Designing a User-Friendly Java NVM Framework

Thomas Shull
University of Illinois at Urbana-Champaign
shull1@illinois.edu

Jian Huang
University of Illinois at Urbana-Champaign
jianh@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

1 Overview

Byte addressable, non-volatile memory (NVM) is emerging as a revolutionary technology that provides near-DRAM performance and scalable memory capacity. To facilitate its usability, many NVM programming models have been proposed. However, most of them require programmers to explicitly specify the data structures or objects that should reside in NVM. Such a limitation inevitably increases the burden on programmers, complicates development, and further introduces correctness and performance bugs.

To rectify this situation, we introduce a new user-friendly Java NVM framework. Because our framework is defined at a high level, it is intuitive, not prone to user bugs, and is flexible enough to allow language implementers to perform many optimizations while still adhering to its requirements.

We implement a persistent version of Memcached in our framework and find that its performance exceeds existing Java offerings and requires minimal program modifications.

2 Limitations of Existing Frameworks

Existing NVM frameworks ask programmers to make many concessions. To use them correctly, a programmer must either precisely mark all memory which should be durable and ensure that data is persisted properly, or refit their code to use specialized libraries where these markings have already been added. Programmers have many difficulties correctly adapting code to be compliant with existing NVM frameworks. Below we highlight the four main limitations of existing Java NVM frameworks.

Is Easy to Introduce Bugs. In current frameworks it is very easy to accidentally omit a marking needed for crash-consistency. Likewise, it is also very easy to be overly conservative and add too many durable markings, which will result in performance bugs.

Expose Low-Level Features To Programmer. Java programmers rely on the runtime to decide where to place objects, how to layout objects, and when an object's resources can be freed. Contrarily, existing NVM frameworks require the developer to consider low-level details of the Java implementation currently abstracted away by the runtime.

Do Not Perform Runtime Checks. The Java runtime inserts many runtime checks, such as array bounds checks, to catch programmer errors before they can harm its environment. Existing NVM frameworks let erroneous code execute, allowing data to potentially become corrupted.

Cannot Persistently Use Built-in Libraries. Java programmers rely on the rich set of libraries built into its distributions. Unfortunately, as these libraries do not have the requisite durable markings, programmers cannot use them when handling persistent objects in current NVM frameworks.

3 A New Java NVM Framework

We separate the explanation of our new Java NVM framework [3] into three subsections. We first describe the keywords used to label a durable program in our framework. Next, we describe the runtime invariants our framework must uphold. Finally, we explain our framework's benefits.

3.1 Framework Keywords

Our framework defines four keywords for the development of durable programs in Java: `@durable_root`, `begin_failure_atomic()`, `end_failure_atomic()`, and `@unrecoverable`. The `@durable_root` annotation can be used to mark any static field. When a field is marked with this annotation, it indicates that our framework will ensure all objects reachable from the field, i.e. its *transitive closure*, will also be stored in NVM.

We also add support for failure-atomic regions with the addition of two new methods: `begin_failure_atomic()` and `end_failure_atomic()`. Within a failure-atomic region, all stores to objects reachable from a `@durable_root` are not part of the recoverable program state until the region has ended. At `end_failure_atomic()`, all stores within the region are atomically added to the recoverable program state.

Finally, the `@unrecoverable` annotation indicates that the marked object does not need to be recoverable across a crash. This annotation can be used to limit the performance impact of persistency for objects which can be recovered or recreated via other means at application restart.

3.2 Execution Invariants

In our implementation of the framework, we modify the Java Virtual Machine (JVM) runtime to include a non-volatile heap portion, contain new runtime features, and have extra behaviors for several JVM bytecodes. Given a durably annotated program, our framework's JVM implementation must ensure correct persistent execution. To accomplish this, we establish three execution invariants.

Execution Invariant 1. *All objects reachable from the durable root set must be recoverable and in non-volatile memory.*

To maintain this invariant, the runtime may need to move objects to NVM when it detects they will become reachable from a `@durable_root`. Note that Java implementations already provide automatic memory management and the location and layouts of Java objects are transparent to the programmer; dynamically moving objects to non-volatile memory throughout execution fits well with the Java ethos.

Execution Invariant 2. *Outside of explicit failure-atomic regions, each store to memory reachable from a durable root is persistently completed before a new store to non-volatile memory can proceed.*

This invariant establishes that outside of failure-atomic regions our framework uses a sequentially persistent model [2]. To maintain this invariant, the runtime must insert cachline writebacks and fences after stores to objects reachable from a `@durable_root` to ensure they have persistenly completed before other stores to NVM occur.

Execution Invariant 3. *All stores to durable objects within a failure-atomic region appear to be performed atomically and persistently at the end of the region.*

This invariant establishes that our framework's failure-atomic regions use an epoch persistency model. To maintain this invariant, the runtime must track stores to objects reachable from a `@durable_root` within a failure-atomic region and ensure they only become a part of the crash-recoverable state atomically at the region's end.

3.3 Framework Benefits

Requires very few durable markings. Our framework only requires the user to identify durable roots and failure-atomic regions. As durable roots will often point to large data structures containing persistent data, having the framework ensure the transitive closure of the durable root set is persistent greatly reduces the amount of markings needed.
Can use existing libraries. Since our framework is integrated into the JVM and modifies the JVM bytecode semantics, users can use both existing built-in and third-party libraries to manipulate persistent objects. greatly improving programability.
Implementation has optimization flexibility. As our implementation is integrated into the JVM, we have much flexibility to dynamically optimize the internal behavior of our implementation in a way transparent to the user.

4 Evaluation

4.1 Evaluation Environment

We implement our framework within the Maxine Java Virtual Machine (JVM) [4] and modify both its baseline and optimizing compilers. We use a development platform with 128GB Intel Optane DC persistent memory modules and 324GB DDR4 DRAM. In all of our experiments, we reserve 20GB for both the volatile and non-volatile heap spaces.

To test a realistic use case of our framework, we implement a persistent version of Memcached using our framework. Specifically, we modify QuickCached, a pure Java implementation of Memcached to use persistent data structures internally for its key-value storage and compare with a the state-of-the-art solution.
IntelKV. To test against the state-of-the-art design, we use Intel's `pmemkv` library [1], using its kvtree storage engine along with its Java bindings as one of our backends, which we call *IntelKV*. This backend uses a hybrid b+ tree written in C++ using the PMDK library version 1.5. In their implementation only the leaf nodes are in persistent memory. Note

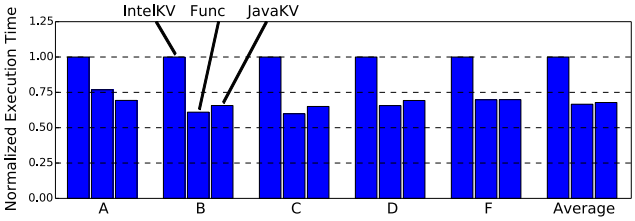

Figure 1. YCSB performance (lower is better).

that the *IntelKV* backend does not need the support of our framework and hence runs on an unmodified JVM.
Func and JavaKV. We implement two backends using our framework. *Func* uses the PCollection library while *JavaKV* uses the same hybrid B+ tree structure as *IntelKV*.

4.2 Framework Usability

A key benefit of our framework is that it requires the developer to add very few markings to their program to attain crash consistency. Using our framework, both the *JavaKV* and *Func* backends needed only four markings each: one for identifying the `@durable_root` and three for labeling `@unrecoverable` data. Clearly, our framework greatly reduces the amount of programmer involvement and the likelihood of introducing performance or correctness bugs.

4.3 Framework Performance

We show the execution time of running different persistent Memcached backends on the YCSB workloads in Figure 1. The performance is normalized to that of *IntelKV*. Overall, on average *Func* and *JavaKV* reduce the execution time by 33% and 32%, respectively, compared to *IntelKV*. This is because in *IntelKV* objects must be serialized to be stored in the C++ `pmemvk` library while in our framework persistent objects can be directly stored in our Java non-volatile heap portion. These results show that our framework and offers an enticing combination of performance and programmability.

Acknowledgments

We would like to thank Sanjay Kumar and Intel for allowing us early access to a system with Intel Optane DC persistent memory modules.

References

[1] [n. d.]. Pmemkv: Key/Value Datastore for Persistent Memory. https://github.com/pmem/pmemkv
[2] Steven Pelley et al. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276.
[3] Thomas Shull et al. 2018. Defining a High-level Programming Model for Emerging NVRAM Technologies. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. ACM, New York, NY, USA, Article 11, 7 pages.
[4] Christian Wimmer et al. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages.