CHAPTER 1

Introduction to Persistent Memory Programming

This book describes programming techniques for writing applications that use persistent memory. It is written for experienced software developers, but we assume no previous experience using persistent memory. We provide many code examples in a variety of programming languages. Most programmers will understand these examples, even if they have not used the specific language before. Every example in this book is provided on-line in a GitHub repository, along with instructions for building and running it.

Note All code examples are available on: https://github.com/pmem/book

Additional documentation on persistent memory, example programs, tutorials, and details on the Persistent Memory Development Kit (PMDK), used heavily in this book, can be found at the website: http://pmem.io

There are various ways to use the persistent memory products on the market, and many of these ways are transparent to applications. For example, all persistent memory products we've encountered support the storage interfaces, providing the standard API just like any solid-state disk (SSD). But accessing data on an SSD is simple and well-understood, so we consider these use cases outside the scope of this book. Instead, this book concentrates on memory-style access, where applications manage byte-addressable data structures that reside in persistent memory. Some use cases we describe are *volatile*, using the persistent memory only for its capacity and ignoring the fact it is persistent. However, most of this book is dedicated to the *persistent* use cases, where data structures placed in persistent memory are expected to survive crashes and power failures, and the techniques described in this book keep those data structures consistent across those events.

A High-Level Example Program

To illustrate how persistent memory is used, let's start with a sample program demonstrating the key-value store provided by a library called `libpmemkv`. In Listing 1-1 we see a full C++ program that stores three key-value pairs in persistent memory and then iterates through the key-value store, printing all the pairs. This example may seem trivial, but there's quite a bit of interesting components at work here. First, let's walk through the lines of the program, explaining what it does.

Listing 1-1. A sample program using libpmemkv

```
#include <iostream>
#include <cassert>
#include <libpmemkv.hpp>

using namespace pmem::kv;
using std::cerr;
using std::cout;
using std::endl;
using std::string;

/*
 * for this example, create a 1 Gig file
 * called "/daxfs/kvfile"
 */
auto PATH = "/daxfs/kvfile";
const uint64_t SIZE = 1024 * 1024 * 1024;

/*
 * kvprint -- print a single key-value pair
 */
int kvprint(string_view k, string_view v) {
    cout << "key: "    << k.data() <<
         " value: " << v.data() << endl;
    return 0;
}

int main() {
    // start by creating the db object
    db *kv = new db();
    assert(kv != nullptr);
 67
 68      // create the config information for
 69      // libpmemkv's open method
 70      config cfg;
 71
 72      if (cfg.put_string("path", PATH) != status::OK) {
 73          cerr << pmemkv_errormsg() << endl;
 74          exit(1);
 75      }
 76      if (cfg.put_uint64("force_create", 1) != status::OK) {
 77          cerr << pmemkv_errormsg() << endl;
 78          exit(1);
 79      }
 80      if (cfg.put_uint64("size", SIZE) != status::OK) {
 81          cerr << pmemkv_errormsg() << endl;
 82          exit(1);
 83      }
 84
 85
 86      // open the key-value store, using the cmap engine
 87      if (kv->open("cmap", std::move(cfg)) != status::OK) {
 88          cerr << db::errormsg() << endl;
 89          exit(1);
 90      }
 91
 92      // add some keys and values
 93      if (kv->put("key1", "value1") != status::OK) {
 94          cerr << db::errormsg() << endl;
 95          exit(1);
 96      }
 97      if (kv->put("key2", "value2") != status::OK) {
 98          cerr << db::errormsg() << endl;
 99          exit(1);
100      }
101      if (kv->put("key3", "value3") != status::OK) {
102          cerr << db::errormsg() << endl;
103          exit(1);
104      }
105
106      // iterate through the key-value store, printing them
107      kv->get_all(kvprint);
108
109      // stop the pmemkv engine
110      delete kv;
111
112      exit(0);
113  }
```

- Line 57: Here we define a small helper routine, `kvprint()`, which prints a key-value pair when called.
- Line 63: This is the first line of `main()` which is where every C++ program begins execution. We start by instantiating a key-value engine using the

engine name `"cmap"`. We'll talk about other types of engines later in Chapter 9 – pmemkv: A Persistent In-Memory Key-Value Store.

- Line 70: The cmap engine takes config parameters from a config structure. The parameter `"path"` is configured to `"/daxfs/kvfile"`, which is the path to a persistent memory file, the parameter "`size"` is set to `SIZE`.
- Line 93: Here we add several key-value pairs to the store. The trademark of a key-value store is the use of simple operations like `put()` and `get()` – we only show `put()` in this example.
- Line 107: Using the `get_all()` method, we iterate through the entire key-value store, printing each pair when `get_all()` calls our `kvprint()` routine.

What's Different?

There are a wide variety of key-value libraries available in practically every programming language. What's different about the persistent memory example above is that the key-value store itself resides in persistent memory. For comparison, Figure 1-1 shows how a key-value store using traditional storage is laid out.

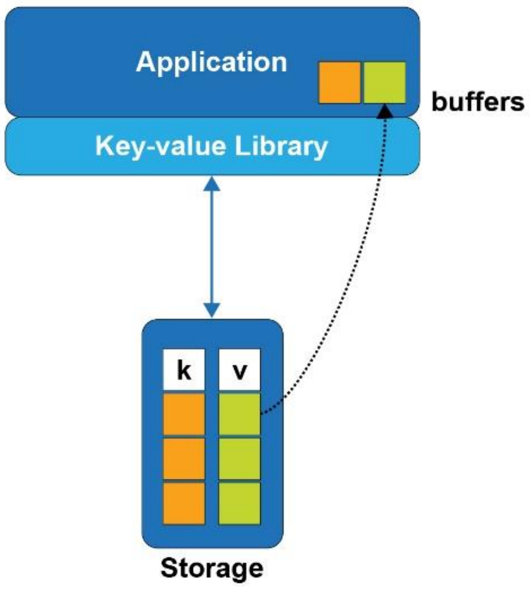

Figure 1-1. A Key-Value Store on Traditional Storage

When the application in Figure 1-1 wants to fetch a value from the key-value store, a buffer must be allocated in memory to hold the result. That's because the values are kept on block storage, which cannot be addressed directly by the application. The only way to access a value is to bring it into memory, and the only way to do that is to read full blocks from the storage device, which can only be accessed via block I/O. Now consider Figure 1-2, where the key-value store resides in persistent memory like our sample code.

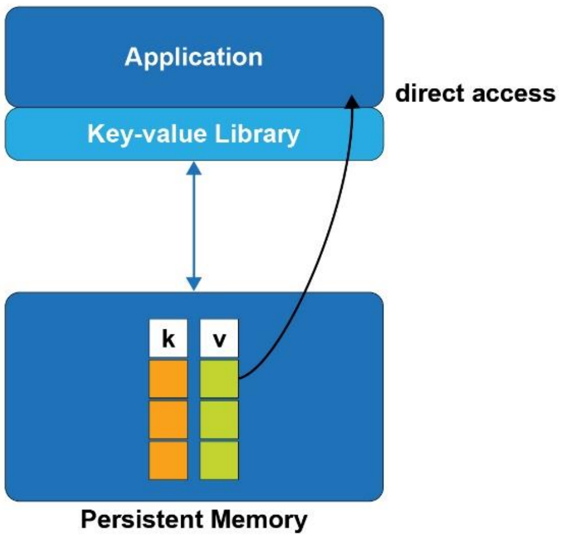

Figure 1-2. A Key-Value Store in Persistent Memory

With the persistent memory key-value store, values are accessed by the application directly, without the need to allocate buffers in memory first. The `kvprint()` routine we showed in Listing 1-1 will be called with references to the actual keys and values, directly where they live in persistence – something that isn't even possible with traditional storage. In fact, even the data structures used by the key-value store library to organize its data are accessed directly. When a storage-based key-value store library needs to make a small update, for example 64 bytes, it must read the block of storage containing those 64 bytes into a memory buffer, update the 64 bytes, and then write out the entire block to make it persistent. That's because storage accesses can only happen using block I/O, typically 4K byte at a time, so the task to update 64 bytes requires reading 4K and then writing 4K. But with persistent memory, the same example of changing 64 bytes would only write the 64 bytes directly to persistence.

The Performance Difference

Moving a data structure from storage to persistent memory doesn't just mean smaller I/O sizes are supported, there's a fundamental performance difference. To illustrate this, Figure 1-3 shows a hierarchy of latency among the different types of media where data can reside at any given time in a program.

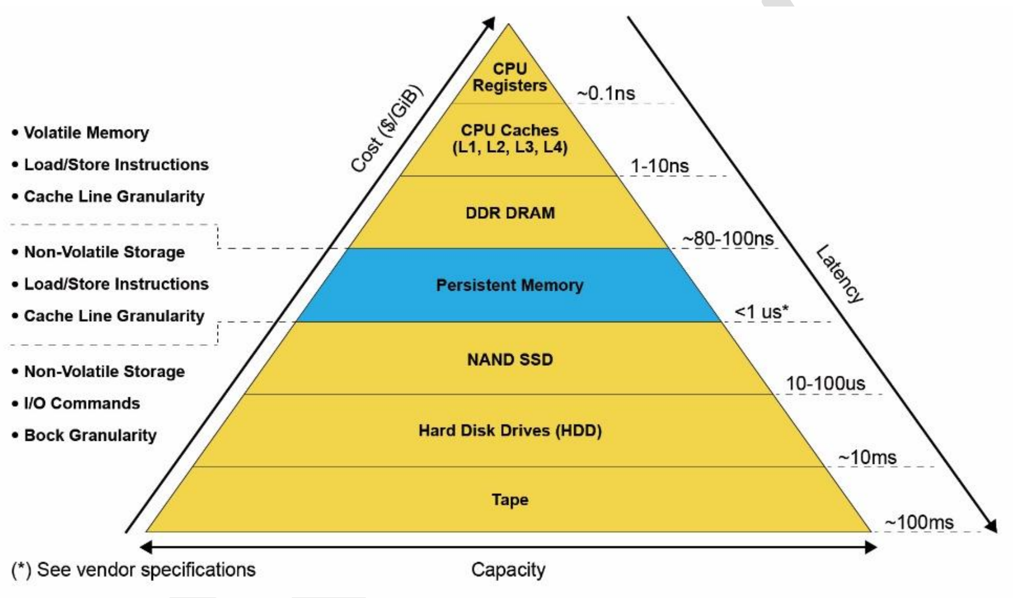

Figure 1-3. The memory/storage hierarchy pyramid with estimated latencies.

As the pyramid shows, persistent memory provides latencies similar to memory, measured in nanoseconds, while providing persistency. Block storage provide persistency with latencies starting in the microseconds and going up from there, depending on the technology. Persistent memory is unique in this ability to act like both memory and storage at the same time.

Program Complexity

Perhaps the most important point of our example is that the programmer still uses the familiar get/put interfaces normally associated with key-value stores. The fact that the data structures are in persistent memory is abstracted away by the high-level API provided by `libpmemkv`. This principal of using the highest level of abstraction possible, as long as it meets the application's needs, will be a recurring theme throughout this book. We start by introducing very high-level APIs but in later chapters of the book we will delve into the lower-level details for programmers who need them. At the lowest level, programming directly to raw persistent memory requires detailed knowledge of things like hardware atomicity, cache flushing, and transactions. High-level libraries like `libpmemkv` abstract away all that complexity and provide much simpler, less error-prone interfaces.

How Does libpmemkv Work?

All the complexity hidden by high-level libraries like `libpmemkv` will be described more fully in later chapters, but let's take a look at the building blocks used to construct a library like this. Figure 1-3 shows the full software stack involved when an application uses `libpmemkv`.

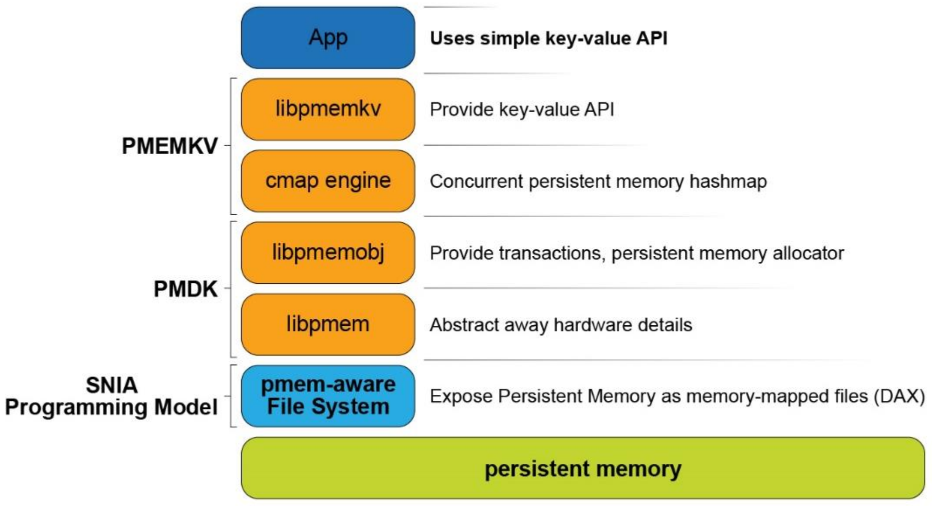
App
Uses simple key-value API
PMEMKV
libpmemkv
Provide key-value API
cmap engine
Concurrent persistent memory hashmap
PMDK
libpmemobj
Provide transactions, persistent memory allocator
libpmem
Abstract away hardware details
SNIA Programming Model
pmem-aware File System
Expose Persistent Memory as memory-mapped files (DAX)
persistent memory

Figure 1-4. The software stack when using `libpmemkv`.

Starting from the bottom of Figure 1-4 and working our way up, we see these components:

- The *persistent memory* hardware, typically connected to the system memory bus and accessed using common memory load/store operations
- A *pmem-aware file system*, which is a kernel module that exposes persistent memory to applications as files. Those files can be memory-mapped to give applications direct access (abbreviated as *DAX*). This method of exposing

persistent memory was published by *SNIA* (Storage Networking Industry Association) and will be described in much more detail in Chapter 3.

- The `libpmem` library, part of the PMDK (Persistent Memory Development Kit). This library abstracts away some of the low-level hardware details like cache flushing instructions. This book will have a great deal to say about PMDK, as it is used heavily in our programming examples.
- The `libpmemobj` library is a full-featured transaction and allocation library for persistent memory. We describe `libpmemobj` in more detail in Chapter and its C++ cousin in Chapter 8. If you are unable to find data structures that meet your needs, the most likely approach you will take is to implement what you need using this library as described in Chapter 11.
- The *cmap* engine, a concurrent hashmap optimized for persistent memory.
- The `libpmemkv` library, providing the API we demonstrated in Listing 1-1.
- And finally, at the top, the application that uses the API provided by `libpmemkv`.

You can see there's quite a stack of components in use here, but that doesn't mean there's necessarily a large amount of code that runs for each operation. Some of the components are used during the initial setup only. For example, the pmem-aware file system is used to find the persistent memory file, and to perform permission checks, but after that it is out of the application's data path. The PMDK libraries are designed to leverage the direct access allowed by persistent memory as much as possible.

What's Next?

Chapters 1 through 3 of this book are designed to provide the essential background that every programmer needs to know in order to start persistent memory programming. Now that we've set the stage with a simple example in this chapter, the next two chapters provide the essential details of persistent memory at the hardware and operating system levels, respectively. Of course, there is always much more detail for those interested, and the later, more advanced chapters of this book will provide much more information. The goal here is to get you programming as quickly as possible, so the recommended way forward is to continue to read through chapters 2 and 3, to finish the essential background, and then dive right in to Chapter 4 where we start showing more detailed persistent memory programming examples.

Summary

In this chapter, we've seen how high-level APIs like `libpmemkv` can be used for persistent memory programming, hiding complex details of persistent memory from the application developer. We've seen how using persistent memory can allow finer-grained access and higher performance than block-based storage. We recommend using the highest-level, simplest APIs possible and only introducing the complexity of lower-level persistent memory programming as necessary. The rest of this book covers persistent memory programming in much more detail.