AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability

Thomas Shull, Jian Huang, Josep Torrellas
University of Illinois at Urbana-Champaign
June 24, 2019
PLDI’19 – Memory Management Session
Emerging Non-Volatile Memory (NVM) devices offer an enticing combination of performance, capacity, and persistency

Applications no longer have to serialize data out to secondary storage for durability

NVM is byte-addressable
Real NVM Systems
Creating persistent applications in NVM is tricky:

- Volatile processor caches are in the memory hierarchy
- Data must be written back from caches to achieve persistency
  - CLWB (cacheline writeback) - force writeback to NVM
  - SFENCE (store fence) - block until writebacks have completed
- Software logging is needed to ensure the failure-atomicity of a collection of writes
  - Hardware only guarantees atomicity at a cacheline granularity
- Solution: use frameworks to hide much of the complexity
1 Describe current NVM framework landscape
   • Current features
   • Drawbacks

2 Present AutoPersist: an Easy-To-Use Java NVM framework
   • Programming model
   • Programming examples
   • Implementation details

3 Comparison of AutoPersist against existing Java NVM frameworks
Currently, the user must perform some of the following actions:

1. Manually identifying persistent objects
2. Add CLWBs and SFENCEs after stores
3. Add logging for failure-atomic regions
4. Using previously marked data structures or libraries
struct DurableList{
    int element; DurableList* next;
}

DurableList* prepend(DurableList* old_head, int element){
    DurableList* new_head = new DurableList();
    new_head->element = element;

    new_head->next = old_head;

    return new_head;
}
struct DurableList{
    int element; DurableList* next;
}

DurableList* prepend(DurableList* old_head, int element){
    DurableList* new_head = Durable_New DurableList();
    new_head->element = element;
    Clwb(&(new_head->element));
    new_head->next = old_head;
    Clwb(&(new_head->next));
    Sfence();
    return new_head;
}
Current NVM Frameworks – Takeaways

Drawbacks of current NVM frameworks:

- Need to identify objects which should be allocated in NVM and also insert CLWBs and SFENCEs
- 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
Contribution: AutoPersist

- Solution to existing shortcomings: design AutoPersist – a new \textit{easy-to-use} NVM framework
- Focus on programmability first
  - Rely on compiler optimizations to get high performance
- Make framework tailored to managed languages
  - Effectively leverage managed language runtime features
- AutoPersist significantly outperforms existing Java NVM offerings
In the AutoPersist programming model, the user must identify only a **durable root set** and **failure-atomic regions**

- **Durable root set:** objects directly referred to at recovery time
- User annotates desired durable roots with the **@durable_root** marking
- **Failure-atomic regions:** regions which should be atomic from a crash-consistency perspective
AutoPersist’s runtime automatically:

1. Dynamically detects and monitors the *transitive closure* of durable roots
   - Ensures everything reachable from a durable root is in NVM
   - Dynamically moves objects to NVM throughout execution

2. Ensure stores to persistent objects are performed correctly
   - Automatically inserts the needed logging, CLWBs, and SFENCEs
@durable_root

Dynamically Moving Objects to NVM
Dynamically Moving Objects to NVM

@durable_root

| INT |
| INT |
| INT |
| INT |
Dynamically Moving Objects to NVM

@durable_root

@durable_root

OBJ

OBJ

OBJ

OBJ

OBJ
Dynamically Moving Objects to NVM

@durable_root

[Diagram of dynamically moving objects to NVM]

Shull et al. AutoPersist: An Easy-To-Use Java NVM Framework
Dynamically Moving Objects to NVM

@durable_root

@durable_root

OBJ

OBJ

OBJ

OBJ
Dynamically Moving Objects to NVM

@durable_root
Dynamically Moving Objects to NVM
class DurableList{
    private int element; private DurableList next;

    public static DurableList head;

    public static void prepend(int element){
        DurableList newHead = new DurableList();
        newHead.element = element;
        newHead.next = head;
        head = newHead;
    }
}
class DurableList{
    private int element; private DurableList next;
    @durable_root
    public static DurableList head;

    public static void prepend(int element){
        DurableList newHead = new DurableList();
        newHead.element = element;
        newHead.next = head;
        head = newHead;
    }
}
Dynamically Moving Objects to NVM

Volatile Memory  Non-Volatile Memory

1
2
@durable_root
3

Shull et al. AutoPersist: An Easy-To-Use Java NVM Framework
Dynamically Moving Objects to NVM

Volatile Memory

Non-Volatile Memory

@durable_root

Shull et al.  AutoPersist: An Easy-To-Use Java NVM Framework
Dynamically Moving Objects to NVM

Volatile Memory

Non-Volatile Memory

2

@durable_root

1

4

3

Shull et al. AutoPersist: An Easy-To-Use Java NVM Framework
Dynamically Moving Objects to NVM
Dynamically Moving Objects to NVM

Volatile Memory

Non-Volatile Memory

2

3

@durable_root

1

4

@durable_root

Shull et al. AutoPersist: An Easy-To-Use Java NVM Framework
Ensuring Stores are Made Persistent

AutoPersist Persistency Model:

- **Epoch Persistency**
  - Used within failure-atomic regions
  - Undo logging is performed for atomicity
- **Sequential Persistency**
  - Used everywhere else
  - Inserts CLWBs and SFENCEs after each store
**Elements in persistent list**

DurableList a = ...  
DurableList b = ...  
DurableList c = ...  

a.element = 10;

c.element = 11;

b.element = 12;
**Elements in persistent list**

DurableList a = ...
DurableList b = ...
DurableList c = ...

a.element = 10;
\textit{Clwb(a.element); Sfence}

c.element = 11;
\textit{Clwb(c.element); Sfence}

b.element = 12;
\textit{Clwb(b.element); Sfence}
**Elements in persistent list**

DurableList a = ... 
DurableList b = ... 
DurableList c = ... 

a.element = 10;  
Clwb(a.element); Sfence

c.element = 11;  
Clwb(c.element); Sfence

b.element = 12;  
Clwb(b.element); Sfence
**Persistent Int Array**

**Persistent Int Array**

```java
int[] parray = ...
```

**Removing the front of array**

```java
for (int i = 1; i < parray.length; i++){
    parray[i-1] = parray[i];
}
```
**Persistent Int Array**

```java
int[] parray = ...
```

**removing the front of array**

Create Undo Log

```java
for (int i = 1; i < parray.length; i++) {
    log original value and location
    parray[i-1] = parray[i];
    writeback parray index
}
```

Clear Undo Log

Shull et al. AutoPersist: An Easy-To-Use Java NVM Framework
**Persistent Int Array**

```java
int[] parray = ...;
```

**removing the front of array**

```java
BeginFailureAtomic
Create Undo Log
for (int i = 1; i < parray.length; i++) {
    Log original value and location
    parray[i-1] = parray[i];
    Writeback parray index
}
Clear Undo Log
EndFailureAtomic
```
Modifying JVM Runtime

To provide its runtime support, AutoPersist:

1. Extends the semantics of several JVM bytecodes to perform AutoPersist’s runtime actions
2. Adds an extra object header word to contain persistent state metadata
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
- Compare against Espresso [Wu et al. ASPLOS’18]
  - Persistent objects must be explicitly allocated in NVM
  - CLWBs and SFENCEs must be manually inserted into the code
Evaluation Applications

- **KVStore: Key-Value Store Server**
  - Modify implementation to use AutoPersist-annotated data structures
  - Test with two data structures
    - **JavaKV**: B\(^+\) Tree implementation
    - **Func**: Functional Hashmap
  - Evaluate KVStore performance running YCSB workloads
- **Kernels: simple data structures**
  - Collection of different arraylist and linked list implementations
  - Run random sequence of get, set, update, and delete operations on each data structure
Comparison of # Marks Needed

<table>
<thead>
<tr>
<th>Framework</th>
<th>KVStore Applications</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Func</td>
<td>JavaKV</td>
<td>Total</td>
</tr>
<tr>
<td>AutoPersist</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Espresso</td>
<td>55</td>
<td>45</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Framework</th>
<th>MArray</th>
<th>MList</th>
<th>FARArray</th>
<th>FArray</th>
<th>FList</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoPersist</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Espresso</td>
<td>49</td>
<td>48</td>
<td>63</td>
<td>47</td>
<td>14</td>
<td>321</td>
</tr>
</tbody>
</table>
• On average, AutoPersist is 31% and 28% faster than Espresso for the Func and JavaKV implementations.

• Significant performance improvements for update-heavy workloads (A, D, F)
Explaining AutoPersist’s Performance Advantages

- In Java, the machine-level layout of objects is hidden from the programmer
- Makes it expensive to persist entire objects in Espresso
  - Must conservatively insert CLWBs to each field name
- In AutoPersist, when an object becomes reachable from a \texttt{@durable} \texttt{root}, then the object is made persistent efficiently
  - Because AutoPersist knows the object’s internal layout, only needs to insert one CLWB per cacheline
- **Takeaway**: AutoPersist is able to limit the number of CLWBs needed when initially persisting objects
On average, AutoPersist is 59% faster than Espresso.

MArray, FArray, and FList create many new objects for insertions.

- Highlights AutoPersist’s efficient object persisting mechanism.
Also in the Paper

- More details about AutoPersist’s programming model (Introspection API, Recovery API)
- In depth explanation of how to amend the JVM bytecode semantics
- Algorithms for how to concurrently move objects to NVM in a thread-safe manner
- Profiling techniques to improve AutoPersist’s performance
- Additional evaluation – (H2, Intel’s pmemkv)
AutoPersist Summary

- NVM Challenge: difficult to program
- Solution: introduce AutoPersist
  - Needs minimal markings
  - Leverages the JVM runtime to automatically perform much of the complicated actions associated with NVM
  - Outperforms existing Java NVM frameworks by minimizing # of CLWBs when persisting complete objects