KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing

Xingda Wei*, Fangming Lu, Rong Chen* and Haibo Chen

The institute of parallel and distributed systems (IPADS)
Shanghai Jiaotong university
* Shanghai AI Laboratory
Remote Direct Memory Access (RDMA)

A high-performance user-space networking feature
- With high bandwidth (up to 400Gbps)
- Low latency (down to 2us)

RDMA and its primitives
- **One-sided**: RNIC read/writes memory bypassing CPU
- **Two-sided**: a messaging primitive (send/recv)

Improve the performance of distributed systems
- E.g., key-value stores (RACE), transactions (FaRM-v2), etc.

* RACE: One-sided rdma-conscious extendible hashing for disaggregated memory@ATC’21
* FaRM-v2: Fast general distributed transactions with opacity@SIGMOD’19
* RNIC: RDMA-capable Network Card
Problem: creating RDMA connections is slow

To use RDMA, user must create RCQP (control plane)

- Reliable connected (RC) queue pair (QP)
- Creating and connecting RCQPs may take a long time
Problem: creating RDMA connections is slow

The creation has three parts

① Loading the driver context at the user-space
Problem: creating RDMA connections is slow

The creation has three parts

① Loading the driver context at the user-space

② Creating and configuring the hardware queues

![Diagram showing the creation process and time breakdown with QPs and one-sided requests.]

- Data (2us)
- Control (15ms)

Time (μs):
- Data: 2us
- Control: 4ms, 15ms

One-sided requests
Problem: creating RDMA connections is slow

The creation has three parts

① Loading the driver context at the user-space
② Creating and configuring the hardware queues
③ Exchange the connection information with remote end
Problem: creating RDMA connections is slow

The creation has three parts

① Loading the driver context at the user-space
② Creating and configuring the hardware queues
③ Exchange the connection information with remote end

Challenging to reduce due to configuring and creating the hardware resources

<table>
<thead>
<tr>
<th>Time (µs)</th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>2us</td>
<td></td>
<td>15ms</td>
</tr>
</tbody>
</table>

*QP: One-sided requests*
No problem for traditional applications

Traditional RDMA–enabled applications are not affected
- E.g., RDMA–enabled databases, filesystems, scientific applications
- Because they run a sufficient long time

What about new applications that require elasticity?
Impact the performance of elastic applications

Example: RDMA–enabled disaggregated key–value stores (KVS)

- Nodes are separated:
  - **Memory nodes.** store the KV–pairs
  - **Compute nodes.** use RDMA to read the KV–pairs from memory nodes
Impact the performance of elastic applications

Benefits: handle loads in a resource efficient way elastically

- If the load changes, we can dynamically add/remove nodes to cope with them

However, new nodes need new RCQPs to the memory nodes

- Requests handled by the new nodes inevitably face the high tail latency

*RACE* is a state-of-the-art RDMA-enabled elastic KV
Goal & this work: reducing RDMA connection time from ms to us and compatible to existing RDMA hardware & software
Basic idea: connection pooling & reusing

Cache RCQPs in a connection pool
- The QPs in the pool can be reused by future applications with no connection cost
Basic idea: connection pooling & reusing

Cache RCQPs in a connection pool

- The QPs in the pool can be **reused** by future applications with no connection cost.
Basic idea: connection pooling & reusing

Cache RCQPs in a connection pool
- The QPs in the pool can be reused by future applications with no connection cost

Example: App#2 can reuse App#1’s QP without connection
Challenge#1. User-space QPs cannot be shared

Different process/container cannot be shared the same QP

- RDMA is in default used in **user-space** (not designed for share among applications)
- User-space QP has a complex data structures (both at the user-space and in kernel)
- Further, cannot reduce the driver loading costs
Solution #1. share QPs in a kernel–space QP pool

Kernel–space RDMA driver also support full–fledged RDMA

- Provide a near–same functionality as user–space QPs
- E.g., `ibv_qp` (user–space RDMA QP) has an equivalent `ib_qp` in the kernel
Solution #1. share QPs in a kernel-space QP pool

Kernel-space RDMA driver also support full-fledged RDMA

- Provide a near-same functionality as user-space QPs
- E.g., `ibv_qp` (user-space RDMA QP) has an equivalent `ib_qp` in the kernel

Idea: put the qp pool in the kernel

- Therefore, different applications can share the same QP
- i.e., we translate the API to the kernel-space QP
Solution #1. share QPs in a kernel–space QP pool

Kernel–space RDMA driver also support full–fledged RDMA

- Provide a near–same functionality as user–space QPs
- E.g., `ibv_qp` (user–space RDMA QP) has an equivalent `ib_qp` in the kernel

Further, a kernel–space solution avoid user–space driver loading

- i.e., kernel can pre–load all the driver context during boot time
Challenge #2. Massive QPs cached in the pool

RCQP is a one-to-one mapping

- Needs a dedicated QP to connect to a different server
- Also, different CPU may have dedicated QP for the best performance \[1\]

Therefore, we need M X N QPs cached in the pool

- \(M\): the number of machines in the cluster
- \(N\): the number of cores on the machine

Causes GBs of memory on modern clusters w/ >10K nodes

\[1\] Fast remote memory@NSDI’14
Opportunity: **Dynamically connected transport (DCT)**

A less–used (but widely support) advance RDMA transport

- E.g., NVIDIA supports DCT through NICs later than Connect–IB (released in 2014)
Opportunity: **Dynamically connected transport (DCT)**

A less-used (but widely support) advance RDMA transport

- E.g., NVIDIA supports DCT through NICs later than Connect-IB (released in 2014)

A DCQP can connect to multiple nodes w/o user connection

- The hardware can piggyback the connection with request and is extremely fast

![Diagram showing RCQP and DCQP with timing details](image)
Solution #2. Retrofit DCT as the shared connection

i.e., the server QP pool use DCQP as the default connection

- No need for a separate RCQP for each machine in the cluster

Problem: DCT metadata discovery

- To communicate with a specific host, the server must first create a **DC Target**, and hand-off the metadata associated with the target to the client
Solution #2. Retrofit DCT as the shared connection

Problem: DCT metadata discovery
- To communicate with a specific host, the server must first create a DC Target, and hand-off the metadata associated with the target to the client

Naïve solution
- Use unreliable-datagram (UD)-based RPC for the discovery

Drawbacks of RPC in our scenarios
① Each server must use dedicated polling threads to handle the DCT discovery requests
② RPC’s latency can vibrate (to 10ms) due to queuing at the server-side
Our design: MetaServer

We dedicate few nodes in the cluster to store the DCT metadata

- Possible: DCT metadata is extremely small (12B)
Our design: MetaServer

We dedicate few nodes in the cluster to store the DCT metadata

- Possible: DCT metadata is extremely small (12B)

A separate architecture allows query metadata with one-sided RDMA

- i.e., implement the MetaServer in an RDMA-enabled key-value store
- Each machine maintains QPs connected to nearby MetaServers

Our design:

MetaServer

We dedicate few nodes in the cluster to store the DCT metadata.

- Possible: DCT metadata is extremely small (12B)

A separate architecture allows query metadata with one-sided RDMA.

- i.e., implement the MetaServer in an RDMA-enabled key-value store.
- Each machine maintains QPs connected to nearby MetaServers.

<table>
<thead>
<tr>
<th>Server addr</th>
<th>DCT meta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0d:9a03:...</td>
<td>73</td>
</tr>
</tbody>
</table>

One-sided requests

Query + control + data (10us)
Challenge #3. Correct QP multiplexing

We let each DCQP in the pool to be shared by multiple applications

- Thus, each application can always choose a QP in the pool

The shared QP is further virtualized to multiple user–space QP

- Provide the same semantic as RCQP to simplify development
Challenge #3. Correct QP multiplexing

We let each DCQP in the pool to be shared by multiple applications

- Thus, each application can always choose a QP in the pool

Problem: shared QPs can be corrupted by various reasons

- Worser, even applications correctly use the shared QP, the QP can be corrupted
- A corrupted QP will prevent progress (and requires re-connection)
We add additional checks to prevent QP corruptions

Behavior the same as a single QP, yet may use a shared QP

How to achieve so? We add additional checks to each request:

1. Malformed requests
2. Queue overflows
3. Completion dispatch

Things to check
Put it all together: KRCore

A networking library that provides us-scale RDMA connections

- On commodity RNICS that support DCT
- 1,500X faster than ibverbs (in connection latency)

We also apply other optimizations

- DCT metadata caching
- Dynamic switch between DCQP & RCQP
KRCore implementation

Implemented as a loadable kernel module

- 10,000 LoC+ Rust code
- With a C-shim layer to translate RDMA request to systemcalls

We are the first to port DCT to the kernel-space RDMA driver

- With ~250 LoC C code added to the mlnx-ofed-4.9 driver

Available on GitHub with continuously developments

- https://github.com/SJTU-IPADS/krcore-artifacts
Evaluations

Questions aim to answer

1. How fast is KRCore’s control plane?
2. What are the costs KRCore added to RDMA’s data plane?
3. Can KRCore benefit existing applications that require elasticity?
Evaluations setup

Evaluation setup

1. A rack-scale cluster consists of 10 machines
2. Each with one ConnectX-4 100Gbps RNIC

Comparing targets

1. **Verbs** — the de facto user-space library for using RDMA
2. **LITE**[^1] — kernel-space RDMA solution that use RCQP pool

[^1]: LITE Kernel RDMA Support for Datacenter Applications@SOSP’17
Control plane performance of KRCore

Case #1

- Multiple client connecting to the same server

Bottlenecked by creating hardware queues

Skip user-space driver loading

Connection latency (us)

Connection creation throughput
Control plane performance

Case #1
- Multiple client connecting to the same server

Case #2
- Creating full-mesh connections

Connection latency (us)

Connection creation throughput

Number of clients connected

Connection time (us)
Data plane performance

Workload: synchronous one-sided RDMA

- The client keeps sending one-sided RDMA READ to a server in a run-to-completion way
- Request payload: 8B
Data plane performance

Workload: synchronous one-sided RDMA

- The client keeps sending one-sided RDMA READ to a server in a run-to-completion way
- Request payload: 8B

![Graph showing data plane performance with throughput on the x-axis and median latency (us) on the y-axis. The graph compares LITE, verbs, and KRCore. System call latency (~1us) is indicated.]
Data plane performance

Workload: synchronous one-sided RDMA

- The client keeps sending one-sided RDMA READ to a server in a run-to-completion way
- Request payload: 8B

![Graph showing median latency vs. throughput for different workloads. The graph indicates that LITE and KRCore perform better than verbs, with LITE having the lowest latency. Additionally, there is a note indicating that DCT has reconnection overhead.]
Accelerating disaggregated RDMA–enabled KVS

Target: RACEHashing@ATC’21
Summary and discussion

A microsecond-scale RDMA control plane
- By retrofitting DCT with kernel-space RDMA connection pool

Limitation
- KRCore trades data path due to kernel interception & DCT overhead
- Thus, it does not suit all the application scenarios
Conclusion of KRCore

The first to provide a microsecond-scale RDMA control plane

- While compatible to existing RDMA hardware/software

Elastic application can benefit from KRCore with little data path costs

- E.g., RDMA for disaggregated key-value store, serverless computing

Thanks & QA