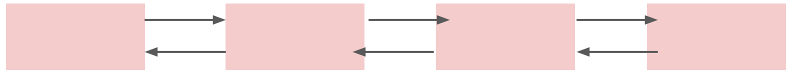
NVHeap - How to do pointer management

How to do pointer management?

Smart pointers and overloading

- Not supported: inter NV heap pointers or NV pointers to volatile structures
- **Operator overloading** : a pointer internally contain offset and heap id to identify which heap they belong to
  - Thus, creation of an inter NVheap pointer can be rejected
- Two types of NV pointers : *normal and weak*
  - **Normal :** increment the reference count
  - **Weak:** do not increment the reference count (doubly link lists), can lead to a NULL but not corruption of NVHeap





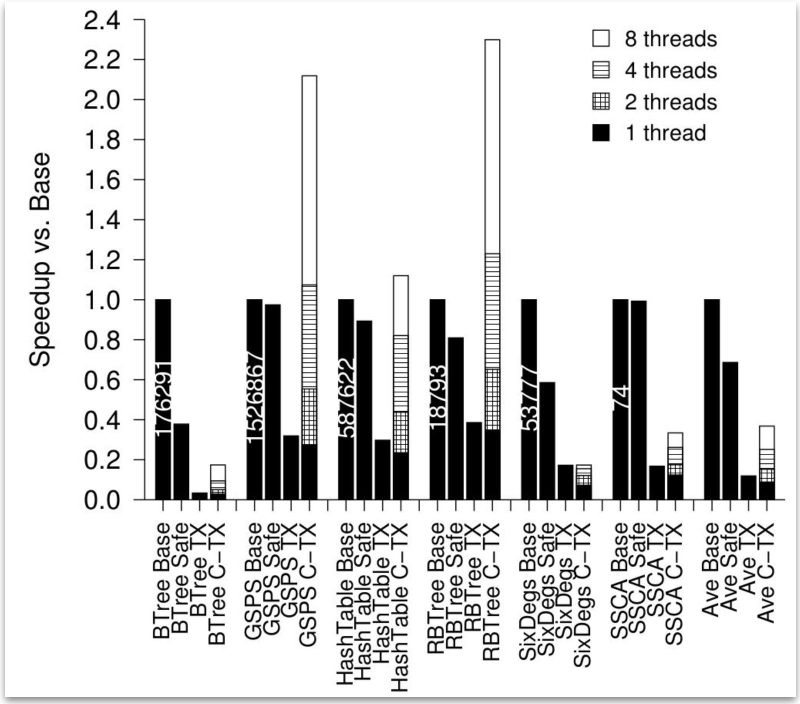
NVHeap - How to do pointer management

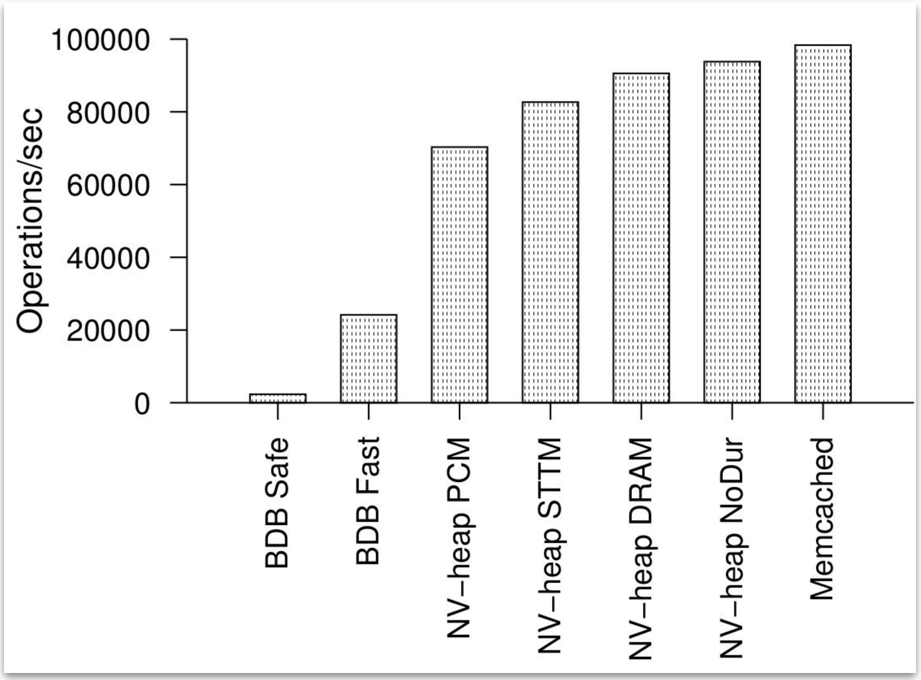
How to do pointer management?

Smart pointers and overloading

- Not supported: inter NV heap pointers or NV pointers to volatile structures
- **Operator overloading** : a pointer internally contain offset and heap id to identify which heap they belong to
    - Thus, creation of an inter NVheap pointer can be rejected
- Two types of NV pointers : *normal and weak*
    - **Normal :** increment the reference count
    - **Weak:** do not increment the reference count (doubly link lists), can lead to a NULL but not corruption of NVHeap
- Volatile pointers to NV Heap pointers
    - These references must count for incrementing references, however in case of a failure?
    - Same trick as generational locks: use generational volatile→ NV reference counts

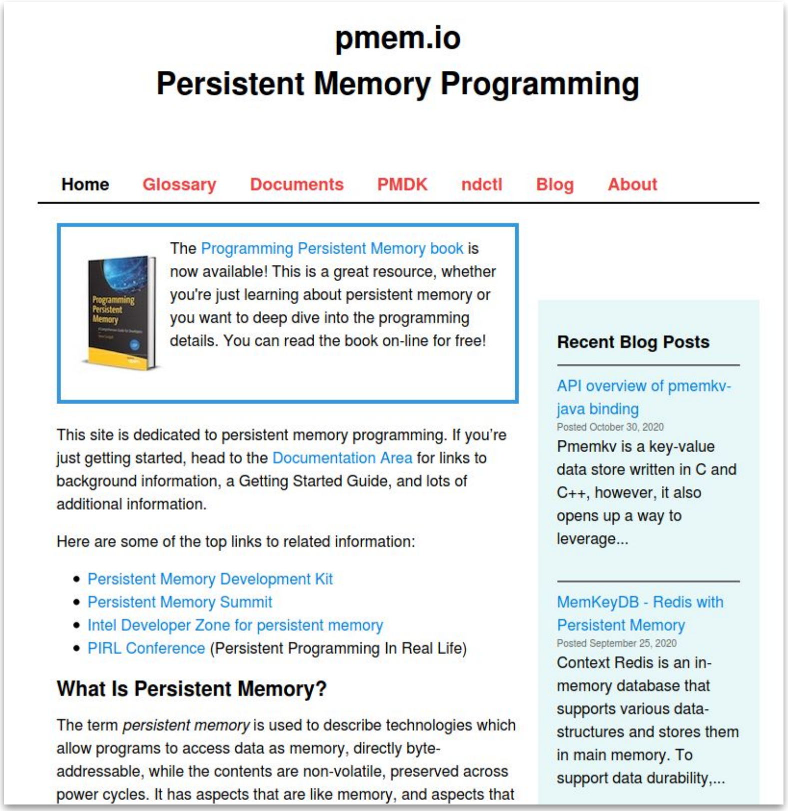
NV Performance





- Variants: base, safe (pointers+mm), Tx (with logging), C-TX (concurrency)
- Comparison with other alternatives: BerkeleyDB, and memcached

Today, These Ideas...



pmem.io

Persistent Memory Programming

Home Glossary Documents PMDK ndctl Blog About

The Programming Persistent Memory book is now available! This is a great resource, whether you're just learning about persistent memory or you want to deep dive into the programming details. You can read the book on-line for free!

This site is dedicated to persistent memory programming. If you're just getting started, head to the Documentation Area for links to background information, a Getting Started Guide, and lots of additional information.

Here are some of the top links to related information:

- Persistent Memory Development Kit
- Persistent Memory Summit
- Intel Developer Zone for persistent memory
- PIRL Conference (Persistent Programming In Real Life)

What Is Persistent Memory?

The term *persistent memory* is used to describe technologies which allow programs to access data as memory, directly byte-addressable, while the contents are non-volatile, preserved across power cycles. It has aspects that are like memory, and aspects that

Recent Blog Posts

API overview of pmemkv-java binding

Posted October 30, 2020

Pmemkv is a key-value data store written in C and C++, however, it also opens up a way to leverage...

MemKeyDB - Redis with Persistent Memory

Posted September 25, 2020

Context Redis is an in-memory database that supports various data-structures and stores them in main memory. To support data durability,...

Persistent Memory Development Kit (PMDK)

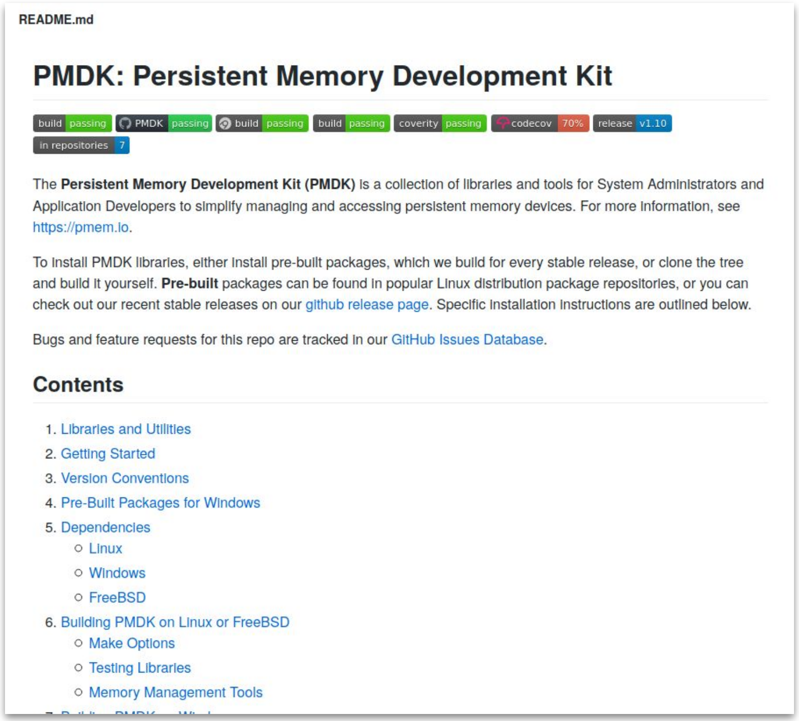
A set of libraries and framework to manage NVDIMMs as

1. Persistent memory;
2. Volatile, but large memory

Contains helper routines to allocate object, persistent them, transactions, log, bulk copying, and remote pmem access

Binding for multiple languages

https://pmem.io/



README.md

PMDK: Persistent Memory Development Kit

build passing | PMDK passing | build passing | build passing | coverity passing | codecov 70% | release v1.10 | in repositories 7

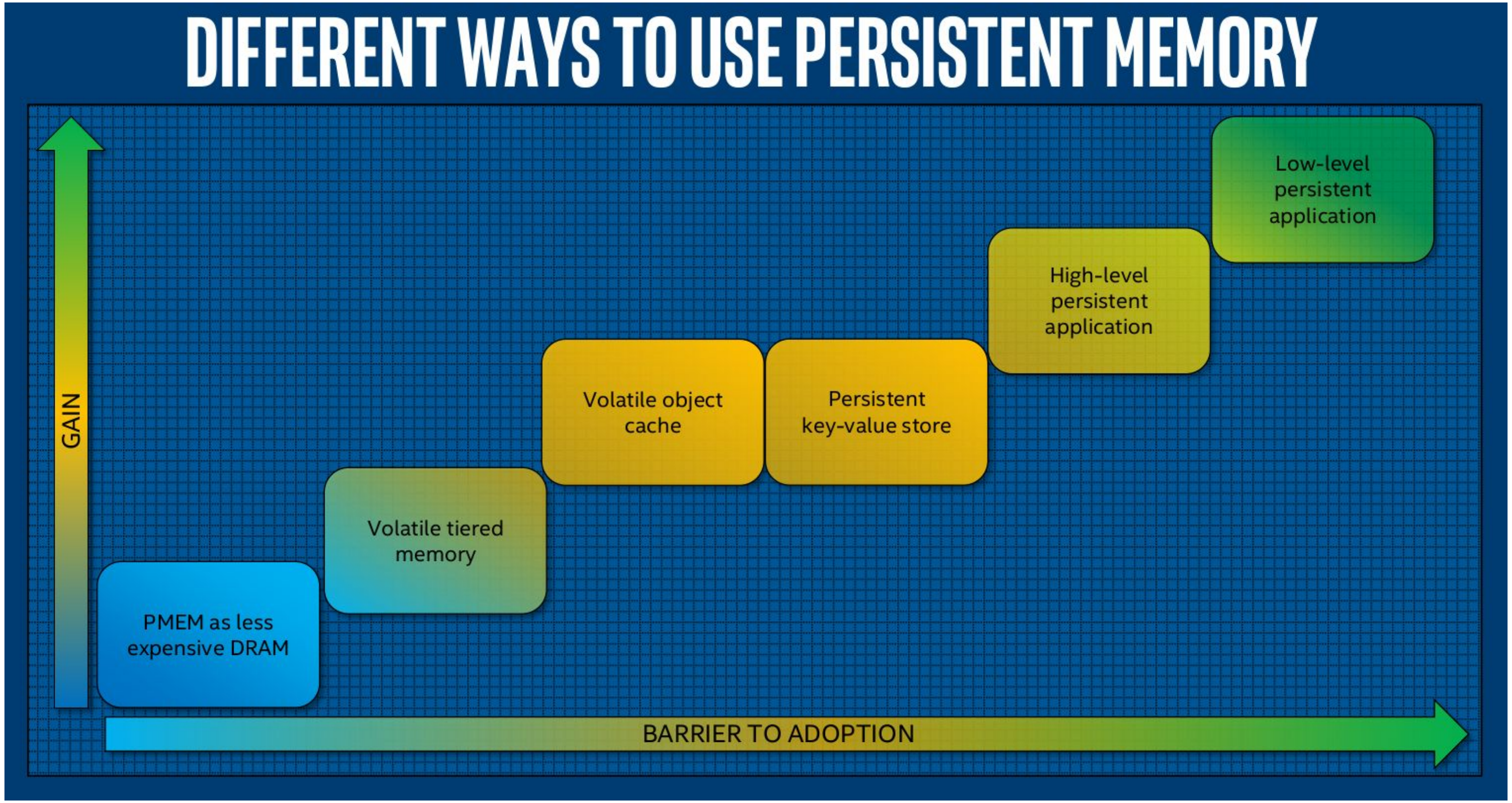
The **Persistent Memory Development Kit (PMDK)** is a collection of libraries and tools for System Administrators and Application Developers to simplify managing and accessing persistent memory devices. For more information, see https://pmem.io.

To install PMDK libraries, either install pre-built packages, which we build for every stable release, or clone the tree and build it yourself. **Pre-built** packages can be found in popular Linux distribution package repositories, or you can check out our recent stable releases on our github release page. Specific installation instructions are outlined below.

Bugs and feature requests for this repo are tracked in our GitHub Issues Database.

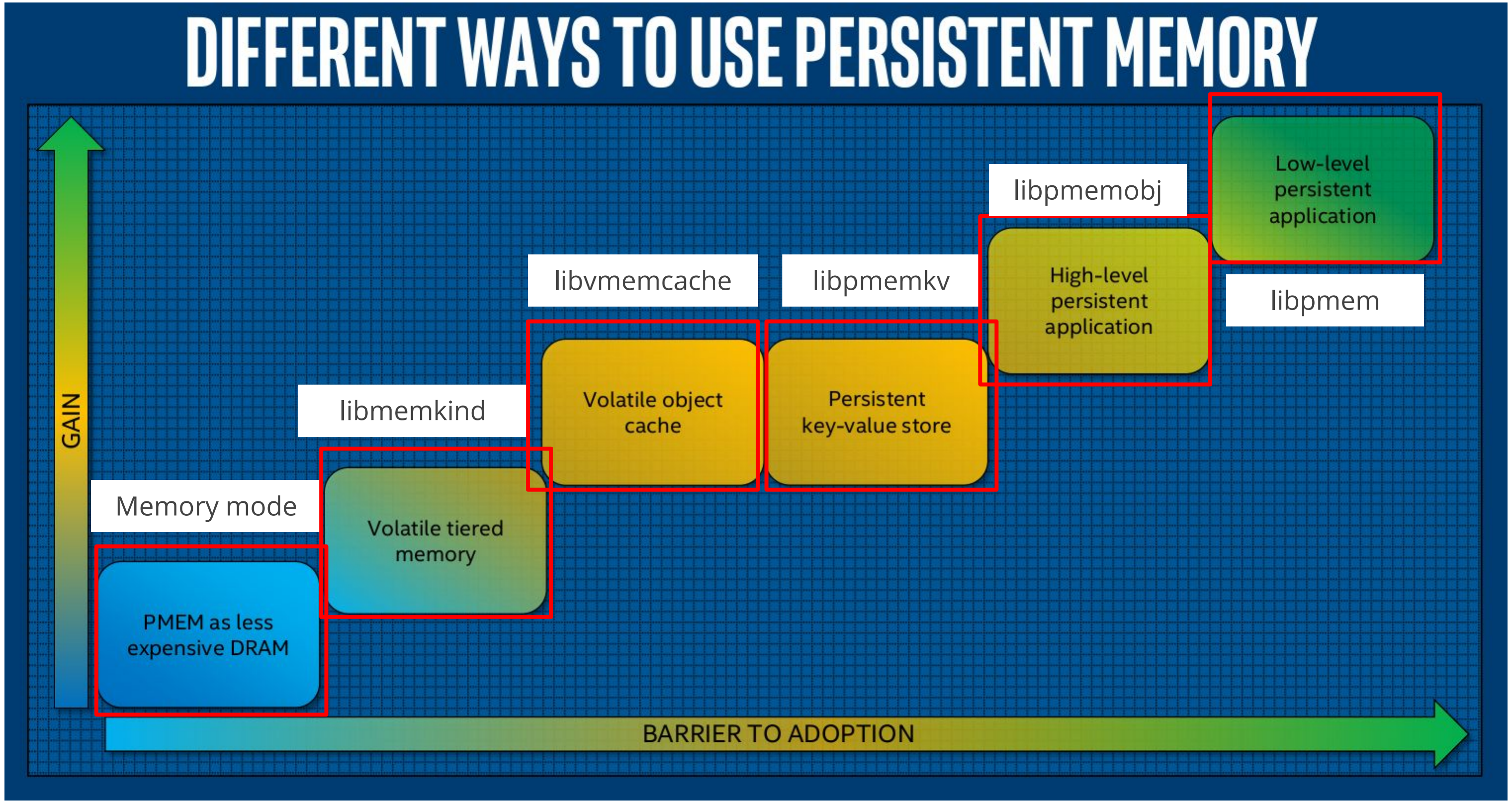
Contents

Adoption Spectrum


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

https://www.slideshare.net/IntelSoftware/pmdk-essentials-parts-1-and-2-andy-rudoff-and-pawel-skowron

Adoption Spectrum


DIFFERENT WAYS TO USE PERSISTENT MEMORY
GAIN
BARRIER TO ADOPTION
Memory mode
PMEM as less expensive DRAM
libmemkind
Volatile tiered memory
libvmemcache
Volatile object cache
libpmemkv
Persistent key-value store
libpmemobj
High-level persistent application
libpmem
Low-level persistent application

https://www.slideshare.net/IntelSoftware/pmdk-essentials-parts-1-and-2-andy-rudoff-and-pawel-skowron

Example: libpmemobj

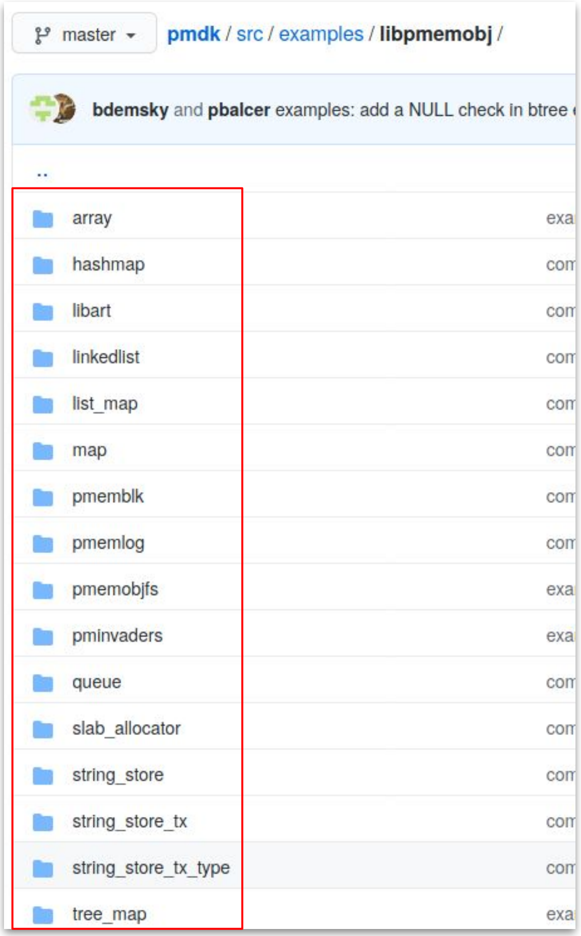
Transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming:

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

```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;
}
```

https://pmem.io/pmdk/libpmemobj/ (examples and documentation)

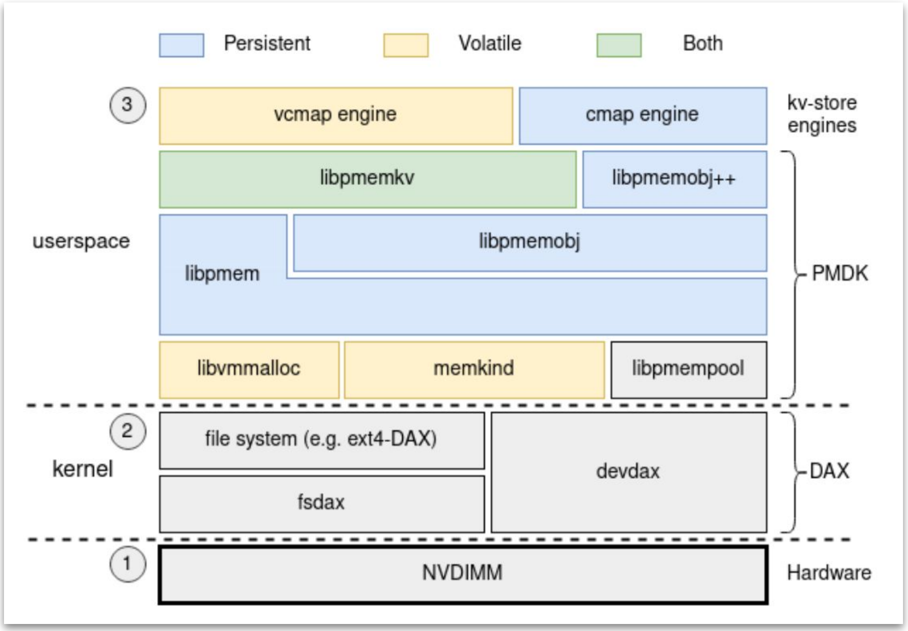

master
pmdk / src / examples / libpmemobj /

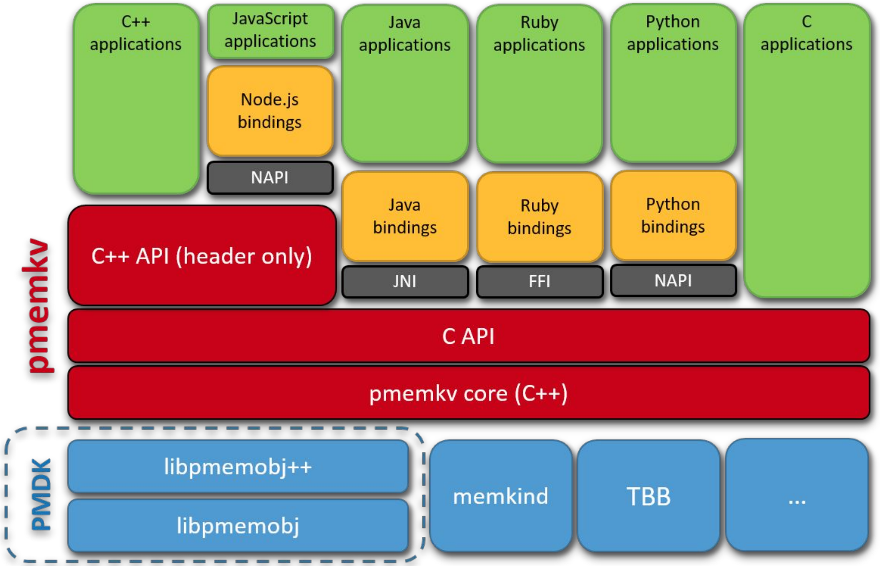
bdemsky and pbalcer examples: add a NULL check in btree

..

array	exa
hashmap	com
libart	com
linkedlist	com
list_map	com
map	com
pmemblk	com
pmemlog	com
pmemobjfs	exa
pminvaders	exa
queue	com
slab_allocator	com
string_store	com
string_store_tx	com
string_store_tx_type	com
tree_map	exa

PMDK Stack Overview



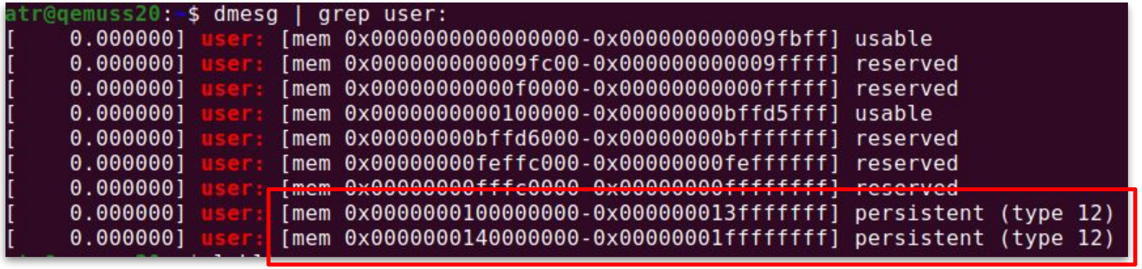


- Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack, BSc thesis, Nick-Andian Tehrany
- https://pmem.io/2020/03/04/pmemkv-bindings.html
- https://pmem.io/pmdk/ ← all libraries and their documentation

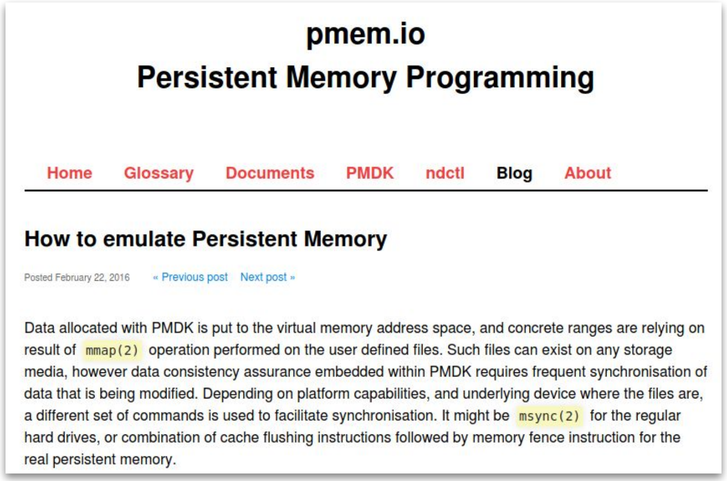
Want to Try NVDIMMs?

Use QEMU

```
qemu-system-x86_64 \
-drive file=ubuntu.img,format=raw,index=0,media=disk \
-m 4G,slots=4,maxmem=32G \
-smp 4 \
-machine pc,accel=kvm,nvdimm=on \
-enable-kvm \
-net nic \
-net user,hostfwd=tcp::2222-:22 \
-object memory-backend-file,id=mem1,share,mem-path=./dimm0,size=4G \
-device nvdimm,memdev=mem1,id=nv1,label-size=2M \
-object memory-backend-file,id=mem2,share,mem-path=./dimm1,size=4G \
-device nvdimm,memdev=mem2,id=nv2,label-size=2M \
```



```
atr@qemuss20:~$ dmesg | grep user:
[    0.000000] user: [mem 0x0000000000000000-0x000000000009fbff] usable
[    0.000000] user: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[    0.000000] user: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[    0.000000] user: [mem 0x0000000000100000-0x00000000bffd5fff] usable
[    0.000000] user: [mem 0x00000000bffd6000-0x00000000bfffffff] reserved
[    0.000000] user: [mem 0x00000000feffc000-0x00000000feffffff] reserved
[    0.000000] user: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
[    0.000000] user: [mem 0x0000000100000000-0x000000013fffffff] persistent (type 12)
[    0.000000] user: [mem 0x0000000140000000-0x00000001ffffffff] persistent (type 12)
```



pmem.io

Persistent Memory Programming

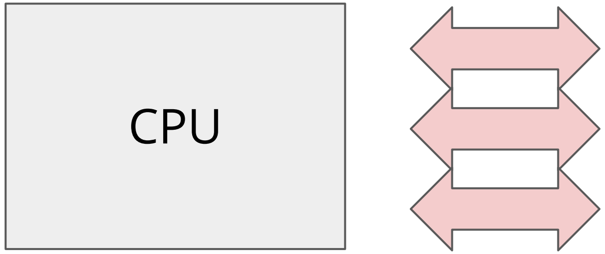
How to emulate Persistent Memory

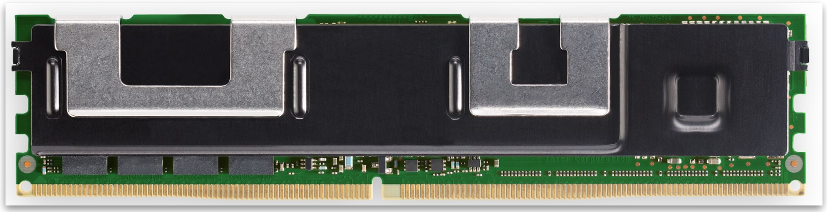
Posted February 22, 2016 « Previous post Next post »

Data allocated with PMDK is put to the virtual memory address space, and concrete ranges are relying on result of `mmap(2)` operation performed on the user defined files. Such files can exist on any storage media, however data consistency assurance embedded within PMDK requires frequent synchronisation of data that is being modified. Depending on platform capabilities, and underlying device where the files are, a different set of commands is used to facilitate synchronisation. It might be `msync(2)` for the regular hard drives, or combination of cache flushing instructions followed by memory fence instruction for the real persistent memory.

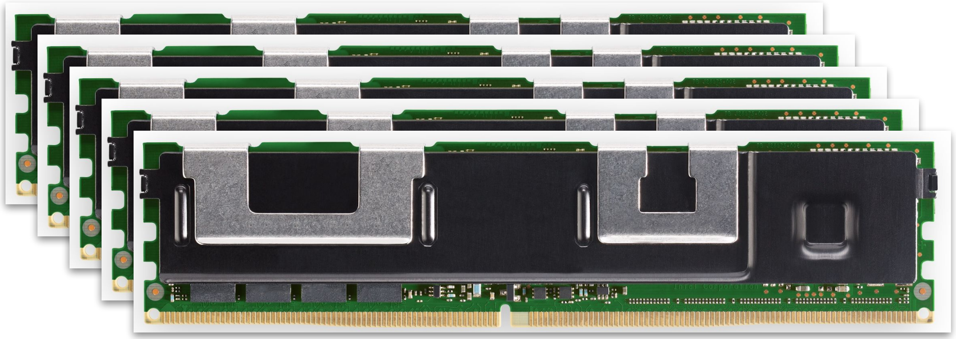
https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/virtualization/qemu

Now Image a System





512 GB of Optane DIMM



A CPU

- With a typical 12 DIMM slots
- Dual socket = 24
- 24 x 512 GB = 12.3 TB of persistent storage

No DRAM, only persistent memory

Imagine a System with Just Optane DRAM

1. What do storage and memory mean then? Two-level of storage?
    a. Virtual memory, paging, address translation?
    b. File systems, buffer caches, files, permissions?
    c. Single address space operating systems

2. What does execution mean?
    a. What does application installation (on fs) and execution (in DRAM) mean?
    b. What do updates mean? A new checkpointed application state?

3. Operating system design
    a. Booting? What is that
    b. How does OS (no temporary state) interacts with devices (have DRAM, can restart)
    c. Data corruption, fault isolation in architecture specific code (less portability)

Operating System Implications of Fast, Cheap, Non-Volatile Memory, https://www.usenix.org/legacy/events/hotos11/tech/final_files/Bailey.pdf

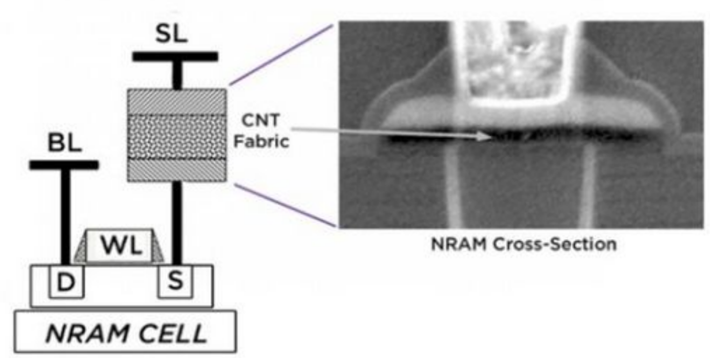
Most Importantly - This will not work anymore!

https://www.pinterest.com/pin/61572719880076742/

More New Technologies are Coming

First carbon nanotube NRAM products due in 2020, says Nantero

April 14, 2020 //By Peter Clarke 1 Comment



NRAM™ cell with CMOS select transistor and CNT resistive change memory element shown in SEM cross-section.

https://www.eenewsanalog.com/news/first-carbon-nanotube-nram-products-due-2020-says-nantero

What You Should Know From This Lecture

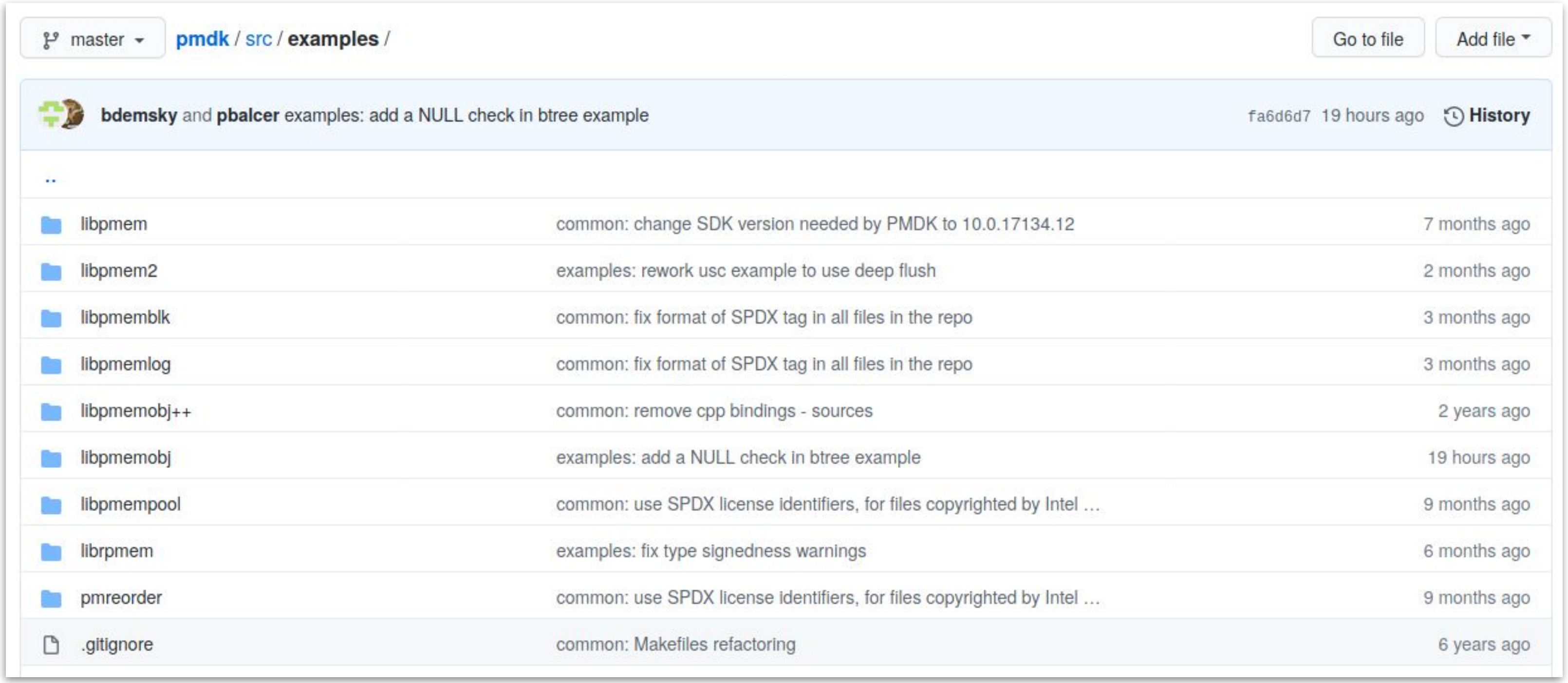
1. NV memory (Optane) integration options
2. Optane ballpark performance numbers
3. Concerns with the integration of NVRAM
    a. How do they integrate into the caching hierarchy
    b. Various options to write back
    c. What is ADR (eADR) and why is it necessary
4. Basic idea of a NVRAM file system (e.g., NOVA)
5. Basic idea and challenges in building a persistent heap (NVHeap)
6. PMDK and pmem project, and what do they do and what they provide

This is a very active area of research as the real hardware becomes available

Further Reading

1. DRAM internals, https://course.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:lectures:onur-740-fall11-lecture25-mainmemory.pdf
2. Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09).
3. Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20).
4. Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent memcached: bringing legacy code to byte-addressable persistent memory. In Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'17). USENIX Association, USA, 4.
5. Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Data structures for Persistent Memory. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20).
6. Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII).
7. http://cseweb.ucsd.edu/~swanson/Pubs.php
8. Lu Zhang and Steven Swanson. 2019. Pangolin: a fault-tolerant persistent memory programming library. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). USENIX Association, USA, 897–911.
9. Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, Ethan L. Miller, Twizzler: a Data-Centric OS for Non-Volatile Memory, USENIX ATC 2020, https://www.usenix.org/system/files/atc20-bittman.pdf

Examples on Github



master pmdk / src / examples /

Go to file | Add file

bdemsky and **pbalcer** examples: add a NULL check in btree example fa6d6d7 19 hours ago **History**

..

libpmem	common: change SDK version needed by PMDK to 10.0.17134.12	7 months ago
libpmem2	examples: rework usc example to use deep flush	2 months ago
libpmemblk	common: fix format of SPDX tag in all files in the repo	3 months ago
libpmemlog	common: fix format of SPDX tag in all files in the repo	3 months ago
libpmemobj++	common: remove cpp bindings - sources	2 years ago
libpmemobj	examples: add a NULL check in btree example	19 hours ago
libpmempool	common: use SPDX license identifiers, for files copyrighted by Intel …	9 months ago
librpmem	examples: fix type signedness warnings	6 months ago
pmreorder	common: use SPDX license identifiers, for files copyrighted by Intel …	9 months ago
.gitignore	common: Makefiles refactoring	6 years ago