CHAPTER 4

Fundamental Concepts of Persistent Memory Programming

In Chapter 3, you saw how operating systems expose persistent memory to applications as memory-mapped files. This chapter builds on this fundamental model and examines the programming challenges that arise. Understanding these challenges is an essential part of persistent memory programming, especially when designing a strategy for recovery after application interruption due to issues like crashes and power failures. However, do not let these challenges deter you from persistent memory programming! Chapter 5 describes how to leverage existing solutions to save you programming time and reduce complexity.

What's Different?

Application developers typically think in terms of *memory-resident* data structures and *storage-resident* data structures. For data center applications, developers are careful to maintain consistent data structures on storage, even in the face of a system crash. This problem is commonly solved using logging techniques such as *write-ahead logging*, where changes are first written to a log and then flushed to persistent storage. If the data modification process is interrupted, the application has enough information in the log to finish the operation on restart. Techniques like this have been around for many years; however, correct implementations are challenging to develop and time-consuming to maintain. Developers often rely on a combination of databases, libraries, and modern file systems to provide consistency. Even so, it is ultimately the application developer's responsibility to design in a strategy to maintain consistent

data structures on storage, both at run-time and when recovering from application and system crashes.

Unlike storage-resident data structures, application developers are concerned about maintaining consistency of memory-resident data structures at run-time. When an application has multiple threads accessing the same data structure, techniques like *locking* are used so that one thread can perform complex changes to a data structure without another thread seeing only part of the change. When an application exits, crashes, or the system crashes, the memory contents are gone, so there is no need to maintain consistency of memory-resident data structures between runs of an application like there is with storage-resident data structures.

These explanations may seem obvious, but these assumptions that the storage state stays around between runs and memory contents are volatile are so fundamental in the way applications are developed that most developers don't give it much thought. What's different about persistent memory is, of course, that it is persistent, so all the considerations of both storage and memory apply. The application is responsible for maintaining consistent data structures between runs and reboots, as well as the thread-safe locking used with memory-resident data structures.

If persistent memory has these attributes and requirements just like storage, why not use code developed over the years for storage? This approach does work; using the storage APIs on persistent memory is part of the programming model we described in Chapter 3. If the existing storage APIs on persistent memory are fast enough and meet the application's needs, then no further work is necessary. But to fully leverage the advantages of persistent memory, where data structures are read and written in-place on persistence, and accesses happen at the byte granularity, instead of using the block storage stack, applications will want to memory map it and access it directly. This eliminates the buffer-based storage APIs in the data path.

Atomic Updates

Each platform supporting persistent memory will have a set of *native* memory operations that are atomic. On Intel® hardware, the atomic persistent store is 8 bytes. Thus, if the program or system crashes while an aligned 8-byte store to persistent memory is in-flight, on recovery those 8 bytes will either contain the old contents or the new contents. The Intel® processor has instructions that store more than 8 bytes, but those are not failure atomic, so they can be *torn* by events like a power failure. Sometimes an update to a memory-resident data structure will require multiple instructions, so naturally those changes can be torn by power failure as well since

power could be lost between any two instructions. Run-time locking prevents other threads from seeing a partially done change, but locking doesn't provide any failure atomicity. When an application needs to make a change that is larger than 8 bytes to persistent memory, it must construct the atomic operation by building on top of the basic atomics provided by hardware, such as the 8-byte failure atomicity provided by Intel hardware.

Transactions

Combining multiple operations into a single atomic operation is usually referred to as a *transaction*. In the database world, the acronym ACID describes the properties of a transaction: *Atomicity, Consistency, Isolation, and Durability*.

Atomicity

As described earlier, atomicity is when multiple operations are composed into a single atomic action that either happens entirely or does not happen at all, even in the face of system failure. For persistent memory, the most common techniques used are:

- Redo logging, where the full change is first written to a log so during recovery it can be *rolled forward* if interrupted.
- Undo logging, where information is logged that allows a partially done change to be *rolled back* during recovery.
- Atomic pointer updates, where a change is made active by updating a single pointer atomically, usually changing it from pointing to old data to new data.

The above list is not exhaustive, and it ignores the details that can get relatively complex. One common consideration is that transactions often include memory allocation/deallocation. For example, a transaction that adds a node to a tree data structure usually includes the allocation of the new node. If the transaction is rolled back, the memory must be freed to prevent a memory leak. Now imagine a transaction that performs multiple persistent memory allocations and free operations, all of which must be part of the same atomic operation. The implementation of this transaction is clearly more complex than just writing the new value to a log or updating a single pointer.

Consistency

Consistency means that a transaction can only move a data structure from one valid state to another. For persistent memory, programmers usually find that the locking they use to make updates thread-safe often indicate consistency points as well. If it is not valid for a thread to see an intermediate state, locking prevents it from happening, and when it is safe to drop the lock, that is because it is safe for another thread to observe the current state of the data structure.

Isolation

Multithreaded (concurrent) execution is commonplace in modern applications. When making transactional updates, the isolation is what allows the concurrent updates to have the same effect as if they were executed sequentially. At run-time, isolation for persistent memory updates is typically achieved by locking. Since the memory is persistent, the isolation must be considered for transactions that were in-flight when the application was interrupted. Persistent memory programmers typically detect this situation on restart and roll partially done transactions forward or backward appropriately before allowing general-purpose threads access to the data structures.

Durability

A transaction is considered durable if it is on persistent media when it is complete. Even if the system loses power or crashes at that point, the transaction remains completed. As described in Chapter 2, this usually means the changes must be flushed from the CPU caches. This can be done using standard APIs, such as the Linux `msync` call, or platform-specific instructions such as Intel's `CLWB`. When implementing transactions on persistent memory, pay careful attention to ensure that log entries are flushed to persistence before changes are started, and flush changes to persistence before a transaction is considered complete.

Another aspect of the durable property is the ability to find the persistent information again when an application starts up. This is so fundamental to how storage works that we take it for granted. Metadata such as filenames and directory names are used to find the durable state of an application on storage. For persistent memory, the same is true due to the programming model described in Chapter 3, where persistent memory is accessed by first opening a file on a direct access (DAX) file

system, and then memory-mapping that file. However, a memory-mapped file is just a range of raw data; how does the application find the data structures resident in that range? For persistent memory, there mustr be at least one well-known location of a data structure to use as a starting point. This is often referred to as a *root object* (described in Chapter 7). The root object is used by many of the higher-level libraries within PMDK to access the data.

Flushing is Not Transactional

It is important to separate the ideas of flushing to persistence from transactional updates. Flushing changes to storage using calls like `msync` or `fsync` on Linux and `FlushFileBuffers` on Windows, have never provided transactional updates. Applications assume the responsibility for maintaining consistent storage data structures in addition to flushing changes to storage. With persistent memory, the same is true. In Chapter 3, a simple program stored a string to persistent memory and then flushed it to make sure the change was persistent. But that code was not transactional, and in the face of failure the change could be in just about any state—from completely lost to partially lost to fully completed.

A fundamental property of caches is that they hold data temporarily for performance, but they do not typically hold data until a transaction is ready to commit. Normal system activity can cause cache pressure and evict data at any time, and in any order. If the examples in Chapter 3 were interrupted by power failure, it is possible for any part of the string being stored to be lost and any part to be persistent, in any order. It is important to think of the cache flush operation as *flush anything that hasn't already been flushed*, and not as *flush all my changes now*.

Finally, we showed a decision tree in Chapter 2 (Figure 2-5) where an application can determine at start-up that no cache flushing is required for persistent memory. This can be the case on platforms where the CPU cache is flushed automatically on power failure, for example. Even on platforms where flush instructions are not needed, transactions are still required to keep data structures consistent in the face of failure.

Start-Time Responsibilities

In Chapter 2 (Figures 2-5 and 2-6) we showed flowcharts outlining the application's responsibilities when using persistent memory. These responsibilities included

detecting platform details, available instructions, media failures, and so on. For storage, these types of things happen in the storage stack in the operating system. Persistent memory, however, allows direct access, which removes the kernel from the data path once the file is memory-mapped.

As a programmer, you may be tempted to map persistent memory and start using it, as shown in the Chapter 3 examples. For production-quality programming, you want to ensure these start-time responsibilities are met. For example, if you skip the checks in Figure 2-5, you will end up with an application that flushes CPU caches even when it is not required, and that will perform poorly on hardware that does not need the flushing. If you skip the checks in Figure 2-6, you will have an application that ignores media errors and may use corrupted data resulting in unpredictable and undefined behavior.

Tuning for Hardware Configurations

When storing a large data structure to persistent memory, there are several ways to copy the data and make it persistent. You can either copy the data using the common store operations and then flush the caches (if required), or use special instructions like Intel's non-temporal store instructions that bypass the CPU caches. Another consideration is that persistent memory write performance may be slower than writing to normal memory, so you may want to take steps to store to persistent memory as efficiently as possible, by combining multiple small writes into larger changes before storing them to persistent memory. The optimal write size for persistent memory will depend on both the platform it is plugged into and the persistent memory product itself. These examples show that different platforms will have different characteristics when using persistent memory, and any production-quality application will be tuned to perform best on the intended target platforms. Naturally, one way to help with this tuning work is to leverage libraries or middleware that has already been tuned and validated.

Summary

This chapter provides an overview of the fundamental concepts of persistent memory programming. When developing an application that uses persistent memory, you must carefully consider several areas:

- Atomic updates

- Flushing is not transactional
- Start-time responsibilities
- Tuning for hardware configurations

Handling these challenges in a production-quality application requires some complex programming and extensive testing and performance analysis. The next chapter introduces the Persistent Memory Development Kit, designed to assist application developers in solving these challenges.