Defining a High-Level Programming Model for Emerging NVRAM Technologies

Thomas Shull, Jian Huang, Josep Torrellas
University of Illinois at Urbana-Champaign
September 13, 2018
• Introduce Non-Volatile Random Access Memory (NVRAM)
  • Describe features of existing NVRAM frameworks
  • Explain the limitations of existing NVRAM frameworks
• Propose new NVRAM programming model for managed languages
• Partially apply model to Java and show initial results
• Describe Future Work
Non-Volatile Random-Access Memory (NVRAM)

- Non-Volatile Random-Access Memory (NVRAM) is soon to become available
  - Byte-addressable durable storage
  - Performance is within an order of magnitude of DRAM
  - Capacity is much higher than DRAM
  - Much faster than traditional block-based storage offerings
- Enterprise Applications will use NVRAM
  - Better performance
  - Removes separation between application working memory and storage
Using NVRAM Today

• Measures still must be taken to ensure crash-consistency
  • Volatile data must be written back from caches
  • Fences must be inserted to ensure a specific ordering of persistence stores
• Current frameworks expect the programmer to manage all operations needed for crash consistency
  • Allocation into NVRAM
  • Cache writebacks
  • Fences
Limitations of Existing NVRAM Frameworks

- Low-level model does not match the level of abstraction expected by managed language users
- Easy to introduce bugs
- At odds with automatic memory management support
- Existing libraries must be modified to be crash consistent
- Difficult for the compiler to perform optimizations
Goals for New NVRAM Programming Model

1. Require minimal markings by programmer
2. Libraries and other pre-existing codes should not need to be changed to work correctly in a durable program
3. Failure-atomic support should be provided and need only minimal markings

• More details in paper
Requirements for NVRAM Programming Model

• Specific enough for consistent behavior
• General enough to give implementors flexibility

1. Only objects directly addressed at recovery time should require markings
   • We call these objects the *durable root set*
   • The runtime should dynamically ensure all objects reachable from the durable root set are in NVRAM
   • May require moving objects at runtime to NVRAM

2. Failure-atomic regions should only need markings to delimit the code region
   • Runtime should automatically provide needed logging

3. Outside of failure-atomic regions, stores to recoverable memory should complete in sequential order
   • More details in paper
Applying Model to Java

- We only describe the beginning of applying our model’s requirements to Java
  - Identifying Durable Roots
  - Modifying Behavior of JVM Bytecodes
- Identifying Durable Roots
  - Annotate select static fields with new @durable_root annotation
  - All objects reachable from this annotation must be durable
  - All stores to objects reachable from this annotation must be performed durably and in sequential order
Modifying Behavior of JVM Bytecodes

- **putfield**(O, F, V)
  - Original: Store value V into field F of object O

Modified → when O is durable:

1. Ensure V and its transitive closure is durable, moving objects to NVRAM if necessary
2. Store V into O.F
3. Write back store to NVRAM
4. Insert fence after writeback to ensure sequential ordering

- **putstatic** and `{a,b,c,d,f,i,l,s}astore` are also modified in similar ways
- More details in paper
Evaluation

- Intention of evaluation is to present an initial approximation of our model’s overhead in Java
- Implement changes on Maxine VM version 2.0.5
  - Modify only baseline compiler (T1X)
  - Disable use of optimizing compiler
- Do not use NVRAM, but do insert all necessary cache writebacks and fences
  - Writebacks and fences are the main overhead of NVRAM
- Three Configurations
  - *Baseline* (*B*): unmodified JVM
  - *WChecks* (*C*): modified JVM, but no durable roots
  - *AllDurable* (*A*): modified JVM and all objects are durable
Initial Results

- Run DaCapo Benchmarks
- \textit{WChecks} does not have significant overheads
- \textit{AllDurable} is over twice as slow as \textit{Baseline}
  - Gap may be wider in reality
Future Work

- Fully define NVRAM programming model extension for Java
  - If also defined for JVM, then can be incorporated into many languages
- Determine the runtime support necessary to dynamically move objects between volatile and non-volatile memory
- Determine the runtime support necessary for failure-atomic regions
- Introduce compiler optimizations to minimize the overheads introduced by our model
Conclusion

Paper Contributions:

- Describes how existing NVRAM frameworks are not a good match for managed languages
- Describes what goals a NVRAM programming model for managed languages should meet
- Proposes a new NVRAM programming model which is intuitive for programmers and a good fit for managed languages
- Explains how the initial steps of how Java can be modified to implement our NVRAM programming model

- I have a poster in the following session