CHAPTER 2

PERSISTENT MEMORY ARCHITECTURE

In this chapter, we provide an overview of the persistent memory architecture while focusing on the essential of the hardware to emphasize requirements and decisions that developers need to know.

Applications that are designed to recognize the presence of persistent memory in a system can run much faster than using storage devices because data doesn't have to transfer back and forth between the CPU and slower storage devices. Applications that only utilize persistent memory may be slower than DRAM. Therefore, applications should decide what data resides in DRAM, persistent memory, and storage.

In the introduction we described the memory-storage hierarchy with the introduction of persistent memory to fil the gap. This use of persistent memory opens new possibilities for application architectures and features. Applications can persistently store in-memory data structures natively without having to serialize and de-serialize the data into some other format, such as storing data in files or a database. Storing data structures in their native state can reduce code complexity and significantly expedite application startup times since the application need only open and memory map the file to access the data rather than having to read data in from external sources, and convert it to in-memory data. Since the data is persistent in memory, applications no longer need to wait for caches to warm up, the data is immediately available.

With the capacity of persistent memory expected to be many times larger than DRAM, the volume of data that applications can potentially store and process in place is also significantly larger, thus reducing the number of disk I/Os. Large datasets that cannot fit into DRAM have to be processed in segments or streamed. Applications can benefit from the larger memory capacities offered by persistent

memory to perform in-memory processing without needing to checkpoint or page data to or from storage, or at least reduce disk I/Os.

Persistent Memory Characteristics

As with every new technology, there are always new things to consider. Persistent memory is no exception. Below are some of these characteristics to consider when architecting and developing solutions.

- Performance (throughput, latency, and bandwidth) is much better than NAND but typically slower than DRAM
- Durable like memory. Persistent memory typically has orders of magnitude better endurance than NAND and should exceed the lifetime of the server without wearing out.
- Persistent memory module capacities are typically much larger than DRAM DIMMs and can co-exist on the same memory channels.
- Applications can update data in-place
- Byte addressable like memory. Applications can update just the data needed without any read-modify-write overhead
- Data is CPU Cache Coherent
- Provides DMA and RDMA operations
- Data is not lost when power is removed. Data may be retained for years without risk of data degradation or corruption
- Data is directly accessible from user space. No Kernel code, file system page caches, or interrupts are in the data path.
- Instant on
    - Data is available as soon as power is applied to the system
    - Applications don’t need to spend time warming up caches
    - Data has no DRAM footprint unless the application copies data to DRAM for faster access
- Data is local to the system - applications are responsible for replicating data across systems

Platform Support for Persistent Memory

Platform vendors such as Intel, AMD, and ARM will decide how persistent memory should be implemented at the lowest hardware levels. Throughout this book we try to attain a general perspective and will call out platform specific details along the way.

For systems with persistent memory, failure atomicity guarantees systems can always recover to a consistent state following a power or system failure. Such failure atomicity can be achieved with journaling and flushing as with filesystems for storage. Similarly, failure atomicity can be achieved with user applications using logging, flushing, and barriers that order such operations. Logging, either undo or redo logging, ensures atomicity when a failure interrupts the last atomic operation from completion. Cache flushing ensures that data held within volatile caches reach the persistence domain, so it won't be lost if a sudden failure occurs. Memory store barriers, such as an SFENCE operation on x86, help prevent potential reordering in the memory hierarchy, as caches and memory controllers may reorder memory operations. For example, a barrier ensures the undo log copy of the data gets persisted onto the persistent memory before the data is mutated in-place, so it's guaranteed that the last atomic operation can be rewound, should a failure occur. However, it's non-trivial to add such failure atomicity in user applications with low-level operations such as write logging, cache flushing, and barriers. The persistent memory development kit was developed to isolate developers from having to re-implement the hardware intricacies.

Cache Hierarchy

We use load/store operations to read and write to persistent memory rather than using block based I/O to read and write to traditional storage. It's a good idea to read CPU architecture documentation for an in-depth description since each successive generation may introduce new features, methods, and optimizations.

A CPU cache typically has three distinct levels, L1, L2, and L3. The hierarchy makes references to the distance from the CPU core, its speed and size of the cache. The L1 cache is closest to the CPU. It is extremely fast, but very small. L2 and L3 caches are increasingly larger in capacity and are also slower. Figure 2-1 shows a typical CPU microarchitecture with three levels of CPU cache and a memory controller with three memory channels, each supporting DRAM and persistent memory (PMEM). All levels of CPU cache are volatile so any content that

has not been flushed to non-volatile storage will be lost if the system crashes or power is lost.

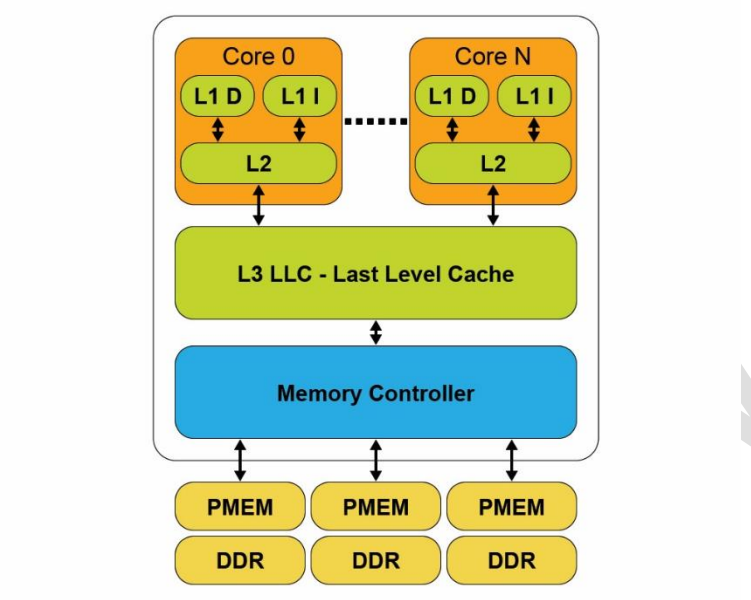


Figure 2-1: CPU Cache and Memory Hierarchy

A L1 (Level 1) cache is the fastest memory that is present in a computer system. In terms of priority of access, the L1 cache has the data the CPU is most likely to need while completing a specific task. The L1 cache is also usually split two ways, into the instruction cache (L1 I) and the data cache (L1 D). The instruction cache deals with the information about the operation that the CPU has to perform, while the data cache holds the data on which the operation is to be performed.

A L2 (Level 2) cache is slower than the L1 cache but bigger. L2 cache holds data that is likely to be accessed by the CPU next. In most modern CPUs, the L1 and L2 caches are present on the CPU cores themselves, with each core getting dedicated caches.

A L3 (Level 3) cache is the largest cache memory unit, and also the slowest. It is also a shared resource among all the cores on the CPU, and it may be internally partitioned to allow each core to have dedicated L3 resources.

Data read from DRAM or persistent memory, is transferred through the memory controller to the L3 cache, then the L2, and finally L1 where the CPU core consumes it. When the processor is looking for data to carry out an operation, it first tries to find it in the L1 cache. If the CPU can find it, the condition is called a cache

hit. It then proceeds to find it in L2, and then L3. If it doesn't find the data in any cache, it tries to access it from memory. This is called a cache miss.

When the CPU writes data, it is initially written to the L1 cache. Due to ongoing activity within the CPU, at some point in time the data will be evicted from the L1 cache into the L2 cache. The data may be further be evicted from L2 and placed into L3 and eventually evicted from L3 into the memory controllers write buffers where it is then written to the memory device.

In a system that does not possess persistent memory, the data in DRAM is persisted to a non-volatile storage device such as an SSD, HDD, SAN, NAS, or a volume in the cloud. This protects data from application or system crashes. Critical data can be manually flushed using calls such as `msync()`, `fsync()` or `fdatasync()` which flushes uncommitted dirty pages from volatile memory to the non-volatile storage device. File systems protect applications from torn pages, perform regular dirty page data flushes, and provide recovery mechanisms in the case of system crashes. Persistent memory provides no such guarantees without software assistance. Given applications are mapping the persistent memory address region directly into its own memory address space, the application must, therefore, assume responsibility for guaranteeing data integrity. The rest of this chapter describes the responsibilities for application developers in a persistent memory environment and how to achieve data consistency and integrity.

Power Fail Protected Domains

A computer system may include one or more CPUs, volatile or persistent memory modules, and non-volatile storage devices such as solid-state drives or hard disk drives.

System platform hardware supports the concept of a persistence domain. Once data has reached a persistence domain, it may be recoverable during a process that results from a system restart. Recoverability depends on whether the pattern of failures affecting the system during the restart can be tolerated by the design and configuration of the persistence domain.

Multiple persistence domains may exist within the same system. It is an administrative act to align persistence domains with volumes and file systems. This must be done in such a way that SNIA NVM Programming Model behavior is assured from each compliant volume or file system.

Volatile memory loses its contents when the computer system's power is interrupted. Just like non-volatile storage devices, persistent memory keeps its

contents even in the absence of system power. Data that has been physically saved to the persistent memory media is called “data at rest.” “Data in flight” refers to writes sent to the persistent memory device but has not yet been physically committed to the media or any write that is in progress, but not yet complete. Data in flight also refers to data that has been temporarily buffered or cached either the CPU caches or memory controller.

When a system is gracefully rebooted or shut down, the system maintains power and can ensure all contents of the CPU caches and memory controllers are flushed such that any data in flight is successfully written to persistent memory or non-volatile storage. In the situation where an unexpected power failure occurs, the system only has enough stored energy within the power supplies and capacitors dotted around the system to flush data before the power drains completely. Any data that is not flushed is lost and not recoverable.

ADR stands for *Asynchronous DRAM Refresh*. ADR is a feature supported on Intel chipsets that triggers a hardware interrupt to the memory controller which flushes the write-protected data buffers and place the DRAM in self-refresh. This process is critical during a power loss event or system crash to ensure the data is in a “safe” state on persistent memory. By default, ADR does not flush the processor caches. To do so, an NMI routine would need to be executed before ADR, called an Enhanced Asynchronous DRAM Refresh event (eADR). Figure 2-2 shows both the ADR and eADR power fail protection domains.

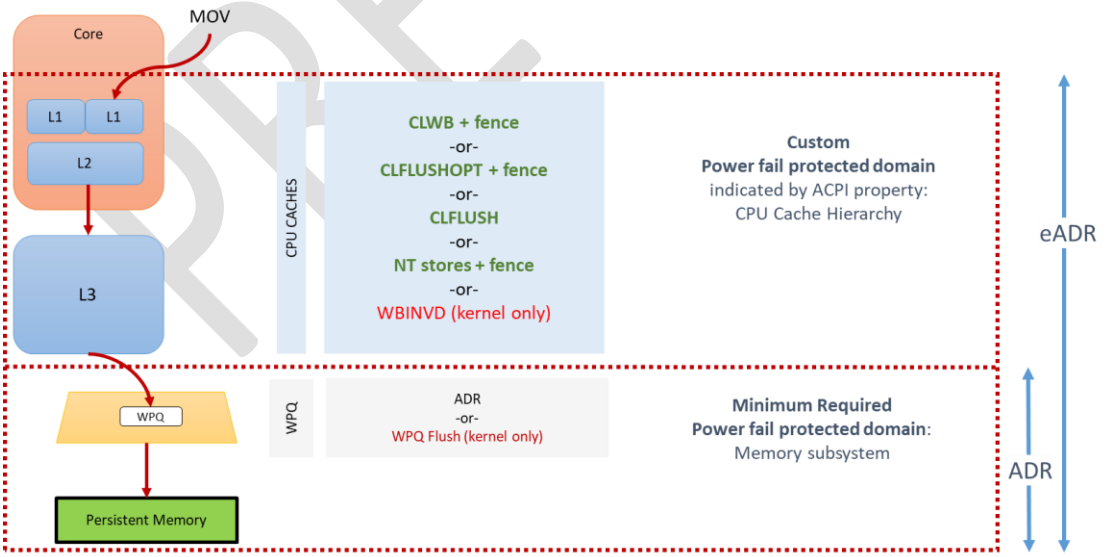


Figure 2-2: ADR and eADR Power Fail Protection Domains

ADR is a platform requirement for persistent memory. The write pending queue (WPQ) within the memory controller acknowledges receipt of the data to the writer once all the data is received. Although the data has not yet made it to the persistent media, a platform supporting ADR guarantees that it will be successfully written should a power loss event occur. If ADR is not available or it has been disabled, the application is fully responsible for flushing data to the media. This is called a deep flush event and can incur significant performance penalties. Data that is in flight through the CPU caches can only be guaranteed to be flushed to persistent media if the platform supports eADR. It will be lost on platforms that only support ADR.

The problem with extending the persistence domain to include the CPU caches is that the CPU caches are quite large and it would take more energy than the capacitors in a typical power supply can practically provide. This usually means the platform would have to contain batteries or utilize an external uninterruptable power supply (UPS). Requiring a battery for every server supporting persistent memory is not generally practical or cost effective and would require additional maintenance which reduces server uptime. It is undoubtedly possible for server or appliance OEM's to include a battery in their product. Doing so would allow the cache flush instructions described above to be skipped. However, the SFENCE instruction is still required as a store barrier. Stores should be considered persistent only when they are globally-visible, which the SFENCE guarantees.

Because some appliance and server vendors plan to use batteries, and because platforms will someday include the CPU caches in the persistence domain, a property is available ACPI such that the BIOS can notify the operating system when the CPU flushes can be skipped. On platforms with eADR, there is no need for manual cache line flushing which allows the operating system to implement calls like `msync()` in the most optimal way.

The Need for Flushing, Ordering, and Fencing

Except for WBINVD, which is a kernel mode only operation, the machine instructions in Table 2-1 are supported in user mode by Intel and AMD CPUs. Intel adopted the SNIA NVM Programming Model for working with persistent memory. This model allows for direct access (DAX) using byte-addressable operations (i.e., load/store). However, the persistence of the data in the cache is not guaranteed until it has

entered the persistence domain, often called the power fail protected domain. x86 provides a set of instructions for flushing cache lines in a more optimized way. In addition to existing x86 instructions such as non-temporal stores, CLFLUSH, and WBINVD, two new instructions were added. These were CLFLUSHOPT and CLWB. Both instructions must follow by an SFENCE to ensure all flushes are completed before continuing. Flushing a cache line using CLWB, CLFLUSHOPT, or CLFLUSH and using non-temporal stores are all supported from user space. Details for each machine instruction can be found in the software developer manuals for the architecture. On Intel platforms for example, this detailed information can be found in the Intel® 64 and IA-32 Architectures Software Developer Manuals (https://software.intel.com/en-us/articles/intel-sdm).

Non-Temporal Stores means that the data being stored is not going to be read again soon; i.e. no “temporal locality” so there is no benefit to keeping the data in the processor’s cache(s), and there may be a penalty if the stored data displaces other useful data from the cache(s).

Flushing to persistent memory directly from user space negates calling into the kernel, which make it highly efficient. The feature is documented in the SNIA persistent memory programming model specification as “Optimized Flush.” The specification document describes Optimized Flush as optionally supported by the platform, depending on the hardware and operating system support. Despite the CPU support, it is essential for applications to only use optimized flushes when the operating system says it is safe to use. The operating system may require the control point provided by calls like `msync()` when, for example, there are changes to file system metadata that need to be written as part of the `msync()` operation.

To better understand instruction ordering, let's look at a very simple linked list example. Our pseudo code has three simple steps to add a new node into an existing list that already contains two nodes:

1. Create the new node (Node 2)
2. Update the node pointer (next pointer) to point to the last node in the list (Node 2 -> Node 1)
3. Update the head pointer to point at the new node (Head -> Node 2)

Figure 2-3 depicts the initial state and the three steps to add the new node.

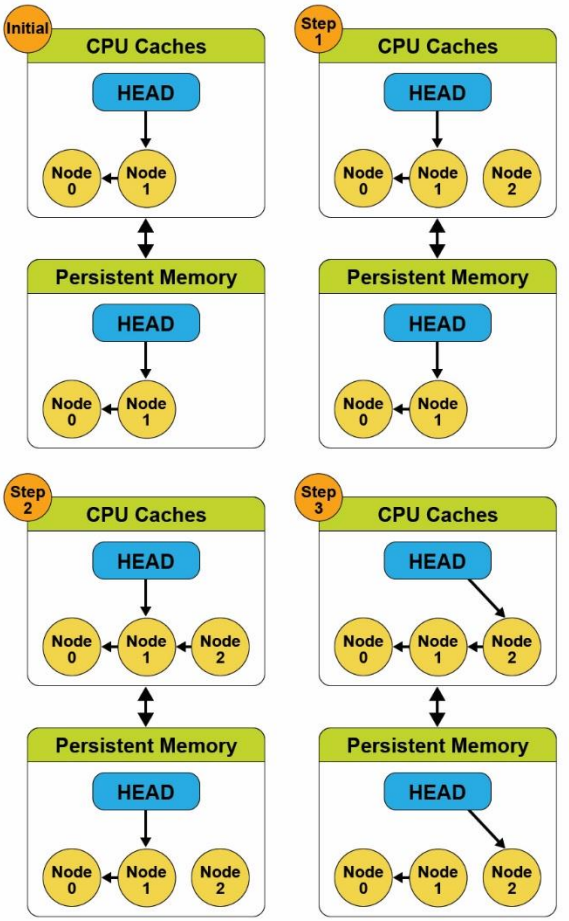


Figure 2-3: A simple linked list example without a memory barrier/fence

Figure 2-3 shows the head pointer was updated in the cached version, but the node 2 to node 1 pointer hasn't been updated in persistent memory yet. This is because the hardware can choose which cache lines to commit and the order may not match the source code flow. If the system or application were to crash at this point, the persistent memory state is inconsistent, and the data structure is no longer usable.

The solution to this problem is to introduce a memory barrier, also called a fence, such that the code now looks like this:

1. Create the new node
2. Update the node pointer (next pointer) to point to the last node in the list
3. Memory Barrier/Fence
4. Update the head pointer to point at the new node

As shown in Figure 2-4, the code now works as expected and maintains a consistent data structure in the volatile CPU caches and on persistent memory.

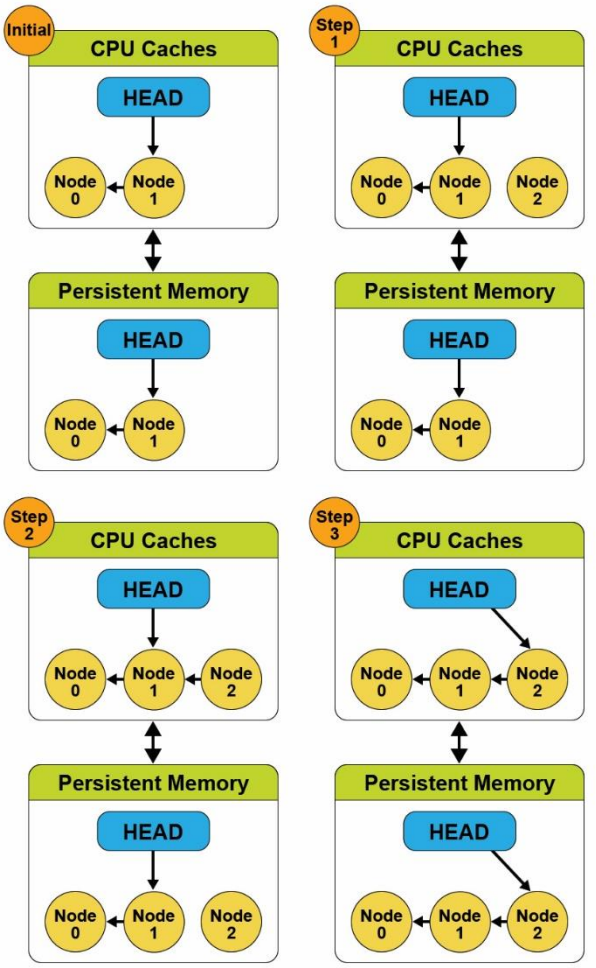


Figure 2-4: A simple linked list example with a memory barrier/fence

The Persistent Memory Development Kit (PMDK) detects the platform, CPU, and persistent memory features when the memory pool is opened, and then uses the optimal instructions and ordering. Memory pools are files that are memory mapped into the process address space.

To insulate application developers from the complexities of hardware, and to keep them from having to research the current state of affairs while programming for persistent memory, the libpmem library provides a function which tells the application when Optimized Flush is safe.

Developers are encouraged to use libraries, such as libpmem, that determine whether it is safe to perform user space flushing or fail back to the standard way of flushing stores to memory mapped files. The libpmem library is also designed to detect the case of the platform with a battery, turning flush calls into simple SFENCE instructions instead. Chapter 5 introduces and describes the core libraries within the PMDK in more detail.

Data Visibility

Understanding when data is visible to other processes or threads and when it is safely in the persistent domain is a critical aspect to using persistent memory in applications. In Figure 2-2 and 2-3 we showed that updates made to locally cached data could become visible to other processes or threads that run on CPU threads that have access to the same cached version. Depending on which CPU cache the data is in, i.e. L1, L2, or L3, will depend on the visibility of the data across the CPUs in the system. Only when the data is written to persistent memory will it become globally visible. This is one reasons we use flushing and fencing operations.

An example pseudo C code example may look like this:

```
open()   // Open a file on a file system
...
mmap()   // Memory map the file
...
strcpy() // Execute a store operation
...      // Data is now visible but not persistent
msync()  // Data is now persistent
```

Developing for persistent memory follows this decades-old model. The difference is that stores are cached in different caches on the system. Applications have very little physical control over when data is evicted from the various hardware caches unless explicitly flushing using one of the user space machine instructions described in Table 2-1.

Intel Machine Instructions for Persistent Memory

Applicable to Intel and AMD based systems, executing an Intel® 64 and IA-32 architecture store instruction is not enough to make data persistent since the data may be sitting in the CPU caches indefinitely and could be lost by a power failure. Additional cache flush actions are required to make the stores persistent. Importantly, these non-privileged cache flush operations can be called from user space; meaning applications decide when and where to fence and flush data. Table 2-1 summarizes each of these instructions. For more detailed information, the "Intel® 64 and IA-32 Architectures Software Developer Manuals" can be found at https://software.intel.com/en-us/articles/intel-sdm.

As a developer, you should primarily focus on CLWB and Non-Temporal Stores if available, and fall back to the others as necessary. We list other opcodes for completeness and describe how libraries such as libpmem from the PMDK detects platform capabilities and uses the most optimal flushing operation.

Table 2-1: Intel Architecture Instructions for Persistent Memory

OPCODE	***Description***
CLFLUSH	This instruction, supported in many generations of CPU, flushes a single cache line. Historically, this instruction is serialized, causing multiple CLFLUSH instructions to execute one after the other, without any concurrency.
CLFLUSHOPT (followed by an SFENCE)	This instruction, newly introduced for persistent memory support, is like CLFLUSH but without the serialization. To flush a range, the

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)

CLWB stands for “Cache Line Write Back.” The effect is the same as CLFLUSHOPT except that the cache line may remain valid in the cache but no longer dirty since it was flushed. This makes it more likely to get a cache hit on this line if the data is accessed again later.

Non-Temporal stores (followed by an SFENCE)

Another feature that has been around for a while in x86 CPUs is the non-temporal store. These stores are “write combining” and bypass the CPU cache, so using them does not require a flush. The final SFENCE instruction is still required to ensure the stores have reached the persistence domain.

SFENCE

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes in program order the SFENCE instruction is globally visible before any store instruction that follows the SFENCE instruction is globally visible. The SFENCE instruction is ordered with respect store instructions, other SFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions or the LFENCE instruction.

WBINVD

This kernel-mode-only instruction flushes

and invalidates every cache line on the CPU that executes it. After executing this on all CPUs, all stores to persistent memory are certainly in the persistence domain, but all cache lines are empty, impacting performance. Also, 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.

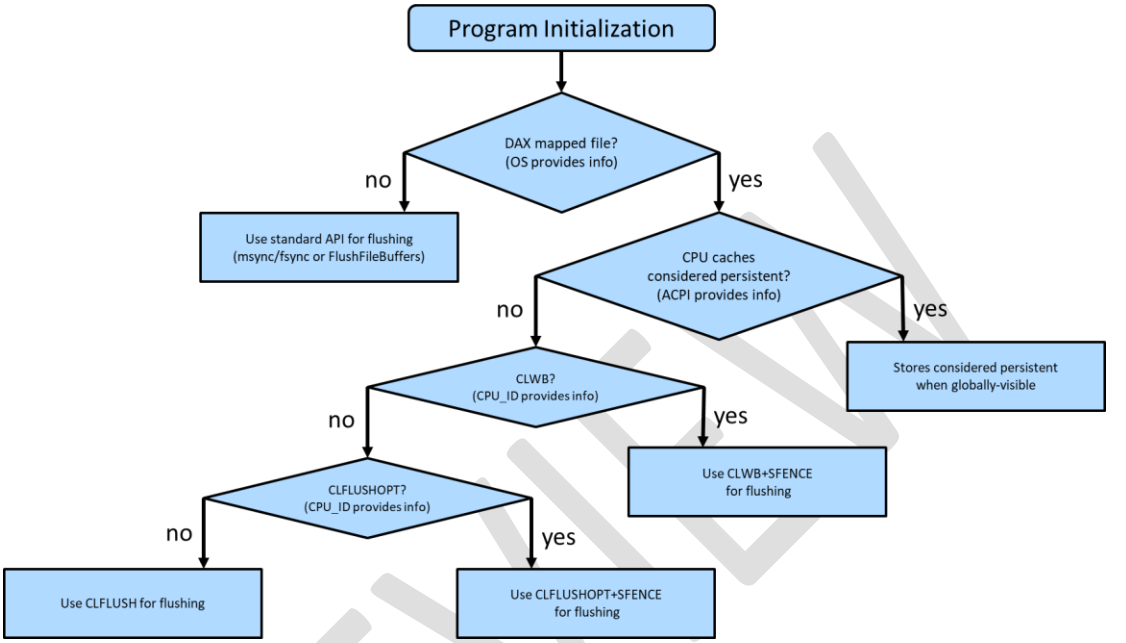
Detecting Platform Capabilities

Server platforms, including CPU and persistent memory features and capabilities, are exposed to the operating system and can be queried by applications. Applications should not assume they are running on hardware with all the optimizations available. Even if the physical hardware supports it, virtualization technologies may or may not expose those features to the guests, or your operating system may or may not implement them. As such, applications should either use libraries such as the persistent memory development kit that perform the required feature checks or implement the checks within the application code base.

Figure 2-5 shows the flow implemented by libpmem which initially verifies the memory mapped file, called a memory pool, resides on a filesystem that has the *Direct Access* (DAX) feature enabled and is backed by physical persistent memory modules. We describe DAX in more detail in chapter 3. On Linux, direct access is achieved by mounting the XFS or ext4 file system with the '-o dax' option. On Windows, NTFS automatically enables DAX when the volume is created and mounted on top of persistent memory. If the filesystem is not DAX enabled, applications should fall back to the legacy approach of using `msync()`, `fsync()` or `FlushFileBuffers()`. If the filesystem is DAX enabled, the next check is to determine whether the platform supports ADR or eADR by verifying whether the CPU caches are considered persistent or not. On an eADR platform where CPU caches are considered persistent, no further action is required. Any data written will be considered persistent, and thus there is no requirement to perform any flushes, which is a significant performance optimization. On an ADR platform, the next

sequence of events identifies the most optimal flush operation based on Intel machine instructions previously described.

Figure 2-5: Flow chart showing how applications can detect platform features



Application Startup and Recovery

In addition to detecting platform features, applications should verify whether the platform was previously stopped and restarted gracefully or ungracefully. Figure 2-6 shows the checks performed by `libpmem` from the persistent memory development kit. `libpmem` provides low-level persistent memory support. In particular, support for the persistent memory instructions for flushing changes to persistent memory is provided. This library is discussed in more detail in chapter 6.

Some persistent memory devices, such as Intel's Optane DC persistent memory, provide counters that can be queried to check the health and status. Several libraries such as libpmemobj query BIOS, ACPI, OS, and persistent memory module information and performs the necessary validation steps and decides which optimal operations to use.

If a system loses power there should be enough stored energy within the power supplies and platform to successfully flush the contents of the memory

controllers write pending queue (WPQ) and the write buffers on the persistent memory devices. Data will be considered consistent upon successful completion. If this process fails, the persistent memory modules will report a "Dirty Shutdown." A dirty shutdown indicates data on the device may be inconsistent. If a dirty shutdown is detected, the operating system, virtualization technology, or hardware may mark the device as offline and request user intervention before the data can be accessed. This may or may not result in restoring data. You can find more information on this process and what errors and signals are sent in the RAS documentation for the platform and persistent memory device.

Assuming the dirty shutdown counter is not set, indicating a clean system reboot, the application should check to see if the persistent memory media is reporting any known poison blocks. These are areas on the physical media that are known to be bad.

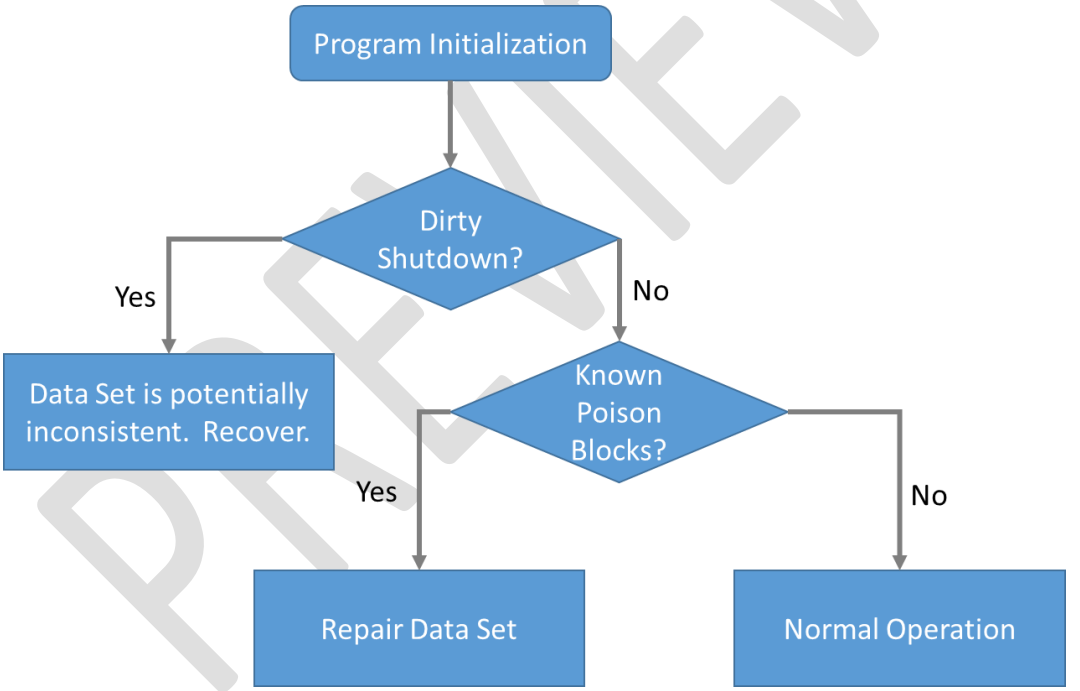


Figure 2-6: Application Startup and Recovery Flow

If an application were not to check these things at startup, due to the persistent nature of the media, the application could get stuck in an infinite loop such as this:

1. Application starts
2. Reads a memory address
3. Encounters poison
4. Application Crashes or System Crashes and reboots
5. Application starts and resumes operation from where it left off
6. Performs a read on the same memory address that triggered the previous restart
7. Application or System crashes
8. …
9. Repeats infinitely until manual intervention

The ACPI specification defines an *Address Range Scrub* (ARS) operation which the operating system implements. This allows the OS to perform a run time to scan the memory address range of the persistent memory in the background or permit the system administrator to initiate one manually. The intent is to identify bad or potentially bad memory regions before the application. It can provide a status notification to the OS and application that can be consumed and handled gracefully. If the bad address range contains data, some method to reconstruct or restore the data needs to be employed. We describe ARS in more detail in chapter N.

Applications are free to implement these features directly, however, using libraries contained within the PMDK handle these complex situations and will be maintained for each product generation while maintaining stable APIs. This gives application developers a future proof option without needing to understand the intricacies of each CPU or persistent memory product.

What's Next?

In chapter 3 we continue to provide foundational information from the perspective of the kernel and user spaces. We describe how operating systems such as Linux and Windows adopted and implemented the SNIA Non-Volatile Programming Model which defines recommended behavior between various user space and operating system kernel components supporting persistent memory. Future chapters build on the foundations provided in chapters 2 and 3.

Summary

In this chapter, we've defined persistent memory, its characteristics, recapped how CPU caches work, and described why it's crucial for applications directly accessing persistent memory to assume responsibility for flushing CPU caches. This chapter was focused primarily on hardware implementations. User libraries such as those delivered with the persistent memory development kit take on the responsibilities for architecture and hardware specific operations and allow developers to use the simple APIs to implement them. Later in the book, we describe the PMDK libraries in more detail and how to use them in your application. Additional resources can be found on https://pmem.io and https://software.intel.com/pmem.