Fast and Consistent Remote Direct Access to Non-volatile Memory

Jingwen Du, Fang Wang, Dan Feng, Weiguang Li, Fan Li
Huazhong University of Science and Technology, China
Corresponding author: wangfang@hust.edu.cn
Outline

- Background and Motivation
- Our work: eFactory
- Performance Evaluation
- Conclusion
Modern data-intensive applications require **fast access to massive amounts of persistent data**.

**Opportunities**
- Emerging hardware technologies: NVM & RDMA

<table>
<thead>
<tr>
<th>Non-volatile Main Memory (NVMM)</th>
<th>Remote Direct Memory Access (RDMA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Non-volatility</td>
<td>• Kernel-bypass network</td>
</tr>
<tr>
<td>• Byte addressability</td>
<td>• high bandwidth (100Gbps)</td>
</tr>
<tr>
<td>• Large capacity</td>
<td>• Low latency (2(\mu s))</td>
</tr>
<tr>
<td>• Low latency (100ns)</td>
<td>• <strong>One-sided</strong>: bypass remote CPUs</td>
</tr>
<tr>
<td></td>
<td>• Two-sided: fast message passing</td>
</tr>
</tbody>
</table>
Client-active Scheme*

- Data writing is offloaded to clients
- Process more requests and gain higher throughput

*proposed and used in previous work such as Octopus@ATC’17, Orion@FAST’19, etc.

[1] RNIC = RDMA-enabled Network Interface Card
Challenge: Remote Data Consistency

- RDMA write doesn’t have persistence semantics currently
- Metadata is updated before data writing

[1] RNIC = RDMA-enabled Network Interface Card
Current Solutions: SAW & IMM

- **Send after Write(SAW)**¹
  - RDMA write followed by an extra RDMA send with DDIO enabled

- **IMM**²
  - replace RDMA write with write_with_imm primitive

---

• Client-active scheme w/o persistence (36% better than RPC)
• Writing durably with IMM & SAW loses latency advantage over RPCs (5%)
Current Solutions: Forca & Erda

- Use the client-active scheme w/o persisting immediately
- Verify data integrity with CRC* when reading

Put:

Client Server

Get Offset

Write Data

AM

UM

Get: Integrity Verification* Persisting

Client Server Client Server

Get Offset

Write Data

AM

UM

Read Meta

Read Data

Read Meta

Read Data

Forca\(^3\)

Erda\(^4\)

* Using CRC (Cyclic Redundancy Check)

[3] Huang H et al. Forca: Fast and atomic remote direct access to persistent memory, ICCD 2018
With value size increases, CRC overhead seriously degrades the read performance.

GET Latency Breakdown

(a) Forca

(b) Erda

RPC excluding CRC calculating
Fetch data
CRC calculating

35%
45%
Motivation

- Existing solutions sacrifice either read or write performance in exchange for consistency guarantees

<table>
<thead>
<tr>
<th></th>
<th>Data Consistency</th>
<th>Read Performance</th>
<th>Write Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SAW</strong>(^1)</td>
<td>√</td>
<td>√</td>
<td>x</td>
</tr>
<tr>
<td><strong>IMM</strong>(^2)</td>
<td>√</td>
<td>√</td>
<td>x</td>
</tr>
<tr>
<td><strong>Forca</strong>(^3)</td>
<td>√</td>
<td>x</td>
<td>√</td>
</tr>
<tr>
<td><strong>Erda</strong>(^4)</td>
<td>--</td>
<td>x</td>
<td>√</td>
</tr>
<tr>
<td><strong>eFactory</strong></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

(“x”: bad, “√”: good, “--”: moderate)

[3] Huang H et al. Forca: Fast and atomic remote direct access to persistent memory, ICCD 2018
Our Solution: eFactory

- Multi-version Log-structuring Design
- Data Workflow
  - Client-active Write with Asynchronous Durability
  - Single Background Thread for Efficient Consistency
  - Hybrid Read Scheme
- Lock-free Log Cleaning Scheme  
  Please check it in our paper!
Multi-version Log-structuring Design

① Out-of-place update with log structuring mechanism
② Multi-version linked list for each object
③ Embed durability flag in each object
④ 2-offset region & backlink pointer for lock-free log cleaning
Multi-version Log-structuring Design

① Out-of-place update with log structuring mechanism
② Multi-version linked list for each object
③ Embed durability flag in each object
④ 2-offset region & backlink pointer for lock-free log cleaning

✓ Concurrent access
✓ Crash consistency
① Out-of-place update with log structuring mechanism
② Multi-version linked list for each object
③ Embed durability flag in each object
④ 2-offset region & backlink pointer for lock-free log cleaning
Multi-version Log-structuring Design

① Out-of-place update with log structuring mechanism
② Multi-version linked list for each object
③ Embed durability flag in each object
④ 2-offset region & backlink pointer for lock-free log cleaning

✓ Help cooperation between foreground thread and background thread
✓ Support the hybrid read scheme for high performance
Client-active Write with Async Durability

✓ High performance for write

Client

PUT(K1, C2)

addr1

Server

Hash Table

PUT(C2)

CPU

High performance for write

Asynchronous Integrity Verification and Durability
Reduce persistence and CRC\textsuperscript{[1]} overheads on the critical path

\begin{itemize}
  \item Calculating CRC
  \item Persisting value if the object is complete
  \item Set the durability flag
\end{itemize}

\textsuperscript{[1]} Cyclic Redundancy Check
Hybrid Read: Basic Methods for GET

- **RPC+RDMA read**
  - Easy to ensure data consistency
  - Relatively slow

- **Pure RDMA read**
  - High performance
  - Difficult to ensure data consistency
• **Challenge:** Is it possible to achieve high performance while providing data consistency?

• **Solution:** Hybrid Read Scheme
  - Use pure RDMA Read method Optimistically
  - Fall back to RPC+RDMA read method
    - Selective durability guarantee
Hybrid Read: Use Pure Read Optimistically

- **GET(Key2): Case 1**

  - Check Durability Flag
  - If durability flag = 1, read complete!
**Hybrid Read: Fall back to RPC+RDMA Read**

- **GET(Key2): Case 2**

  - **Check Durability Flag**
    - If durability flag = 0, change read strategy

  - **Hash(key2)**

  - **Data Pool**
    - A B A C B C ...

  - **Hash Table**

  - **Server**

  - **Client**

  - **Cursor**

  - **DF:**
    - 1 1 0 0 0 0 0
Hybrid Read: Fall back to RPC+RDMA Read

If durability flag = 0, change read strategy.
Hybrid Read: Fall back to RPC+RDMA Read

If durability flag = 0, change read strategy
Hybrid Read: Fall back to RPC+RDMA Read

If durability flag = 0, change read strategy

Return value’s location

Read complete!!
Hybrid Read: Selective Durability Guarantee

If durability flag = 0, change read strategy

GET Client

Hash(key2)

GET Client

Get(Key2)

Check durability flag

DF: 1 1 1 0 0 0

A B A C B C

Data Pool

A B A C B C

A B A C B C

Data Pool

Server

Hash Table

CPU

Server

Hash Table

cursor
Hybrid Read: Selective Durability Guarantee

GET Client

Get(Key2)

CPU

Hash Table

Server

Data Pool

A B A C B C ... 

DF: 1 1 1 0 0 0

Check durability flag
Selective Durability Guarantee

Request Handler
① Calculating CRC over Object B
Hybrid Read: Selective Durability Guarantee

- Case 1: Calculated CRC == Stored CRC

1. Calculating CRC over Object B
2. Persisting value if it is complete
Hybrid Read: Selective Durability Guarantee

- **Case 1: Calculated CRC == Stored CRC**

  **Request Handler**
  ① Calculating CRC over Object B
  ② Persisting value if it is complete
  ③ Set durability flag
**Case 1: Calculated CRC == Stored CRC**

1. Calculating CRC over Object B
2. Persisting value if it is complete
3. Set durability flag

Request Handler

- Calculating CRC over Object B
- Persisting value if it is complete
- Set durability flag
Case 2: Calculated CRC != Stored CRC

Hybrid Read: Selective Durability Guarantee

GET
Client

Get(Key2)

Server

CPU

Hash Table

Data Pool

A  B  A  C  B  C  ...

DF:

1  1  1  0  0  0

Check durability flag
Selective Durability Guarantee

Request Handler
① Calculating CRC over Object B
Hybrid Read: Selective Durability Guarantee

Case 2: Calculated CRC != Stored CRC

1. Calculating CRC over Object B
2. Traverse the version list until find the completely durable version

Hash Table
Check durability flag
Selective Durability Guarantee

Server

Client

GET

Get(Key2)

Return Offset X

CPU

Data Pool

Offset X

1 1 1 0 0 0

Request Handler

Read complete!!
Hybrid Read Scheme

✓ High Performance
  – Use pure RDMA read method optimistically
  – Durability flag: verify data consistency efficiently
  – Selective durability guarantee: no need to wait bg thread

✓ Consistency Guarantee
  – Background thread: async verification and durability
  – Request Handler: selective durability guarantee
Performance Evaluation

• Environment
  – Use PMDK\textsuperscript{[1]} to emulate persistent memory
  – YCSB\textsuperscript{[2]} benchmark (Zipfian distribution)
  – Mellanox ConnectX-5 InfiniBand NIC
  – Two 10-Core CPUs with 25MB L3 cache

• Apple-to-apple Comparison on the same codebase
  – SAW: Send after Write @SDC’15
  – IMM: Solutions using write\_with\_imm @FAST’19
  – Forca @ICCD’18
  – Erda @arvix’19

\textsuperscript{[1]} 2016. How to emulate Persistent Memory. http://pmem.io/2016/02/22/pm-emulation.html
\textsuperscript{[2]} YCSB: https://github.com/brianfrankcooper/YCSB @ SOCC’10
Read Throughput

- Comparable read throughput as IMM and SAW (Gap: 2%)
- Better than Erda (1.3x-1.96x) and Forca (1.24x-1.67x)
Write Throughput

Throughput (IOPS) * 10000

Value Size (Bytes)

Better than IMM (0.42x-2.79x) and SAW (0.66x-2.85x)

Slightly higher than Erda and Forca
Scalability

When write dominates, IMM and SAW fail to scale well.
Conclusion

- Existing solutions sacrifice either read or write performance to guarantee data consistency of RDMA-based NVM systems.
- We propose eFactory, a multi-version log-structuring design:
  - Client-active Write with Asynchronous Durability
  - Single Background Thread for Efficient Consistency
  - Hybrid Read Scheme
  - Lock-free Log Cleaning Scheme
- Results: high performance for both read and write while providing data consistency:
  - Outperforms IMM and SAW by 0.42x-2.79x and 0.66x-2.85x for write
  - 1.3x-1.96x and 1.24x-1.67x of Erda and Forca for read
Thanks for listening

Q&A
Throughput with read-intensive workload

➢ Accounts for 95% of IMM’s throughput
➢ Outperforms Erda and Forca by 0.26x-0.74x and 0.2x-0.61x
Throughput with write-intensive workload

➢ Achieves the highest throughput for all the value sizes
Multi-version Log-structuring Design

1. Out-of-place update with log structuring mechanism
2. Multi-version linked list for each object
3. Embed durability flag in each object
4. 2-offset region & backlink pointer for lock-free log cleaning
Lock-free Log Cleaning Scheme

- 2 Stages: Log Compressing and Log Merging

Old Data Pool

A₁  B₁  A₂  C₁  B₂

New Data Pool


Lock-free Log Cleaning Scheme

• 2 Stages: Log Compressing and Log Merging

Old Data Pool

New Data Pool

New Writes

Update Pointer

D₁ F₁
Lock-free Log Cleaning Scheme

- **2 Stages: Log Compressing and Log Merging**

Old Data Pool

- $A_1$, $B_1$, $C_2$, $B_3$, $D_1$, $F_1$, ...

New Data Pool

- $B_2$, $C_1$, $A_2$, $D_2$

Old Data Pool

- $A_1$, $B_1$, $A_2$, $C_1$, $B_2$, $C_2$, $B_3$, $D_1$, $F_1$, ...

New Data Pool

- $B_2$, $C_1$, $A_2$, $D_2$

Reserved Space

New Writes

Update Pointer
Performance Impact by Log Cleaning

- Performance decreases since the reading process falls back to RPC+RDMA read scheme

Overall, log cleaning incurs 1%-21% performance overhead