Expert Opinion

Persistent Memory: Abstractions, Abstractions, and Abstractions

Yan Solihin
University of Central Florida

Nonvolatile memory technologies have matured to the extent that commercial DIMM memory products are available today, for example Intel Optane DC Persistent Memory. Compared to DRAM, they provide higher density, better scaling potential, lower cost per bit, and are nonvolatile. Just like DRAM, they provide random access, byte addressability, and can be accessed through load/store instructions. However, writes are slow, expensive, and are limited in endurance. Nevertheless, on the whole, nonvolatile main memory (NVMM) is here and will replace or augment DRAM as main memory fabric.

Utilizing NVMM for capacity is obvious, but the biggest potential for NVMM is providing fabric for hosting data persistently. Currently, when program wants to keep data persistently across process lifetime, it must interface with the file system. When a process wants to work on data, it must create data structures in its address space, read data from a file to populate them. When the process terminates, its address space is reclaimed; data must be written to the file prior to that. The transition between memory and storage is expensive and unnecessary when the main memory is nonvolatile.

There are many challenges in utilizing NVMM. The fact is that memory and storage have had separate interfaces for decades and have been optimized in isolation for different goals. Hence, NVMM is a disruptive technology that requires revisiting this dichotomy for many years to come. The thesis of this paper is that how well systems and programs use NVMM depends on the design of abstractions. Abstraction affects the programming complexity, efficiency of persistency, and scalability.

In programming languages, there are in general three layers of abstraction: assembly language (AL), high-level language (HLL), and domain-specific language (DSL). These abstractions evolved over decades and provide distinct efficiently views of the machine. I argue that utilizing NVMM requires similar layers of abstractions as well.

Current research work in persistency programming mainly focuses on the persistency model. A persistency model defines *durability ordering*, i.e., in what order stores are made durable (reaching the nonvolatile domain) relative to the order in which they appear in the program, and *atomic durability*, in which multiple stores must be made durable together or none at all. An example is *epoch persistency*, which defines epochs separated by persist barriers. Within an epoch, all stores are made durable in any order. However, stores of the one epoch must be durable prior to any store in the next epoch is made durable. Instruction set architecture support for persistency includes instructions to force a cache line to reach the NVMM (e.g., CLFLUSH, CLFLUSHOPT, and CLWB), while a persist barrier is provided through a serializing instruction (e.g., SFENCE). Atomic durability is provided through durable

Digital Object Identifier 10.1109/MM.2018.2885589
Date of current version 21 February 2019.

transactions provided either in hardware or software.

Current persistency support overheads are too high. Even with speculation past a persist barrier,[1] substantial slowdown still results. Logging is another source of overheads, if performed in software. Logging in hardware restricts the count and size of durable transactions, hence a hybrid scheme promises the best of both worlds.[2]

> The thesis of this paper is that how well systems and programs use NVMM depends on the design of abstractions.

Current persistency model abstraction is also too low level, akin to AL abstraction, where programmers must be aware of the memory hierarchy and using appropriate assembly instructions. HLL persistency model is needed to simplify programming, for example language support that allows programmers to define which data is persistent, and which regions must be durable atomically. Abstraction simplicity leads to shorter software development time, and higher quality and reliability of the resulting code.

HLL model must also interact with the system. Hence, the system must provide data abstraction in order to host data persistently in the main memory. Persistency requires data to be encapsulated as a persistent memory object (PMO) that can be located across process lifetime, crashes and reboots, similar to a file. A PMO also needs system-managed permission, sharing, and efficient meta data. Recent research has looked into NVMM file system (e.g., PMFS, BPFS, NOVA, and NOVA-Fortis), or memory mapped files (e.g., Mnemosyne). Both approaches have significant drawbacks: the former incurs much software overhead, while the latter needs to reconcile two systems (virtual memory and file system) that are not necessarily compatible. I believe a PMO abstraction needs to adopt only necessary features from a file system and keep simple metadata.

Beyond HLL, DSL abstraction should also be developed. For example, key-value stores translate quite well to NVMM. Being higher level than HLL, DSL persistency abstraction needs not necessarily lead to high performance overheads, because certain semantics applicable for a particular domain may present an opportunity to relax the persistency model. For example, consider checkpointing in HPC systems. Instead of taking a snapshot of memory to create a checkpoint in the storage system, one may simply persist data structures regularly. If a failure occurs, one may recover by reading the data structures, and recomputing all noncompleted and partially completed regions of code.[3] We may also perform *lazy persistency*, where code regions are protected by error detection codes that detect incomplete persistency.[4] Upon failures, if some regions are detected to not have fully persisted, they are re-executed. Lazy persistency removes the need to perform logging, flush cache lines, and use persist barriers. The result is much faster execution and negligible write amplification, but at the cost of slower and more complex recovery.

Coming up with the right abstractions is very important for NVMM to harness its best potentials. In this paper, I touched upon a few aspects of abstractions, but many more aspects require future research.

REFERENCES

1. S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 175–186.
2. S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for NVM," in *Proc. Int. Symp. Microarchit.*, 2017, pp. 178–190.
3. H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2017, pp. 318–329.
4. M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *Proc. Int. Symp. Comput. Archit.*, 2018, pp. 439–451.

Yan Soiihin is the Director of Cybersecurity & Privacy Cluster and Charles N. Millican Professor of Computer Science at University of Central Florida, Orlando, USA. Contact him at yan.solihin@ucf.edu.