Designing a User-Friendly Java NVM Framework

Thomas Shull, Jian Huang, Josep Torrellas
University of Illinois at Urbana-Champaign
March 12, 2019
NVMW’19 Session 13: Objects II
• Non-Volatile Memory (NVM) offers an enticing combination of performance, capacity, and persistency
• Programs will no longer have to serialize data out to secondary storage for durability
• Now have access to persistent memory at a byte-level granularity
Leveraging NVM to create persistent applications is tricky:

- Entire memory hierarchy is not durable
  - Processor caches are volatile
- Data must be written back from caches to achieve persistency
  - Perform combination of non-temporal stores, cacheline writebacks (CLWBs), and fences
- Software measures must be taken to ensure failure-atomicity for a collection of writes
  - Hardware only guarantees atomicity at cacheline level granularity
- To simply this process, frameworks for developing persistent applications have been introduced
Outline

1. Describe current NVM framework landscape
   - Current NVM framework features
   - Drawbacks of current frameworks

2. Present some of my work on creating a new high-level Java NVM framework
   - New NVM programming model
   - How we implement our model
   - Creating a high-performance model implementation
Current Techniques for Persistent Applications

- Manual – add explicit assembly instructions and system calls
- Industrial Libraries and Frameworks
  - Persistent Memory Development Kit (PMDK)
- Academic Frameworks
  - Atlas, NVL-C, Espresso, Mnemosyne, ARP, NVThreads, NV-Heaps, more
In current NVM frameworks the user must perform some combination of the following:

- Manually identifying persistent objects
- Wrapping stores needing persistent or transactional support
- Using previously persistently-marked data structures or libraries
Current NVM Frameworks – Drawbacks

Drawbacks of current NVM frameworks:

- Need many markings to identify persistent objects and direct persistent store mechanisms
- Easy to introduce bugs
  - Correctness bugs – markings are missing
  - Performance bugs – too many markings
- Difficult for the compiler to perform optimizations
  - Programmer’s intention is not visible to the compiler
Most NVM frameworks are designed for C/C++, not managed languages like Java, C#, Scala, etc.

- Cannot use existing built-in libraries
  - Built-ins do not contain necessary persistent markings
- Expose low-level features to programmer
  - Does not abstract away enough details
- Do not perform runtime checks
  - Catch problems before more damage is done
Contribution: Create a New NVM Framework

• Solution to existing shortcomings: Design a new NVM framework
• Focus on programmability first
  • Rely on compiler optimizations to get high performance
• Make framework tailored to managed-languages
  • Build upon their automatic memory management support and transparent object representation
Desirable Traits

Three important programmability goals for our NVM framework’s programming model:

1. Require minimal markings by programmer
2. Making libraries and other pre-existing codes persistent should be simple
3. Failure-atomic support should be provided and need only minimal markings
New NVM Framework Highlights

1. Require minimal markings by programmer
   - In our model the user must identify only a *durable root set* and failure-atomic regions
   - Durable Root Set: the set of objects directly referred to at recovery time
   - The runtime then ensures all objects reachable from the durable root set are in NVM
     - *Transitive closure* of the durable root set is placed in NVM automatically
     - Requires dynamically moving objects to NVM throughout execution
   - Durable roots should be program’s container/parent objects. (E.g. the *DATABASES* map object in H2’s *Engine.java*)
Making libraries and other pre-existing codes persistent should be simple

- Extend the semantics of existing JVM bytecodes
  - E.g. `putfield`, `aastore`, others
- Existing code can now be persistently handled if reachable from a durable root
Failure-atomic support should be provided and need only minimal markings

- Label only failure-atomic regions’ start and finish
- The runtime then ensures all persistent objects within the region are properly logged
Our programming model requires the framework to:

1. Dynamically detect and monitor the transitive closure of durable roots
   - Must ensure everything reachable from a durable root is in NVM

2. Ensure stores to persistent objects are performed correctly
   - Behavior is dependent on whether the store is in an failure-atomic region or not
Dynamically Moving Objects to NVM

(a) Initial Heap State

Volatile Memory

Non-Volatile Memory

A
B
C
D
E
F
G
@durable_root
Dynamically Moving Objects to NVM

(a) Initial Heap State

(b) Model Violation

Shull et al. Designing a User-Friendly Java NVM Framework
Dynamically Moving Objects to NVM

(a) Initial Heap State

(b) Runtime Maintaining Correct Heap State
Ensuring Persistent Stores

Two cases: outside and inside failure-atomic regions

• Outside failure-atomic region - enforce sequential persist order
  • after each store perform CLWB and FENCE
• Inside failure-atomic region - enforce atomic commit at end
  • Epoch Persistency – stores within region can be reordered
  • Logging should be performed to create appearance of atomicity
Recovery Procedure

At recovery time we expect the program to:

1. Check if data from previous execution exists
2. Load previous data if available
   - Checking and Loading is performed by interacting with @durable_roots
3. Jump to proper execution point
   - E.g. Server-side event loop
Implementing New Model

- Model requires many guarded actions before accesses
  - Storing to `@durable_root`?
  - Storing to an object reachable from a `@durable_root`?
  - In a failure-atomic region?

Solution:

1. Add extra object header word to contain persistent state and metadata
2. Extend the semantics of several JVM bytecodes to perform the necessary checks and guarded actions
   - More details in papers
Modified store operation

1: **procedure** STOREFIELD(Object holder, Field f, Value v)

2: writeField(holder, f, v)

3: **end procedure**
Modified store operation

1: **procedure** STOREFIELD(Object holder, Field f, Value v)  
   
   2: **if** isPersistent(holder) **then**
   3:     *Move value to NVM if necessary*
   4:     *Log (object, field, value) tuple if in failure-atomic region*
   5: **end if**
   
   6: **end procedure**
Our implementation collects profiling information to limit the performance overhead:

- Limit check overhead
  - Predict whether a given object access site usually handles persistent or volatile objects
  - Can reduce check overhead for sites predicted to handle volatile objects

- Preallocate objects in NVM
  - Predict whether a given allocation site usually allocates objects which become reachable from a @durable_root
  - Can originally allocate these objects in NVM to limit object movement
Evaluation Environment

- Modify Maxine 2.0.5
  - Research JVM originally developed by Oracle
- Run on system containing two 24-core Intel® second generation Xeon® Scalable processors (codenamed Cascade Lake)
- System has 128GB Intel Optane DC persistent memory modules
Performance Results

- IntelKV: Intel’s pmemkv library (kvtree3 engine) using its Java API
- JavaKV: Implementing same data structure used in pmemkv in our framework
- Use Quickcached (Java KV-Store) and run YCSB

A B C D F Average
Normalized Execution Time

IntelKV JavaKV

- JavaKV reduces execution time by 32%
- Have more (& fairer) results and comparisons in papers

Shull et al.  Designing a User-Friendly Java NVM Framework
Papers About Our Model & Framework

Motivating the need for new Java-specific NVM programming model – **ManLang’18**: Defining a High-level Programming Model for Emerging NVRAM Technologies

How to limit our framework’s runtime check overhead on volatile objects – **VEE’19**: QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks

How our framework dynamically moves objects between DRAM and NVM – **PLDI’19**: AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability
Motivating the need for new Java-specific NVM programming model – ManLang’18: Defining a High-level Programming Model for Emerging NVRAM Technologies

How to limit our framework’s runtime check overhead on volatile objects – VEE’19: QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks

How our framework dynamically moves objects between DRAM and NVM – PLDI’19: AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability

Questions?