RDMA is Turing complete, we just did not know it yet!

Waleed Reda, Marco Canini, Dejan Kostić, Simon Peter
Benefits of RDMA networking

• Bypasses the kernel and allows zero-copy data transfers

• Offers one-sided operations
  • e.g. RDMA READ or RDMA WRITE

• Requires no CPU involvement
  • But can only perform simple memory transfers!
Massive growth in RDMA processing power
Massive growth in RDMA processing power

Almost 2x increase / year!
Existing designs for RDMA-based systems

Commodity RNIC offloads

One-sided (e.g. FaRM-KV)
- Limited by RDMA API.
- Incurs extra roundtrips to serve requests

Two-sided (e.g. HERD)
- Requires remote CPU involvement.

Smart NIC offloads

FPGA-offload (e.g. KV-DIRECT)
- Expensive and difficult to program

NIC-CPU offload (e.g. UneFS)
- Expensive and uses slow “wimpy” cores.
Alternative Design: Exploit RNIC Processing Power

Insight #1: Perform complex operations using RDMA chains
Alternative Design: Exploit RNIC Processing Power

Insight #1: Perform complex operations using RDMA chains

Use RDMA WAIT feature

☑ Adds execution dependency between operations
Alternative Design: Exploit RNIC Processing Power

Insight #1: Perform complex operations using RDMA chains

Use RDMA WAIT feature

- Adds execution dependency between operations
- Allows clients to trigger server RDMA code
Alternative Design: Exploit RNIC Processing Power

Insight #1: Perform complex operations using RDMA chains

- Use RDMA WAIT feature
  - Adds execution dependency between operations
  - Allows clients to trigger server RDMA code

Rich API for offloads + Uses commodity RNICs

Client

NIC

Server

Work Queue 1

RDMA SEND

RDMA RECV

Work Queue 2

RDMA WAIT

RDMA READ

Work Queue 3

RDMA WAIT

RDMA WRITE
But is this Turing complete?

• So far, we only managed to construct an imperative language for RDMA NICs

• To be Turing complete, two requirements must be met:
  - **R1** The ability to read/write to an arbitrary amount of memory
  - **R2** Conditional branching (e.g. support for if/else statements)
But is this Turing complete?

• So far, we only managed to construct an imperative language for RDMA NICs

• To be Turing complete, two requirements must be met:
  
  **R1** The ability to read/write to an arbitrary amount of memory
  **R2** Conditional branching (e.g. support for if/else statements)
But is this Turing complete?

• So far, we only managed to construct an imperative language for RDMA NICs

• To be Turing complete, two requirements must be met:
  
  **R1** The ability to read/write to an arbitrary amount of memory
  
  **R2** Conditional branching (e.g. support for if/else statements)
But is this Turing complete?

• So far, we only managed to construct an imperative language for RDMA NICs

• To be Turing complete, two requirements must be met:
  
  R1 The ability to read/write
  R2 Conditional branching (e.g. support for if/else statements)
  R3 Support for loops or recursion
Conditional Branching – is it possible?

Client

NIC

Mem

CPU

Server

RDMA SEND

RDMA RECV

RDMA WAIT

RDMA WRITE

Work Queue 1

Work Queue 2

Work Queue 3
Conditional Branching – is it possible?

Insight #2: Use self-modifying RDMA code to control execution
Conditional Branching – is it possible?

Insight #2: Use self-modifying RDMA code to control execution
Conditional Branching – is it possible?

Insight #2: Use self-modifying RDMA code to control execution

RDMA Compare-and-Swap (CAS) to check conditions
- Typically used for simple transactions
- Supported by commodity RDMA NICs
Branching with Self-Modifying Code

**Simple Example**

Input $x, y$

If ($x == y$)

    return `foo`;

else

    return `bar`;

**RDMA code (server-side):**

**Work Queues (WQs)**
Branching with Self-Modifying Code

Simple Example

Input $x, y$
If ($x == y$)
    return $\text{foo}$;
else
    return $\text{bar}$;

RDMA code (server-side):

$$\text{opcode id}$$

- **R1**: `RECV`
- **R2**: `CAS` old: `NOOP` new: `WRITE`
- **R3**: `NOOP` data: `foo`
- **R4**: `WRITE` data: `bar`

Work Queues (WQs)
Branching with Self-Modifying Code

Simple Example

Assume $x == y$ is true

Input $x, y$

If ($x == y$)
    return foo;
else
    return bar;

RDMA code (server-side):

<table>
<thead>
<tr>
<th>opcode</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>RECV</td>
</tr>
<tr>
<td>R2</td>
<td>CAS</td>
</tr>
<tr>
<td>R3</td>
<td>NOOP</td>
</tr>
<tr>
<td>R4</td>
<td>WRITE</td>
</tr>
</tbody>
</table>

Work Queues (WQs)
Branching with Self-Modifying Code

**Simple Example**

Assume \( x == y \) is true

Input \( x, y \)

If \( x == y \)

return \texttt{foo};

else

return \texttt{bar};

**RDMA code (server-side):**

**Client**

\[ \text{opcode \ id} \]

- **R1:** \texttt{RECV}

- **R2:** \texttt{CAS} old: NOOP new: WRITE

- **R3:** NOOP data: \texttt{foo}

- **R4:** \texttt{WRITE} data: \texttt{bar}

**Work Queues (WQs)**
Branching with Self-Modifying Code

Simple Example

Assume $x == y$ is true

Input $x, y$
If ($x == y$)
    return foo;
else
    return bar;

RDMA code (server-side):

Work Queues (WQs)

Client

opcode id

R1 RECV
R2 CAS old: NOOP new: WRITE
R3 NOOP data: foo
R4 WRITE data: bar

SEND data: $x, y$
Branching with Self-Modifying Code

Simple Example

Assume $x == y$ is true

Input $x, y$

If $(x == y)$

return $\text{foo}$;

else

return $\text{bar}$;

RDMA code (server-side):

```
WRITE data: bar
CAS old: NOOP new: WRITE
RECV
NOOP data: foo
WRITE data: bar
```

Client

```
SEND data: $x, y$
```

Work Queues (WQs)
Branching with Self-Modifying Code

**Simple Example**

Assume $x == y$ is true

Input $x, y$

If ($x == y$)
    return foo;
else
    return bar;

**RDMA code (server-side):**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Id</th>
<th>Data</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECV</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAS</td>
<td>R2</td>
<td></td>
<td>NOOP</td>
<td>WRITE</td>
</tr>
<tr>
<td>WRITE</td>
<td>R3</td>
<td>foo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td>R4</td>
<td>bar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Client**

SEND data: $x, y$

**Work Queues (WQs)**
Branching with Self-Modifying Code

Simple Example

Assume \( x == y \) is true

Input \( x, y \)
If \( x == y \)
    return foo;
else
    return bar;

RDMA code (server-side):

\[
\begin{align*}
\text{WRITE} & \quad \text{data:} \quad \text{bar} \\
\text{CAS} & \quad \text{old:} \quad \text{NOOP} \quad \text{new:} \quad \text{WRITE} \\
\text{RECV} & \quad \text{data:} \quad \text{foo} \\
\text{WRITE} & \quad \text{data:} \quad \text{foo} \\
\text{WRITE} & \quad \text{data:} \quad \text{foo}
\end{align*}
\]

Client

Assume \( x == y \) is true

SEND data: \( x, y \)

Work Queues (WQs)
Branching with Self-Modifying Code

Simple Example

Assume $x == y$ is true

Input $x$, $y$
If ($x == y$)
  return foo;
else
  return bar;

RDMA code (server-side):

```
WRITE     data: bar
CAS          old: NOOP      new: WRITE
RECV
```

Client

Assume $x == y$ is true

SEND      data: $x$, $y$

R1
R2
R3
R4

 opcode id

foo

Work Queues (WQs)
What about loops?

• **Observation:** RDMA operations are not deleted after execution
What about loops?

• **Observation:** RDMA operations are not deleted after execution

**Insight #3:** Recycle previously posted RDMA operations
What about loops?

- **Observation**: RDMA operations are not deleted after execution

**Insight #3**: Recycle previously posted RDMA operations
What about loops?

- **Observation:** RDMA operations are not deleted after execution

**Insight #3:** Recycle previously posted RDMA operations

Use **RDMA ENABLE** at the end to re-trigger chain
What about loops?

• **Observation:** RDMA operations are not deleted after execution

**Insight #3:** Recycle previously posted RDMA operations

Use **RDMA ENABLE** at the end to re-trigger chain
RedN Framework - Overview

Convert to RDMA Code
RedN Framework - Overview

```
if (x == 5)
  return true;
else
  return false;
```

Example offload

Convert to RDMA Code
RedN Framework - Overview

Client

RDMA NIC

Example offload
if (x == 5)
return true;
else
return false;

Post RDMA Code (chain of work requests)

Setup Offload (done once)

CPU

Convert to RDMA Code

Host Memory

Work Queues (WQs)

User buffers

Example: RDMA chain

if (x == 5)
return true;
else
return false;

Example: RDMA chain

if (x == 5)
return true;
else
return false;

if (x == 5)
return true;
else
return false;
**RedN Framework - Overview**

**Client**

1. **Setup Offload** (done once)  
2. **Convert to RDMA Code**
3. **Trigger Function** (invoked as necessary)

**Server**

1. **Convert to RDMA Code**
2. **Post RDMA Code (chain of work requests)**
3. **RPC request triggers WAIT**
4. **Example offload**
   ```java
   if (x == 5)
       return true;
   else
       return false;
   ```
5. **Response Ready**

**RDMA NIC**

- Example: RDMA chain
- Example: READ
- Example: WRITE
- Example: CAS
- Example: NOP

**Host Memory**

- User buffers
- Work Queues (WQs)

**Example: RDMA chain**

**RedN Framework**

1. **REDN**
2. **REDN**
3. **REDN**
4. **REDN**
5. **REDN**
RedN Framework - Overview

**Trigger Function** (invoked as necessary)

1. **Setup Offload** (done once)
   - Convert to RDMA Code

2. **CPU**
   - Example offload
     ```
     if (x == 5) return true;
     else return false;
     ```

3. **RDMA NIC**
   - Convert to RDMA Code (chain of work requests)

4. **RPC request triggers WAIT**

5. **Response Ready**

**Conditional branching + Loops**

= RDMA NIC is Turing Complete
Evaluation

• Our experimental testbed consists of 3× dual-socket Haswell servers:
  • 3.2 GHz, with a total of 16 cores
  • 128 GB of DRAM
  • 100 Gbps dual-port Nvidia ConnectX-5 Infiniband RNICs.
  • Nodes are connected via back-to-back Infiniband links

• We evaluate RedN using microbenchmarks and real applications
  (e.g. Memcached)
Use case: Memcached Lookups

Client inputs

1. $H(x)$
2. RECV
3. CAS
4. NOOP

set opcode to WRITE iff $x == \text{key}$
Use case: Memcached Lookups

Client inputs

1. REC
2. READ
3. CAS
4. NOOP

H(x)
\(x\)

set opcode to WRITE iff \(x\) == key

13
Use case: Memcached Lookups

Client inputs

1. \( H(x) \)  
2. \( x \)  
3. NOOP  
4. CAS

set opcode to WRITE iff \( x == \text{key} \)

RECV \rightarrow READ

buckets

0 1 2 3 4 5 6 7 8

key ptr value
Use case: Memcached Lookups

Client inputs

1. \( H(x) \)
2. \( x \) \rightarrow \text{RECV}
3. \( x \) \rightarrow \text{READ}
4. \( x \) \rightarrow \text{NOOP}
5. \( x \) \rightarrow \text{CAS}

set \text{opcode} \text{ to WRITE} \text{ iff } x == \text{key}

\text{buckeets}

H(x)

\( x \)

key
ptr
value
Use case: Memcached Lookups

Client inputs

\[ H(x) \]
\[ x \]

1. \( \rightarrow \) RECV
2. \( \rightarrow \) READ
3. \( \rightarrow \) CAS
4. \( \rightarrow \) NOOP

set opcode to WRITE iff \( x == \) key

buckets

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>key</td>
<td>ptr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ H(x) \]
\[ x \]
Use case: Memcached Lookups

Client inputs

1. $H(x)$
2. REC
3. READ
4. WRITE

set opcode to WRITE iff $x == key$

bucket allocation:

<table>
<thead>
<tr>
<th>buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
| 2       | key, ptr
| 3       |
| 4       |
| 5       |
| 6       |
| 7       |
| 8       |

Client inputs:

$x$
Use case: Memcached Lookups

Client inputs

1. \( H(x) \) → RECV
2. RECV → READ
3. READ → CAS
4. WRITE → Return value to client

set opcode to WRITE iff \( x == \) key

buckets

\[
\begin{array}{c|c|c}
\hline
0 & 1 & 2 \\
\hline
key & ptr & value \\
\hline
\end{array}
\]
Results: Memcached *get* latency

![Bar chart showing latency for different value sizes and access modes.](image-url)

- **RedN**
- **One-sided**
- **Two-sided (VMA)**

Latency (us) vs. Value Size (B): 64, 1K, 4K, 16K, 64K
Results: Memcached *get* latency

Requires 2 RTTs

- **RedN**
- **One-sided**
- **Two-sided (VMA)**

Latency (us)

- 64
- 1K
- 4K
- 16K
- 64K

Value Size (B)
Results: Memcached *get* latency

- **Requires 2 RTTs**
- **Extra memory copies**

- **RedN**
- **One-sided**
- **Two-sided (VMA)**

Latency (µs)

Value Size (B)

<table>
<thead>
<tr>
<th>64</th>
<th>1K</th>
<th>4K</th>
<th>16K</th>
<th>64K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results: Memcached *get* latency

- **Requires 2 RTTs**
- **Extra memory copies**

**Legend:**
- RedN
- One-sided
- Two-sided (VMA)

**Graph:**
- Latency (us)
- Value Size (B)
- Size values: 64, 1K, 4K, 16K, 64K
Results: Memcached get latency

- Requires 2 RTTs
- Extra memory copies

<table>
<thead>
<tr>
<th>Value Size (B)</th>
<th>RedN</th>
<th>One-sided</th>
<th>Two-sided (VMA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64K</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Latency (us)

2.6x
Results: Memcached *get* latency

RedN accelerates Memcached *gets* by up to 2.6x

- **RedN**
- **One-sided**
- **Two-sided (VMA)**

Requires 2 RTTs

Extra memory copies

Latency (us)

Value Size (B)

- 64
- 1K
- 4K
- 16K
- 64K

2.6x
Results: Memcached *get* (contention)

![Graph showing latency (us) vs level of contention for Memcached get operations with different contention levels and metrics. The graph includes lines for RedN Avg., RedN 99th-%ile, Two-sided Avg., and Two-sided 99th-%ile.]
Results: Memcached *get* (contention)

- **RedN Avg.**
- **RedN 99<sup>th</sup>–%ile**
- **Two–sided Avg.**
- **Two–sided 99<sup>th</sup>–%ile**
Results: Memcached *get* (contention)

RedN improves latency by up to 35x under contended settings
Conclusion

• **RedN** shows that RDMA is Turing complete

Unlocks the door for innovations in many areas

- Distributed Locking
- Consensus
- Database Transactions
- Distributed Deep Learning
- Network Telemetry
- RDMA Security

• Source code: [redn.io](http://redn.io)