Non-Volatile Memory and Java: Part 3

Mario Wolczko

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

In the first two articles I described the main hardware and software characteristics of Intel's new Optane persistent memory.

In this article I will discuss the implications of these characteristics on how we build software.

Part 3: Benefits and challenges of Non-Volatile RAM

NVRAM offers several attractions:

1. *Reduced startup time.* After a system or application restart, data will be immediately accessible in memory without the delay of reading blocks from secondary storage (i.e., SSDs and HDs) and constructing an in-memory data structure. The challenge here is not in using the data exactly as they were before the restart (which could have been due to a failure) but in recovering to the most recent *consistent* state. Much more on this below.
2. *Reduced latency of durable updates.* Currently, to make an update durable, we have to write to block storage, which at the very least requires traversing a system call and then waiting for a relatively slow medium.
3. *Higher memory density and lower cost/bit than DRAM*, which will make possible larger in-memory datasets and hence faster, more efficient

applications. This can be seen as a one-time efficiency boost to the by-now anticipated progression due to Moore's Law, and does not in itself require dramatic changes in software. JEP 316 proposes to allow the Java heap to be allocated in NVRAM to take advantage of these economies, without exploiting non-volatility.

To fully benefit from persistence, as we shall see, will require changes to application architectures.

Volatility as both curse and cure

It is obvious that the delay incurred by reloading data after a restart is undesirable, and that eliminating this delay will bring the immediate benefit of higher availability. What is less obvious is that reloading data brings with it vital benefits, and the loss of these benefits cannot be tolerated and they must be realized in some other way.

Restarts are used to work around memory corruption bugs

Wiping memory clean and restarting from secondary storage is a panacea for a large class of bugs. More than a half-century of system design and implementation has entrenched this approach: programmers are extremely careful to ensure that updates to secondary storage are correct, but much less careful in avoiding problems in RAM, choosing instead to restart (a process, an application, a node, potentially a whole system) rather than engineering the level of correctness required for continuous, uninterrupted operation. This has been expedient: systems have to cope with unscheduled restarts (due to power loss, hardware errors, operator error, etc.) so why not use the same mechanism to clear up corrupted data?

One can view the current popularity of scale-out architectures (esp. "cloud") as reinforcing this approach: build redundancy at the node level, and deal with node failures by restarting a node, on different hardware if needed. The immense cost of engineering a node that incurs downtime only, say, once every few years, is borne only in scenarios where that downtime is even more expensive (e.g., industrial process control) and in which the inherent complexity of the system being designed is far smaller than that in a contemporary general-purpose software stack.

Coping with hardware faults

Hardware is not totally reliable, and it is uneconomical to attempt to build hardware beyond a certain level of reliability (e.g., to protect against memory corruption by cosmic rays would require immense shielding).

Transient memory faults typically result in the corruption of a small amount of data, perhaps as little as a single bit. In DRAM, they can be caused by stray radiation, and have been carefully characterized. We do not yet know the failure characteristics of 3D XPoint memory. (However, there are results characterizing radiation effects in related technologies, which suggest a high degree of immunity; see *Emerging Memory Technologies*, Yu and Chen, 2016.)

A *correctable* transient memory fault is caught by the Error-Correcting Code (ECC) logic in the memory interface and repaired without the application ever being aware of the problem, using redundant information in the ECC bits. However, when an *uncorrectable* fault is detected the only reasonable response is to restart immediately, before the faulty datum or data can propagate. In a conventional system, this means killing the process using the memory (or the OS, if the memory is OS memory), restarting, falling back to the data held in external storage and reloading the application's data into DRAM.

A *permanent* memory fault means that some region of memory is no longer usable. Enterprise systems to date have used techniques such as chipkill to provide and exploit redundant hardware, switching over from a bad chip to a good spare on the fly.

How are these concepts handled in a system with NVRAM? This UCSD paper states that Optane PMMs contain a mapping table which is used to re-map blocks for wear leveling and bad block management. Also, since they report Optane write latency as very similar to DRAM we can surmise that the initial write is to a buffer, with the write to 3D-XPoint memory taking place later. This will allow time to consult the wear and bad block data, to decide on a final placement, and to update the mapping. Blocks which have incurred correctable errors are presumably avoided; each DIMM is likely to have spare capacity to accommodate some number of bad blocks. However, until we see the detailed Cascade Lake documentation we do not know how uncorrectable errors are reported, nor what mitigation is provided. The details are needed to design an effective recovery system.

In any case, an enterprise system will need a strategy to deal both with uncorrectable transient faults, and permanent failures. One strategy may be to keep a copy of the critical data in secondary storage, just as must be done in current, DRAM-based systems. In the event of the loss of data in NVRAM, the state is reconstructed from the copy. This could involve reading data from block storage, or perhaps replaying a log. In this approach, the application developer cannot eliminate all the code that maintains the external data and rely just on the in-memory structures. This approach may be attractive when enhancing an existing application for NVRAM, in that the code to manage the secondary copy already exists. However, we lose one advantage in that durable updates to the copy in secondary storage will still have longer latency.

An alternative would be to use a RAID-like approach to NV-DIMMs (e.g., mirroring) — simpler in design than backing up to block storage, but less economical in hardware. One can envisage this facility being supported by a managed language runtime (e.g., duplicating each write to a different NV-DIMM), with no additional hardware support and no burden to the application developer, although at some cost in performance. This may be attractive for new applications, as it would save having to write all the code that manages a copy in secondary storage. However, this does not address node failure; the two copies are still in the same failure domain.

Yet another approach is to maintain a parallel copy of the data on another node, to be used as a hot spare. In this case the system architecture has to propagate the updates to the spare(s). Currently, this approach is even more costly in hardware and requires complex software and the performance cost of traversing a network, but provides additional protection against failures and enables high availability. The Gen-Z industry consortium is developing an interconnect to provide memory-semantic access to remote data and devices.

An intermediate solution is to periodically backup the NV-DIMM to external storage. It may be necessary to schedule a quiescent period in order to ensure that the backup is consistent. A backup process designed in concert with the application, and using shared-memory concurrency techniques to coordinate with updates, could make these periods very short (there are parallels with concurrent garbage collection). An update log may be needed to record important changes made between backups. Recovery will take longer than switching over to a mirror.

For any particular system, a trade-off will have to made among hardware cost, additional software complexity, reliability and system availability. Each of the approaches above has different strengths and weaknesses.

Coping with software faults

Mitigating software faults in the presence of persistent memory is a much thornier problem. Software faults which result in memory corruption (in the broadest sense of that term) are extremely widespread and far more common than faults which result in corrupted storage. How can I claim this? Simple: *everybody* expects to restart an application or a machine with some regularity, to correct transient undesired behavior. That a simple restart corrects a problem strongly evinces that the source was in DRAM and not in secondary storage, since the former is reinitialized with state from the latter.

It is unwise to expect the introduction of NVRAM to lead to a sharp reduction in bug rates, but it does seem prudent for the industry to invest more in tools and techniques to help in the prevention, detection and correction of bugs which result in incorrect memory states. Systematic approaches have the promise of bigger payoffs. For example, the use of memory-safe languages, which prevent the application programmer from even expressing certain kinds of memory corruption, have immediate appeal, and Java falls squarely into this camp. A particularly pernicious source of problems is malware which exploits bugs to corrupt memory. The introduction of persistent memory increases the attack surface easily reached by malware and can dramatically widen its reach by making it easier to corrupt long-term data.

Data structure evolution currently requires restarts

An additional reason to restart is to handle software evolution. A new version can change the in-memory representation of data when it is loaded from block storage. If these data are now persistent and not constructed from an out-of-memory canonical copy, then we need a way to evolve the representation in response to software modifications. This problem has received some attention in certain contexts (e.g., dynamic code evolution for Java, see *Dynamic Code Evolution for Java* by Würthinger et al or *UpgradeJ: Incremental Typechecking for Class Upgrades* by Bierman et al.) but in the general case is a difficult, unsolved problem, and even in the limited cases is not widely used. It will require concerted industry effort to develop the underlying techniques, integrate them into software development and deployment practices, and retrain staff.

Restarts from secondary state will still be required

Until these problems are overcome (the last of which may be a long time coming) we may still need occasional restarts from external canonical state or to recover a backup.

When can data in NVM be reused after a restart?

- If the restart is to deploy a revised data structure, then it can be reused only after the old structure gets restructured in place; otherwise the new structure should be populated from secondary storage.
- If the restart was not due to a fault but was scheduled and the system shut down cleanly, then the data can be reused.
- If the restart was caused by an uncorrectable error, then the affected data must be reverted to a known good state (e.g., by restoring a backup and replaying the log).
- For all other failures, we either reload data from secondary storage, or provide a mechanism to revert the data in NVM to a known consistent state (hopefully in less time than required to reload, if this is an option). This requires that (a) software explicitly indicates when NVM is in a consistent state, (b) updates are logged, and (c) during a restart after a fault the runtime undoes updates which took place after the last consistent state: in short, a transacted system, with commits, logging and recovery.

The remainder of this series will explore how to achieve this in Java.