P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks

Apostolos Kokolis, Thomas Shull, Jian Huang and Josep Torrellas
University of Illinois at Urbana-Champaign

International Symposium on Microarchitecture (MICRO) 2020
Non-Volatile Memory (NVM)

- NVM offers appealing characteristics
  - Capacity
  - Performance
  - Persistence
- Applications have direct access to NVM at byte-granularity
- NVM is already available
P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks

Programming NVM

- NVM success depends on its programming frameworks
- Frameworks should be **efficient** and **easy to use**
- NVM programming challenges
  - API presented to the programmer
  - Identifying persistent objects
  - Performing updates to NVM
Most NVM Frameworks are Hard to Use

- Rely on user involvement
  - Identify persistent data structures
  - Add `clwb` and `sfence`
  - Add `Logging` for transactions
  - Use libraries with NVM support

- Efficient 😊
- Increases programming effort 😞
- Code is prone to bugs and inefficiencies 😞
- Hard to apply to legacy code 😞

```python
function foo() {
    f = persistent F()
    f.element = value
    clwb f
    sfence
}
```
Programmable Framework: Persistence by Reachability

- Programmer only identifies entry points to persistent structures (durable roots)
- The compiler/runtime are responsible to persist necessary objects dynamically

![Diagram showing DRAM and NVM with nodes A, B, C, D, E, F and a durable root]

P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks
Programmable Framework: Persistence by Reachability

- Programmer only identifies entry points to persistent structures (**durable roots**)
- The compiler/runtime are responsible to persist necessary objects dynamically
- Simplify programming 😊
- Compatible with legacy code 😊
- Only move data to NVM when needed 😊

The runtime introduces overheads to check object state during execution 😞
Contribution: P-INSPECT

- Transpareently **accelerates** Persistence by Reachability (P-by-R) frameworks using simple hardware
- Introduces hardware support to minimize software checks and to speed up persistent writes
- In a state-of-the-art P-by-R framework, P-INSPECT decreases the executed instructions by 26% and the execution time by 16%
AutoPersist: A Persistence by Reachability Framework[1]

- $F$ is a durable root
- $F$ wants to point to $A$

[1]: Shull et al., “AutoPersist: An Easy-To-Use Java NVM Framework based on Reachability”, PLDI’19
AutoPersist: A Persistence by Reachability Framework

- Queue bit: object & its *transitive closure* are being processed

- F is a durable root
- F wants to point to A
- A and its transitive closure must become persistent

Runtime:
- Creates NVM copy of A
- Sets its Queued bit
AutoPersist: A Persistence by Reachability Framework

- Queued bit: object & its transitive closure are being processed
- Forwarding bit: object points to its NVM copy

DRAM

NVM

- F is a durable root
- F wants to point to A
- A and its transitive closure must become persistent
- Runtime:
  - Creates NVM copy of A
  - Sets its Queued bit
  - Sets Forwarding bit of DRAM obj. A
  - Creates forwarding pointer for DRAM A
AutoPersist: A Persistence by Reachability Framework

- $F$ is a durable root
- $F$ wants to point to $A$
- $A$ and its transitive closure must become persistent
- Runtime:
  - Creates NVM copy of $A$
  - Sets its Queued bit
  - Sets Forwarding bit of DRAM obj. $A$
  - Creates forwarding pointer for DRAM $A$
  - Repeats for transitive closure of $A$
AutoPersist: A Persistence by Reachability Framework

- $F$ is a durable root
- $F$ wants to point to $A$
- $A$ and its transitive closure must become persistent
- **Runtime**:  
  - Creates NVM copy of $A$
  - Sets its Queued bit
  - Sets Forwarding bit of DRAM obj. $A$
  - Creates forwarding pointer for DRAM $A$
  - Repeats for transitive closure of $A$
Required Runtime Software Checks in AutoPersist

Runtime must determine if:
- Object is or needs to be persistent
- Object is forwarding
- Object’s trans. closure is processing
- Logging is required

**Load**: \( \text{reg} = \text{holder}_{\text{obj}} \)

**Store**: \( \text{holder}_{\text{obj}}.f = \text{value}_{\text{obj}} \)

<table>
<thead>
<tr>
<th>Runtime Checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holder’s &amp; value’s memory locations</td>
</tr>
<tr>
<td>Holder’s &amp; value’s forwarding bits</td>
</tr>
<tr>
<td>Value’s queued bit</td>
</tr>
<tr>
<td>If execution is inside a Xaction</td>
</tr>
</tbody>
</table>
Motivation: AutoPersist has High Overhead

![Bar charts showing execution time for different operations]

- Application Ops
- Runtime checks
- Persistent Writes
- Runtime actions
Motivation: AutoPersist has High Overhead

Runtime Checks & Persistent Writes contribute 10% - 60% of the total execution time.

Execution time

<table>
<thead>
<tr>
<th>Application Ops</th>
<th>Runtime checks</th>
<th>Persistent Writes</th>
<th>Runtime actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTree-A</td>
<td>pTree-B</td>
<td>pmap-A</td>
<td>pmap-B</td>
</tr>
<tr>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
</tr>
<tr>
<td>Htree-A</td>
<td>Htree-B</td>
<td>hashmap-A</td>
<td>hashmap-B</td>
</tr>
<tr>
<td>0.4</td>
<td>0.6</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>pTree-4</td>
<td>pTree-8</td>
<td>pmap-4</td>
<td>pmap-8</td>
</tr>
<tr>
<td>0.6</td>
<td>0.8</td>
<td>1.0</td>
<td>1.2</td>
</tr>
<tr>
<td>Htree-4</td>
<td>Htree-8</td>
<td>hashmap-4</td>
<td>hashmap-8</td>
</tr>
<tr>
<td>0.8</td>
<td>1.0</td>
<td>1.2</td>
<td>1.4</td>
</tr>
<tr>
<td>pTree-12</td>
<td>pTree-16</td>
<td>pmap-12</td>
<td>pmap-16</td>
</tr>
<tr>
<td>1.0</td>
<td>1.2</td>
<td>1.4</td>
<td>1.6</td>
</tr>
<tr>
<td>Htree-12</td>
<td>Htree-16</td>
<td>hashmap-12</td>
<td>hashmap-16</td>
</tr>
<tr>
<td>1.2</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.0</td>
<td>1.2</td>
<td>1.4</td>
</tr>
</tbody>
</table>
**P-INSPECT: Design**

- **Idea:** Introduce hardware to perform the checks

**Before:** SW reads object’s header to find if object is in NVM  
**Now:** HW checks the Virtual Address of the object

<table>
<thead>
<tr>
<th>Runtime Checks</th>
<th>Hardware Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holder’s &amp; value’s memory location</td>
<td>Virtual Address</td>
</tr>
<tr>
<td>Holder’s &amp; value’s forwarding bits</td>
<td></td>
</tr>
<tr>
<td>Value’s queued bit</td>
<td></td>
</tr>
<tr>
<td>Execution inside a Xaction</td>
<td></td>
</tr>
</tbody>
</table>
P-INSPECT: Design

- **Idea**: Introduce hardware to perform the checks

Bloom filter (BF) is a cheap HW structure to check for membership
Objects with *Forwarding* bit set are placed in the *Forwarding* BF
Objects with *Queued* bit set are placed in the *Trans. Clos.* BF
On an object access, HW checks for BF membership

<table>
<thead>
<tr>
<th>Runtime Checks</th>
<th>Hardware Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holder’s &amp; value’s memory location</td>
<td>Virtual Address</td>
</tr>
<tr>
<td>Holder’s &amp; value’s forwarding bits</td>
<td><em>Forwarding</em> bloom filter</td>
</tr>
<tr>
<td>Value’s queued bit</td>
<td><em>Transitive Closure</em> bloom filter</td>
</tr>
<tr>
<td>Execution inside a Xaction</td>
<td></td>
</tr>
</tbody>
</table>
P-INSPECT: Design

- **Idea:** Introduce hardware to perform the checks

<table>
<thead>
<tr>
<th>Runtime Checks</th>
<th>Hardware Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holder’s &amp; value’s memory location</td>
<td>Virtual Address</td>
</tr>
<tr>
<td>Holder’s &amp; value’s forwarding bits</td>
<td><em>Forwarding</em> bloom filter</td>
</tr>
<tr>
<td>Value’s queued bit</td>
<td><em>Transitive Closure</em> bloom filter</td>
</tr>
<tr>
<td>Execution inside a Xaction</td>
<td>Control register bit</td>
</tr>
</tbody>
</table>
P-INSPECT: Operation of the Bloom filters

- Software adds addresses of objects to the BFs
P-INSPECT: Operation of the Bloom filters

- Software adds addresses of objects to the BFs

Diagram:
- Core
  - Object Address
  - Hash
- L1 Cache Controller
  - Forwarding BF: 010010000
  - Transitive Closure BF: 00010101
P-INSPECT: Operation of the Bloom filters

- On a read/write access to an object, HW checks for BF membership (and other checks)
  - If not present and all other checks pass: Read/write completes
P-INSPECT: Operation of the Bloom filters

- On a read/write access to an object, HW checks for BF membership (and other checks)
  - If not present and all other checks pass: Read/write completes
  - Else: Invoke SW handler to make the state consistent and read/write

function handler()
  
  readObjHeader();
  determinePersistence();
  checkMakeRecoverable();
  checkLogging();
  performOperation();
  
}
Clearing the *Forwarding* Bloom filter

- A special thread regularly
  - Traverses the volatile heap & fixes the *forwarding* pointers
  - After that it clears the *Forwarding* bloom filter
- At Garbage Collection time: *Forwarding* objects are deleted
Persistent writes add overhead to application execution
Worst case: Need 2 round trips to NVM
P-INSPECT: Speeds up Persistent Writes

**Idea:** Combine \textit{write}+\textit{clwb}+\textit{sfence} in one transaction

- **Operation**
  - Write proceeds to NVM
  - NVM is updated
  - Updated cache line is brought into the cache in Exclusive state

**Faster execution: Only 1 round trip to NVM**
Evaluation Methodology

- Full system simulations with Simics + SST

- Pintool simulations for long execution intervals

- Applications
  - Kernels
  - Key Value Store with many backends running YCSB

- AutoPersist JVM runtime (modified Maxine 2.0.5) to communicate the runtime instructions to the simulators
Reduction in Executed Instructions

Baseline: AutoPersist
P-INSPECT--: no persistent write optimization
P-INSPECT
Ideal-SW: no P-by-R overheads and no persistent write optimization

P-INSPECT greatly reduces number of instructions
Reduction in Execution Time

Baseline: AutoPersist
P-INSPECT--: no persistent write optimization
P-INSPECT
Ideal-SW: no P-by-R overheads and no persistent write optimization

---

P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks
Reduction in Execution Time

- **Baseline**: AutoPersist
- **P-INSPECT--**: no persistent write optimization
- **P-INSPECT**: no P-by-R overheads and no persistent write optimization
- **Ideal-SW**: no P-by-R overheads and no persistent write optimization

- **P-INSPECT** reduces execution time by 32% and 16%
- Persistent write optimization improves performance

P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks
Also in the paper…

- More details on P-INSPECT
  - Operation and instructions
  - Software handlers
  - Bloom filter operations

- Deeper Evaluation
P-INSPECT uses hardware to speed up programmable Persistence by Reachability (P-by-R) frameworks

- Keeps *P-by-R programming* advantages
- Performs checks with simple hardware and accelerates persistent writes
- Achieves performance similar to an ideal runtime system with no *P-by-R*
P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks

Apostolos Kokolis, Thomas Shull, Jian Huang and Josep Torrellas
University of Illinois at Urbana-Champaign
{kokolis2, shull1, jianh, torrella}@Illinois.edu
International Symposium on Microarchitecture (MICRO) 2020

THANK YOU!!
Non-Blocking Pointer Update Thread

- The Pointer Update Thread (PUT) traverses the live objects of the volatile heap and updates pointers to FWD objects.
- PUT operation is non-blocking.
- We keep 2 FWD bloomfilters:
  - Active
  - Inactive
- Both are checked on a special load/store.
- Insert to active.
- Clear inactive when PUT completes.
Pointer Update Thread Overhead
## Pointer Update Thread Overhead

<table>
<thead>
<tr>
<th>Applic.</th>
<th># Inst. between PUT calls (mill.)</th>
<th># FWD checks per insert (thous.)</th>
<th>Avg. FWD occup.</th>
<th>PUT instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>26,326</td>
<td>3,006.0</td>
<td>14.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>LinkedList</td>
<td>3,175</td>
<td>163.5</td>
<td>15.9%</td>
<td>0.2%</td>
</tr>
<tr>
<td>ArrayListX</td>
<td>43,778</td>
<td>4,937.4</td>
<td>15.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td>HashMap</td>
<td>928</td>
<td>134.8</td>
<td>15.9%</td>
<td>1.6%</td>
</tr>
<tr>
<td>BTree</td>
<td>237</td>
<td>10.4</td>
<td>15.9%</td>
<td>6.5%</td>
</tr>
<tr>
<td>BPlusTree</td>
<td>45,367</td>
<td>3,201.0</td>
<td>15.9%</td>
<td>0.0%</td>
</tr>
<tr>
<td>pTree-D</td>
<td>478</td>
<td>22.4</td>
<td>16.0%</td>
<td>3.6%</td>
</tr>
<tr>
<td>HpTree-D</td>
<td>426</td>
<td>11.1</td>
<td>15.8%</td>
<td>3.8%</td>
</tr>
<tr>
<td>hashmap-D</td>
<td>969</td>
<td>85.2</td>
<td>16.1%</td>
<td>1.8%</td>
</tr>
<tr>
<td>pmap-D</td>
<td>92</td>
<td>1.9</td>
<td>15.9%</td>
<td>18.4%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>12,177</strong></td>
<td><strong>1,157.4</strong></td>
<td><strong>15.8%</strong></td>
<td><strong>3.6%</strong></td>
</tr>
</tbody>
</table>
Coherent Bloomfilters

- Bloomfilters (BFs) span multiple cache lines
- To read the BF a core must first fetch the cache lines to the L1 cache controller
- For a write access to the BF one of the lines of the BF acts as the Seed
  - The Seed is requested in Exclusive state
  - Once it attains the Seed locks it and proceeds to obtain the rest of the cache lines
  - When all the lines are present the write operation is performed
Speeding-up persistent writes

- Interaction with coherence protocol
  - If the cache line is in one of the caches it is incorporated in the write
  - Once it reaches the directory, the directory is locked. The directory indicates if the line is in another’s core cache
  - The NVM ACKs the directory. Directory marks the line as Exclusive for the originating persistent-write core
New instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>What it Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkStoreBoth [Ha],Va</td>
<td>Performs checks, then Mem[Ha] = Va</td>
</tr>
<tr>
<td>checkStoreH [Ha],value</td>
<td>Performs checks, then Mem[Ha] = value</td>
</tr>
<tr>
<td>checkLoad [Ha],dest</td>
<td>Performs checks, then dest = Mem[Ha]</td>
</tr>
<tr>
<td>insertBF&lt;sub&gt;FWD&lt;/sub&gt; Addr</td>
<td>Inserts Addr in the FWD bloom filter</td>
</tr>
<tr>
<td>insertBF&lt;sub&gt;TRANS&lt;/sub&gt; Addr</td>
<td>Inserts Addr in the TRANS bloom filter</td>
</tr>
<tr>
<td>clearBF&lt;sub&gt;FWD&lt;/sub&gt;</td>
<td>Clears the FWD bloom filter</td>
</tr>
<tr>
<td>clearBF&lt;sub&gt;TRANS&lt;/sub&gt;</td>
<td>Clears the TRANS bloom filter</td>
</tr>
</tbody>
</table>

A possible implementation in x86 is to use the existing store and load instructions with a prefix.
Complete list of hardware checks

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Where is Base(Ha)?</strong></td>
<td><strong>Base(Ha) in FWD?</strong></td>
</tr>
<tr>
<td>NVM</td>
<td>-</td>
</tr>
<tr>
<td>DRAM</td>
<td>false</td>
</tr>
<tr>
<td>DRAM</td>
<td>false</td>
</tr>
<tr>
<td>DRAM</td>
<td>true</td>
</tr>
<tr>
<td>NVM</td>
<td>-</td>
</tr>
<tr>
<td>NVM</td>
<td>-</td>
</tr>
</tbody>
</table>