Non-Volatile Memory and Java

A series of short articles about the impact of non-volatile memory (NVM) on the Java platform.

In the first article I described the main hardware characteristics of Intel's new Optane persistent memory. In this article I will discuss several software issues.

Part 2: The view from software

In use, NVRAM will look just like DRAM, and be accessed through the usual memory-related instructions (load, store, etc.) as virtually-addressed memory.

Data structures in a filesystem

To enable the long-term organization and access of data in NVRAM, data will be encapsulated in a filesystem. The file system will allow independent structures to be contained within separate files, with the attendant metadata (file name, etc.). Readers of a certain age will recall the emblematic feature of the 1980s PC, the RAM disk—but in this case the contents are not lost when the machine is rebooted. And like a RAM disk, data can also be accessed using filesystem operations, but that would lose the benefits (immediacy and granularity) of the load-store interface.

To get the best performance and true memory semantics the filesystem driver in the OS must provide direct load-store access and not copy data. The typical access pattern will be to open a file in this filesystem, map it into the application's address space, operate upon the data directly and then close the file, thereby unmapping the data. This is known as the Direct Access (DAX) model and is already supported by some of the major OSes (e.g., Linux and Windows).[1]

Placing long-lived data in a filesystem has obvious benefits. There are established solutions for backup and archiving, hierarchical organization, access control, space accounting, and so on.

Position independence

It is desirable that the data in a DAX file are arranged to be position independent. If the contents had to be loaded at a specific address there would be the possibility of conflict between independent files, and this would increase if a file was to be portable to a variety of systems and applications. Furthermore, the data in a DAX file could be simultaneously mapped to different addresses, in different processes or even within the same process. Additionally, modern OSes provide Address Space Layout Randomization to make it harder for malware to modify data: at each run, the address of a data segment is randomized so that learning the address of a datum in one run is of no benefit when wishing to tamper with the datum in subsequent runs.

---

[1] https://www.intel.com/content/www/us/en/support/articles/000032860/memory-and-storage/data-center-persistent-memory.html

One way to achieve position independence is to have all internal references be self-relative. Another is to embed relocation metadata allowing the contents to be relocated, either all at once when the file is mapped, or incrementally (e.g., triggered by an initial page fault). Some files may be very large (terabytes or more) and so non-incremental relocation might negate the faster start-up advantages of non-volatility. Also, relocation does not accommodate simultaneous mapping.

Current software does not typically operate on self-relative data, and programming languages do not typically provide support for it. Changing an application written in an unmanaged language, e.g., C, to use self-relative addressing could be a major undertaking.[2] This is a place where a managed runtime, e.g., the JVM, can help, by using self-relative addressing transparently to the application.

The volatile memory hierarchy above NVRAM

Accesses to NVRAM are usually mediated by the memory hierarchy, which contains several levels of caching and buffering. This hierarchy is built from conventional, volatile memory and will lose its contents on power loss.[3] Hence a store to NVRAM will not become durable until the cache line containing the modified data is written to NVRAM.[4]

Intel has added two instructions to x64 to assist:

- CLFLUSHOPT (Flush Cache Line Optimized) evicts every cache line containing a specified address. Unlike the older CLFLUSH, it is not ordered with respect to writes to other cache lines.
- CLWB (Cache Line Write Back) writes back a modified cache line but does not force its eviction.

---

[2] NVM-Direct (https://github.com/oracle/nvm-direct) handles this by extending C to distinguish references within persistent memory and makes those self-relative; the programmer must express the distinction, and the compiler takes care of the details.

[3] it seems unlikely that state held on the CPU (registers, caches) will become non-volatile within, say, a decade. Although some technologies under development have promise as a non-volatile replacement for SRAM (e.g., Spin-Transfer Torque RAM) these are unlikely to be competitive in much less than a decade (see *Emerging Memory Technologies*, DOI 10.1109/MSSC.2016.2546199, 2016). An exception is the emerging field of so-called Non-Volatile Processors (NVPs). The idea here is to use non-volatile state within processors intended for energy-harvesting IoT applications (i.e., which run ephemerally, only when enough energy can be obtained from their environment). However, we will not consider this further, as it is not relevant to processors in data centers and on desktops.

[4] The astute reader will ask: at what point does a write become durable? Between the caches and the NVRAM chips is control logic holding volatile state, such as additional buffering. How does one know that the data have left these buffers and made it to the NVM chips? The answer is: it doesn't matter. Intel guarantees that in the event of power loss the volatile state of the memory controllers will be committed to NVM. Presumably there is a source of energy (such as a supercapacitor) which is sufficient to accomplish this goal. See https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction for details.

These instructions will have to be inserted into applications after non-volatile updates; by the programmer (or library writer) in an unmanaged language (or perhaps by the compiler, if it knows when non-volatile data are being modified), or by the runtime of a managed language. These instructions update NVRAM asynchronously; a fence is used to determine when the update is complete.[5] (An alternative is to use `msync()` or an equivalent, but this is just a wrapper for the write backs and fence and incurs system call overhead.) Unfortunately, this approach still suffers from an observability problem which would require additional coordination between threads relying on a datum becoming durable.[6]

An alternative is to have the system store sufficient energy that in the event of power loss it can write back from the caches *all* unsaved non-volatile data. In a system implementing this scheme writes would, in effect, become durable immediately, and render the special instructions unnecessary.[7] [8]

In the next article I'll start looking at the implications of the hardware and software characteristics.

---

[5] Actually, the fence determines when the update becomes visible according to the coherence protocol; the update can still be pending in the memory controller's write queue. However, memory controllers for Optane will be signaled upon power loss and guarantee that all pending writes will be completed, a property confusingly called Asynchronous DRAM Refresh (ADR) — it has nothing to do with DRAM refresh, so far as I can tell. See *Persistent Memory Programming*, Andy Rudoff, login 42(2), https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf

[6] Bill Bridge, The Observability Problem with Persistent Memory, Oracle memo. Summary: because only the thread issuing the fence can directly know when a datum becomes durable, other threads could read the datum and erroneously propagate the information before it is guaranteed to persist. Avoiding this requires extra coordination between the threads.

[7] This has become known as *extended ADR;* no announced system has this property.

[8] The WBINVD instruction is somewhat related, but not a complete solution: it writes back the contents of all caches, but then invalidates them (undesirable). Also, there not does appear to be a way to know when it has completed the writebacks, as execution continues immediately. Finally, it is a privileged instruction and thus incurs system call overhead.