CHAPTER 2

Persistent Memory Architecture

This chapter provides an overview of the persistent memory architecture while focusing on 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 other storage devices because data does not have to transfer back and forth between the CPU and slower storage devices. Because applications that only use persistent memory may be slower than dynamic random-access memory (DRAM), they should decide what data resides in DRAM, persistent memory, and storage.

The capacity of persistent memory is expected to be many times larger than DRAM; thus, the volume of data that applications can potentially store and process in place is also much larger. This significantly reduces the number of disk I/Os, which improves performance and reduces wear on the storage media.

On systems without persistent memory, large datasets that cannot fit into DRAM must be processed in segments or streamed. This introduces processing delays as the application stalls waiting for data to be paged from disk or streamed from the network.

If the working dataset size fits within the capacity of persistent memory and DRAM, applications can perform in-memory processing without needing to checkpoint or page data to or from storage. This significantly improves performance.

Persistent Memory Characteristics

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

- Performance (throughput, latency, and bandwidth) of persistent memory is much better than NAND but potentially slower than DRAM.
- Persistent memory is durable unlike DRAM. Its endurance is usually orders of magnitude better than NAND and should exceed the lifetime of the server without wearing out.
- Persistent memory module capacities can be much larger than DRAM DIMMs and can coexist on the same memory channels.
- Persistent memory-enabled applications can update data in place without needing to serialize/deserialize the data.
- Persistent memory is byte addressable like memory. Applications can update only the data needed without any read-modify-write overhead.
- Data is CPU cache coherent.
- Persistent memory provides direct memory access (DMA) and remote DMA (RDMA) operations.
- Data written to persistent memory is not lost when power is removed.
- After permission checks are completed, data located on persistent memory is directly accessible from user space. No kernel code, file system page caches, or interrupts are in the data path.

- Data on persistent memory is instantly available, that is:
    - Data is available as soon as power is applied to the system.
    - Applications do not need to spend time warming up caches. They can access the data immediately upon memory mapping it.
    - Data residing on persistent memory has no DRAM footprint unless the application copies data to DRAM for faster access.
- Data written to persistent memory modules is local to the system. Applications are responsible for replicating data across systems.

Platform Support for Persistent Memory

Platform vendors such as Intel, AMD, ARM, and others will decide how persistent memory should be implemented at the lowest hardware levels. We try to provide a vendor-agnostic perspective and only occasionally call out platform-specific details.

For systems with persistent memory, failure atomicity guarantees that systems can always recover to a consistent state following a power or system failure. Failure atomicity for applications can be achieved using logging, flushing, and memory store barriers that order such operations. Logging, either undo or redo, 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 will not be lost if a sudden failure occurs. Memory store barriers, such as an `SFENCE` operation on the x86 architecture, help prevent potential reordering in the memory hierarchy, as caches and memory controllers may reorder memory operations. For example, a barrier ensures that the undo log copy of the data gets persisted onto the persistent memory before the actual data is modified in place. This guarantees that the last atomic operation can be rolled back should a failure occur. However, it is nontrivial 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 (PMDK) was developed to isolate developers from having to re-implement the hardware intricacies.

Failure atomicity should be a familiar concept, since most file systems implement and perform journaling and flushing of their metadata to storage devices.

Cache Hierarchy

We use load and store operations to read and write to persistent memory rather than using block-based I/O to read and write to traditional storage. We suggest reading the CPU architecture documentation for an in-depth description because each successive CPU generation may introduce new features, methods, and optimizations.

Using the Intel architecture as an example, 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, but they are relatively slower. Figure 2-1 shows a typical CPU microarchitecture with three levels of CPU cache and a memory controller with three memory channels. Each memory channel has a single DRAM and persistent memory attached. On platforms where the CPU caches are not contained within the power-fail protected domain, any modified data within the CPU caches that has not been flushed to persistent memory will be lost when the system loses power or crashes. Platforms that do include CPU caches in the power-fail protected domain will ensure modified data within the CPU caches are flushed to the persistent memory should the system crash or loses power. We describe these requirements and features in the upcoming “Power-Fail Protected Domains” section.

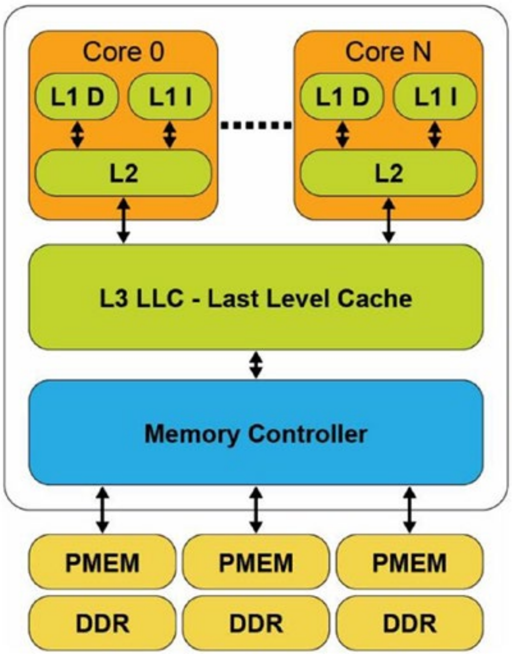

Figure 2-1. *CPU cache and memory hierarchy*

The L1 (Level 1) cache is the fastest memory in a computer system. In terms of access priority, 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.

The L2 (Level 2) cache has a larger capacity than the L1 cache, but it is slower. 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.

The L3 (Level 3) cache is the largest cache memory, but it is also the slowest of the three. It is also a commonly shared resource among all the cores on the CPU and 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 into the L3 cache, then propagated into the L2 cache, and finally the L1 cache where the CPU core consumes it. When the processor is looking for data to carry out an operation, it first tries to find it into the L1 cache. If the CPU can find it, the condition is called a *cache hit*. If the CPU cannot find the data within the L1 cache, it then proceeds to search for it first within L2, then L3. If it cannot find the data in any of the three, it tries to access it from memory. Each failure to find data in a cache is called a *cache miss*. Failure to locate the data in memory requires the operating system to page the data into memory from a storage device.

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 evicted from L2 and placed into L3 and eventually evicted from L3 into the memory controller's write buffers where it is then written to the memory device.

In a system that does not possess persistent memory, software persists data by writing it 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 flush uncommitted dirty pages from volatile memory to the non-volatile storage device. File systems provide `fdisk` or `chkdsk` utilities to check and attempt repairs on damaged file systems if required. File systems do not protect user data from torn blocks. Applications have a responsibility to detect and recovery from this situation. That's why databases, for example, use a variety of techniques such as transactional updates, redo/undo logging, and checksums.

Applications memory map the persistent memory address range directly into its own memory address space. Therefore, the application must assume responsibility for checking and guaranteeing data integrity. The rest of this chapter describes your responsibilities 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 SSDs or HDDs.

System platform hardware supports the concept of a *persistence domain*, also called *power-fail protected domains*. Depending on the platform, a persistence domain may include the persistent memory controller and write queues, memory controller write queues, and CPU caches. Once data has reached the persistence domain, it may be recoverable during a process that results from a system restart. That is, if data is located within hardware write queues or buffers protected by power failure, domain applications should assume it is persistent. For example, if a power failure occurs, the data will be flushed from the power-fail protected domain using stored energy guaranteed by the platform for this purpose. Data that has not yet made it into the protected domain will be lost.

Multiple persistence domains may exist within the same system, for example, on systems with more than one physical CPU. Systems may also provide a mechanism for partitioning the platform resources for isolation. This must be done in such a way that SNIA NVM programming model behavior is assured from each compliant volume or file system. (Chapter 3 describes the programming model as it applies to operating systems and file systems. The "*Detecting Platform Capabilities*" section in that chapter describes the logic that applications should perform to detect platform capabilities including power failure protected domains. Later chapters provide in-depth discussions into why, how, and when applications should flush data, if required, to guarantee the data is safe within the protected domain and persistent memory.)

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* has the following meanings:

- Writes sent to the persistent memory device but have not yet been physically committed to the media
- Any writes that are in progress but not yet complete
- Data that has been temporarily buffered or cached in 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 in-flight or uncommitted data is successfully written to persistent memory or non-volatile storage. When an unexpected power failure occurs, and assuming no uninterruptable power supply (UPS) is available, the system must have enough stored energy within the power supplies and capacitors dotted around it to flush data before the power is completely exhausted. Any data that is not flushed is lost and not recoverable.

Asynchronous DRAM Refresh (ADR) is a feature supported on Intel products which flushes the write-protected data buffers and places 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 and consistent state on persistent memory. By default, ADR does not flush the processor caches. A platform that supports ADR only includes persistent memory and the memory controller's write pending queues within the persistence domain. This is the reason data in the CPU caches must be flushed by the application using the CLWB, CLFLUSHOPT, CLFLUSH, non-temporal stores, or WBINVD machine instructions.

Enhanced Asynchronous DRAM Refresh (eADR) requires that a non-maskable interrupt (NMI) routine be called to flush the CPU caches before the ADR event can begin. Applications running on an eADR platform do not need to perform flush operations because the hardware should flush the data automatically, but they are still required to perform an SFENCE operation to maintain write order correctness. Stores should be considered persistent only when they are globally visible, which the SFENCE guarantees.

Figure 2-2 shows both the ADR and eADR persistence domains.

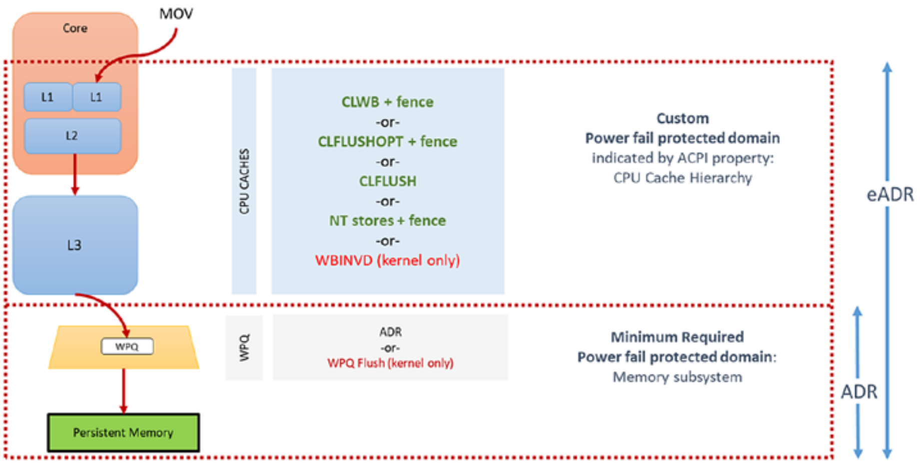

Figure 2-2. *ADR and eADR power-fail protection domains*

ADR is a mandatory 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. During a crash or power failure, 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 challenge with extending the persistence domain to include the CPU caches is that the CPU caches are quite large and it would take considerably more energy than the capacitors in a typical power supply can practically provide. This means the platform would have to contain batteries or utilize an external uninterruptable power supply. Requiring a battery for every server supporting persistent memory is not generally practical or cost-effective. The lifetime of a battery is typically shorter than the server, which introduces additional maintenance routines that reduce server uptime. There is also an environmental impact when using batteries as they must be disposed of or recycled correctly. It is entirely possible for server or appliance OEMs to include a battery in their product.

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 within ACPI such that the BIOS can notify software when the CPU flushes can be skipped. On platforms with eADR, there is no need for manual cache line flushing.

The Need for Flushing, Ordering, and Fencing

Except for WBINVD, which is a kernel-mode-only operation, the machine instructions in Table 2-1 (in the "Intel Machine Instructions for Persistent Memory" section) are supported in user space 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. The x86 architecture 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: `CLFLUSHOPT` and `CLWB`. Both new instructions must be followed 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. You can find details for each machine instruction in the software developer manuals for the architecture. On Intel platforms, for example, this information can be found in the Intel 64 and 32 Architectures Software Developer Manuals (`https://software.intel.com/en-us/articles/intel-sdm`).

Non-temporal stores imply that the data being written is not going to be read again soon, so we bypass the CPU caches. That is, there is 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 makes it highly efficient. The feature is documented in the SNIA persistent memory programming model specification as an *optimized flush*. The specification document[1] 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 use only optimized flushes when the operating system indicates that 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, consider a very simple linked list example. Our pseudocode described in the following has three simple steps to add a new node into an existing list that already contains two nodes. These steps are depicted in Figure 2-3.

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 (Step 3) shows that the head pointer was updated in the CPU cached version, but the Node 2 to Node 1 pointer has not yet been updated in persistent memory. 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 would be inconsistent, and the data structure would no longer be usable.

[1]SNIA NVM programming model spec: `https://www.snia.org/tech_activities/standards/curr_standards/npm`

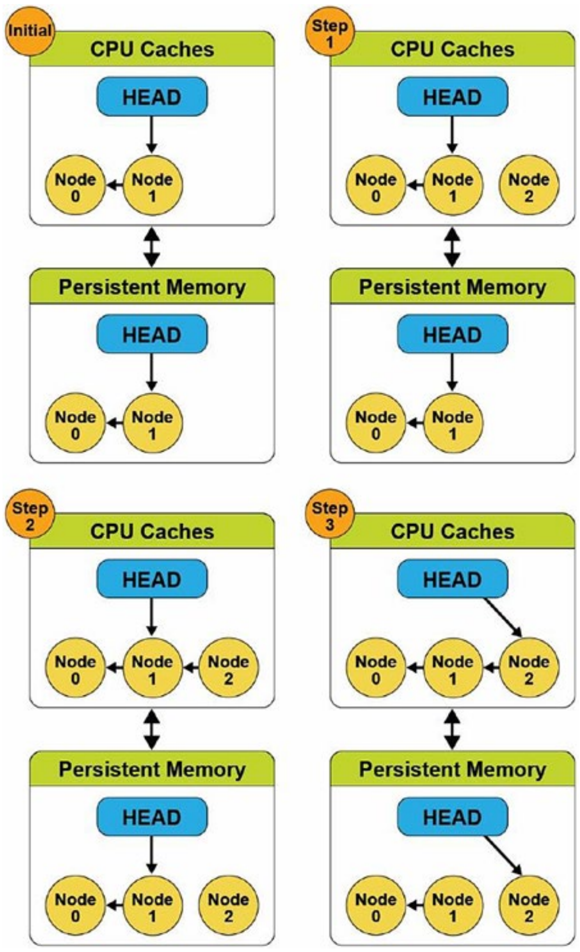

Figure 2-3. *Adding a new node to an existing linked list without a store barrier*

To solve this problem, we introduce a memory store barrier to ensure the order of the write operations is maintained. Starting from the same initial state, the pseudocode 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, and perform a store barrier/fence operation.
3. Update the head pointer to point at the new node.

Figure 2-4 shows that the addition of the store barrier allows the code to work as expected and maintains a consistent data structure in the volatile CPU caches and on persistent memory. We can see in Step 3 that the store barrier/fence operation waited for the pointer from Node 2 to Node 1 to update before updating the head pointer. The updates in the CPU cache matches the persistent memory version, so it now globally visible. This is a simplistic approach to solving the problem because store barriers do not provide atomicity or data integrity. A complete solution should also use transactions to ensure the data is atomically updated.

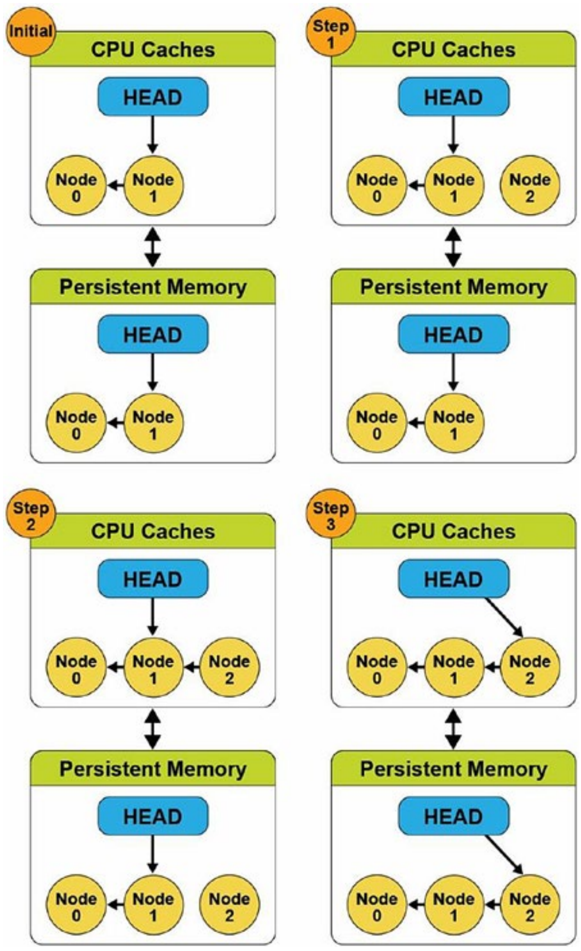

Figure 2-4. *Adding a new node to an existing linked list using a store barrier*

The PMDK detects the platform, CPU, and persistent memory features when the memory pool is opened and then uses the optimal instructions and fencing to preserve write ordering. (Memory pools are files that are memory mapped into the process address space; later chapters describe them in more detail.)

To insulate application developers from the complexities of the hardware and to keep them from having to research and implement code specific to each platform or device, the `libpmem` library provides a function that tells the application when optimized flush is safe to use or fall back to the standard way of flushing stores to memory-mapped files.

To simplify programming, we encourage developers to use libraries, such as `libpmem` and others within the PMDK. The `libpmem` library is also designed to detect the case of the platform with a battery that automatically converts flush calls into simple `SFENCE` instructions. Chapter 5 introduces and describes the core libraries within the PMDK in more detail, and later chapters take an in-depth look into each of the libraries to help you understand their APIs and features.

Data Visibility

When data is visible to other processes or threads, and when it is safe in the persistence domain, is critical to understand when using persistent memory in applications. In the Figure 2-2 and 2-3 examples, updates made to data in the CPU caches could become visible to other processes or threads. Visibility and persistence are often not the same thing, and changes made to persistent memory are often visible to other running threads in the system before they are persistent. Visibility works the same way as it does for normal DRAM, described by the memory model ordering and visibility rules for a given platform (for example, see the Intel Software Development Manual for the visibility rules for Intel platforms). Persistence of changes is achieved in one of three ways: either by calling the standard storage API for persistence (`msync` on Linux or `FlushFileBuffers` on Windows), by using optimized flush when supported, or by achieving visibility on a platform where the CPU caches are considered persistent. This is one reason we use flushing and fencing operations.

A 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 globally visible
msync()  // Data is now persistent
```

Developing for persistent memory follows this decades-old model.

Intel Machine Instructions for Persistent Memory

Applicable to Intel- and AMD-based ADR platforms, executing an Intel 64 and 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 32 Architectures Software Developer Manuals are online at `https://software.intel.com/en-us/articles/intel-sdm`.

Developers should primarily focus on `CLWB` and `Non-Temporal Stores` if available and fall back to the others as necessary. Table 2-1 lists other opcodes for completeness.

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)	The effect of cache line writeback (`CLWB`) is the same as `CLFLUSHOPT` except that the cache line may remain valid in the cache but is 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)	This feature has existed for a while in x86 CPUs. These stores are “write combining” and bypass the CPU cache; using them does not require a flush. A final `SFENCE` instruction is still required to ensure the stores have reached the persistence domain.

(*continued*)

Table 2-1. (*continued*)

OPCODE	Description
SFENCE	Performs a serializing operation on all store-to-memory instructions that were issued prior to 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 can be globally visible. The `SFENCE` instruction is ordered with respect to 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 (at least many megabytes).

Detecting Platform Capabilities

Server platform, CPU, and persistent memory features and capabilities are exposed to the operating system through the BIOS and ACPI that 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, we encourage developers to use libraries, such as those in the PMDK, 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 file system that has the DAX feature enabled, and is backed by physical persistent memory. Chapter 3 describes DAX in more detail.

On Linux, direct access is achieved by mounting an XFS or ext4 file system with the `"-o dax"` option. On Microsoft Windows, NTFS enables DAX when the volume is created and formatted using the DAX option. If the file system is not DAX-enabled, applications should fall back to the legacy approach of using `msync()`, `fsync()`, or `FlushFileBuffers()`. If the file system is DAX-enabled, the next check is to determine whether the platform supports ADR or eADR by verifying whether or not the CPU caches are considered persistent. 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.

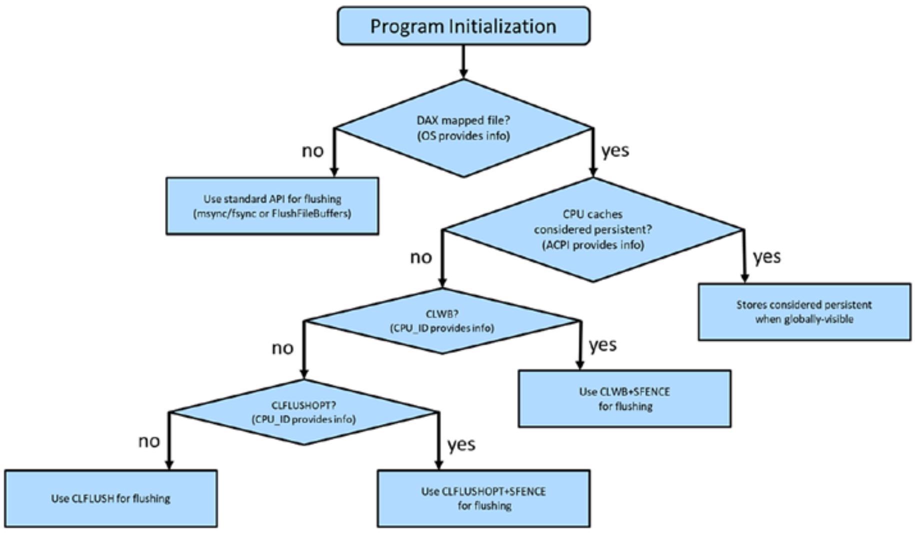
Program Initialization
DAX mapped file? (OS provides info)
no
yes
Use standard API for flushing (msync/fsync or FlushFileBuffers)
CPU caches considered persistent? (ACPI provides info)
no
yes
Stores considered persistent when globally-visible
CLWB? (CPU_ID provides info)
no
yes
Use CLWB+SFENCE for flushing
CLFLUSHOPT? (CPU_ID provides info)
no
yes
Use CLFLUSH for flushing
Use CLFLUSHOPT+SFENCE for flushing

Figure 2-5. *Flowchart 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 the Persistent Memory Development Kit.

Some persistent memory devices, such as Intel Optane DC persistent memory, provide SMART counters that can be queried to check the health and status. Several libraries such as `libpmemobj` query the BIOS, ACPI, OS, and persistent memory module information then perform the necessary validation steps to decide which flush operation is most optimal to use.

We described earlier that 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 controller's WPQ and the write buffers on the persistent memory devices. Data will be considered consistent upon successful completion. If this process fails, due to exhausting all the stored energy before all the data was successfully flushed, the persistent memory modules will report a *dirty shutdown*. A dirty shutdown indicates that data on the device may be inconsistent. This may or may not result in needing to restore the data from backups. You can find more information on this process – and what errors and signals are sent – in the RAS (reliability, availability, serviceability) documentation for your platform and the persistent memory device. Chapter 17 also discusses this further.

Assuming no dirty shutdown is indicated, the application should check to see if the persistent memory media is reporting any known poison blocks (see Figure 2-6). Poisoned blocks are areas on the physical media that are known to be bad.

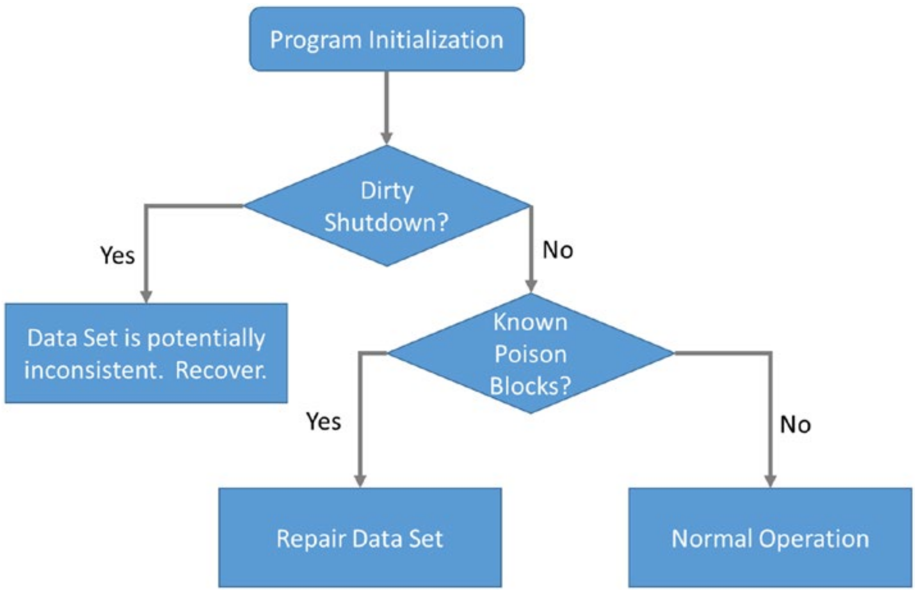

Figure 2-6. *Application startup and recovery flow*

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

1. Application starts.
2. Reads a memory address.
3. Encounters poison.
4. Crashes or system crashes and reboots.
5. 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 that the operating system implements. This allows the operating system to perform a runtime background scan operation across the memory address range of the persistent memory.

System administrators may manually initiate an ARS. The intent is to identify bad or potentially bad memory regions before the application does. If ARS identifies an issue, the hardware can provide a status notification to the operating system and the 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 implemented. Chapter 17 describes ARS in more detail.

Developers are free to implement these features directly within the application code. However, the libraries in the PMDK handle these complex conditions, and they will be maintained for each product generation while maintaining stable APIs. This gives you a future-proof option without needing to understand the intricacies of each CPU or persistent memory product.

What's Next?

Chapter 3 continues to provide foundational information from the perspective of the kernel and user spaces. We describe how operating systems such as Linux and Windows have adopted and implemented the SNIA non-volatile programming model that defines recommended behavior between various user space and operating system kernel components supporting persistent memory. Later chapters build on the foundations provided in Chapters 1 through 3.

Summary

This chapter defines persistent memory and its characteristics, recaps how CPU caches work, and describes why it is crucial for applications directly accessing persistent memory to assume responsibility for flushing CPU caches. We focus primarily on hardware implementations. User libraries, such as those delivered with the PMDK, assume the responsibilities for architecture and hardware-specific operations and allow developers to use simple APIs to implement them. Later chapters describe the PMDK libraries in more detail and show how to use them in your application.