Non-Volatile Memory and Java: Part 4

Mario Wolczko

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

- In Part 1 I introduced the opportunities presented by Intel's new Optane persistent memory and NVRAM in general
- Part 2 presented the software interface of Optane PM.
- In Part 3 I discussed the software engineering challenges arising from the combination of hardware and software characteristics.

In this part, I will address some of the choices specific to the Java platform.

Part 4: Java and non-volatility

In addition to the considerations already described, Java brings additional opportunities and challenges.

Java is the most widely used programming language. In my view, the biggest challenges are to integrate NVM support in a way which does not alienate or confuse many of the existing practitioners, nor requires massive surgery to existing code to take advantage of non-volatility.

Every programming language has a design center — a set of values, characteristics, practices and associated lore to which programmers become accustomed as they become more experienced. Languages vary widely in this respect, which is a good thing — it's best to choose the right tool for the job. Dynamically typed languages value malleability; weakly typed languages value performance. Java, in contrast, requires the programmer to specify types so that type errors cannot occur at runtime.

Java SE VMs have provided excellent peak performance through the use of dynamic feedback-mediated compilation since the late 1990s. Startup performance and pause times have been more of a problem, but a section of the community has been pressing implementors to improve these and the vendors have been delivering: heavyweight compilation has its own thread(s), concurrent low-pause GC has been the subject of much R&D, and ahead-of-time compilation is addressing startup.

Given this background, any extension for NVM should do its best to avoid backward steps in any of these areas. There will almost inevitably be a decrease in peak performance because the underlying NVRAM is slower than DRAM. Additionally, the machinery to implement recovery will impose costs, but it's important to try to minimize these, consistent with the higher-level goals of programmer productivity and compatibility. It's a hard problem, and will take considerable effort from many people to solve.

This is just a hunch but I believe that several viable alternatives will emerge, each with its strengths and weaknesses. This is to be encouraged; the community should understand the trade-offs and can provide invaluable feedback. If no clear winner emerges, it may even be that several different approaches will co-exist for many years. Some approaches may have technical superiority but require more programmer training or adaptation of existing code — these can be adopted over the longer term, but are less likely to gain early adoption. The Java platform has evolved considerably over its more than 20 year lifetime, and if it is to last another 20 years it will evolve still further.

Resilience is key

For NVM to be successful, applications must be able to correctly maintain their data structures over long periods, and must be resilient to failures — if we discard NVM state at every failure, the only gain will be faster planned restarts. This resilience will require additional programmer effort, to specify points of consistency for recovery. Worse, testing recovery is challenging,

to say the least: there are a vast number of points at which failure could occur, compounded by many concurrent states and interactions, and so trying to establish correctness by exhaustive simulation of failures does not appear practical. Instead, the programmer will have to arrive at correctness largely by construction and reasoning, and the design of the facilities will play a large role in determining how difficult this is.

Achieving resilience requires work. Can any work be saved? The biggest prize would be to eliminate the burden of maintaining an additional representation of vital state in secondary storage and all the code that goes along with that. This will only be possible when:

(i) failures are acceptably mitigated (programs correctly implement consistency and recovery works well), and

(ii) the software versioning problem (also known as dynamic software update) has been addressed: i.e., we understand how to evolve in-memory data structures as the program evolves, and the tooling and training are in place.

The benefits from this additional effort are faster restart time and low-latency updates, and so every architect will have to weigh the advantages against the considerable effort required to achieve them.

On- or off-heap persistence?

Should it be possible to persist Java objects, or should persistence be confined to off-heap memory? Until recently, off-heap data had to be accessed via calls to native code through the Java Native Interface (JNI), or via sun.misc.Unsafe, or via java.nio interfaces such as ByteBuffer (see JEP 352). The approach taken by Intel's Persistent Collections for Java library is to use off-heap persistence (more on this library in a future article).

JNI data access is both cumbersome and relatively slow. A new mechanism

to access external code and data is being defined by Project Panama, and this promises to make access to off-heap data considerably easier and quicker.

Will off-heap persistence suffice, or should Java be enhanced to allow Java objects, arrays, classes, etc., to reside in NVRAM? While off-heap access may satisfy immediate needs and is available for use now, using JNI, and soon via Panama, it seems inevitable that there will be a desire to make regular Java entities persistent. After all, the promise of NVRAM is a compatible load-store interface; other than immediate availability, there is no advantage to exploiting NVRAM via a foreign data/code interface and it has the twin disadvantages of programmer complexity and additional overhead (e.g., duplication of metadata on- and off-heap; extra instructions to cross the language boundary, and space overhead).

Additional challenges

Unmanaged or "native" code
A potential source of NVM heap corruption will be bugs in unmanaged code in the same address space. In the best case, the result is a failure in the managed code and data are reverted to an earlier, consistent and correct version. In the worst case, data are corrupted silently and the error is undetected until something really bad has happened (e.g., the erroneous transfer of a large sum of money). This is not a new problem — but the possible corruption of persistent data via erroneous memory operations is new, and the volume of such data could make it more likely.

Garbage collection of very large heaps (terabyte and beyond) is rarely done and there are potential problems: induced pauses, the CPU resources dedicated to GC, the memory wasted by GC falling behind, and the potential of system failure due to memory exhaustion if the GC cannot keep up. Although NVM will be accessed via conventional virtual addressing it is unlikely to be paged since this would compromise the low update latency and so there will be a hard limit on NVM usage. On the plus side, when a

persistent heap region is not in continuous use, GC can be done “off-line” without causing application slowdown. There may be applications in which this is the preferred mode, because they are sensitive to slowdowns and/or do not create a lot of persistent garbage.

The stability of heap formats and object layouts

To date, the layout of objects and the format of the heap is of concern only to the associated JVM instance. They can be changed whenever a JVM is restarted. However, if heap regions are persistent, then object layout and heap format have to be stable over longer periods, and a change will have repercussions, unless the JVM can accommodate multiple formats and layouts (something no current JVM has been designed to do). It may even be desirable to have NVM heap data portable not only among versions of the same JVM implementation, but among different implementations, or even be directly accessible from languages other than Java.

Persistent type information

When a persistent heap is attached to an application, how do we guarantee that the types of objects it contains match the types expected by the application?

A minimum requirement is that the types are checked to ensure that the integrity provided by the JVM is not compromised. To verify structural integrity, the order and type of fields in each object would have to match those being used by the application. For example, if a persistent object contains a pair of doubles, then the associated class in the application would also have to contain a pair of doubles. When a reference type was encountered, the matching would be performed recursively on the referent’s fields, and so on. It may be desirable for this matching to be incremental: eagerly matching all the types in a large persistent heap could take a long time, and some of them may never be accessed in a particular run.

Simple structural matching cannot prevent some elementary errors. For

example, if the application believes the double-pair objects to represent Cartesian coordinates, but they were created as instances of polar coordinates, the error will likely go undetected if simple structural matching is all that is provided. A larger degree of safety can be ensured by also requiring class names to match, but this cannot detect errors due to class version mismatches (e.g., the heap contains Cartesian coordinates with x and y coordinates, but the application assumes they are in the other order). Including and matching field names would eliminate this error, but there can still be mismatches in the code between the version that created an object and the version that consumes the object (e.g., in the assumed orientation of the y-axis).

One possible solution to this problem is have a guaranteed unique ID for every version of a persistent class, and to check this against the class being used in the application. For example, a cryptographic hash of the class, including the bytecode, and dependent on the cryptographic hashes of all dependent classes (inherited classes, etc.) would suffice to ensure that the classes really are the same. Or, the ID could be assigned by the programmer manually (the approach taken by NVM-Direct). The latter has the advantage that the ID need not change if the underlying change in the class file is unimportant; otherwise, such a change would result in a spurious error, but it requires programmers to have a more detailed understanding of the implications of changes.

Type evolution

Requirements change, and software must change in response. This often results in changes to the structure or interpretation of data. In common practice with volatile data structures this is handled by having the canonical version of the data reside externally (i.e., not in RAM); when a new version of the software starts, it builds the updated data structure by populating it from the external data. This approach could be adopted for non-volatile data structures, but it has drawbacks: first, it requires that the external copy exist, together with all the code to maintain it. If resiliency is achieved by

mirroring the non-volatile data, there may not be such a copy. Second, re-loading the data could take a long time, decreasing availability. This is punitive if the structure is large but the change is simple, such as adding a new field to a class. Particularly in managed runtimes, it should be possible to accomplish this without having to rebuild the entire structure; it would be desirable if the update could be performed incrementally, as old objects were reused by the new version. This requires extensions to the programming language that describe such updates, perhaps providing default mechanisms for simple updates (such as adding a field) and allowing the programmer to provide extended logic for non-trivial updates. For Java, a basic mechanism already exists in the form of class redefinition within the Tool Interface, JVMTI. To be used within Java applications it would have to be provided not as a tool interface but in a more central role for use within applications, and it would also have to be implemented completely (currently, it is optional) and with efficiency guarantees (e.g., incrementally).

A more ambitious goal would be to provide direct language support for versioning, in the manner of *UpgradeJ*. One potential direction for exploration would make persistent regions self-describing, in that every object in the region would be accompanied by a complete class definition (perhaps augmented with version information). An application would then access a persistent object only via its accompanying class, with increased confidence that the code was in sync with the data.

This area is relatively unexplored by earlier work and will require considerable thought and experimentation before it is fully understood.

—

In the next article I'll discuss some design choices for Java.