J-NVM: Off-heap Persistent Objects in Java

Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra and Gaël Thomas
Télécom SudParis - IP Paris

1 Motivation

Until recently volatile media were order of magnitude faster than persistent ones. This fundamental difference much impacted the way systems are architectured.

Recent advances in persistent memory technology promise to re-shuffle the cards. In particular, non-volatile main memory (NVMM) is a byte-addressable memory that preserves its content after a power outage. It provides durability with memory performance similar to DRAM, offering the promise of a dramatic increase in storage performance.

To harvest the benefits of NVMM, it is key to integrate it with programming languages. This matters notably for languages used in the design of the distributed storage systems at heart of nowadays computer infrastructures. Such an integration is however challenging because managed object-oriented languages are complex software runtimes which inherit from decades of refinments and optimizations. This paper tackles the problem of integrating NVMM with the Java language.

2 Limitations of the State of the Art

To date, approaches that integrate NVMM with Java use it as a mass storage medium accessible through a file system interface [6, 8, 18], address it through the Java native interface (JNI) [11, 12], or transparently make part of the Java heap persistent [14, 17]. As detailed next, such approaches are generic and unsatisfactory for several reasons.

- The file system and JNI approaches maintain dual representations of data, one in-memory and another on NVMM. This requires to continuously marshal objects back and forth between the persistent and the volatile memory. In particular, complex software mechanisms are necessary to keep the two representations mutually consistent.
- The integrated design solves the dual representation problem: plain Java objects are stored in NVMM and are accessible directly with read and write instructions by the application. However, it requires significant modifications to the Java virtual machine (JVM), and comes with several performance limitations and reliability concerns whose are detailed next.

Garbage collection (GC). Figure 1 shows that garbage collecting just 80 GB can degrade by 3 the completion time, yet NVMM is expected to host hundreds of GBs to TBs of data. Further, we report after studying several NVMM-ready data stores that persistent objects however are often deleted in a very limited number of places. Altogether, the use of garbage collection for persistent objects seems unneeded.

Orthogonal Persistence. The lack of static persistent types raises the need for Java bytecodes instrumentation in order to transparently check whether an object is allocated

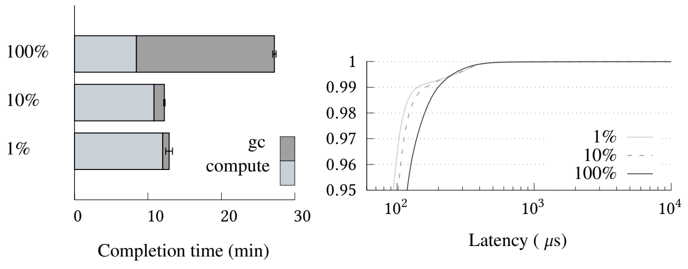

Figure 1. YCSB-F in Java with different cache ratios.

on volatile or persistent memory at runtime. AutoPersist [14] presents a 51% slow down when not even actually using NVMM. (9% with their QuickCheck optimization [15])

Furthermore, when persistent states in the application are not made obvious by types, neither the developper nor the compiler can easily identify bugs since they occur at runtime [2, 9]. Mistaking a volatile object for a persistent one leads to data loss, the opposite to a non-volatile memory leak. Instead of silently loosing data or memory, the runtime should provide help to prevent these situations.

In short, the integrated design offers direct NVMM access but it trades in code simplicity for performance penalty (GC + code instrumentation) and potential reliability issues.

3 Key Insights

This paper proposes to remedy these shortcomings by keeping NVMM outside the Java heap to avoid costly garbage collection while retaining direct NVMM access as in the integrated design. To this end, we introduce a *decoupling principle* between the data structure of a persistent object and its representation in the JVM. Specifically, persistent objects are separated into a data structure that is stored off-heap on NVMM and a proxy Java object that remains on-heap in volatile memory. The data structure holds the fields of the persistent object, while the volatile proxy acts as a gateway to the durable off-heap data structure and implements the methods of the persistent object. With this design, durable data remains outside the Java heap (using a dedicated memory layout), and thus cannot be collected by the Java runtime. The dual representation of data is also avoided thanks to a JVM interface that inlines the low-level instructions that access NVMM directly from Java methods.

4 Main Artifacts

These key ideas are implemented in the J-NVM framework [7], a lightweight pure-Java library that runs on the Hotspot 8 JVM with the minimal addition of three NVMM-specific instructions (`pwb`, `pfence` and `psync` [5]).

J-NVM is a low-level interface that focuses on efficient proxy and memory manipulation. Namely, the bare logic to instantiate and destroy persistent objects and efficiently

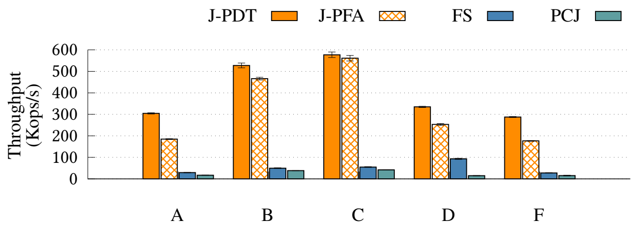

Figure 2. The YCSB benchmark.

access their fields. We build up from J-NVM two higher level interfaces: J-PFA and J-PDT.

- J-PFA provides failure-atomic blocks of code, i.e. a generic way of making any code crash consistent.
- J-PDT is a collection of hand-crafted crash-consistent data structures for NVMM (e.g. arrays, maps, trees), which do not rely on J-PFA for performance.

Moreover, because J-NVM relies on explicit persistent types, we include a code generator to automatically enhance and decouple legacy Java classes into a persistent data structure and a volatile proxy object. It is implemented as an off-line Java bytecode to bytecode post-compilation transformation plugin integrated in the application build system. It scans for annotated classes and apply the transformation for each one, while preserving user-defined functionality in any class hierarchy.

5 Key Results and Contributions

We evaluate J-NVM by implementing several persistent backends for the Infinispan data store [10] and test them on a TPC-B like workload [16] as well as the YCSB benchmark [3]. These implementations are available at [1]. Figure 2 depicts the performance on the YCSB workloads for backends based on J-PFA and J-PDT, the original file-system approach FS sitting atop DAX-ext4, as well as a backend based on PCJ that uses internally the Intel PMDK [13] through the Java Native Interface. J-NVM is significatively more effecient than prior approaches, at least one order of magnitude faster.

Throughout our evaluation campaign, we show that:

- Both the J-PDT and J-PFA systematically outperform the external design. In YCSB, J-PDT is at least 10.5x faster than FS or PCJ, except in a single case where it is only 3.6x faster.
- While the failure-atomic blocks of J-PFA offer an all-around solution, J-PDT, with its hand-crafted persistent data types, executes up to 65% faster. Compared to the Volatile implementation, J-PDT is only 45-50% slower.
- Integrating NVMM in the language runtime hurts performance due to the cost of garbage-collecting the persistent objects. For a Redis-like application written with go-pmem [4], increasing the persistent dataset from 0.3 GB to 151 GB multiplies the completion time of YCSB-F by 3.4

Other relevant insights from the performance analysis of J-NVM:

Marshalling. The low performance of FS comes from (un)marshalling operations to move the persistent objects back and forth between their file and Java representation. PCJ is highly impacted by the cost of JNI calls to escape the Java world. These operations were commonly used and had no significant impact with slower storage media, but can now be bottlenecks with NVMM and should be avoided where possible.

Caching. We observe in the YCSB benchmark that J-PDT does not benefit from caching. Indeed, because data is accessed directly and only proxies are kept in the cache, increasing the cache ratio has almost no impact on read or update latencies.

Recovery. The performance of the recovery procedure is evaluated with a TPC-B like (transactional) workload. J-PFA recovers about 4.7x faster than FS and up to 8.6x faster with a recovery optimization possible for purely transactional workloads. Conversely, FS has to repopulate the 10% in-memory cache eargerly on recovery when J-PFA can only recreate proxies lazily with much less NVMM bandwidth usage.

References

[1] Implementation sources. URL https://github.com/jnvm-project/jnvm.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'11*. ACM, 2011.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the Symposium on Cloud Computing, SoCC'1*. ACM, 2010.

[4] J. S. George, M. Verma, R. Venkatasubramanian, and P. Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *Proceedings of the USENIX Annual Technical Conference, ATC'20*, 2020.

[5] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the international conference on Distributed Computing, DISC'16*. Springer, 2016.

[6] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'19*. ACM, 2019.

[7] A. Lefort, Y. Pipereau, K. Amponsem, P. Sutra, and G. Thomas. J-NVM: off-heap persistent objects in java. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'21*. ACM, 2021. URL https://www-public.imtbs-tsp.eu/~thomas_g/research/biblio/2021/lefort-sosp-jnvm.pdf.

[8] Linux. Direct access for files. URL https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[9] S. Liu, S. Mahar, B. Ray, and S. Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'21*. ACM, 2021.

[10] F. Marchioni and M. Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[11] Oracle. *Java Native Interface Specification*. Java SE 14 edition, 2020.

[12] Persistent Collections for Java. URL https://github.com/pmem/pcj.

[13] Persistent Memory Development Kit, 2018. URL https://pmem.io/pmdk.

[14] T. Shull, J. Huang, and J. Torrellas. AutoPersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI'19*. ACM, 2019.

[15] T. Shull, J. Huang, and J. Torrellas. QuickCheck: Using speculation to reduce the overhead of checks in nvm frameworks. In *Proceedings of the international conference on Virtual Execution Environments, VEE'19*. ACM, 2019.

[16] TPC Benchmark B. URL http://www.tpc.org/tpcb.

[17] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'18*. ACM, 2018.

[18] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'17*. ACM, 2017.