

Storage Systems (StoSys)

XM_0092

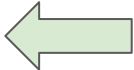
Lecture 6: Byte-Addressable Persistent Memories

Animesh Trivedi

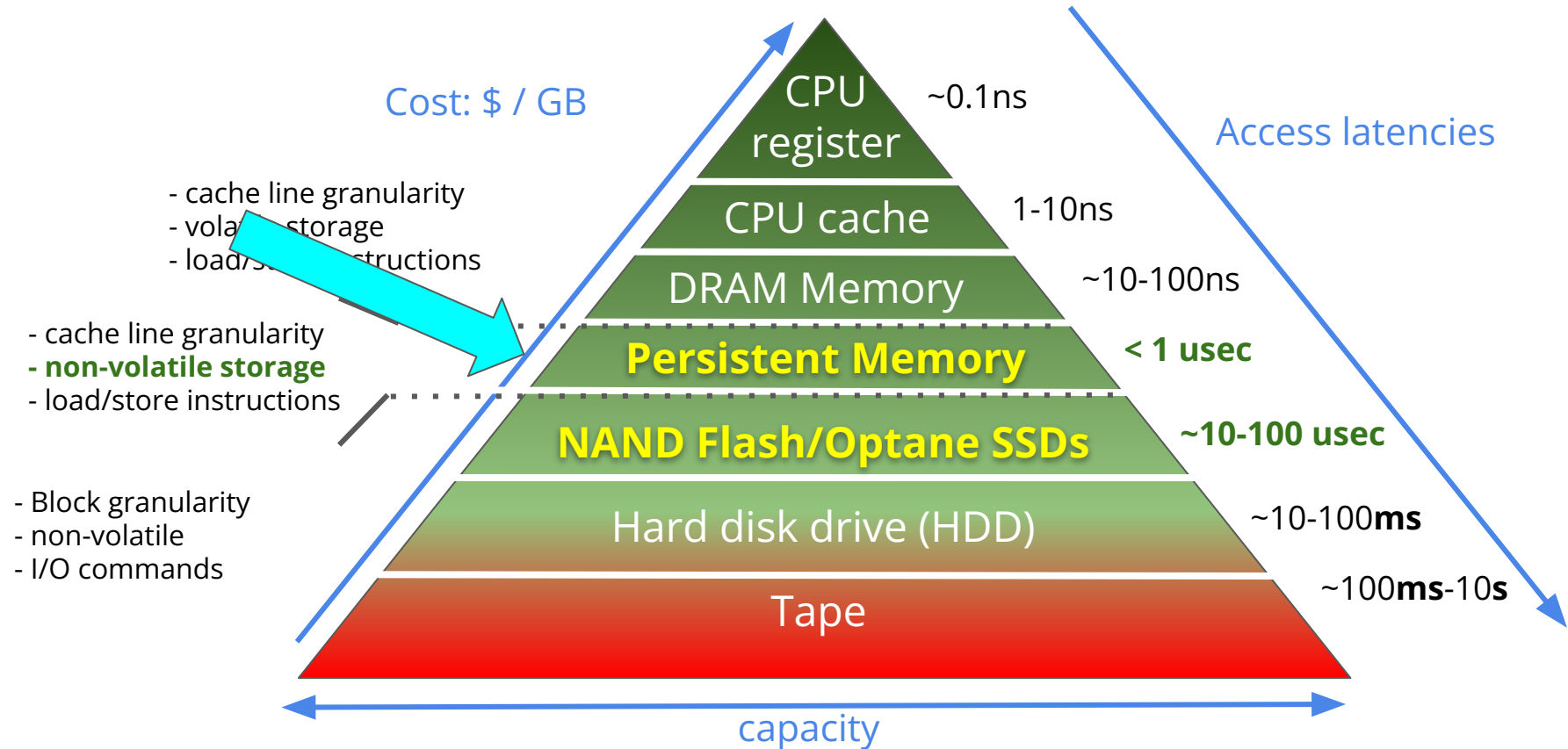
<https://stonet-research.github.io/>

Autumn 2023, Period 1

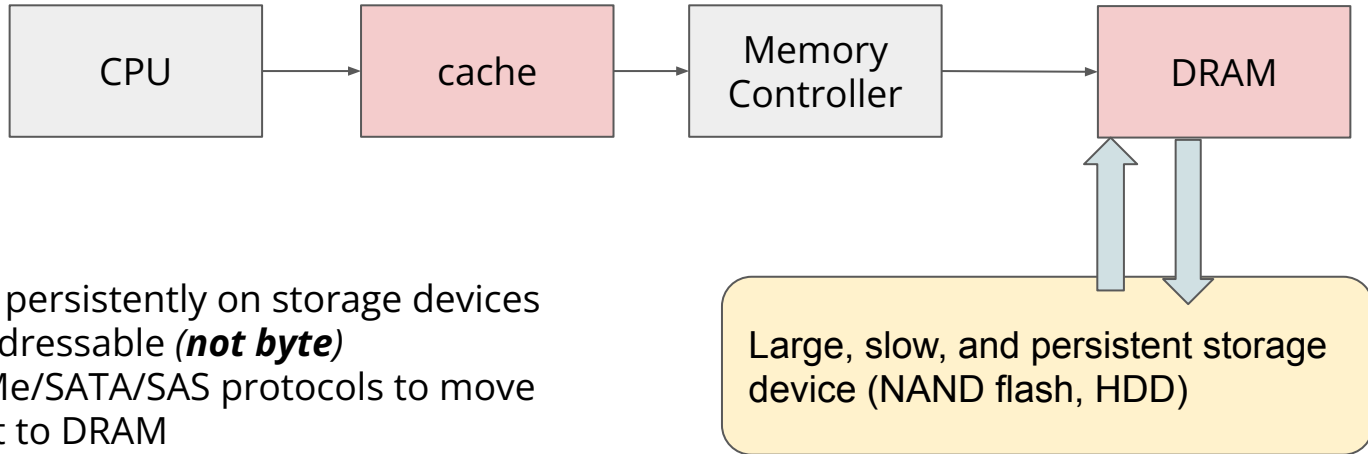
Syllabus outline

- ~~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
11. Emerging Topics

The (new) triangle of storage hierarchy



The Basic Storage Model

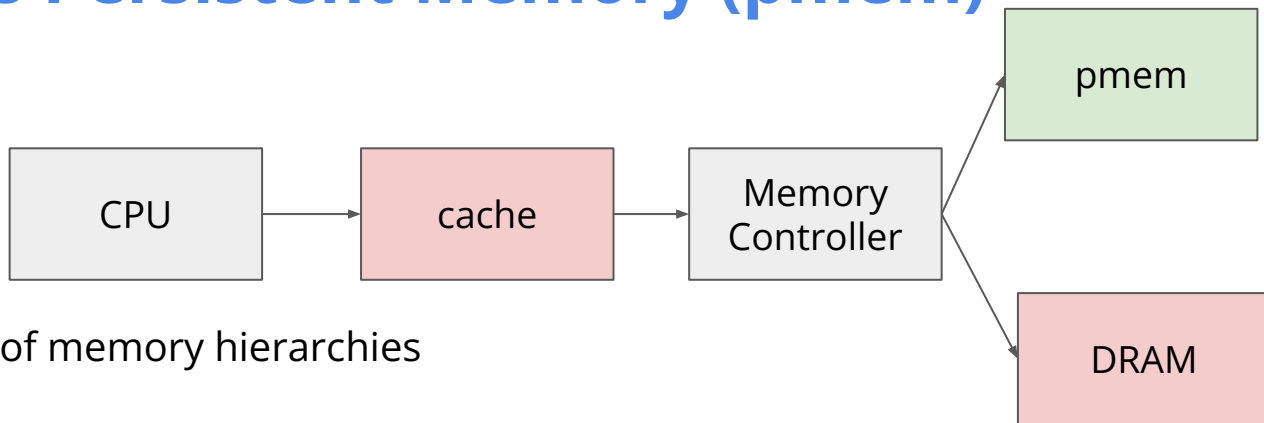


Data is stored persistently on storage devices

- Block addressable (**not byte**)
- 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 - responsibility of the application/OSes

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

NVM as Persistent Memory (pmem)



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 the “software” architectures before they arrived

→ *In this lecture we will cover the high-level concepts (it is a large area, see references)*

Keep in mind that **the abstraction of persistent memories** is much border and can be done on conventional devices also with mmap: https://web.eecs.umich.edu/~tpkelly/papers/Failure_atomic_msync_EuroSys_2013.pdf and https://www.usenix.org/system/files/login/articles/login_winter19_08_kelly.pdf

Today: Intel Optane

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

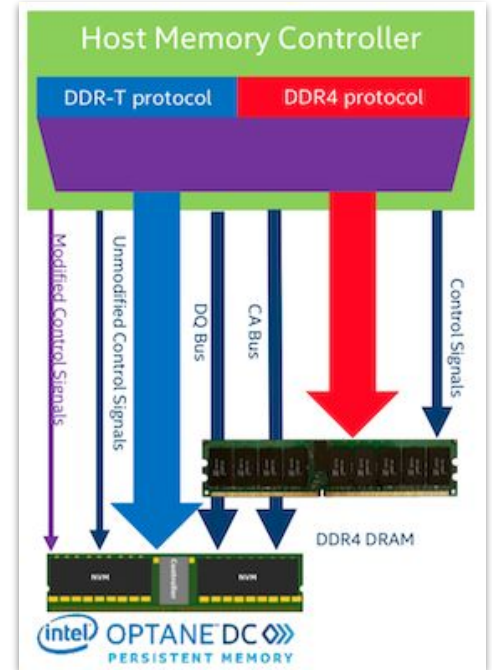
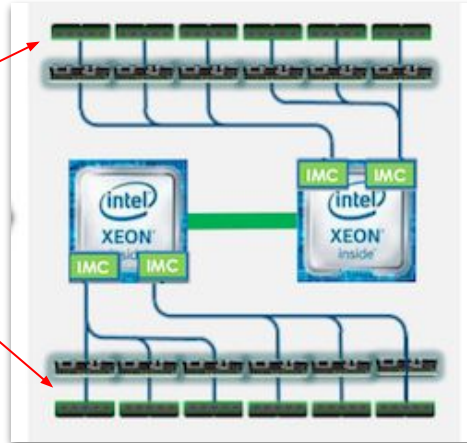
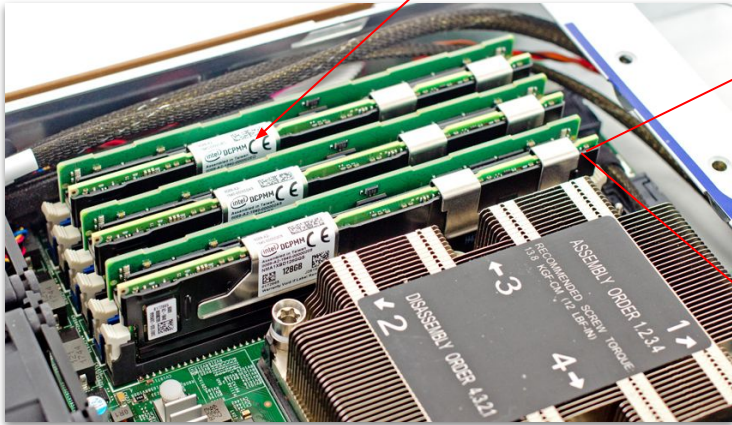
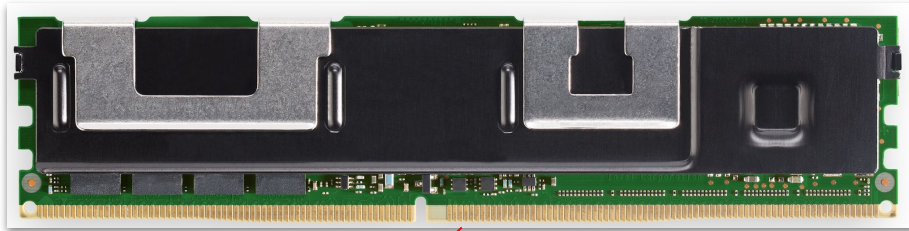
It is a byte-addressable, load-store accessible (from the CPU) storage that can be put in a DDR4 DIMM slot (uses the same mechanical and electrical protocols)

In comparison to DRAM

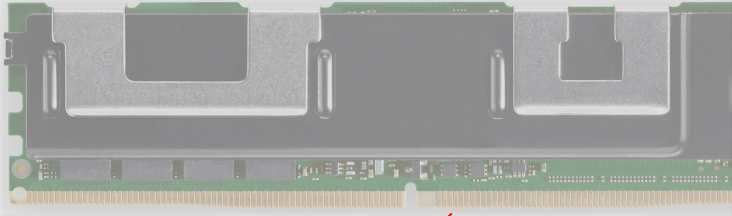
- **More capacity** : 128, 256, and 512 GB DIMMs (DRAMs are usually at 32-64GB then they get super expensive)
- **Cheaper** : than the DRAM (2-4x) times, but more expensive than Flash (10-100x)
- **Energy Efficient** : Unlike DRAM, no need to constantly refresh

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

Today: Intel Optane



Today: Intel Optane



TRENDING Best Tech Deals GPU Benchmark Hierarchy AMD Ryzen 7

Tom's Hardware is supported by its audience. When you purchase through links on our site, we may earn a commission.

Home > News

Intel Kills Optane Memory Business, Pays \$559 Million Inventory Write-Off

By Paul Alcorn published July 28, 2022

3D XPoint at the last crossroad.

 Comments (31)



(Image credit: Lenovo)

Update 08/02/2022 12:30am PT: Intel reached out to clarify that it would bring the next-gen Crow Pass Optane memory DIMMs to market and will use its existing inventory to fulfill orders. This wasn't clear from Intel's previous statement because this is technically a future product. We have clarified that point in the below text.

Optane Memory, writes inventory

chip giant

business, a line of memory that was slightly slower than high input/output operations per second.

\$59 million inventory impairment/write-off as it exits the



- Intel

memory chip business to SK Hynix, but kept onto Optane - a technology it developed with Micron in 2015.

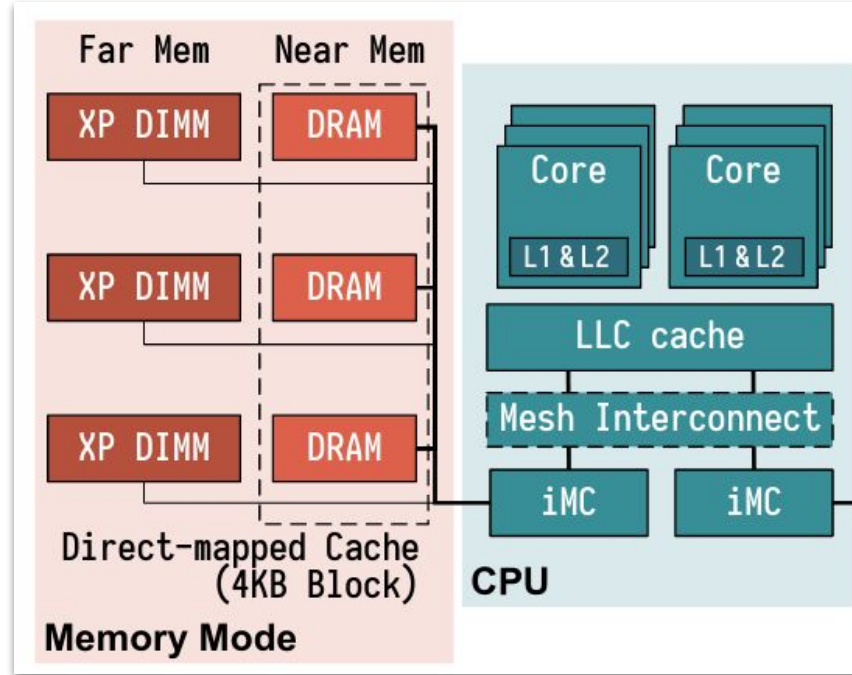
persistent memory DIMMs based on the technology, but was a complete failure in the consumer market, and was discontinued in January 2021.

ended in 2021, and left the market.

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



XP = 3D XPoint

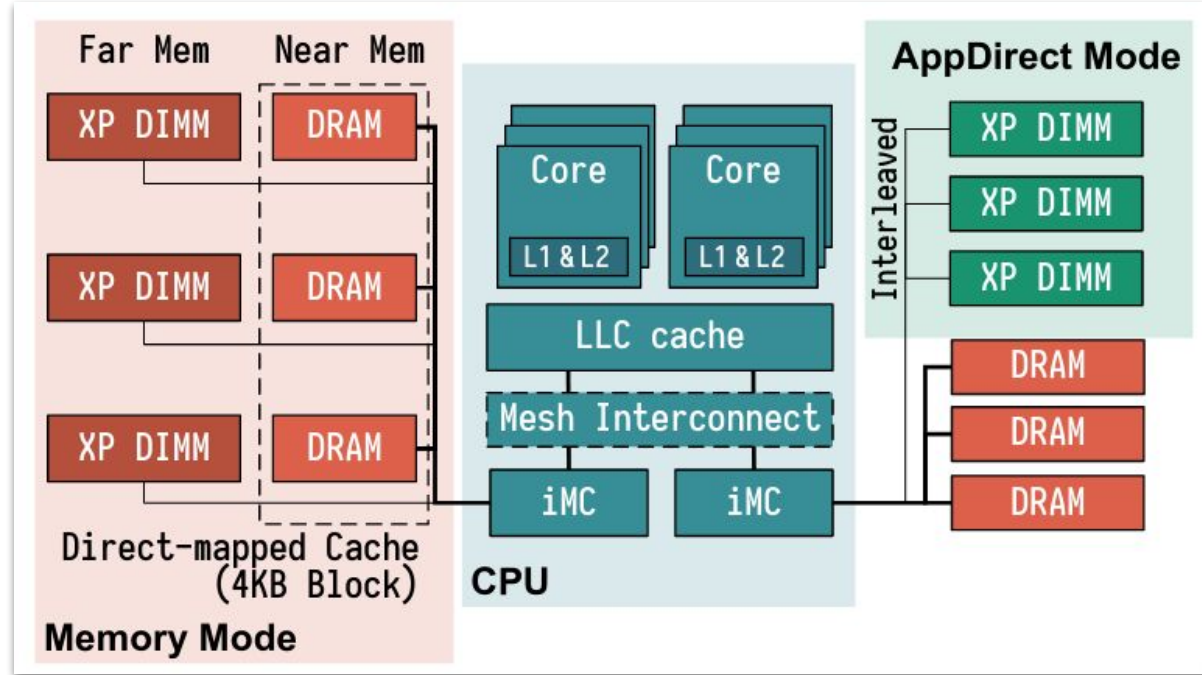
Optane Memory Layout and Operation Modes

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

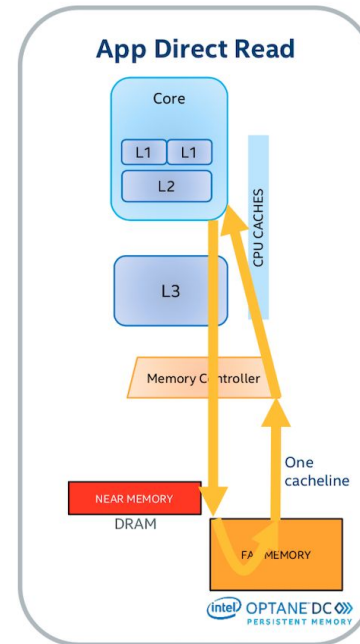
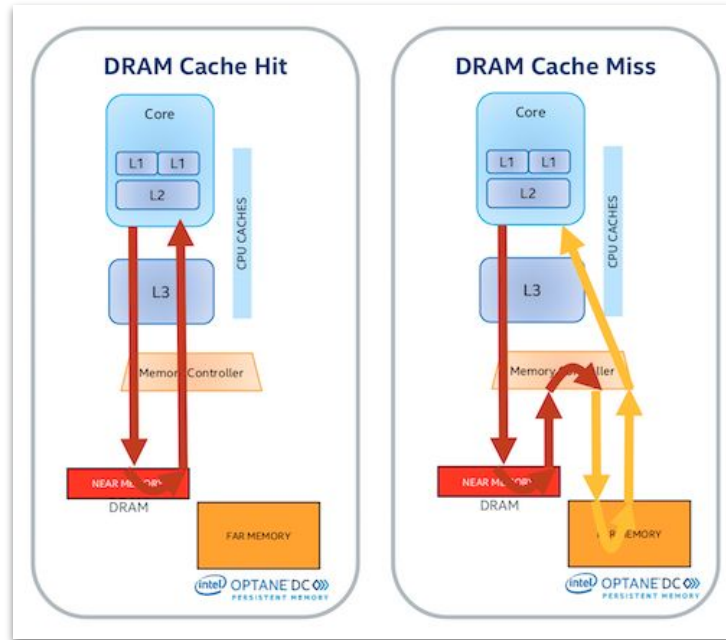
AppDirect Mode: Optane is used as a persistent memory and exposed to the OS/application

- Applications should be aware of its performance and persistence properties



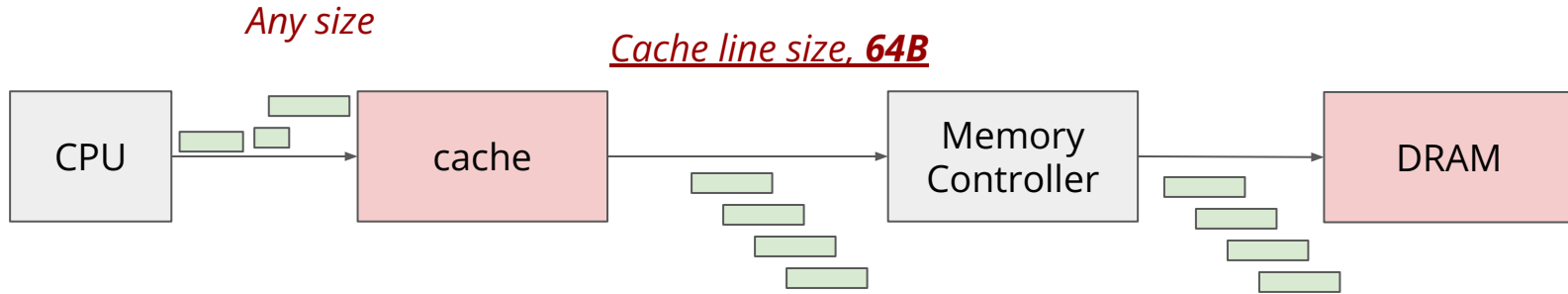
Optane Memory Layout and Operation Modes

Two modes: **Memory Mode** and **App Direct Mode** (these are Optane specific)



Potential challenges?

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

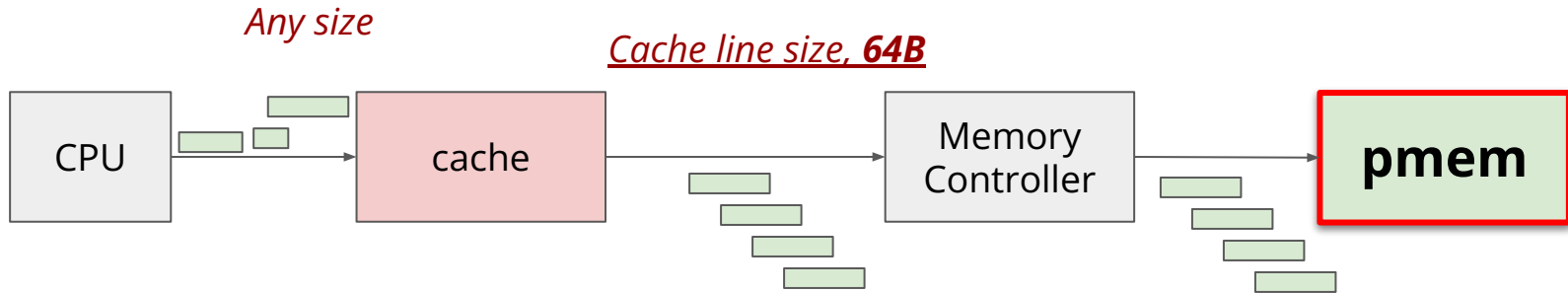
Caches are always managed in the cache line granularity (64B), and this is also the unit of DRAM access (*so, is DRAM truly a byte-addressable memory?*)

A memory controller can **re-order** loads and stores (out of order execution), hence, there is no guarantees in which order instructions get to DRAM (the program order is different than the execution order)

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 and questions important?

How Does the Current CPU Work? (simplified)



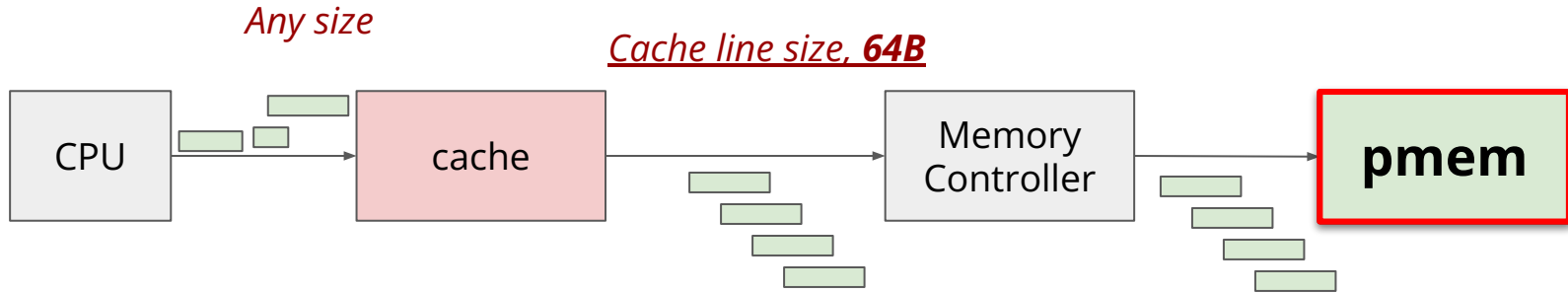
Now instead of DRAM, there is persistent memory

1. What if CPU writes are only stored in the cache (write-back mode)?
2. Do you program cache? Isn't the CPU cache suppose to be a micro-architecture, hence, it is an invisible CPU feature to the programmer?
3. What if CPU writes to pmem are re-ordered?

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 and questions 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

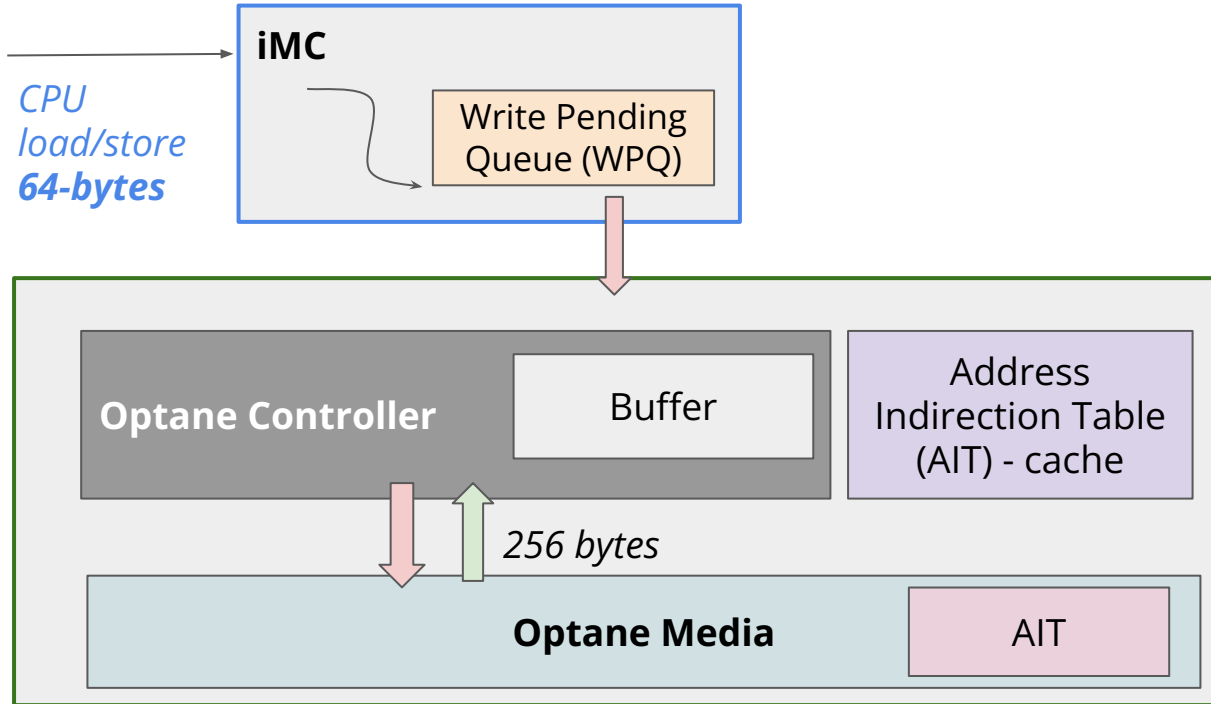
1. The Significance of the x86 SFENCE Instruction, <https://hadibraais.wordpress.com/2019/02/26/the-significance-of-the-x86-sfence-instruction/>
2. Memory part 5: What programmers can do <https://lwn.net/Articles/255364/>
3. https://en.wikipedia.org/wiki/X86_instruction_listings
4. <https://stackoverflow.com/questions/40096894/do-current-x86-architectures-suppor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

CPU Instructions to Control Data Flushing

CLFLUSH	This instruction, supported in many generations of CPU, flushes a single cache line. Historically, this instruction is <u>serialized</u> , causing multiple CLFLUSH instructions to <u>execute one after the other</u> , without any concurrency.
CLFLUSHOPT (followed by an SFENCE)	This instruction, newly introduced for persistent memory support, is like CLFLUSH but <u>without the serialization</u> . 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 <i>cache line write back</i> . 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 <u>non-temporal store</u> . 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 <u>kernel-mode-only instruction flushes and invalidates every cache line on the CPU</u> 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



Writes end up in iMC, at WPQ

Then flushed into Optane DIMM

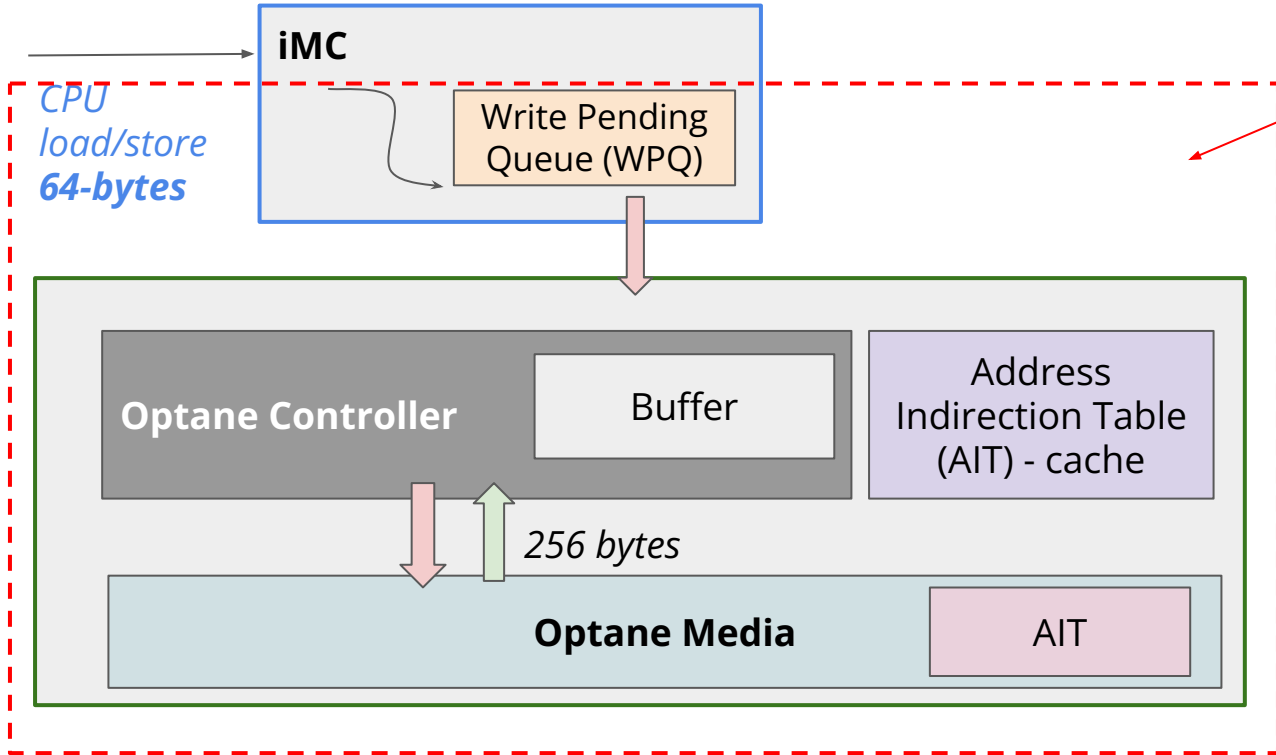
Optane DIMM has a write buffer, where 64 bytes r/w are merged into 256 bytes accesses to Optane

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?

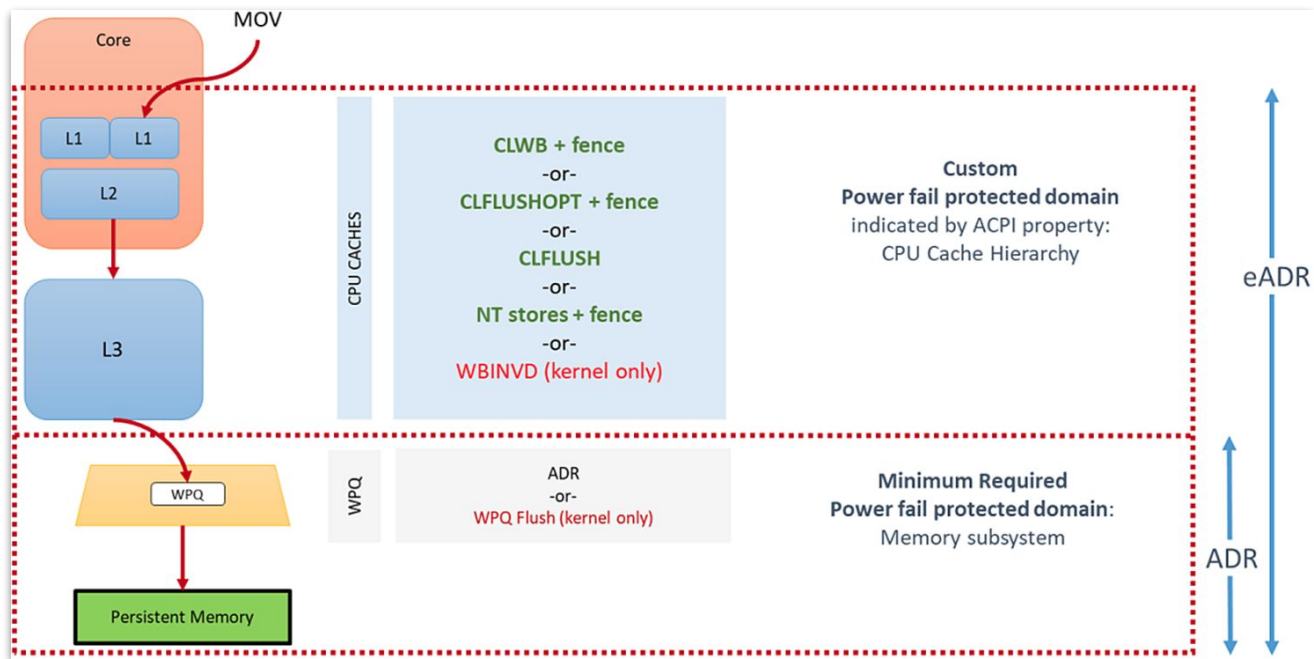
Optane Internals - the Write Path



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)

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[†]

{jiansyang, 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 architectural details on our test machine and the Optane DIMM

System Setup

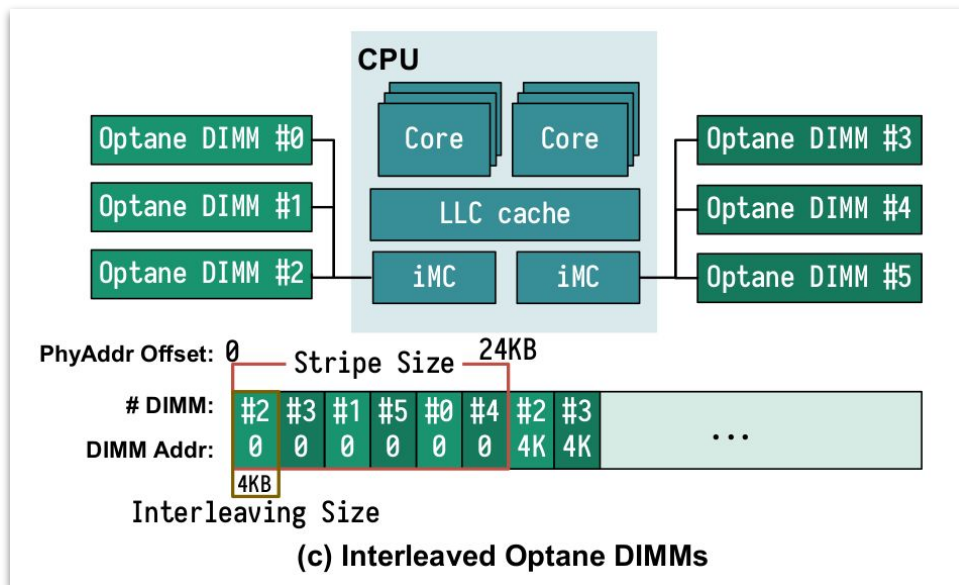
2 x CPU 24 cores Cascade Lake

Each CPU: 2 x iMC with 3 memory channels each

Total 6 channels for DRAM and Optane

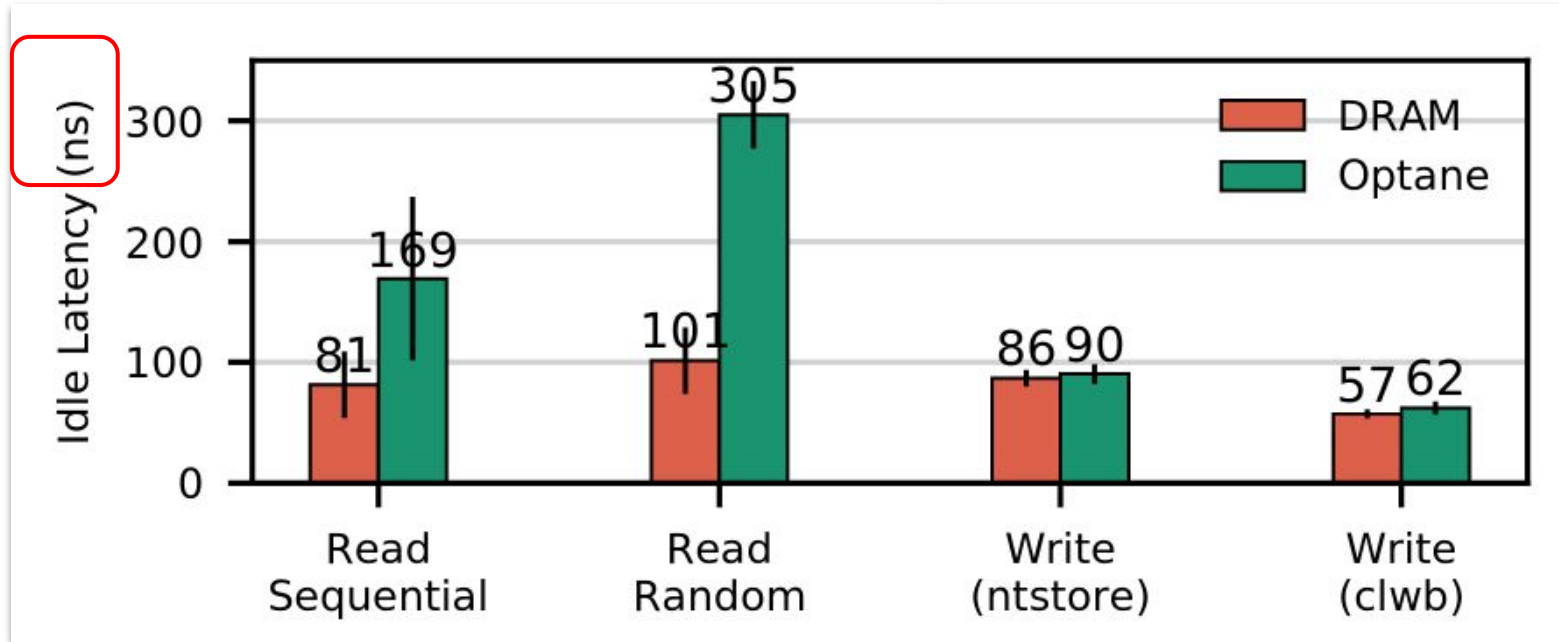
2 CPU x 6 Ch. x 32 GB = **192 GB DRAM**

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



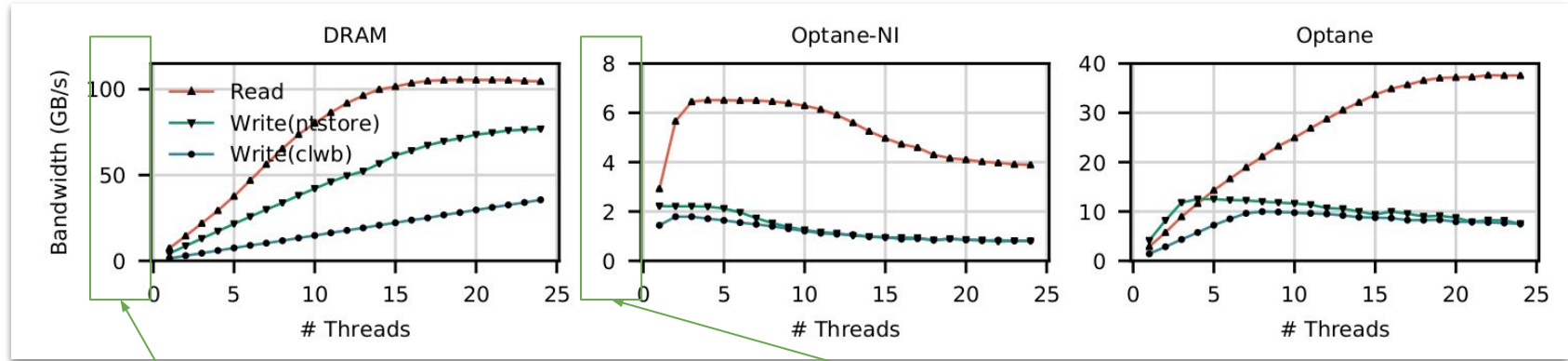
There are 2 of such CPUs

Basic Performance: Latency



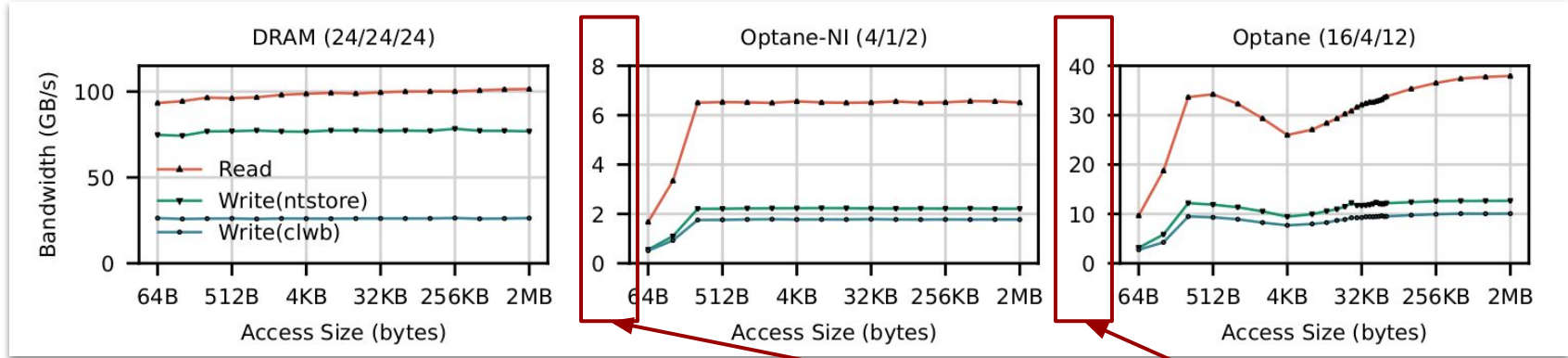
- The read latency for Optane is 2-3x higher than DRAM
- The random-vs-sequential gap is 20% for DRAM but 80% for Optane memory
- Write performance measures writes reaching the ADR domain (not necessarily Optane)

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.

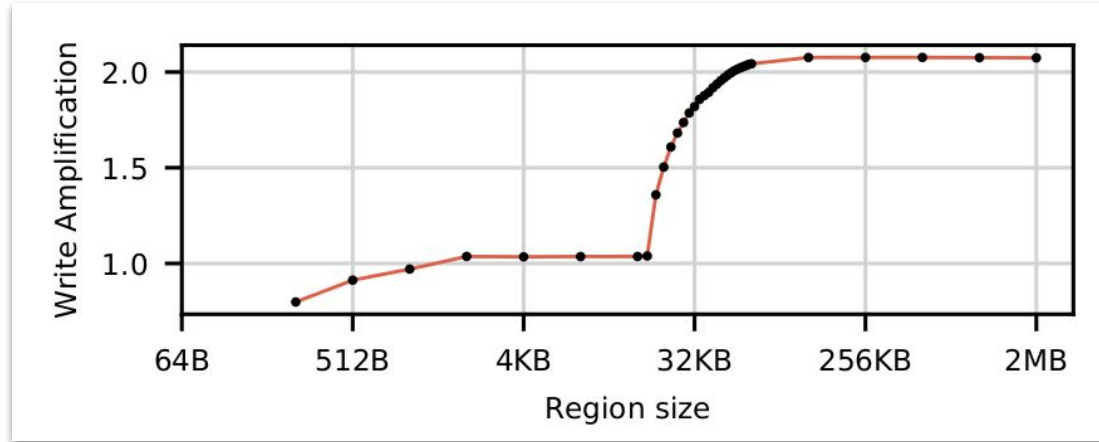
Optane Bandwidth Comparison - Access Size



- DRAM performance is independent of the access 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



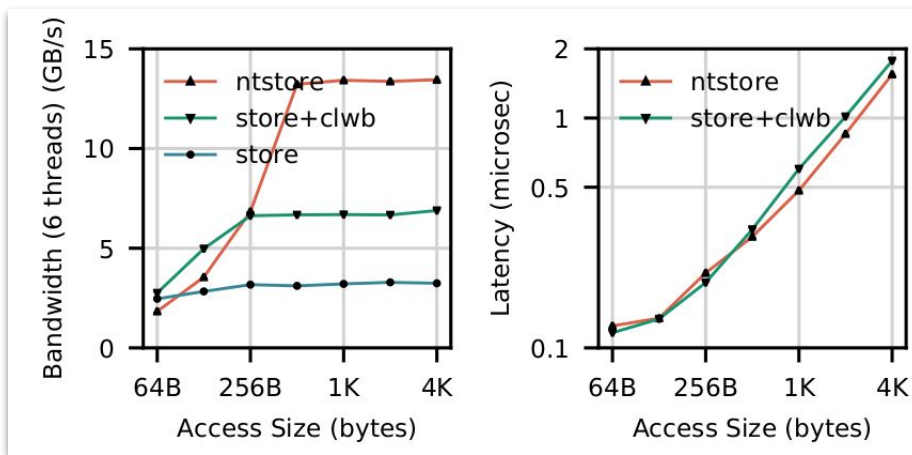
Write addresses repeatedly which are separated by certain XP Line size (256B)

Measure WA (DIMM counter), which shows at 16 lines, $64 \times 256 = 16 \text{ KB}$ buffer

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

Further reading: Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. **Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering.** In Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22). Association for Computing Machinery, New York, NY, USA, 488–505. <https://doi.org/10.1145/3492321.3519556>

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

[Not covering]


What should be the interface between the CPU and persistent memory?

Data copy cost?

DMA engines - how to use them?

- Optimize - pre-pinning, batching, and parallelism

Frees the CPU from “synchronous” data copies!



Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems


Jingbo Su, Jiahao Li, and Luofan Chen, *University of Science and Technology of China*; Cheng Li, *University of Science and Technology of China and Anhui Province Key Laboratory of High Performance Computing*; Kai Zhang and Liang Yang, *SmartX*; Sam H. Noh, *UNIST & Virginia Tech*; Yinlong Xu, *University of Science and Technology of China and Anhui Province Key Laboratory of High Performance Computing*

<https://www.usenix.org/conference/fast23/presentation/su>

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA
978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by



Persistent Memory Programming (2017)

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.

Good Old-Fashioned Persistent Memory

TERENCE KELLY

Terence Kelly studied computer science at Princeton and the University of Michigan, earning his PhD at the latter in 2002. He then spent 14 years at Hewlett-Packard Laboratories. During his final five years at HPL, he developed software support for non-volatile memory. Kelly now teaches and evangelizes the persistent memory style of programming. His publications are listed at: <http://ai.secs.umich.edu/~tpkelly/>, tpkelly@secs.umich.edu

Byte-addressable non-volatile memory (NVM)—Intel Optane—is now shipping in volume. Today's NVM offers performance between that of DRAM memory and flash storage [2, 7] and can be accessed via either storage or memory interfaces [8]. The latter offers the prospect of radically simplifying application software by allowing direct manipulation of persistent data via CPU instructions (LOAD and STORE), thus offering an alternative to traditional persistence technologies such as relational databases and key-value stores. Industrial adoption of NVM and its corresponding style of programming is growing [9].

Given the excitement surrounding novel NVM hardware, now is a good time to remind ourselves that it has long been possible to implement a software abstraction of persistent memory (p-mem) on conventional hardware—ordinary volatile DRAM and block-addressed durable storage devices. The corresponding “p-mem style of programming” resembles the style that NVM invites, and supports similar simplifications, but doesn't require special NVM hardware.

This article illustrates p-mem programming on conventional hardware with C code for UNIX-like operating systems; all code is available at [3]. Spoiler alert: the basic technique is to lay out application data in memory-mapped files, with help from a few easy tricks and patterns. Because conventional `memset()` doesn't guarantee data integrity in the face of failures, crash consistency requires extra support. The right crash consistency mechanism for p-mem programming on conventional hardware is *failure-atomic asyncnc* (FAMS) [6], and this article presents a concise new implementation of FAMS.

A Persistent Linked List

The C program below prepends words from `stdin` to a persistent singly linked list. It relies on a bare-bones persistent memory library, `pmem`, presented later. Notice that the list node data structure's `next` field is not a conventional pointer but rather an *offset*—specifically a `pmem_t` (“persistent memory offset type”), defined as a `uintptr_t` in `pmem.h`. Under the hood, `pmem` computes offsets relative to the base address where persistent data are mapped, which may vary on different runs of the program. Offsets allow data structures to be relocatable, which improves portability and facilitates sharing persistent data between different applications. The alternative of non-relocatable persistent data offers different tradeoffs and is beyond the scope of this article; see [5] for a discussion.

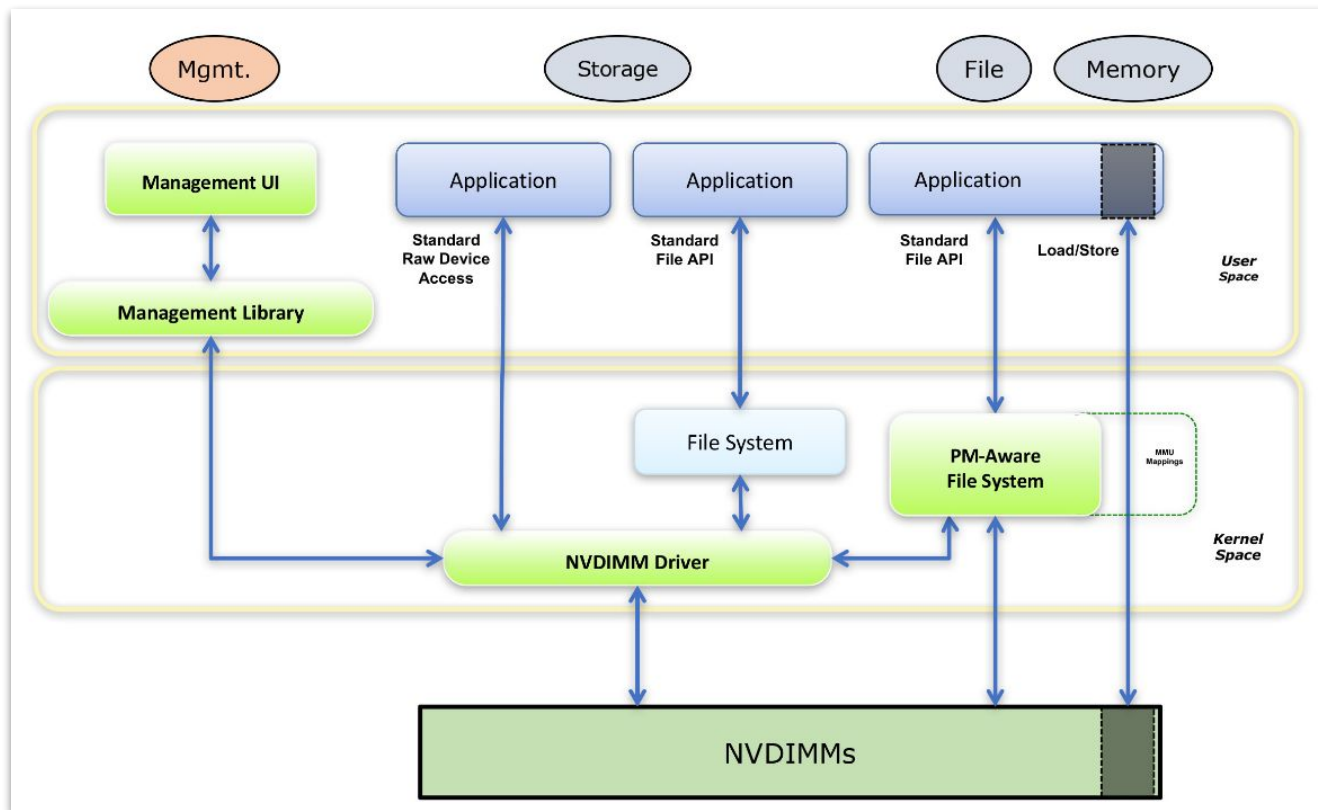
```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "pmem.h"

typedef struct {
    pmem_t next;
    char string[];
} node_t;

#define NPGO ((node_t**)pmem_map(0))
```

See also

So, How Do You Program/Manage Your PMEM DIMMS?



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

Understand: Storage and Memory

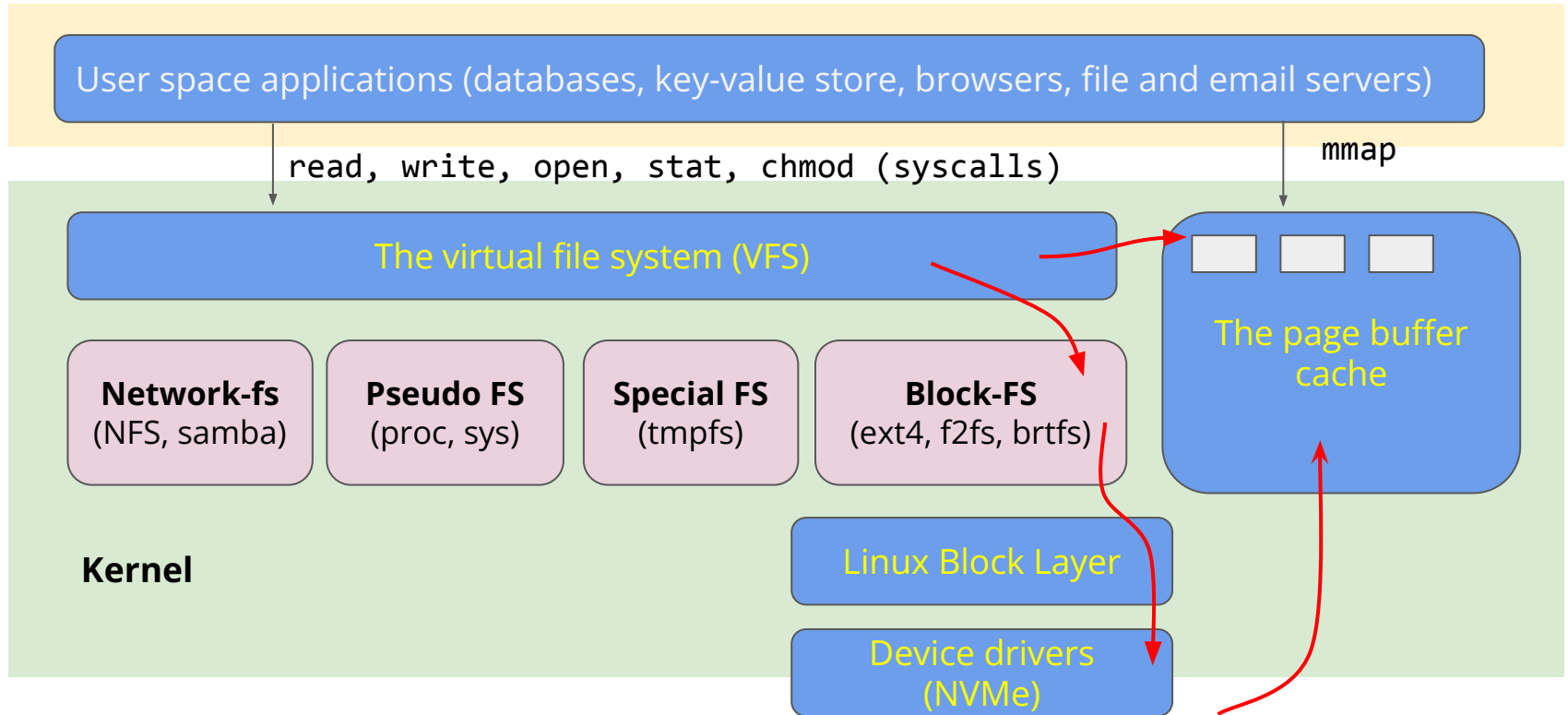
Storage and Memory are what is known as classical two-level storage system

- **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

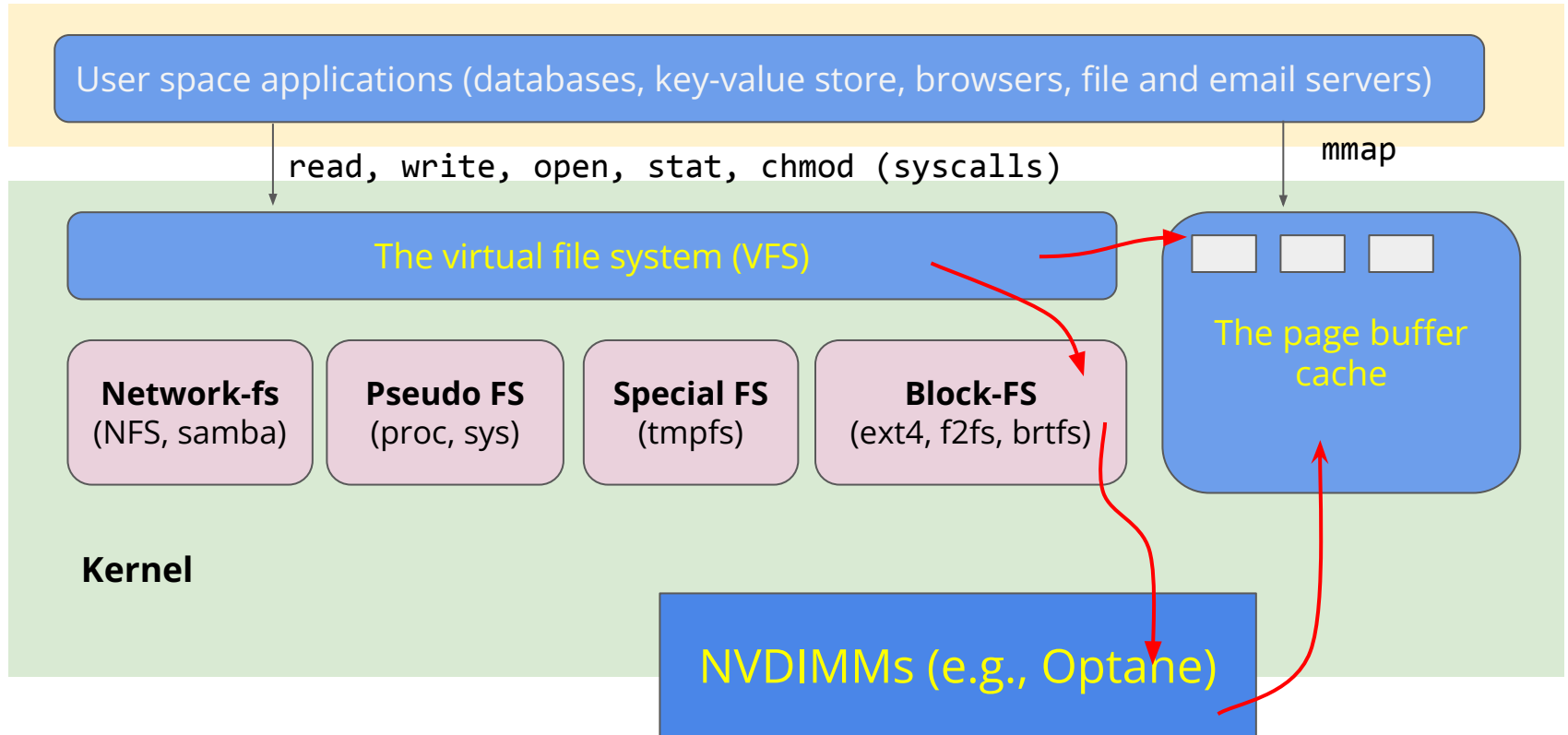
Why do we want to run a file system on top of a persistent memory?

- 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

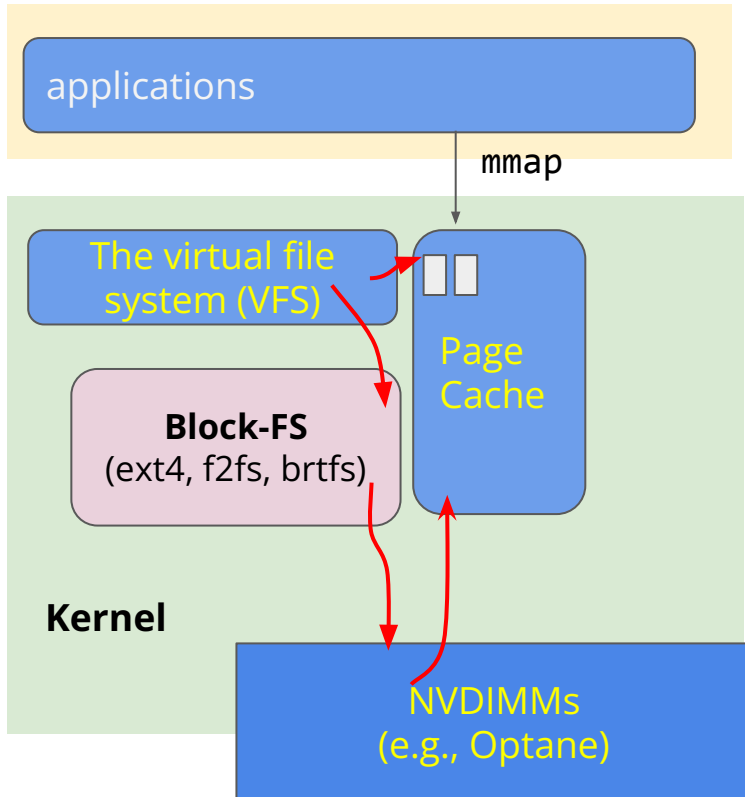
Looking at the Storage Stack Again



Looking at the Storage Stack Again



What Happens When I mmap a Page?



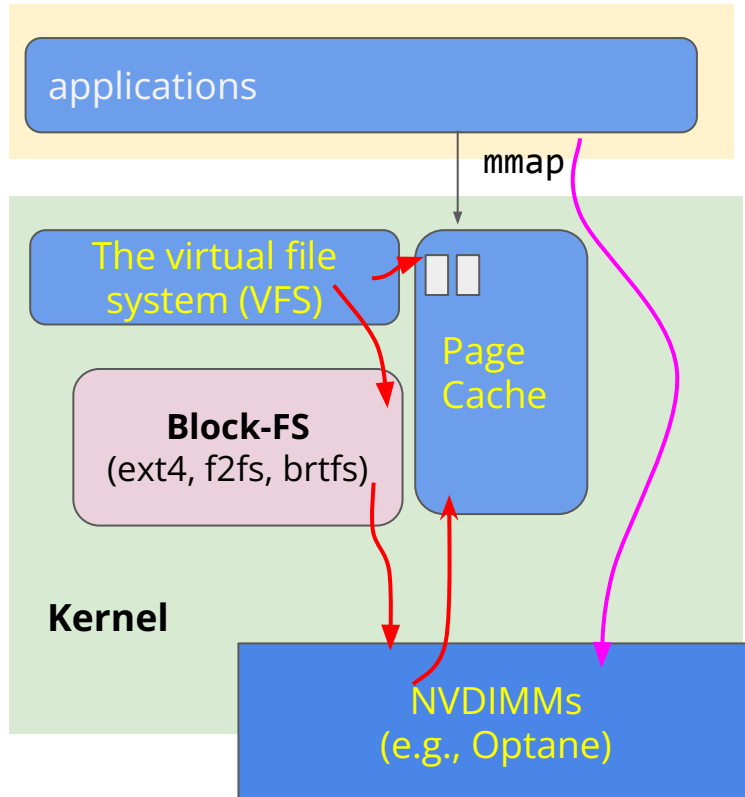
mmap takes pages from the page cache

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? to do so on mmap?

Direct Access (DAX) Extensions for files



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

- read/write calls have their own optimizations with memcpy (need more support from FSes)

Needs modification into the file system to support this operation

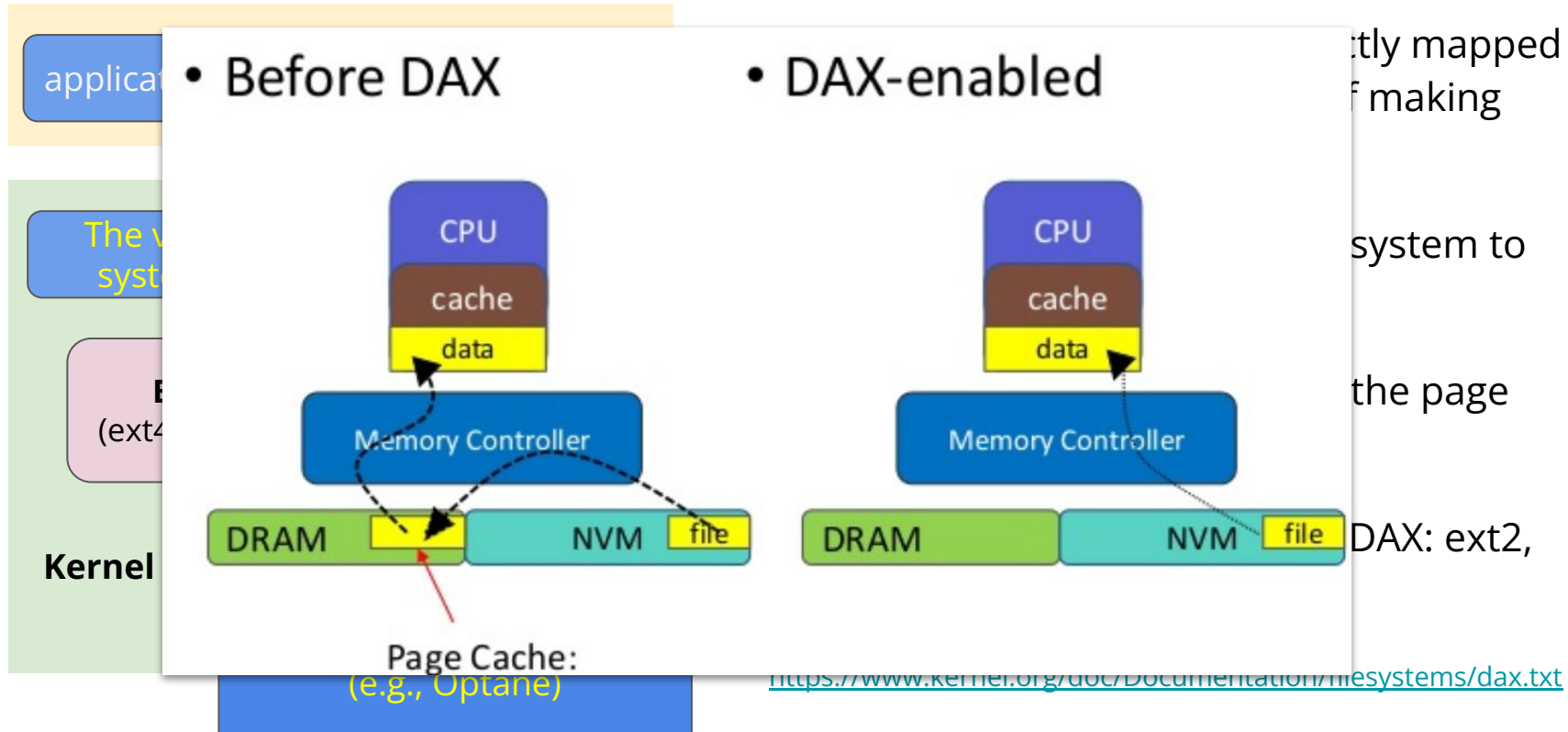
⇒ **Translating file offset to their PMEM locations**

The DIMM block size must be equal to the CPU page size

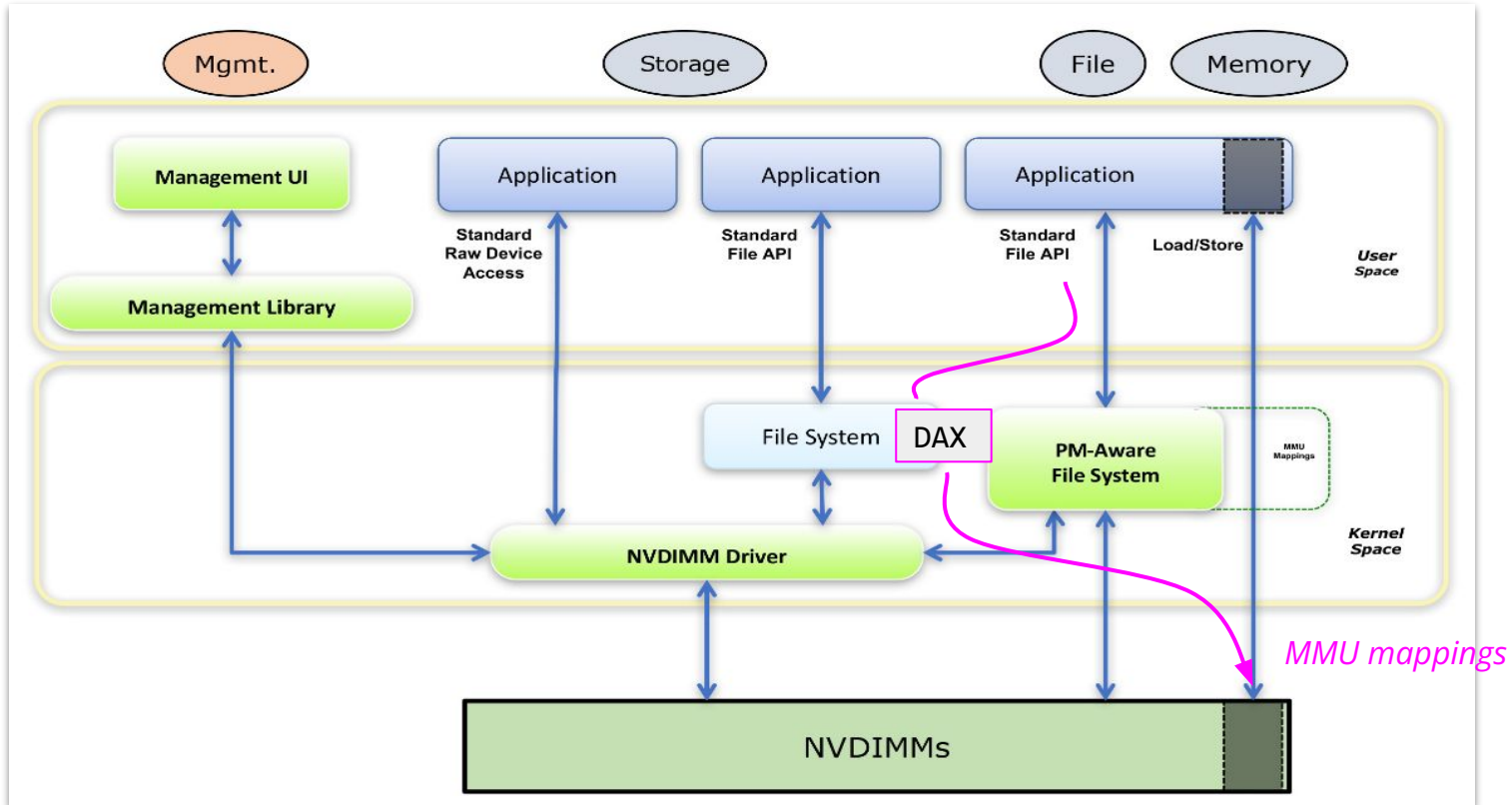
Multiple filesystems support DAX: ext2, ext4 and xfs

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

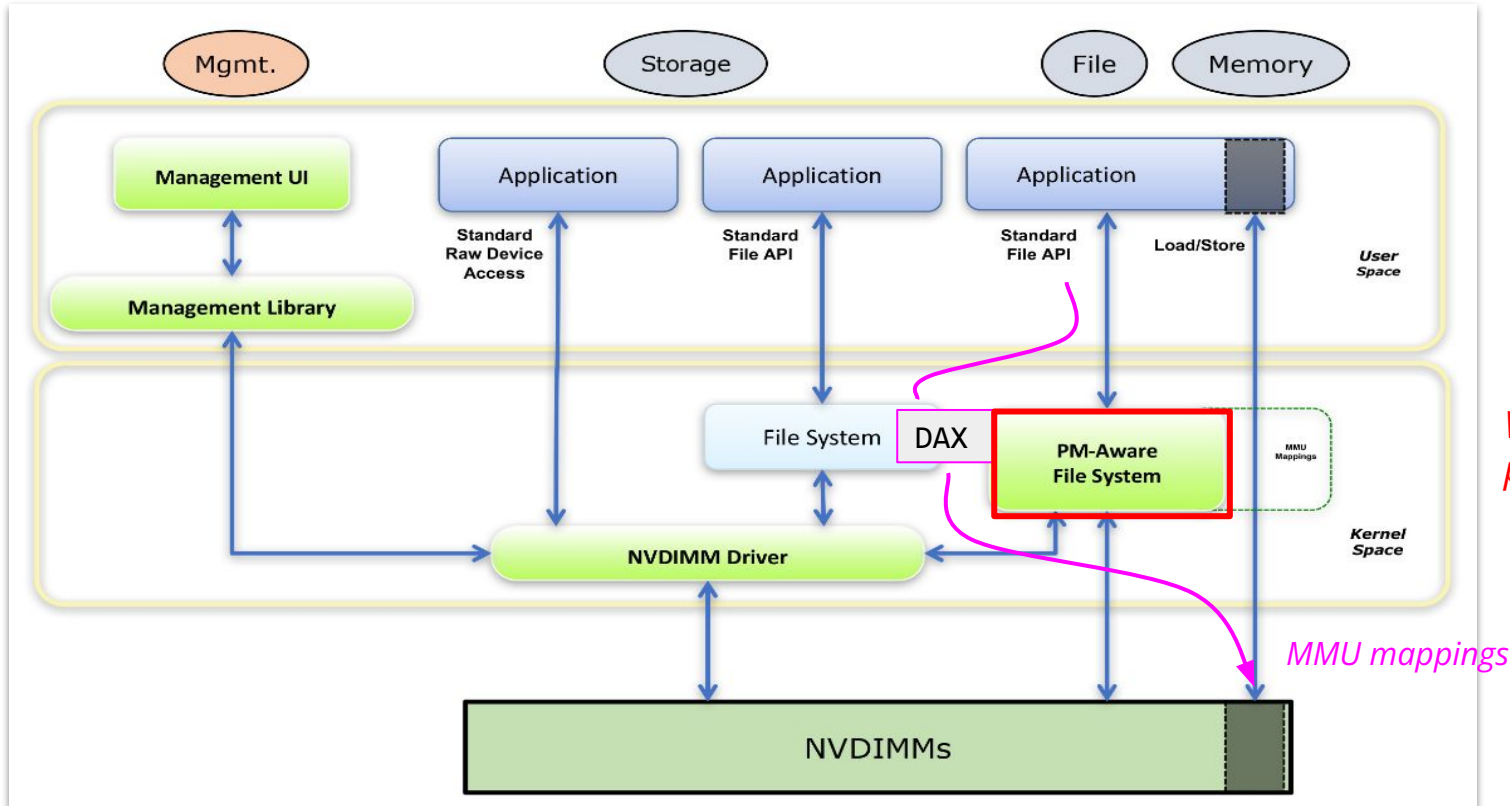
Direct Access (DAX) Extensions for files



Updated Stack Image



Updated Stack Image



What are pmem-aware fs?

There are a lot of Interesting File Systems Designs in this area

High-level Design	File System	User/Kernel Space	DAX	POSIX-compliant	Main Contribution
Influenced by Traditional File Systems (e.g. <i>ext2</i>)	BPFS [18]	Kernel	✗	✓	POSIX-compliant file system that reduces write amplification through adapted shadow paging
	PMFS [25]	Kernel	✓	✓	Bypass OS page cache and generic block layer, avoid extensive I/O stack modifications. Lightweight in-place metadata updates
	HiNFS [57]	Kernel	✓	✓	Elimination of double copy overhead in kernel
	Ext4-DAX [29]	Kernel	✓	✓	Include DAX to PM in the existing <i>ext4</i> file system
	Contiguous File Allocation	SCMFS [77]	Kernel	✗	✗
SplitFS [37]		Hybrid	✓	✗	Introduces a <i>hybrid</i> architecture in which data operations are handled in user space, while metadata operations are processed in the kernel
Aerie [71]		Hybrid	✗	✗	Allow user space applications to update metadata directly in user space
Kuco [15]		User	✓	✓	Address the poor scalability of existing PM hybrid file systems (e.g., SplitFS)
ZoFS [23]		User	✓	✓	Like Aerie, allow user space applications to update metadata directly in user space, however, with less kernel involvement
Log-Structured		NOVA [79]	Kernel	✗	✓
	Strata [39]	Hybrid	✓	✓	Capture unique properties of multiple storage devices in one file system

Table 4: PM File Systems categorized by their high-level design

Persistent Memory File Systems: A Survey

Wiebe van Breukelen
Vrije Universiteit Amsterdam

Abstract

Persistent Memory (PM) is non-volatile byte-addressable memory that offers read and write latencies in the order of magnitude smaller than flash storage, such as SSDs. This survey discusses how file systems address the most prominent challenges in the implementation of file systems for Persistent Memory. First, we discuss how the properties of Persistent Memory change file system design. Second, we discuss work that aims to optimize small file I/O and the associated metadata resolution. Third, we address how existing Persistent Memory file systems achieve (meta) data persistence and consistency.

Keywords. Persistent Memory, Storage Class Memory (SCM), Byte-addressable Memory, Memory-Aware File Systems, Intel Optane, Direct Access (DAX)

1 Introduction

Over the past several decades, data storage has become an indispensable part of modern society. However, modern storage had its origins in the early twentieth century. Charles Babbage, who is considered by some to be the “father of the computer”, introduced a simplistic form of storage in his Analytical Engine: a general-purpose computer that could be programmed by punch cards [19]. With the emergence of faster and more advanced computers in the 1960s, storage demand grew exponentially. As a result, *magnetic storage*, where data is stored on rotating platters, like on a hard disk drive (HDD), quickly gained traction. Until now, this growth has not slowed down.

As storage demands and processing power increased, a new bottleneck emerged. In demanding environments, such as data centers, data access time could not keep up with CPU speed. This speed gap between CPU and storage continues to grow, so faster storage devices are necessary [28, 27].

A Solid-State Drive (SSD), a form of *flash storage*, offers lower read and write latencies than an HDD, especially in a workload that involves a lot of random data accesses [41].

Like HDDs, SSDs exchange data by the smallest unit of access: a block [81]. Exchanging these blocks between the computer (or host) and storage efficiently is an ongoing challenge. Compared to CPUs, storage devices are an order of magnitude slower in terms of latency [33].

Operating systems strive to minimize the impact of high device latency on application speed. For example, the Linux kernel reduces the performance impact as much as possible by maintaining a *page cache*: a chunk of memory where the OS caches chunks of a file for later use. Based on access patterns, disk blocks can be loaded into memory proactively, allowing substantially lower access latencies [14].

Such mitigations are due to the view we had on storage over the past 50 years. We assumed a two-level storage hierarchy: a fast primary memory (e.g., DRAM) and slow secondary memory (e.g., HDD). Both memories have their own unique properties, for example, the access interface, location within the computer architecture, and access latencies. This has a large influence on the overall design of the Operating System. An alternative scheme, the *one-level storage hierarchy*, changes how we view storage as a whole. Instead of a hierarchy in which we combine the strengths of multiple storage devices, we switch to a hierarchy in which we combine storage and memory into a single device. Persistent Memory (PM) enables the use of such hierarchy [2]. It is a form of storage that is very related to DRAM in terms of access latency, the most significant difference being that PM is non-volatile while DRAM is volatile. A well-known example of Persistent Memory is Intel’s Optane Memory [36].

To better illustrate the position of PM in the storage hierarchy, consider Figure 1. PM is located between an SSD and a DRAM module in terms of access latencies and is accessed through CPU load and store instructions at cache line granularity; 64 bytes for the x86-64 architecture [74]. Note that the capacity and cost scale with the access latencies; storage located at the top (e.g., CPU caches) of the pyramid is scarce and costly compared to storage at the bottom of the pyramid, e.g. HDDs. In terms of data bandwidth, DRAM outperforms PM by quite a margin, see Table 1.

ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory


“How to leverage memory translation hardware for file system design”

Why: Unification and performance acceleration

Where: File offset → location

How: Use page tables as the core data structure

We are also building something similar



ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory

Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan, *University of Toronto*

<https://www.usenix.org/conference/fast22/presentation/li>

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies. February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

Open access to the Proceedings of the 20th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

The Extent of the Problem ...

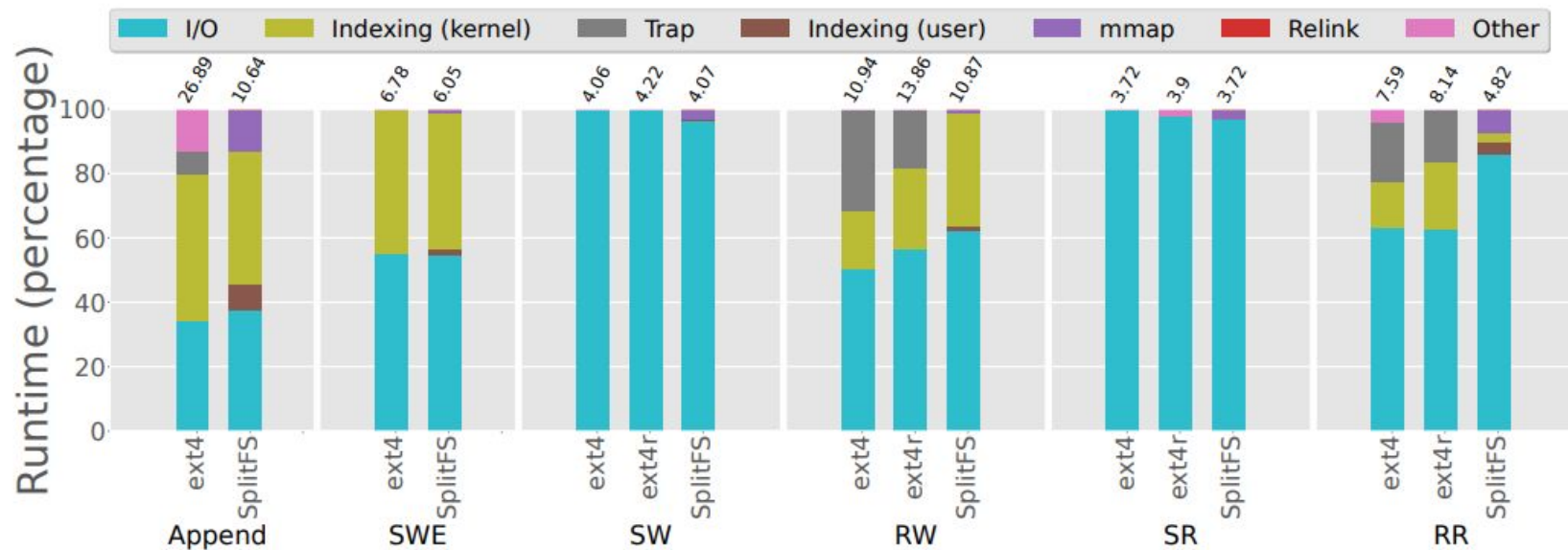


Figure 1: **Performance breakdown** (in percentage) of ext4-DAX and SplitFS on persistent memory. The number above each bar is the total run time in seconds.

Recap x86-64 page table

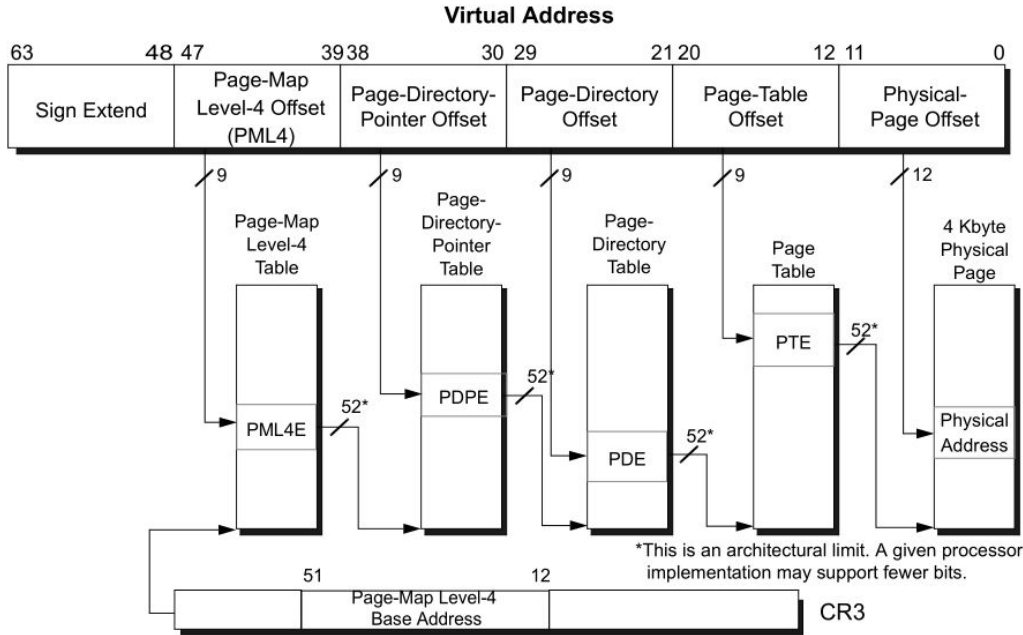
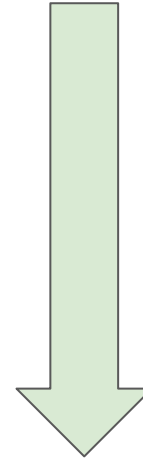
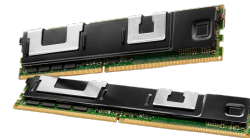


Figure 5-17. 4-Kbyte Page Translation—Long Mode

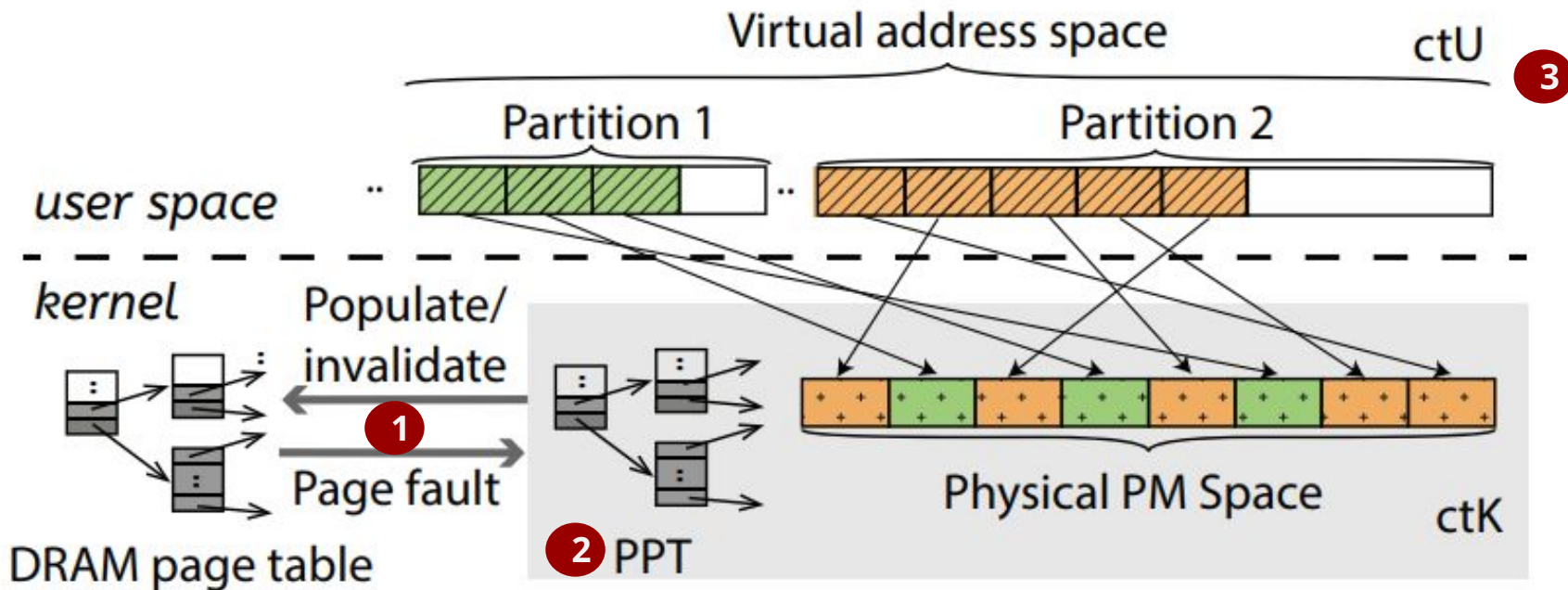
Complete file system (think: DFS)



MMU Accelerated Translation



Architecture of ctFS - Persistent Page Tables (PPT)



1. **DRAM page table vs. PMEM page table (TLBs):** how many levels, where to store
2. **PPT design :** management of pointer copies
3. **User-space split architecture :** kernel manages PPTs, and user-space lays out the FS

Atomic pswap

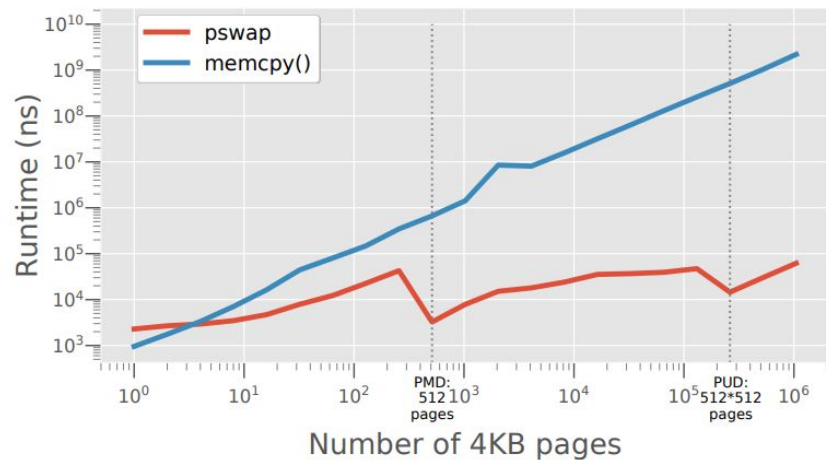
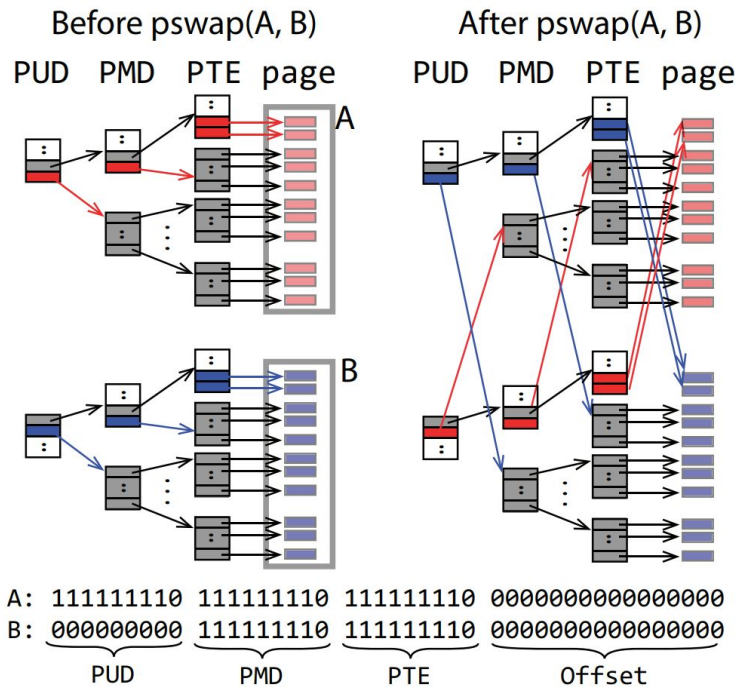
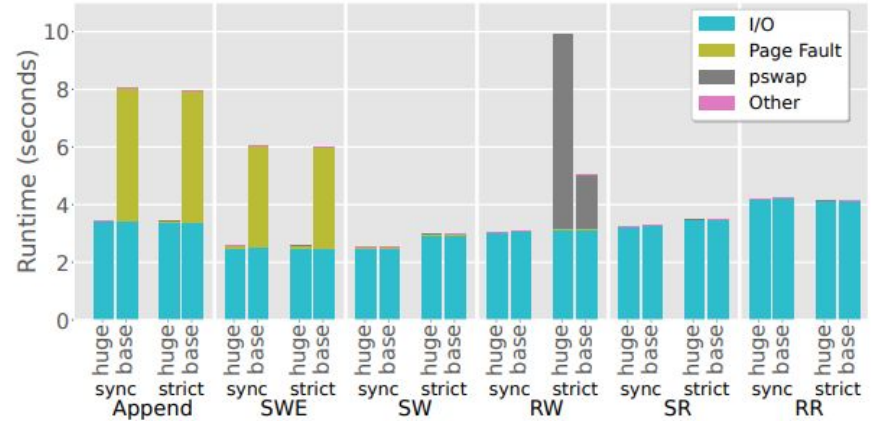
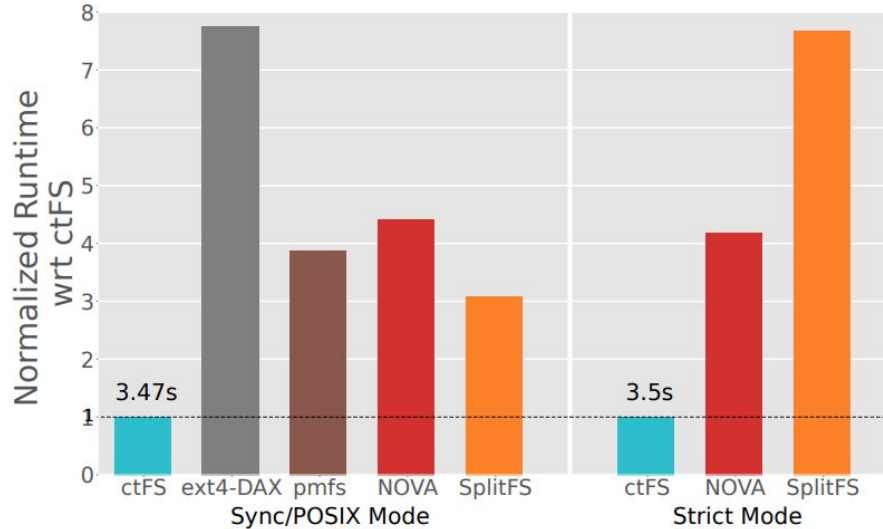


Figure 6: Comparing the performance of **pswap** and **memcpy()**. Both the X and Y axis are log scale.

Results...

Appends as 4KB on 10GB File



Is File System the Best Way to Use PMEM?

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

With a file system

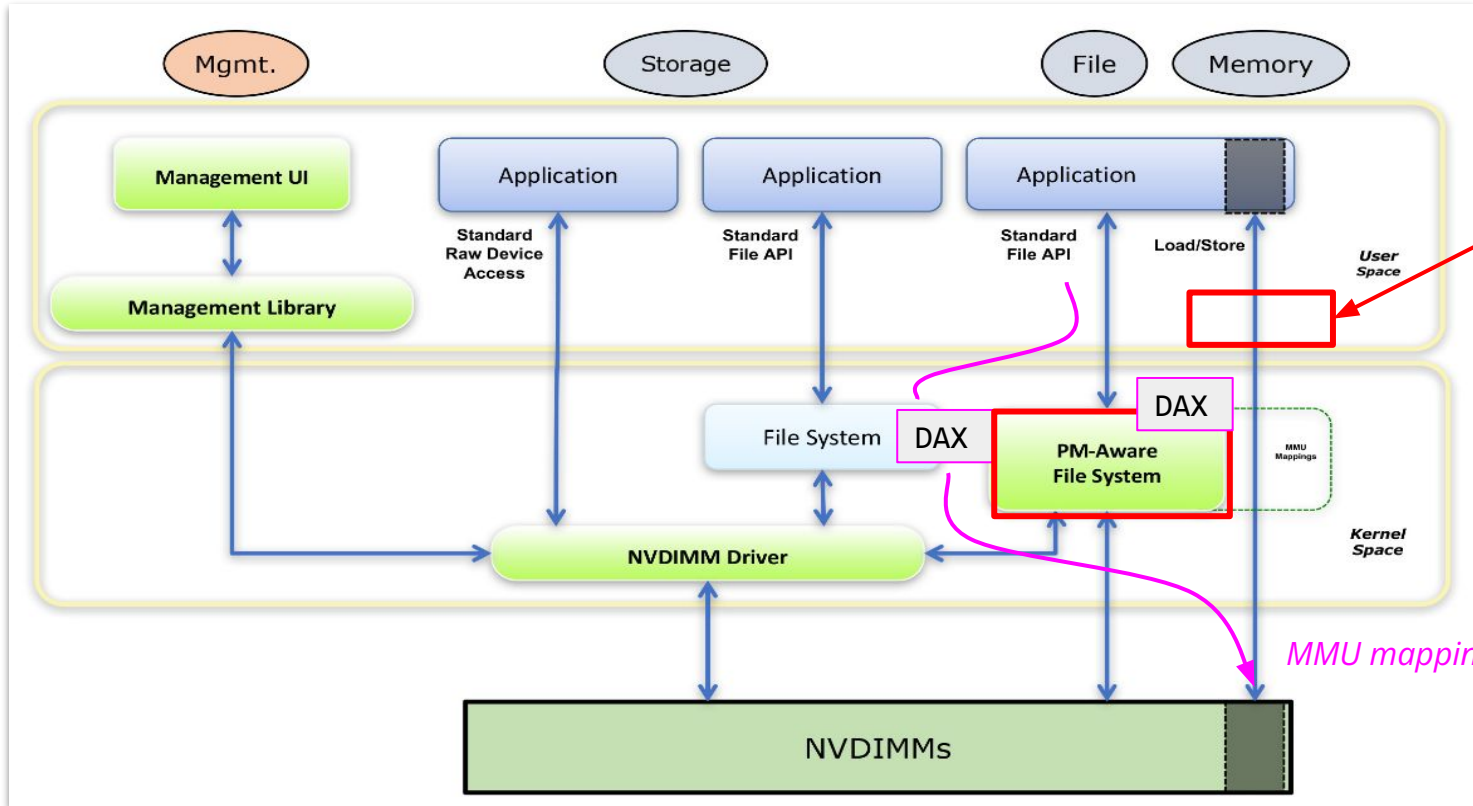
1. Application Data must first be (de)serialized when reading in
2. When writing out application data must be serialized to be written out to a file

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

MMU mappings

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 \times and 244 \times , 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: Motivation

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: Motivation

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

Is "l" 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*

The key problem is how to give proper system support to help mitigate these bugs and build safe, high performance, concurrent and durable data structures in pmem

NV-Heaps: Core Ideas

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?

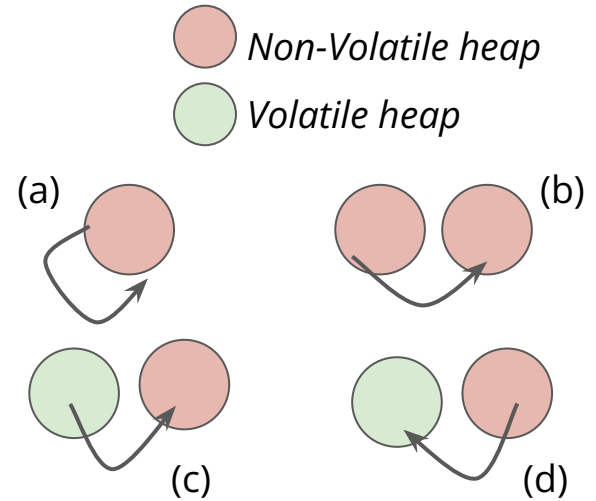
1. Pointer management:

- Non-Volatile (NV) pointers within a single NV heap
- NV pointers to another NV heap pointer
- Volatile (V) pointers to a NV pointer
- NV heap pointer to a volatile pointer

2. Memory management: memory leaks, double free

3. How and when to make structure consistent, and concurrent

- Locking, transaction issues ← hard to get it right even with DRAM



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

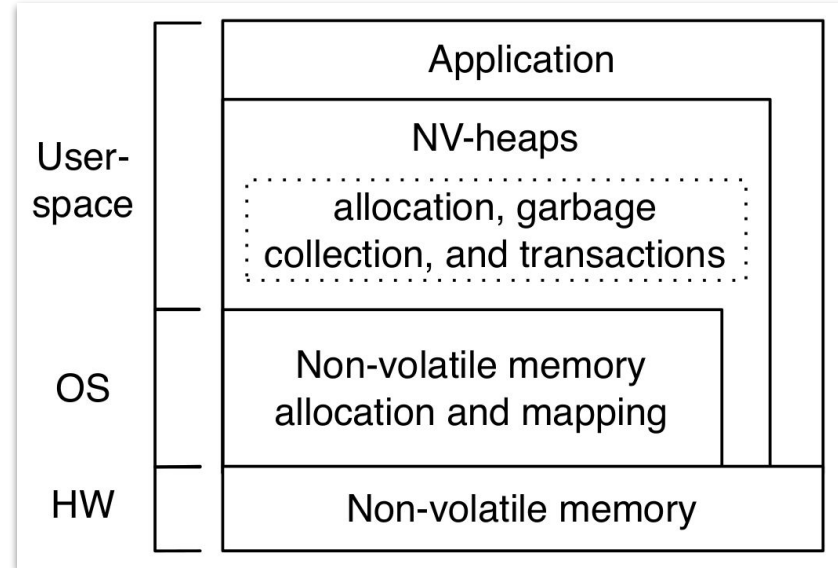
NV-Heaps: Core Ideas

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



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 (for all volatile and non-volatile references) 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: each NV object needs a lock, not scalable
 - 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 (for all volatile and non-volatile references) 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 (simplify): inter NV heap pointers or NV pointers to volatile structures
- **Operator overloading**: a pointer internally contain offset (*use smart pointers and swizzling on loading*) and a heap id to identify which heap they belong to
 - Thus, creation of an inter NVheap pointer can be rejected

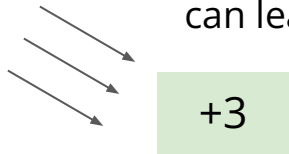
```
class NVList : public NVObject {
    DECLARE_POINTER_TYPES(NVList);
public:
    DECLARE_MEMBER(int, value);
    DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};
```

NVHeap - How to do pointer management

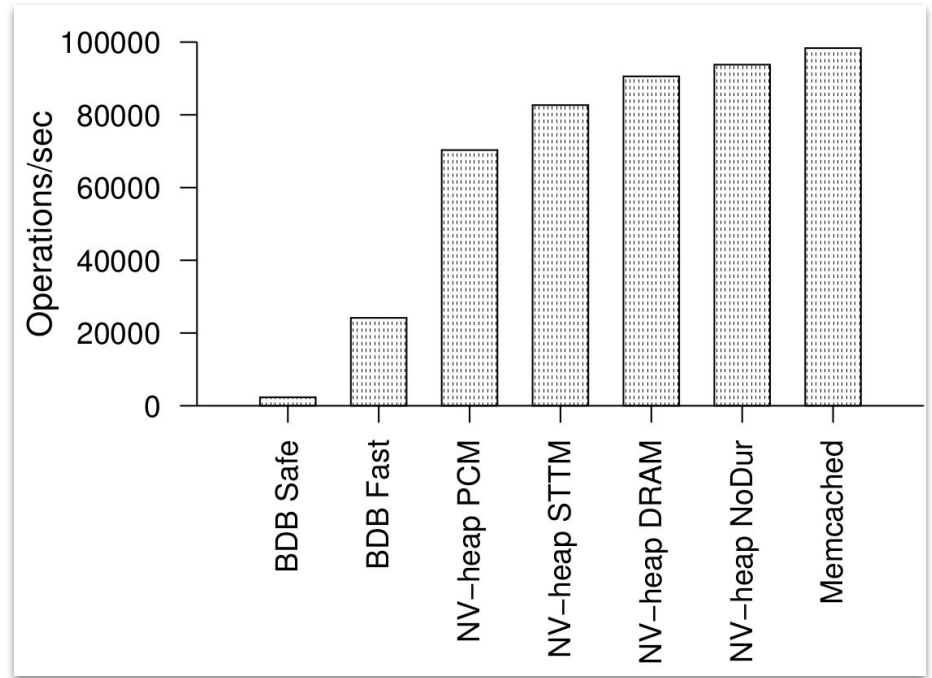
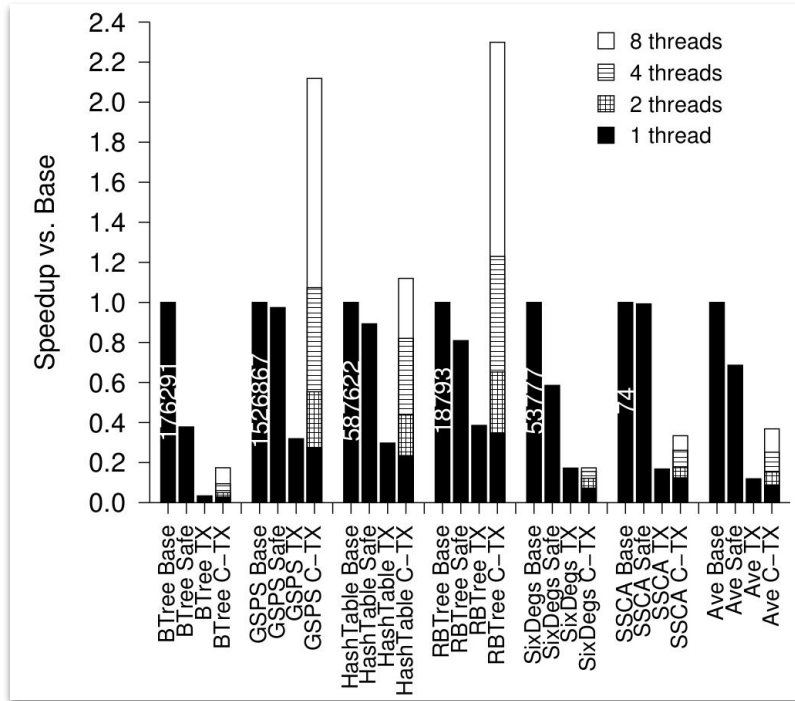
How to do pointer management?

Smart pointers and overloading

- Not supported (simplify): inter NV heap pointers or NV pointers to volatile structures
- **Operator overloading** : a pointer internally contain offset (*use smart pointers and swizzling on loading*) and a 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 (useful for cyclic structures, doubly link lists), can lead to a NULL but not corruption of NVHeap



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!

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,...

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

Persistent Memory Development Kit (PMDK)

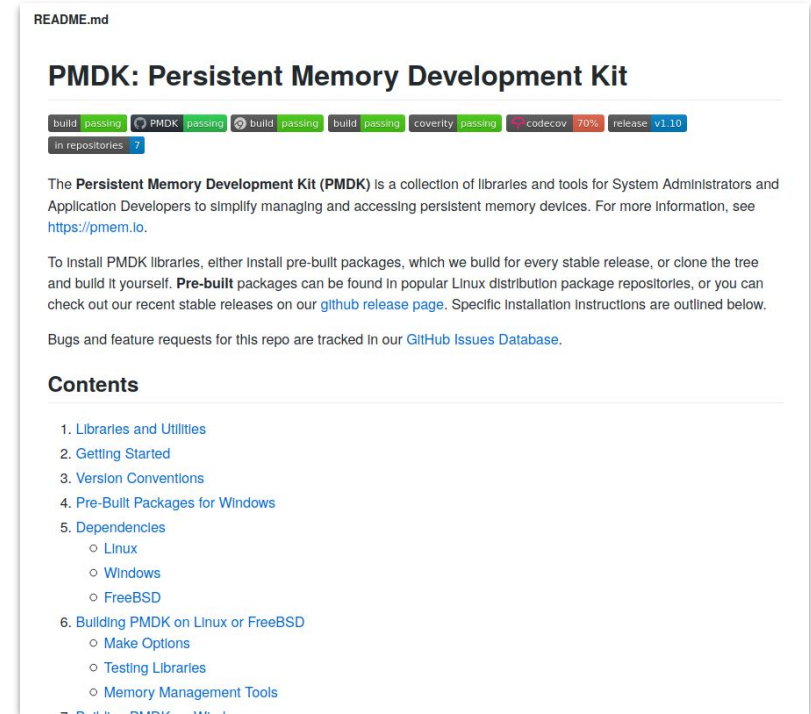
A set of libraries and framework to manage NVDIMMs as

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/>



The screenshot shows the README.md file for the PMDK project. At the top, it says "README.md". Below that is the title "PMDK: Persistent Memory Development Kit". There is a progress bar showing various build and test statuses: "build passing", "PMDK passing", "build passing", "build passing", "coverity passing", "codecov 70%", and "release v1.10". Below the progress bar, it says "in repositories: 7". The main text describes the PMDK as a collection of libraries and tools for System Administrators and Application Developers to simplify managing and accessing persistent memory devices. It provides a link to the project website: <https://pmem.io>. It also mentions that pre-built packages are available for Linux, Windows, and FreeBSD, and provides instructions on how to build the project from source. Finally, it mentions that bugs and feature requests are tracked in the GitHub Issues Database.

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>.

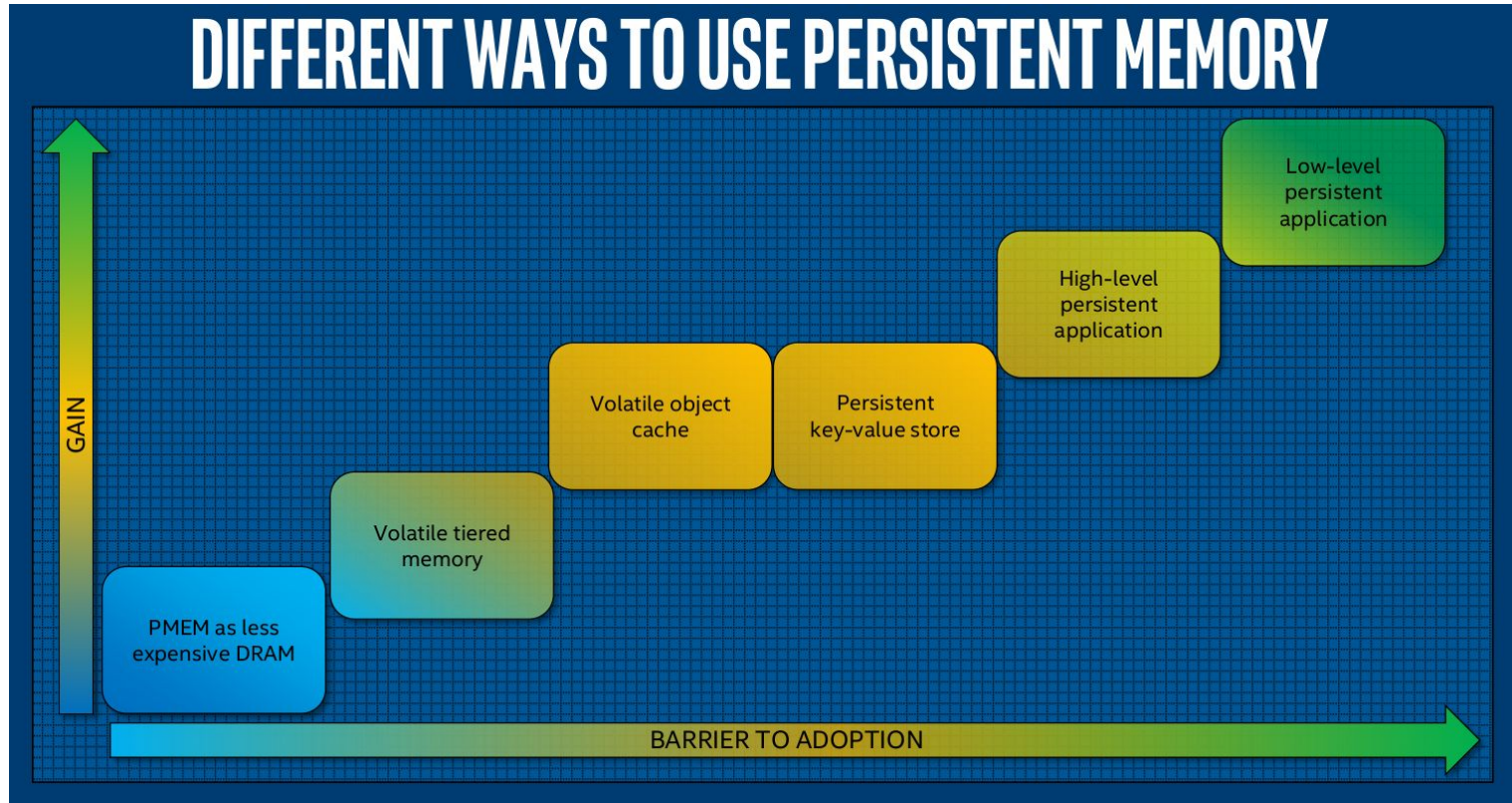
To install PMDK libraries, either install pre-built packages, which we build for every stable release, or clone the tree and build it yourself. **Pre-built** packages can be found in popular Linux distribution package repositories, or you can check out our recent stable releases on our [github release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

Bugs and feature requests for this repo are tracked in our [GitHub Issues Database](#).

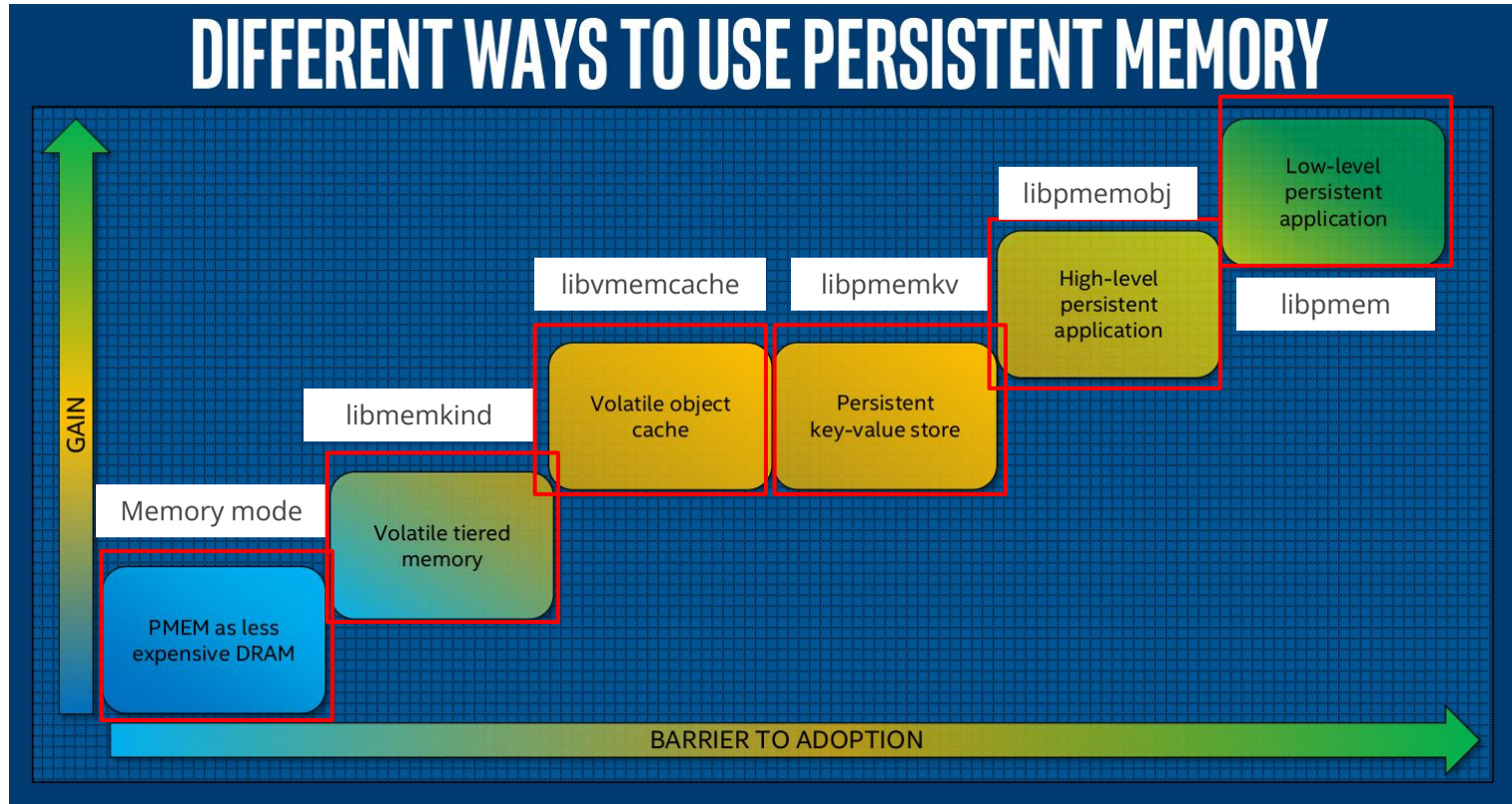
Contents

1. [Libraries and Utilities](#)
2. [Getting Started](#)
3. [Version Conventions](#)
4. [Pre-Built Packages for Windows](#)
5. [Dependencies](#)
 - o [Linux](#)
 - o [Windows](#)
 - o [FreeBSD](#)
6. [Building PMDK on Linux or FreeBSD](#)
 - o [Make Options](#)
 - o [Testing Libraries](#)
 - o [Memory Management Tools](#)

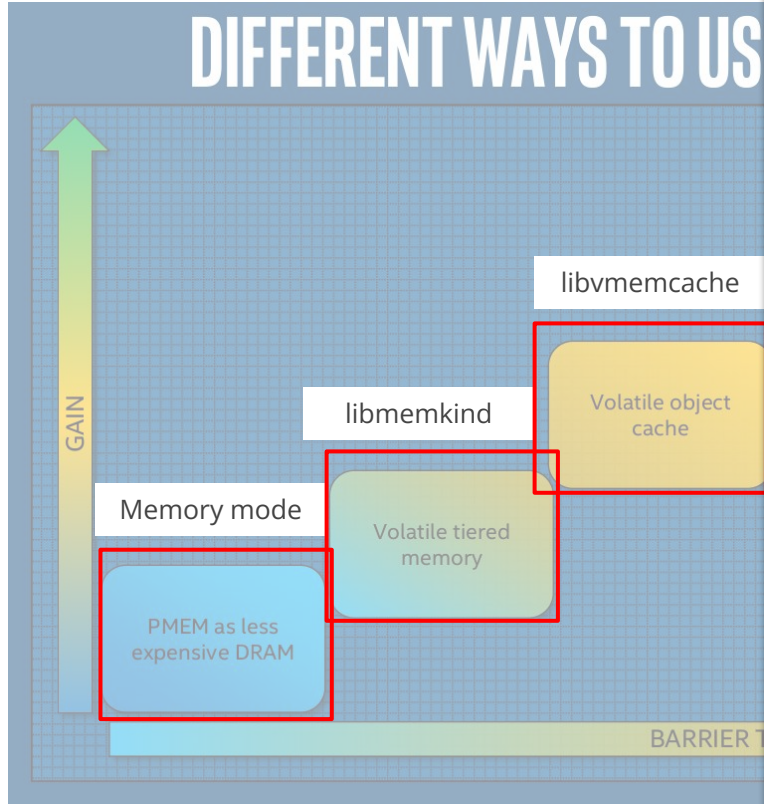
Adoption Spectrum



Adoption Spectrum



Adoption Spectrum



Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory

Virendra J. Marathe Margo Seltzer Steve Byan Tim Harris
{virendra.marathe,margo.seltzer,steve.byan,timothy.l.harris}@oracle.com
Oracle Labs

Abstract

We report our experience building and evaluating `pmemcached`, a version of `memcached` ported to byte-addressable persistent memory. Persistent memory is expected to not only improve overall performance of applications' persistence tier, but also vastly reduce the "warm up" time needed for applications after a restart. We decided to test this hypothesis on `memcached`, a popular key-value store. We took the extreme view of persisting `memcached`'s entire state, resulting in a virtually *instantaneous* warm up phase. Since `memcached` is already optimized for DRAM, we expected our port to be a straightforward engineering effort. However, the effort turned out to be surprisingly complex during which we encountered several non-trivial problems that challenged the boundaries of `memcached`'s architecture. We detail these experiences and corresponding lessons learned.

1 Introduction

Key-value stores with simple `get/put` based interfaces have become an integral part of modern data centers. The list of successful key-value stores is long – Cassandra [22], Dynamo [11], LevelDB [23], `memcached` [14, 24], Redis [29] – to name a few. At the same time, emerging persistent memory technologies [1, 13, 18, 19, 26, 31], such as Intel and Micron's 3D XPoint [1], promise to provide the byte-addressability of DRAM (simple load/store access) and the persistence of traditional storage technologies, at performance 1000X greater than state-of-the-art NAND flash. This can fundamentally change the way applications manage persistent data.

With persistent memory on the horizon, many researchers are developing systems that ensure fast or even instantaneous recovery of application data [3, 5, 7, 27]. The overarching intuition is that by leveraging byte-addressability and the high performance of persistent memory, applications can drastically reduce, or even eliminate, the time needed to recover and "warm up" their state after a restart. While we share this view, we decided to test it in the context of `memcached`, a key-value store primarily used as a DRAM-resident cache. Warming up `memcached`'s state after a restart can take up to several hours for workloads with large data sets [16]. Persisting that state could drastically reduce the warm up time.

During this exercise we wanted to investigate several important questions. Does persisting `memcached`'s state entail any significant performance overheads? In the early years of persistent memory adoption, programmers will be forced to maintain existing application architectures to continue support for platforms without persistent memory. Minimal variation between this legacy code and the new persistent memory optimized code is desirable. How difficult will it be to persist `memcached` without changing its high level architecture? What hurdles will we encounter in this effort? Is there a pattern to these problems? Are there common programming practices that could be used to address them? How generic are these problems? Are there issues that cannot be addressed without rearchitecting `memcached`?

We first summarize the existing structure and operation of `memcached` (§ 2). We frame the description of our experience developing "pmemcached", our persistent memory port of `memcached`, in terms of 10 lessons learned (§ 3). Our findings were interesting, and in some cases, quite surprising. A big takeaway was that this exercise can be surprisingly non-trivial. The required lower level changes were contagious and quickly became pervasive. Failure-atomicity – providing all or nothing semantics across a failure boundary – seems fundamental. We found that we needed failure-atomic transactions more widely than we expected [4, 6, 8, 15, 21, 28, 34]. Other high level surprises and lessons learned include the challenges posed by tricky interactions between *persistent* and *non-persistent* objects, co-location of semantically persistent and nonpersistent data, and unexpected critical section inflation.

We evaluated `pmemcached` on Intel's Software Emulation Platform for persistent memory [12, 36] using the YCSB workload generator [9] (see § 4). We did achieve almost instantaneous warm up. We expected some performance degradation, however it varied significantly across different workloads. Degradation relative to `memcached` was about 10–15% for YCSB's read heavy workloads, but about 40–60% for YCSB's write heavy workloads.

2 memcached Overview

The high level architecture of `memcached` is typical of many key-value stores: It contains a stateful client re-

Example: libpmemobj

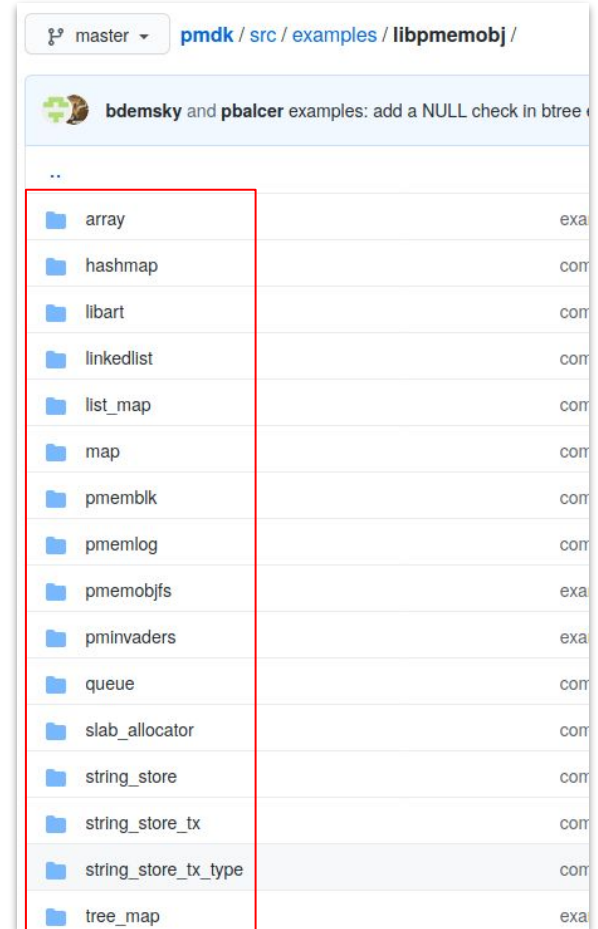
Transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming:

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

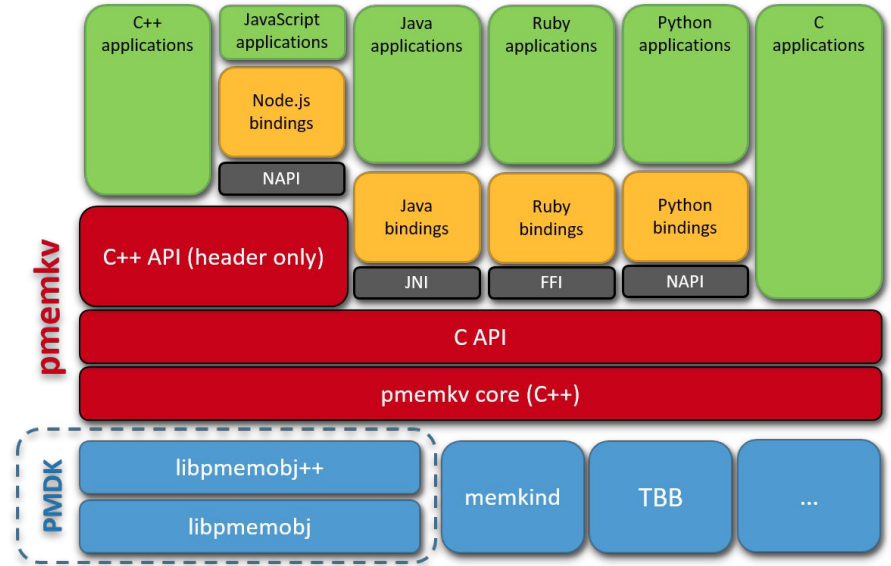
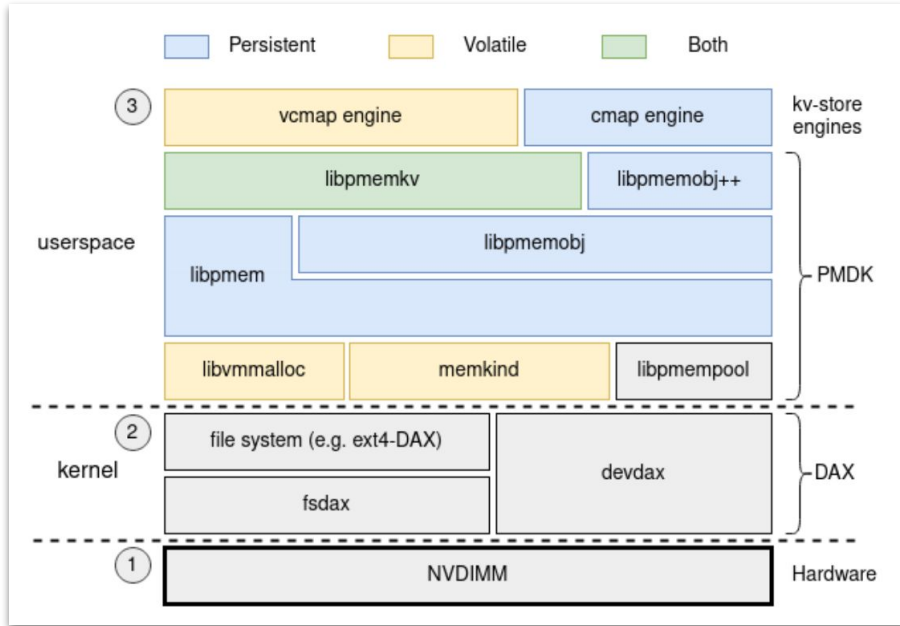
```
typedef struct foo {
    PMEMoid bar; // persistent pointer
    int value;
} foo;

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)



PMDK Stack Overview



- [Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack](#), BSc thesis, Nick-Andian Tehrani
- <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,nvdimms=on \  
-enable-kvm \  
-net nic \  
-net user,hostfwd=tcp::2222-:22 \  
-object memory-backend-file,id=mem1,share,mem-path=./dimm0,size=4G \  
-device nvdimms,memdev=mem1,id=nv1,label-size=2M \  
-object memory-backend-file,id=mem2,share,mem-path=./dimm1,size=4G \  
-device nvdimms,memdev=mem2,id=nv2,label-size=2M \  

```

```
atr@qemu20: $ dmesg | grep user:  
[ 0.000000] user: [mem 0x0000000000000000-0x00000000000009fbff] usable  
[ 0.000000] user: [mem 0x00000000000009fc00-0x00000000000009ffff] reserved  
[ 0.000000] user: [mem 0x000000000000f0000-0x000000000000ffffff] reserved  
[ 0.000000] user: [mem 0x00000000001000000-0x0000000000bffd5fff] usable  
[ 0.000000] user: [mem 0x000000000bffd6000-0x000000000bfffffff] 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

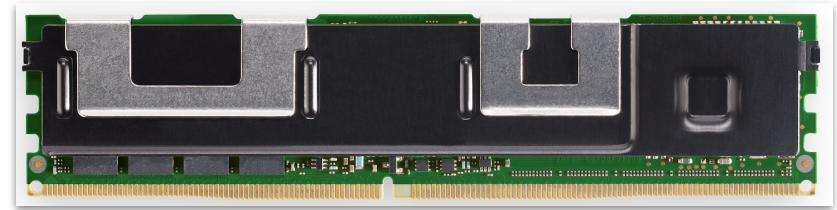
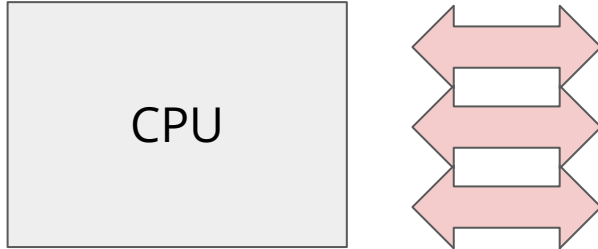
[Home](#) [Glossary](#) [Documents](#) [PMDK](#) [ndctl](#) [Blog](#) [About](#)

How to emulate Persistent Memory

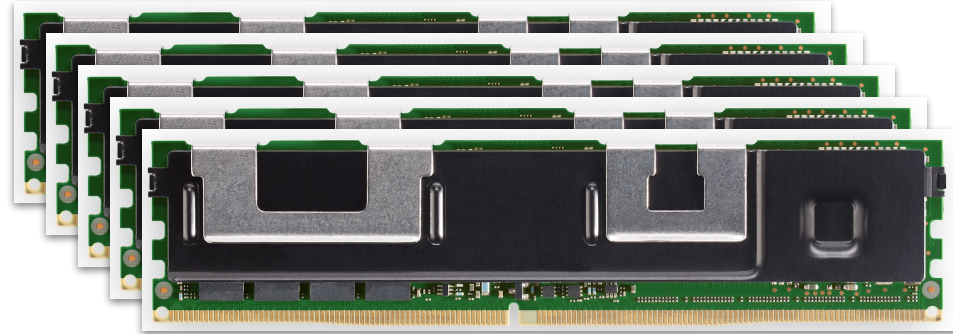
Posted February 22, 2016 [← Previous post](#) [Next post →](#)

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.

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)

Most Importantly - This will not work anymore!

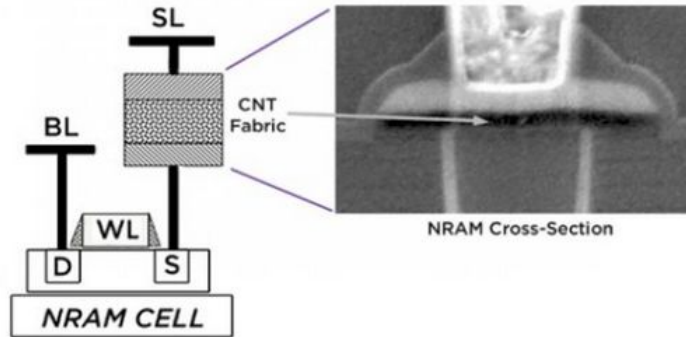


More New Technologies are Coming

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.

Twizzler Operating System (2020)

Twizzler: a Data-Centric OS for Non-Volatile Memory

Daniel Bittman
UC Santa Cruz

Peter Alvaro
UC Santa Cruz

Pankaj Mehra
IEEE Member

Darrell D. E. Long
UC Santa Cruz

Ethan L. Miller
UC Santa Cruz
Pure Storage

Abstract

Byte-addressable, non-volatile memory (NVM) presents an opportunity to rethink the entire system stack. We present Twizzler, an operating system redesign for this near-future. Twizzler removes the kernel from the I/O path, provides programs with memory-style access to persistent data using small (64 bit), object-relative cross-object pointers, and enables simple and efficient long-term sharing of data both between applications and between runs of an application. Twizzler provides a clean-slate programming model for persistent data, realizing the vision of Unix in a world of persistent RAM.

We show that Twizzler is simpler, more extensible, and more secure than existing IO models and implementations by building software for Twizzler and evaluating it on NVM DIMMs. Most persistent pointer operations in Twizzler impose less than 0.5 ns added latency. Twizzler operations are up to 13× faster than Unix, and SQLite queries are up to 4.2× faster than on PMDK. YCSB workloads ran 1.1–2.9× faster on Twizzler than on native and NVM-optimized SQLite backends.

1 Introduction

Byte-addressable non-volatile memory (NVM) on the memory bus with DRAM-like latency [23, 38] will fundamentally shift the way that we program computers. The two-tier memory hierarchy split between high-latency persistent storage and low latency volatile memory may evolve into a single level of large, low latency, and directly-addressable persistent memory. Mere incremental change will leave dramatic improvements in programmability, performance, and simplicity on the table. It is essential that operating systems and system software evolve to make the best use of this new technology.

These opportunities motivate us to revisit how programs operate on persistent data. The separation of volatile memory and high-latency persistent storage at the core of OS design requires the OS to manage ephemeral copies of data and interpose itself on persistence operations, a penalty that will consume an increasing fraction of time as NVM performance increases [64]. The direct-access nature of NVM invites the

use of load and store instructions to directly access persistent data, simplifying applications by enabling persistent data manipulation without the need to transform it between in-memory and on-storage data formats. Thus, the model that best exploits the low latency nature of NVM is one in which persistent data is maintained as in-memory data structures and not serialized or explicitly loaded or unloaded. To avoid serialization, this model must support *persistent pointers* that are valid in *any* execution context, not just the one in which they were created.

Trying to mold NVM into existing models will not enable its fullest potential, just as SSDs did not reach their full potential until they transcended the disk paradigm. To explore a "clean-slate" approach, we are building Twizzler, an OS designed to take full advantage of this new technology by rethinking the abstractions OSes provide in the context of NVM. Twizzler divides NVM into *objects* within a global object space, and pointers are interpreted in the context of the object in which they reside. This decouples pointers from the address space of an individual thread, providing a data-centric programming model rather than a process-centric one. The result is a vastly simpler environment in which the OS's primary function is to support manipulating, sharing, and protecting persistent data using few kernel interpositions.

We implemented a simple, standalone kernel that supports a userspace for NVM-based applications, with compatibility layers for legacy programs. We wrote a set of libraries and portability layers that provide a rich environment for applications to access persistent data that takes into account both semantics (persistent pointers) and safety (building crash-consistent data structures). We then performed a case-study by writing software for Twizzler, taking into account the new flexibility and power gained by our model and evaluating our software for complexity and performance. We ported SQLite to Twizzler, showing how our approach can provide significant performance gains on existing applications as well.

In a world where in-memory data can last forever, the context required to manipulate that data is best coupled with the data rather than the process. This key insight manifests itself in the three primary contributions of this paper:



Twizzler

About Pubs Docs Download
People

Twizzler is a research operating system, written in Rust, designed around the data-centric programming model. It is built from scratch with a new kernel focused on simplicity to facilitate direct access to shared, persistent data for applications with minimal OS involvement and interposition. Twizzler is motivated by the convergence of the memory hierarchy for traditionally "far away" memory, brought about by the increasing closeness of persistent and distributed memory. Twizzler is developed by the CRSS at UC Santa Cruz.

For more information, see our [Publications](#) or our [Documentation](#).

<https://twizzler.io/>

Key Problems

1. NVM are fast

- Should you involve the kernel in data access path?
- How do you involve kernel? sys_calls

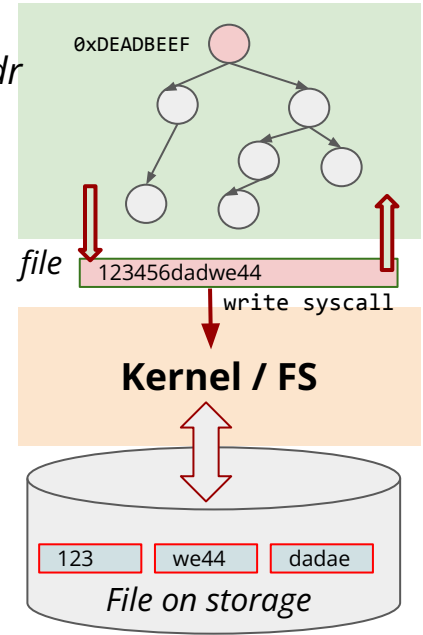
2. Two-level storage hierarchy

- Explicit serialization-deserialization
- Constant data format changes (in-memory or on storage)
- Takes time

3. Addressing data in memory? (which memory?)

- Pointers
- Pointers are only valid with a process**
- Process are ephemeral, hence pointers are ephemeral
- How to identify data in NVM outside a “process”?

*Ephemeral
process vaddr
space*



If these sounds like philosophical questions - then you should realize the magnitude of such changes in the systems research!

Twizzler OS

Thin, **Exo-kernel style OS kernel**

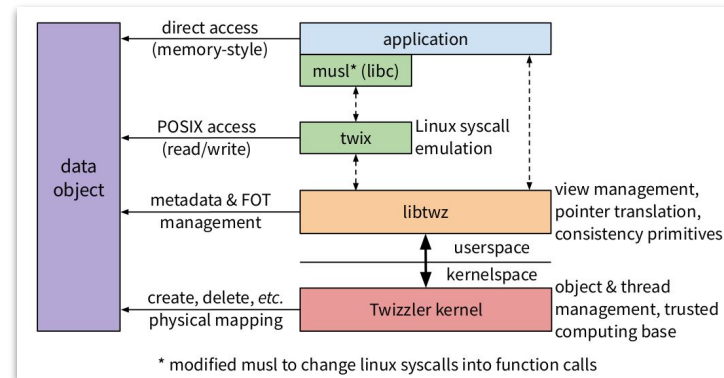
- Does scheduling, synchronization, and management

LibraryOS, libtwz

- Does mapping management and persistent pointers

Backward-compatible twix library

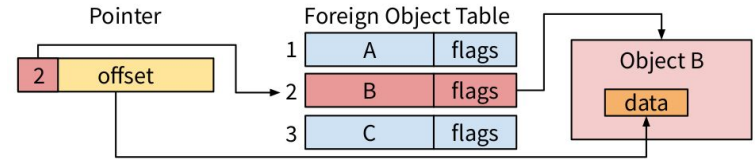
- Converts classic syscalls into function calls (libc)
- read and writes, become memcpy



Key Ideas

Leverages persistent object ideas of NV-Heap

- Have 128-bits persistent objects
 - create, delete object syscall
 - Objects can be 4KB to 1GB
 - Can contain either the full B+ tree or just node
- Kernel and user application share “views” (eqv. of req/resp on the vspace management)
 - When a fault happens, the handler maps objects in the requested view slots
 - Leverages existing virtual memory mechanism
- Allow cross-object “thin” pointers using a Foreign Object Table (FOT)
 - Pointers are still 64 bits (object id + offset)
 - Allows late binding of objects and **cross object references** (this wasn't allowed in NVHeap)



More details in the paper ...an interesting read

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., ctFS)
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

More War (Persistent Data Structures)

Pronto: Easy and Fast Persistence for Volatile Data Structures

Amirsaman Memaripour*
University of California, San Diego
amemari@eng.ucsd.edu

Joseph Izaelevitz
University of Colorado Boulder
joseph.izaelevitz@colorado.edu

Steven Swanson
University of California, San Diego
swanson@cs.ucsd.edu

Abstract

Non-Volatile Main Memories (NVMMs) promise an opportunity for fast, persistent data structures. However, building those data structures is hard because their data must be consistent in the wake of a failure. Existing methods for building persistent data structures require either in-depth code changes to an existing data structure using an NVMM-aware library or rewriting the data structure from scratch. Unfortunately, both of those methods are labor-intensive and error-prone.

Pronto is a new NVMM library that reduces the programming effort required to add persistence to volatile data structures using *asynchronous semantic logging (ASL)*. ASL is generic enough to allow programmers to add persistence to the existing volatile data structure (e.g., C++ Standard Template Library extensions) with very little programming effort. Furthermore, ASL moves most durability code off the critical path, and our evaluation shows Pronto data structures outperform highly-optimized NVMM data structures written with other libraries by a large margin.

CCS Concepts • **Hardware** → Emerging technologies; **Software and its engineering** → Software libraries and repositories; **Information systems** → Data structures; **Computer systems organization** → Processors and memory architectures

Keywords Non-volatile Memory, Persistent Memory, Persistent Objects, Data Structures, Storage Systems, Snapshots, Asynchronous Logging, Semantic Logging

*The author is now at MongoDB, Inc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASPLoS '20, March 16–20, 2020, Lausanne, Switzerland

© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7023-0/20/03.
https://doi.org/10.1145/337376.3378156

ACM Reference Format:

Amirsaman Memaripour, Joseph Izaelevitz, 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)*, March 16–20, 2020, Lausanne, Switzerland. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/337376.3378156>

1 Introduction

Emerging non-volatile main memory (NVMM) technologies such as 3D XPoint [14, 20] offer higher density than DRAM with comparable latency and bandwidth, allowing computer architects to attach them to processors via the memory bus. Programs can then use load and store instructions to access persistent data directly. Bypassing the storage stack and directly accessing NVMM is essential for unleashing the performance benefits that NVMM offer [18]. However, this strategy requires careful reasoning to ensure a consistent-state in NVMM in the wake of a crash — data in the caches will not survive [28, 36].

NVMMs appear to be an exceptional opportunity for building fast, persistent, data structures, and researchers have approached this problem in two ways. NVMM failure-atomicity libraries (e.g., [11, 5]) allow programmers to identify *failures-atomic* updates to persistent data — writes within the update become persistent all at once. By identifying failure-atomic code regions and persistent writes, programmers can adapt an existing data structure to NVMM using these libraries [6, 12]. Alternatively, researchers have built custom data structures from scratch for NVMM (e.g., [43, 50]). Unfortunately, both of these design options are labor-intensive, require detailed program knowledge, and are a fertile source of subtle errors [54]. Furthermore, these options effectively ignore the wide range of useful, volatile data structures currently available (e.g., the C++ Standard Template Library or the Java Collection data structures).

In this work, we propose *Pronto*, a library that reduces the programming effort required to add persistence to off-the-shelf, volatile data structures, preserving the original operation of the data structure and, for concurrent data structures, their concurrency scheme. Furthermore,

MOD: Minimally Ordered Durable Datastructures for Persistent Memory

Swarnil Haria*
University of Wisconsin-Madison
swah@cs.wisc.edu

Mark D. Hill
University of Wisconsin-Madison
markhill@cs.wisc.edu

Michael M. Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

Abstract

Persistent Memory (PM) makes possible recoverable applications that can preserve application progress across system reboots and power failures. Actual recoverability requires careful ordering of cache-line flushes, currently done in two extreme ways. On one hand, expert programmers have reasoned deeply about consistency and durability to create applications centered on a single custom-crafted durable data structure. On the other hand, less-expert programmers have used software transaction memory (STM) to make atomic one or more updates, albeit at a significant performance cost due largely to ordered log updates.

In this work, we propose the middle ground of composable persistent datastructures called Minimally Ordered Durable Datastructures (MOD). We prototype MOD as a library of C++ datastructures — currently, map, set, stack, queue and vector — that often perform better than STM and yet are relatively easy to use. They allow multiple updates to one or more datastructures to be atomic with respect to failure. Moreover, we provide a recipe to create additional recoverable datastructures.

MOD is motivated by our analysis of real Intel Optane PM hardware showing that allowing unordered, overlapping flushes significantly improves performance. MOD reduces ordering by adapting existing techniques for out-of-place updates (like shadow paging) with space-reducing structural sharing (from functional programming). MOD exposes a *Basic Interface* for single updates and a *Composition Interface* for atomically performing multiple updates. Relative to widely used Intel PMDK v1.5.5, STM MOD improves map, set, stack, queue microbenchmark performance by 40%, and speed up application benchmark performance by 38%.

*Now at Google.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Requested permissions from permissions@acm.org.
ASPLoS '20, March 16–20, 2020, Lausanne, Switzerland.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7023-0/20/03
https://doi.org/10.1145/337376.3378156

CCS Concepts • **Software and its engineering** → Software libraries and repositories; **Software fault tolerance** → Storage class memory; **Information systems** → Storage class memory.

Keywords crash-consistency, durability, datastructures, persistent memory.

ACM Reference Format:

Swarnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS '20)*, March 16–20, 2020, Lausanne, Switzerland. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/337376.3378172>

1 Introduction

Persistent Memory (PM) is here—Intel Optane DC Persistent Memory Modules (DCPMM) began shipping in 2019 [21]. Such systems expose fast, byte-addressable, non-volatile memory (NVM) devices as main memory and allow applications to access this persistent memory via regular load/store instructions. In fact, we ran all experiments in this paper on a system with engineering samples of Optane DCPMM [22, 23].

The durability of PM enables *recoverable applications* that preserve in-memory data beyond process lifetimes and system crashes, a desirable quality for workloads like databases, key-value stores and long-running scientific computations [4, 28]. Such applications use cache-line flush instructions to move data from volatile caches to durable PM and order these flushes carefully to ensure consistency. For instance, applications must durably update data before updating a persistent pointer to the data, or atomically do both.

However, few recoverable PM applications have been developed so far, though libraries like Mememory [46] and Intel Persistent Memory Development Kit (PMDK) [19] have existed for several years. Currently, there are two approaches to building such applications: single-purpose custom data structures (e.g., persistent B-trees [6, 46, 48]) or general-purpose transactions. Both approaches have some benefits, but we believe that neither is suitable for developers building new PM applications.

Although custom data structures are typically fast, significant effort is needed in designing these structures to ensure that updates are performed atomically with respect to failure, i.e., either all modified data is made durable in PM or none.

Corundum: Statically-Enforced Persistent Memory Safety

Morteza Hoseinzadeh
University of California, San Diego
San Diego, California, USA
mhosein@ucsd.edu



Steven Swanson
University of California, San Diego
San Diego, California, USA
swanson@cs.ucsd.edu

ABSTRACT

Fast, byte-addressable, persistent main memories (PM) make it possible to build complex data structures that can survive system failures. Programming for PM is challenging, not just because it combines well-known programming challenges like locking, memory management, and pointer safety with novel PM-specific bug types. It also requires logging updates to PM to facilitate recovery after a crash. A mistake in any of these areas can corrupt data, leak resources, or prevent successful recovery after a crash. Existing PM libraries in a variety of languages — C, C++, Java, Go — simplify some of these problems, but they still require the programmer to learn (and flawlessly apply) complex rules to ensure correctness. Opportunities for data-destroying bugs abound.

This paper presents Corundum, a Rust-based library with an idiomatic PM programming interface and leverages Rust's type system to statically avoid most common PM programming bugs. Corundum lets programmers develop persistent data structures using familiar Rust constructs and have confidence that they will be free of those bugs. We have implemented Corundum and found its performance to be as good as or better than Intel's widely-used PMDK library, HP's Adas, Mememory, and go-pmem.

CCS CONCEPTS

Information systems → Storage class memory; **Software and its engineering** → Formal software verification; **Software testing and debugging** • **Hardware** → Non-volatile memory.

KEYWORDS

non-volatile memory programming library, static bug detection, crash-consistent programming

ACM Reference Format:

Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS '21)*, April 23–27, 2021, Virtual, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3445013.3446770>

1 INTRODUCTION

Persistent main memory (PM) is the first new memory technology to arrive in the memory hierarchy since the appearance of



This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
ASPLoS '21, April 23–27, 2021, Virtual, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9152-0/21/04
https://doi.org/10.1145/3445013.3446770

DRAM in the early 1970's. PM offers numerous potential benefits including improved memory system capacity, lower latency and higher bandwidth relative to disk-based storage, and a unified programming model for persistent and volatile programs state. However, it also poses a host of novel challenges. For instance, it requires memory controller and ISA support, new operating system facilities, and it places large, new burdens on programmers.

The programming challenges it poses are daunting and stem directly from its non-volatility, high-performance, and direct connection to the processor's memory bus. PM's raw performance demands the removal of system software from the common-use access path, its non-volatility requires that (if it is to be used as storage) updates must be robust in the face of system failures, and its memory-like interface forces application software to deal directly with issues like fault tolerance and error recovery rather than relying on layers of software.

In addition, programming with PM exacerbates the impact of existing types of bugs and introduces novel classes of programming errors. Common errors like memory leaks, dangling pointers, concurrency bugs, and data structure corruption have permanent effects (rather than disappearing on restart). New errors are also possible: A programmer might forget to log an update to a persistent structure or create a pointer from a persistent data structure to volatile memory. The former errors manifest during recovery while the latter is inherently unsafe since, after the pointer to volatile memory is meaningless and dereferencing it will result in (at best) an exception.

The challenges of programming correctly with PM are among the largest potential obstacles to wide-spread adoption of PM and our ability to fully exploit its capabilities. If programmers cannot reliably write and modify code that correctly and safely manages persistent data structures, PM will be hobbled as a storage technology.

Some of the bugs that PM programs suffer from have been the subject of years of research and practical tool building. The solutions and approaches to these problems range from programming disciplines to improved library support to debugging tools to programming language facilities.

Given the enhanced importance of memory and concurrency errors in PM programming, it makes sense to adopt the most effective and reliable mechanisms available for avoiding them.

The Rust programming language provides programming language-based mechanisms to avoid a host of common memory and concurrency errors. Its type system, standard library, and "borrow checker" allow the Rust compiler to statically prevent data races, synchronization errors, and most memory allocation errors. Further, the performance of the resulting machine code is comparable with that of compiled C or C++. In addition to these built-in static checks, Rust also provides facilities that make it easy (and idiomatic) to

More work (P

Pronto: Easy and Fast Persistence for Volatile Data Structures

Amirsaman Memaripour* University of California, San Diego amemari@eng.ucsd.edu
Joseph Izaevlevit University of Colorado Boulder joseph.izaevlevit@colorado.edu
Steven Swanson* University of California, San Diego swanson@cs.ucsd.edu

Abstract

Non-Volatile Main Memories (NVMMs) promise an opportunity for fast, persistent data structures. However, building these data structures is hard because their data must be consistent in the wake of a failure. Existing methods for building persistent data structures require either in-depth code changes to an existing data structure using an NVMM-aware library or rewriting the data structure from scratch. Unfortunately, both of these methods are labor-intensive and error-prone.

Pronto is a new NVMM library that reduces the programming effort required to add persistence to volatile data structures using *asynchronous semantic logging (ASL)*. ASL is generic enough to allow programmers to add persistence to the existing volatile data structure (e.g., C++ Standard Template Library containers) with very little programming effort. Furthermore, ASL moves most durability code off the critical path, and our evaluation shows Pronto data structures outperform highly-optimized NVMM data structures written with other libraries by a large margin.

CCS Concepts. • Hardware → Emerging technologies; • Software and its engineering → Software libraries and repositories; • Information systems → Data structures; • Computer systems organization → Processors and memory architectures.

Keywords. Non-volatile Memory, Persistent Memory, Persistent Objects, Data Structures, Storage Systems, Snapshots, Asynchronous Logging, Semantic Logging

*The author is now at MongoDB, Inc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). ASPLOS '20, March 16–20, 2020, Losanone, Switzerland. © 2020 Copyright held by the owner/author(s). ACM ISBN 978-1-4503-7010-2/20/03. https://doi.org/10.1145/3373776.3373786

ACM Reference Format:

Amirsaman Memaripour, Joseph Izaevlevit, 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)*, 3: 16–20, 2020, Losanone, Switzerland. ACM, New York, USA, 18 pages. https://doi.org/10.1145/3373776.3373786

1 Introduction

Emerging non-volatile main memory (NVMM) technologies such as 3D XPoint [14, 23] offer higher density DRAM with comparable latency and bandwidth, letting computer architects to attach them to processing the memory bus. Programs can then use load and instructions to access persistent data directly. By using the storage stack and directly accessing NVMM, essential for unleashing the performance benefits NVMM offer [48]. However, this strategy requires full reasoning to ensure a consistent-state in NV in the wake of a crash — data in the caches will survive [28, 36].

NVMMs appear to be an exceptional opportunity for building fast, persistent, data structures, and research have approached this problem in two ways. NV failure-atomicity libraries (e.g., [11, 51]) allow programmers to identify failure-atomic code regions and ensure that by delineating failure-atomic code regions an persistent writes, programmers can adapt an existing structure to NVMM using these libraries [6, 12, 22]. Alternatively, researchers have built custom data structures from scratch for NVMM (e.g., [43, 50]). Unfortunately, both of these design options are labor-intensive, require detailed program knowledge, and are a fertile source of subtle errors [54]. Furthermore, these options often ignore the wide range of useful, volatile data structures currently available (e.g., the C++ Standard Template Library or the Java Collection data structures).

In this work, we propose *Pronto*, a library that reduces the programming effort required to add persistence to off-the-shelf, volatile data structures, preserving the original operation of the data structure and, for certain data structures, their concurrency scheme. Furthermore,

AGAMOTTO: How Persistent is your Persistent Memory Application?

Ian Neal University of Michigan
Ben Reeves University of Michigan
Ben Stoler University of Michigan
Andrew Quinn University of Michigan
Youngjin Kwon KAIST
Simon Peter University of Texas at Austin
Baris Kasikci University of Michigan

Abstract

Persistent Memory (PM) can be used by applications to directly and quickly persist any data structure, without the overhead of a file system. However, writing PM applications that are simultaneously correct and efficient is challenging. As a result, PM applications contain correctness and performance bugs. Prior work on testing PM systems has low bug coverage as it relies primarily on extensive test cases and developer annotations.

In this paper we aim to build a system for more thoroughly testing PM applications. We inform our design using a detailed study of 63 bugs from popular PM projects. We identify two application-independent patterns of PM misuse which account for the majority of bugs in our study and can be detected automatically. The remaining application-specific bugs can be detected using compact custom oracles provided by developers.

We then present AGAMOTTO, a generic and extensible system for discovering misuse of persistent memory in PM applications. Unlike existing tools that rely on extensive test cases or annotations, AGAMOTTO symbolically executes PM systems to discover bugs. AGAMOTTO introduces a new symbolic memory model that is able to represent whether or not PM state has been made persistent. AGAMOTTO uses a state space exploration algorithm, which drives symbolic execution towards program locations that are susceptible to persistence bugs. AGAMOTTO has so far identified 84 new bugs in 5 different PM applications and frameworks while incurring no false positives.

1 Introduction

Persistent Memory (PM) is a promising new technology that offers an appealing performance-cost tradeoff for application developers. PM technologies, such as Intel Optane DC [36], can offer persistent memory accesses with latencies that are only 2–3× higher than the latencies of DRAM [70]. Moreover, such PM technologies are cheaper than DRAM per GB

of capacity [3]. As byte-addressable memory, PM can also be accessed via processor load and store instructions. Application developers have already started building systems that use PM directly, without relying on heavyweight system calls to ensure durability, including ports of popular systems such as memcached [24] and Redis [21].

While using PM directly via persistent data structures can offer performance, it is challenging to write PM-based applications that are simultaneously correct and efficient [12, 18, 33, 52, 54, 60, 71, 76]. Persistent memory writes in the CPU cache must be explicitly flushed to PM using specific instructions or APIs. In certain cases, PM flush operations need to be ordered using memory fences to enforce crash consistency. Incorrect usage of these mechanisms can result in *persistence bugs* which break crash-consistency guarantees or degrade application performance. Persistence bugs are challenging to diagnose because their symptoms are easily masked. For example, crash-consistency bugs may be masked because PM writes are implicitly flushed when dirty (or updated) cache lines are evicted from the CPU—furthermore, flushes which are required for proper crash consistency under one execution path may be redundant and unnecessary under a different program execution path, leading to performance degradations.

Several systems have been built to aid with testing PM applications; however, these existing approaches are either specific to a target application or require significant manual developer effort. Intel designed Yat [44] and pmemcheck [65] specifically to test the crash consistency and durability of PMFS (Persistent Memory File System) [27] and PMDK (Persistent Memory Development Kit) [20], respectively. To find bugs, Yat exhaustively tests all possible update orderings, and pmemcheck tracks annotated updates. Both of these tools are specific to a single system (PMFS and PMDK, respectively) and are hard to generalize. Other tools like Persistence Inspector [62], PMTest [50], and XiDetector [49] are applicable to general PM systems, but require developer annotations and extensive test suites to thoroughly test PM applications.

In order to determine the extent to which persistence bug finding can be automated (i.e., not require program annota-



ures)...

Statically-Enforced Persistent Memory Safety

Ahoseinzadeh California, San Diego aahoein@cs.ucsd.edu
Steven Swanson University of California, San Diego swanson@cs.ucsd.edu



DRAM in the early 1970's, PM offers numerous potential benefits including improved memory system capacity, lower latency and higher bandwidth relative to disk-based storage, and a unified programming model for persistent and volatile programs state. However, it also poses a host of novel challenges. For instance, it requires memory controller and ISA support, new operating system facilities, and it places large, new burdens on programmers.

The programming challenges it poses are daunting and stem directly from its non-volatility, high-performance, and direct connection to the processor's memory bus. PM's raw performance demands the removal of system software from the common-use access path, its non-volatility requires that (if it is to be used as storage) updates must be robust in the face of system failures, and its memory-like interface forces application software to deal directly with issues like fault tolerance and error recovery rather than relying on layers of system software.

In addition, programming with PM exacerbates the impact of existing types of bugs and introduces novel classes of programming errors. Common errors like memory leaks, dangling pointers, concurrency bugs, and data structure corruption have permanent effects (rather than disappearing on restart). New errors are also possible: A programmer might forget to log an update to a persistent structure or create a pointer from a persistent data structure to volatile memory. The former errors may manifest during recovery while the latter is inherently undetectable since after restart, the pointer to volatile memory is meaningless and dereferencing it will result in a fault or an exception.

The challenges of programming correctly with PM are among the largest potential obstacles to wide-spread adoption of PM and our ability to fully exploit its capabilities. If programmers cannot reliably write and modify code that correctly and safely maintains persistent data structures, PM will be hobbled as a storage technology.

Some of the bugs that PM programs suffer from have been the subject of years of research and practical tool building. The solutions and approaches to these problems range from programming disciplines to improved library support to debugging tools to programming language facilities.

Given the enhanced importance of memory and concurrency errors in PM programming, it makes sense to adopt the most effective and reliable mechanisms available for avoiding them. The Rust programming language provides programming language-based mechanisms to avoid a host of common memory and concurrency errors. Its type system, standard library, and "borrow checker" allow the Rust compiler to statically prevent data races, synchronization errors, and most memory allocation errors. Further, the performance of the resulting machine code is comparable with that of compiled C or C++. In addition to these built-in static checks, Rust also provides facilities that make it easy (and idiomatic) to

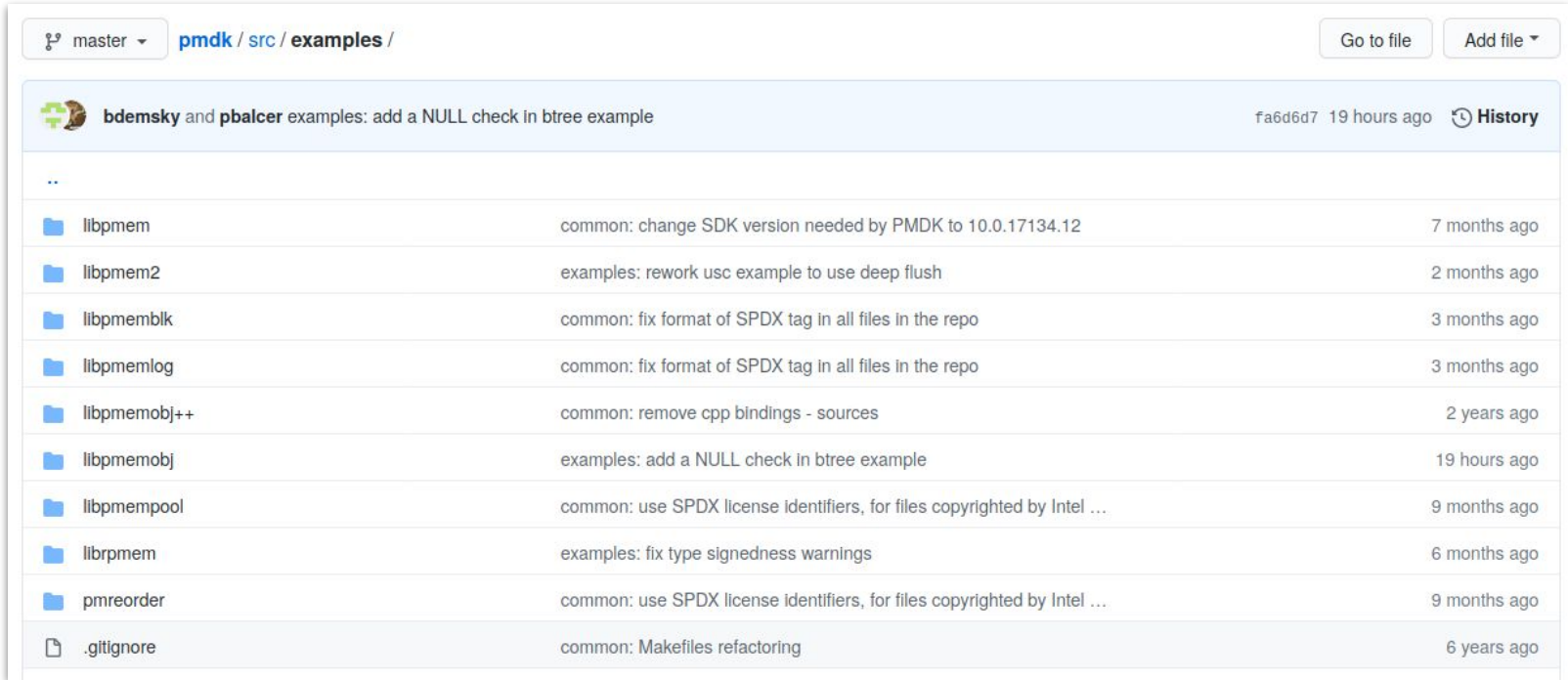
Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 2021 ACM International Conference on Programming Language and Systems (PLoS '21)*, Virtual, USA. ACM, New York, 10:11–25. https://doi.org/10.1145/3443501.3446707

USENIX Association
14th USENIX Symposium on Operating Systems Design and Implementation 1047

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



The screenshot shows a GitHub repository page for the path `pmdk / src / examples /`. The current branch is `master`. The page displays a commit history for the `examples` directory, showing a list of folders and files with their commit messages and dates.

Commit: `fa6d6d7` 19 hours ago [History](#)

File/Folder	Commit Message	Time Ago
..		
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
libpmem	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

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].

(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
 - a. *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

File system	Append Time (ns)	Overhead (ns)	Overhead (%)
ext4 DAX	9002	8331	1241%
PMFS	4150	3479	518%
NOVA-Strict	3021	2350	350%

SplitFS, SOSP 2019

Developed NOVA file system for hybrid DRAM-NVDIMM memories

NOVA Design and Ideas

A Log-structured file system

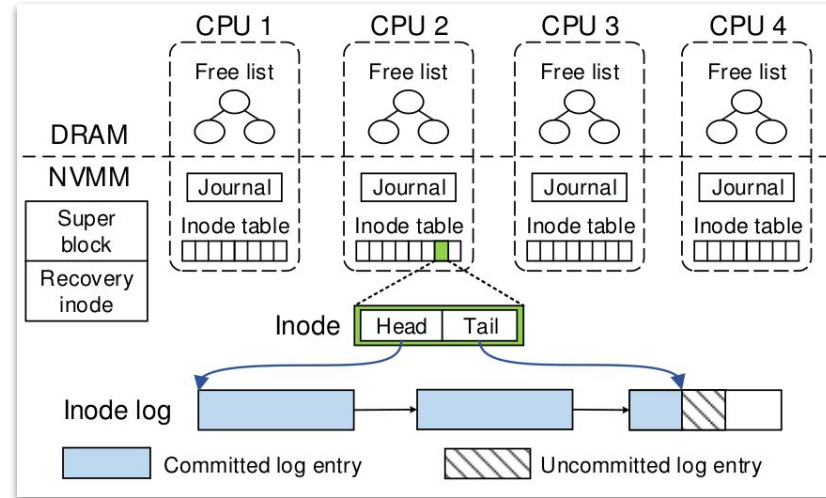
Each inode has its own log (concurrency and parallelism)

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

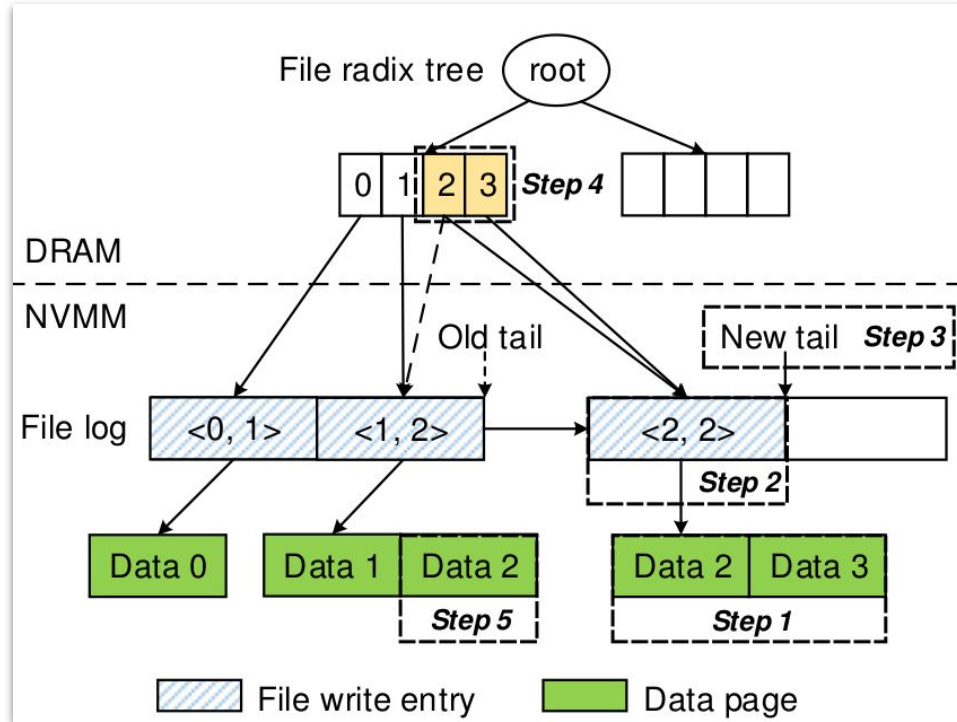
Performance: logs in NV memory, and index in DRAM (can be reconstructed)

Smaller log 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)

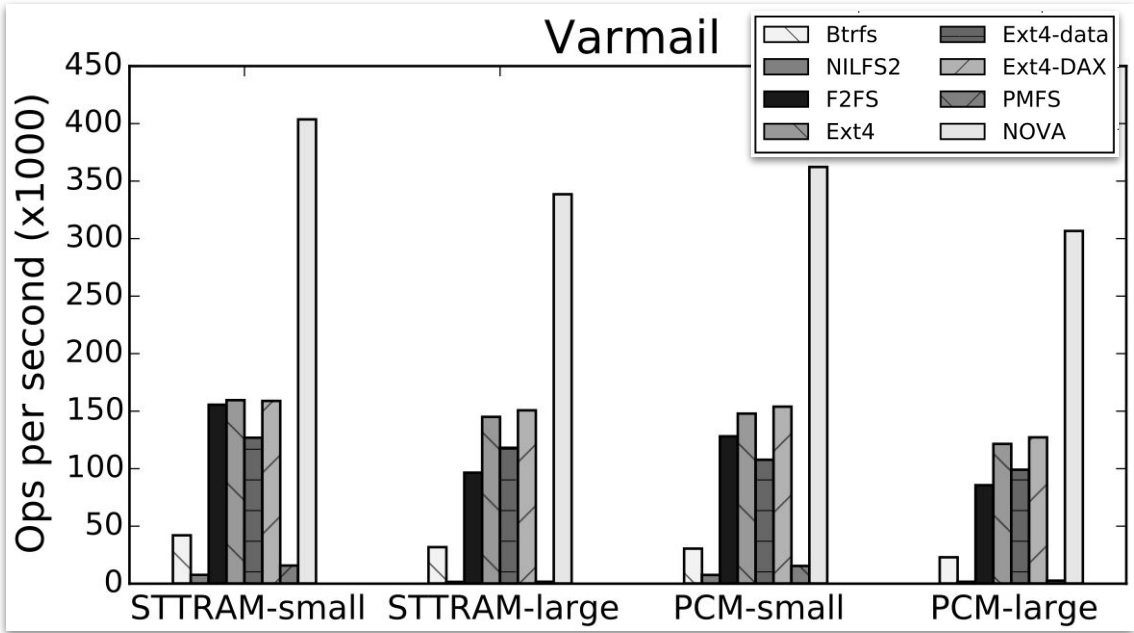
Step 2: Add to the file inode log

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