Architectural Support for Atomic Durability in Non-Volatile Memory

Arpit Joshi, Vijay Nagarajan, Stratis Viglas, Marcelo Cintra

NVMW 2018
Summary

• Non-Volatile Memory (NVM) - on the memory bus
  • enables in-memory persistent data structures
• Persistent data structures require an atomic durability primitive to ensure crash consistency
• Logging is a technique to provide atomic durability
• ATOM: hardware support for atomic durability by way of undo logging
Atomic Durability

- **All or nothing** persists: think transactions (ACID)
Atomic Durability

- **All or nothing** persists: think transactions (ACID)
Atomic Durability

- **All or nothing** persists: think transactions (ACID)

<table>
<thead>
<tr>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 100</td>
</tr>
<tr>
<td>B 100</td>
</tr>
</tbody>
</table>

Atomic_Begin
A = A - 50
B = B + 50
Atomic_End
Atomic Durability

- **All or nothing** persists: think transactions (ACID)

```
Atomic_Begin
A = A - 50
B = B + 50
Atomic_End
```

Initial State

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
</tr>
</tbody>
</table>

Final State

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
</tr>
</tbody>
</table>
Atomic Durability

- **All or nothing** persists: think transactions (ACID)

**Initial State**

<table>
<thead>
<tr>
<th></th>
<th>A 100</th>
<th>B 100</th>
</tr>
</thead>
</table>

**Atomic_Begin**

A = A - 50
B = B + 50

**Atomic_End**

**Final State**

<table>
<thead>
<tr>
<th></th>
<th>A 100</th>
<th>B 100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A 50</th>
<th>B 150</th>
</tr>
</thead>
</table>
Atomic Durability

- **All or nothing** persists: think transactions (ACID)

**Atomic_Begin**

A = A - 50
B = B + 50

**Atomic_End**

Initial State

| A 100 | B 100 |

Final State

| A 100 | B 100 |

| A 50   | B 150 |

Final State

| A 50   | B 100 |

| A 100 | B 150 |
Atomic Durability

- **All or nothing** persists: think transactions (ACID)

**Initial State**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
</tr>
</tbody>
</table>

**Atomic_Begin**

A = A - 50
B = B + 50

**Atomic_End**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>150</td>
</tr>
</tbody>
</table>

**Final State**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
</tr>
</tbody>
</table>

**Final State**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>150</td>
</tr>
</tbody>
</table>

**Final State**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>150</td>
</tr>
</tbody>
</table>
Mechanisms

- Shadow Paging
- Write-Ahead-Logging

- beneficial for coarse grained updates
Mechanisms

Shadow Paging
- beneficial for coarse grained updates
  - reads redirection
  - victim cache

Write-Ahead-Logging
- fine grained log->data ordering

REDO

UNDO
Undo Logging

1. **Compute**: Compute the new value \( V = A - 50 \)

2. **Log**: Write old value of data to log space in persistent memory (Log \([A, 100]\))

3. **Modify**: Modify data in-place \((A = V)\)
Undo Logging

1. **Compute**: Compute the new value \( V = A - 50 \)

2. **Log**: Write old value of data to log space in persistent memory \( \text{Log} [A, 100] \)

3. **Modify**: Modify data in-place \( A = V \)
Undo Logging

1. **Compute**: Compute the new value \( V = A - 50 \)

2. **Log**: Write old value of data to log space in persistent memory (Log \([A, 100]\))

3. **Modify**: Modify data in-place \((A = V)\)
Undo Logging

1. **Compute**: Compute the new value ($V = A - 50$)

2. **Log**: Write old value of data to log space in persistent memory ($\text{Log}[A, 100]$)

3. **Modify**: Modify data in-place ($A = V$)

Log writes reach NVM before data writes. ($\text{Log} \rightarrow \text{Data ordering}$)
Undo Logging

1. **Compute**: Compute the new value \( V = A - 50 \)

2. **Log**: Write old value of data to log space in persistent memory (Log \( [A , 100] \))

3. **Modify**: Modify data in-place \( (A = V) \)

Logging is essentially a data movement task.
System Architecture

Disk Based Persistence

NVM Based Persistence
System Architecture

Disk Based Persistence

NVM Based Persistence

Core

Cache

DRAM

Software Controlled

Secondary Storage

Core

Cache

NVM

Secondary Storage
System Architecture

Disk Based Persistence

Core

Cache

DRAM

Secondary Storage

Software Controlled

NVM Based Persistence

Core

Cache

NVM

Secondary Storage

Hardware Controlled
Logging with Disk

- Compute
- Log
- Modify
- Flush Log
- Flush Data
Logging with Disk

Volatile Phase

Compute → Log → Modify

Persistence Phase

Flush Log → Flush Data
Logging with Disk

Volatile Phase:
- Compute
- Log
- Modify

Persistence Phase:
- Flush Log
- Flush Data

Clear separation of volatile and persistence phases.
Logging with NVM

Diagram:
- Compute
- Log
- Flush Log
- Modify
- Flush Data
Logging with NVM

Volatile and persistence phases overlap.
ATOM

Compute → Modify → Flush Data

Log → Flush Log

In Hardware
Goal: Move logging out of critical path.
while ( ! Done ) {
    Write Undo Log
    Flush Log
    Modify Data
}
Flush Data

ATOMIC_BEGIN
while ( ! Done ) {
    Modify Data
}
Flush Data
ATOMIC_END

Software Logging

ATOM
Baseline Hardware Logging

- **Create Undo Log**
  - on a store, write old value to log
- **Flush Undo Log**
  - enforce log $\rightarrow$ data ordering
Baseline Hardware Logging

- **Create Undo Log**
  - on a store, write old value to log
- **Flush Undo Log**
  - enforce log $\rightarrow$ data ordering
Baseline Hardware Logging

- **Create Undo Log**
  - on a store, write old value to log
- **Flush Undo Log**
  - enforce log $\rightarrow$ data ordering

![Diagram](image-url)
Baseline Hardware Logging

- **Create Undo Log**
  - on a store, write old value to log

- **Flush Undo Log**
  - enforce log $\rightarrow$ data ordering

![Diagram of system architecture showing Core, Cache, Log, Data, and NVM with values A = 50 and L(A) = 100]
ATOM Design Philosophy

Where is log $\rightarrow$ data ordering enforced?

Diagram:
- Core
- Cache
- Memory Controller
- NVM
- Store Queue
- Store Buffer
ATOM Design Philosophy

Where is log —> data ordering enforced?

Core

Cache

Memory Controller

NVM

Baseline Design

Store Queue

Store Buffer
ATOM Design Philosophy

Where is log —> data ordering enforced?

Core

Cache

Memory Controller

NVM

Store Queue

Baseline Design

Store Buffer

ATOM Design
Baseline Implementation

SQ → Cache → Mem Ctrl → Memory
Baseline Implementation

SQ  →  ST(A)  →  Cache  →  Mem Ctrl  →  Memory
Baseline Implementation
Baseline Implementation

SQ
Cache
Mem Ctrl
Memory

ST(A)  L(A)  L(A)
Baseline Implementation

SQ

Cache

Mem Ctrl

Memory

ST(A)

L(A)

L(A)

WRITE L(A)
Baseline Implementation
Baseline Implementation

SQ
Cache
Mem Ctrl
Memory

ST(A)  L(A)  L(A)  L(A)
       L(A)  L(A)  L(A)

WRITE L(A)
Baseline Implementation

SQ

Cache

Mem Ctrl

Memory

ST(A)

L(A)

L(A)

L(A)

L(A)

WRITE L(A)
Baseline Implementation

Store Completion Time

SQ
- ST(A)
- ST(A)

Cache
- L(A)
- L(A)

Mem Ctrl
- L(A)
- L(A)

Memory
- WRITE L(A)
Baseline Implementation

Log persist operation in the critical path of retiring stores.
ATOM

Posted Log

SQ
Cache
Mem Ctrl
Memory
ATOM
Posted Log

SQ  ST(A)  
Cache
Mem Ctrl
Memory
ATOM
Posted Log

SQ
Cache
Mem Ctrl
Memory

ST(A)
L(A)
ATOM
Posted Log

SQ
ST(A)

Cache
L(A)

Mem Ctrl
L(A)

Memory
ATOM
Posted Log

SQ
Cache
Mem Ctrl
Memory

ST(A)
L(A)
L(A)
WRITE L(A)
ATOM

Posted Log

SQ

Cache

Mem Ctrl

Memory

WRITE L(A)
ATOM
Posted Log

SQ
ST(A)

Cache
L(A)
L(A)

Mem Ctrl
L(A)
L(A)

Memory
WRITE L(A)
ATOM
Posted Log

SQ

Cache

Mem Ctrl

Memory

ST(A)   ST(A)

L(A)   L(A)

L(A)   L(A)

WRITE L(A)
ATOM
Posted Log

Store Completion Time

SQ
ST(A)
ST(A)

Cache
L(A)
L(A)
L(A)

Mem Ctrl
L(A)
L(A)

Memory
WRITE L(A)
Remove log persist operations from the critical path by enforcing ordering at memory controller.
ATOM
Store Miss

SQ
Cache
Mem Ctrl
Memory
ATOM
Store Miss

SQ → Cache → Mem Ctrl → Memory

• • RD (A)
ATOM Store Miss

SQ -> Cache -> Mem Ctrl -> Memory

- RD(A)
- RD (A)
ATOM
Store Miss

SQ
Cache
Mem Ctrl
Memory
ATOM

Store Miss

SQ

Cache

Mem Ctrl

Memory

RD(A) L(A)

RD(A)

RD (A)
ATOM
Store Miss

SQ

Cache

RD(A)  L(A)

Mem Ctrl

RD(A)  L(A)

Memory

RD (A)
ATOM
Store Miss

SQ
Cache
Mem Ctrl
Memory

RD(A)   L(A)
RD(A)   L(A)

• RD (A)

WRITE L(A)
ATOM
Store Miss

SQ
Cache
Mem Ctrl
Memory

RD(A)  L(A)
RD(A)  L(A)  L(A)
RD (A)  WRITE L(A)
ATOM
Store Miss

SQ

Cache

Mem Ctrl

Memory

• • RD (A)  WRITE L(A)

• RD(A)  L(A)  L(A)

• RD(A)  L(A)  L(A)
ATOM
Store Miss

SQ
Cache
Mem Ctrl
Memory

RD(A)  L(A)  L(A)  ST(A)

RD(A)  L(A)  L(A)

RD (A)  WRITE L(A)
ATOM

Store Miss

Store Completion Time

SQ

Cache

Mem Ctrl

Memory

• RD (A)

• WRITE L(A)
ATOM
Store Miss

Store Completion Time

SQ
Cache
Mem Ctrl
Memory

Same data goes from Mem Ctrl to Cache and back.
ATOM-OPT
Source Log

SQ
Cache
Mem Ctrl
Memory
ATOM-OPT
Source Log

SQ

Cache

Mem Ctrl

Memory • • RDx(A)
ATOM-OPT

Source Log

SQ

Cache

Mem Ctrl

Memory

• RDx(A)

• WRITE L(A)
ATOM-OPT
Source Log

SQ
Cache
Mem Ctrl
Memory

RDx(A)
RDx(A)
RDx(A)
WRITE L(A)
ATOM-OPT
Source Log

SQ

Cache

Mem Ctrl

Memory

ST(A)

RDx(A)

RDx(A)

RDx(A)

WRITE L(A)
ATOM-OPT

Source Log

Store Completion Time

SQ

ST(A)

RDx(A)

Cache

Mem Ctrl

RDx(A)

Memory

RDx(A)

WRITE L(A)
Remove redundant data movement by creating log entry in the memory controller.
Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>Baseline hardware undo log implementation</td>
</tr>
<tr>
<td>ATOM</td>
<td>Posted log writes to memory controller</td>
</tr>
<tr>
<td>ATOM-OPT</td>
<td>Posted log writes with source logging</td>
</tr>
<tr>
<td>NON-ATOMIC</td>
<td>No logging (Upper bound on performance)</td>
</tr>
</tbody>
</table>

**Atomic Durability Designs**

- **System Configuration**
  - We evaluate proposed design using **GEM5 full-system simulation mode**
  - **32 Core CMP with 32x1MB LLC cache banks** and **4 memory controllers**
Transaction Throughput
Transaction Throughput

Higher is Better
Transaction Throughput

Higher is Better
Transaction Throughput

Higher is Better

- btree
- hash
- queue
- rbtree
- sdg
- sps
- gmean

ATOM
ATOM-OPT
NON-ATOMIC

27%
Transaction Throughput

ATOM-OPT performance is within 11% of optimal design.
Crash Consistency Primitives

- Ordering primitive [Condit’09, Joshi’15, Kolli’16, Haria’17]
  - necessary for crash consistent programming
  - reasoning about crash consistency with ordering as the only primitive is difficult
- Atomic Durability [Doshi’16, Joshi’17, Shin’17]
  - eases the burden of reasoning about crash consistency
  - enables further performance optimizations
Crash Consistency Primitives

Atomic_Begin

Log (A)
Write (A)
Log (B)
Write (B)
Atomic_End
Crash Consistency Primitives

Atomic_Begin

Log (A)

Write (A)

Log (B)

Write (B)

Atomic_End

Explicit Constraint
Crash Consistency Primitives

Explicit Constraint

Atomic_Begin

Log (A)

Write (A)

Log (B)

Write (B)

Atomic_End

Implicit Constraint with Ordering
Crash Consistency Primitives

Atomic_Begin

Log (A)

Write (A)

Log (B)

Write (B)

Atomic_End

Explicit Constraint

Not present with Atomic Durability
Conclusion

- Non-Volatile memory is an opportunity to enable in-memory persistent data structures
- Primitives like Atomic Durability are necessary to ensure crash consistency
- Traditional approaches to logging perform log writes to NVM in the critical path of store operations
- ATOM removes log writes from the critical path by enforcing ordering at the memory controller
Architectural Support for Atomic Durability in Non-Volatile Memory

Arpit Joshi, Vijay Nagarajan, Stratis Viglas, Marcelo Cintra

NVMW 2018