Non-Volatile Memory and Java: Part 2

Mario Wolczko

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, permissions, time stamps, 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 (Linux and Windows).

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 address conflict between independent files, and the likelihood would increase if a file was to be portable to a variety of systems and applications. Furthermore, the data in a DAX file might 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.

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. An alternative is to provide language and compiler support. For example, NVM-Direct, an extended version of C, requires the programmer to distinguish references within persistent memory and makes those self-relative; the compiler takes care of the details. This is a place where a managed runtime, e.g., the JVM, can help, by providing self-relative addressing transparently to the application, which would be unaware of the change.

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. Hence a store to NVRAM will not become durable until the cache line containing the modified data is written to NVRAM.

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. Even after flushing the cache, 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 an available and sufficient source of energy (such as a supercapacitor). This property is, somewhat confusingly, called *Asynchronous DRAM Refresh* (ADR) — the connection to DRAM refresh is somewhat opaque. For more details see *Persistent Memory Programming* and this Intel article by Andy Rudoff.

Incidentally, it is unlikely that state held on the CPU (registers, caches) will become non-volatile within, say, much less than a decade. Although some technologies under development, e.g., Spin-Transfer Torque RAM, show promise as a non-volatile replacement for SRAM (Static RAM, the kind used for caches) these are unlikely to be competitive soon (*Emerging Memory Technologies*, Yu and Chen, 2016). One exception is in the emerging field of

so-called Non-Volatile Processors. 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.

Writing cache lines to NVRAM

Intel has added two instructions to x64 to assist the writeback of cache lines:

- 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, and so is the more useful of the two (or three).

To ensure durability, a write back instruction will have to be inserted into applications after non-volatile updates to each cache line; 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 instruction can be used to wait until the update is durable. An alternative is to use msync() or an equivalent, but this is likely just a wrapper for the writebacks and fence and incurs system call overhead.

One proposed scheme that obviates the writebacks and fences 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. This has become known as *extended ADR;* I don't believe any systems incorporating

this have been announced. The WBINVD (Write Back and Invalidate Cache) instruction is somewhat related, but not a complete solution: it writes back the contents of all caches, but then invalidates them (undesirable); there not does appear to be a way to know when the writebacks have completed, as execution continues immediately; and this is a privileged instruction and thus incurs system call overhead. WBNOINVD (Write Back and Do Not Invalidate Cache, coming in Ice Lake) is the same but without the invalidation.

The Observability Problem

Having visibility separate from durability plants a trap for the unwary, as my colleague, Bill Bridge, has pointed out (see article below for the full story). To summarize: After a thread updates a shared location in NVRAM, the update becomes visible (via cache coherence) before the value becomes durable (after the write back). This leaves a window in which another thread can take actions based on seeing the new value, but before it can know the value is durable. This may include making dependent updates elsewhere, which can become durable before the original; a failure at this point, e.g., due to a crash or power failure, can lead to inconsistent state during recovery.

—

In the next article I'll look at some implications of these hardware and software characteristics.

THE OBSERVABILITY PROBLEM WITH PERSISTENT MEMORY

Bill Bridge, Oracle

I have uncovered an observability problem with all persistent memory designs I have ever heard of. The basic issue is that a store to persistent memory is observable before it becomes persistent. It is possible for

another processor to see the data and make another update that becomes persistent before the observed data. This creates a window where a power failure can leave inconsistent data in persistent memory. The fix is for a load from a persistent memory location to only see persistent data. I think there is always a software way around the problem if you are aware of it, but that is not a reliable solution.

Controlling Persistent Memory Contents

It is generally recognized that software needs to control the order of updates to persistent memory to ensure the data is consistent after a processor reset or power failure.

In today's processors, without persistent memory, a store to memory goes into the processorcache. Eventually the modified cache line may get flushed from the cache to DRAM. If another processor loads the modified data before it goes to DRAM, then it will get the data from the originalprocessor's cache. Thus the new data is observable before it gets to DRAM. This is perfectly fine for DRAM since all DRAM content is lost if there is a power failure or system reset. (This is an important recovery technique for all computers.)

With persistent memory this mechanism is not acceptable since the result of a store could sit in a processor cache indefinitely and never become persistent. The generally proposed solution to this problem is to provide a mechanism for the software to force the modified cache line to be written to persistent memory. The mechanism includes a means of pausing until the persistent memory acknowledges that the data was successfully written and will be visible if there is a reset or power failure.

Suppose a program needs to update two persistent memory locations A and B ensuring B is only persistent if A is persistent. The steps would be as follows:

1. Store new value to A
2. Force A to be persistent
3. Store new value to B
4. Force B to be persistent

This ensures that any program that sees the new value of B will also see the new value of A. This is true even if the system is rebooted after step 4. However if there is a reset or power failure before step 4 completes then the new value of B will not be visible after a reboot, even though it was visible to other processors after step 3.

The Observability Problem

In the example above, another software thread could read the new value of B as soon as step 3 completes. It could then update other persistent memory locations based on seeing the new value in B. The other memory locations could be on a different memory controller so that they could become persistent before the new value is persistent in B.

In none of the persistent memory proposals I have ever heard of is there an efficient means for a reader to know if the data from a load instruction is persistent or not. One could force the memory location to be persistent immediately after the load. Usually the force would not actually find the data to be dirty, but it is still a lot of overhead. The force cannot be done before the load since there could be a new value stored just after the force.

A Realistic Example

A circular buffer is a common technique used to asynchronously communicate between two different threads of execution. One would think that the queue contents could be preserved through a reboot by putting the data structures in persistent memory. However using the same algorithm with persistent memory will encounter the observability problem. The queue can become persistently corrupt even with the correct forces to persistent

memory.

The persistent circular queue in this example has the following characteristics:

- There is a Producer thread that is storing records into the persistent queue.
- There is a Consumer thread that is fetching records from the persistent queue.
- There is a persistent IN pointer that is advanced by the Producer after a record is stored in the queue
- There is a persistent OUT pointer that is advanced by the Consumer after a record is processed
- If OUT and IN are equal the queue is empty.
- Records after the OUT pointer and before the IN pointer are waiting in the queue.

Here is a scenario where the queue becomes corrupt due to the observability problem:

Producer Thread:

1. Reads IN and OUT seeing that they are equal and thus there is room to store a new record.
2. Stores the record in circular buffer slot pointed to by IN.
3. Forces the record data to be persistent.
4. Advances IN to include the new record in the contents of the queue.
5. An interrupt switches to an OS context. Producer execution is resumed after ten microseconds.
6. Forcec the IN pointer to be persistent, but power fails before the new value becomes persistent.

Consumer Thread:

1. Reads IN and OUT seeing that they are equal so there are no records to process.
2. Uses MWAIT to wait for the IN pointer to advance.
3. MWAIT completes and sees new IN pointer indicating there is a record to process. However IN is not persistently updated as yet. (This is the observability problem)
4. Processes the new record faster than the interrupt handler that paused the Producer.
5. Advances OUT to match the new IN pointer indicating the queue is empty.
6. Forces the OUT pointer to be persistent. This completes successfully making OUT persistent.
7. Power fails leaving IN not advanced but OUT has advanced.

After a reboot the queue is not empty as it should be. Since IN was not advanced, OUT equals IN+1. In the logic of a circular buffer this means the buffer is full. If the queue has fixed size records the contents of the queue will be all old records that were already processed. If the queue handles variable size records then the OUT pointer could be in the middle of an old record and the contents will be garbage.

There is a similar problem with a full queue and an OUT pointer advance that does not become persistent while the IN pointer advance does become persistent. With fixed size records this makes a full queue become empty.

The Solution

The most robust solution is to ensure a load from a persistent memory location only sees data that is persistent. This is likely to involve a change to the cache coherency algorithms to pause the load until the memory controller has made the data persistent. This will make persistent memory work the way most people would assume it works.

There is a software only solution for the circular buffer problem. There

would be an IN pointer in DRAM and an IN pointer in persistent memory. Similarly there would be a DRAM and persistent OUT pointers. Adding or removing a record from the queue would first update the persistent pointer and ensure it is persistent. Then the DRAM pointer would be updated to reflect the persistent pointer. Only the DRAM pointers would be read for finding space or records in the queue. At reboot, the DRAM pointers would be reinitialized from the persistent pointers.

An alternative solution is to flush the current value of the IN or OUT pointer to NVM *after* loading it into a local variable. This ensures the value in the local variable is either the current persistent value or a value that is a previous version of the persistent version. Flushing to NVM before loading could see a new value that never gets persisted.

I expect there is a software solution to the problem for every data structure. However there are several issues with leaving this up to software.

- Many developers would not realize there was a problem
- Every data structure is likely to have a different solution even though the basic concept of a DRAM and persistent copy of all metadata will work in many cases.
- The problem is likely to be so infrequent that it would not be found even in stress testing.
- If a problem occurred in stress testing or at a customer site it would be exceedingly difficult to diagnose.
- If the developer did consider the issue and coded for it, there does not seem to be any wayto force the race condition to test the code. As a wise man once said, “If it is not tested, it is broken.”

Conclusion

The observability problem with persistent memory can cause real world corruption of persistent data. The corruptions are so rare that they are unlikely to be diagnosed. The best solution is to design processors so that a

load from a persistent memory location will only see data that is persistent.