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

Xingda Wei[1,2], Fangming Lu[1], Rong Chen*[1,2], and Haibo Chen[1]
[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Shanghai AI Laboratory

Abstract

We present KRCORE, an RDMA library with a microsecond-scale control plane on commodity RDMA hardware for elastic computing. KRCORE can establish a full-fledged RDMA connection within 10μs (hundreds or thousands of times faster than verbs), while only maintaining a (small) fixed-sized connection metadata at each node, regardless of the cluster scale. The key ideas include virtualizing pre-initialized kernel-space RDMA connections instead of creating one from scratch, and retrofitting advanced RDMA dynamic connected transport with static transport for both low connection overhead and high networking speed. Under load spikes, KRCORE can shorten the worker bootstrap time of an existing disaggregated key-value store (namely RACE Hashing) by 83%. In serverless computing (namely Fn), KRCORE can also reduce the latency for transferring data through RDMA by 99%.

1 Introduction

The desire for high resource utilization has led to the development of elastic applications such as disaggregated storage systems [52, 16, 67]. Elasticity provides a quick increase or decrease of computing resources (e.g., processors or containers) based on application demands. Since the resources are dynamically launched and destroyed, minimizing the control path overheads—including process startup and creating network connections—is vital to applications, especially those with ephemeral execution time. Elastic applications typically have networking requirements. For instance, computing nodes in a disaggregated storage system access the data stored at the storage nodes across the network.

RDMA is a fast networking feature widely adopted in datacenters [53, 19, 13]. Unfortunately, RDMA has a slow control path: the latency of creating an RDMA connection (15.7ms) is 15,700X higher than its data path operation (see Figure 1(b)). As the latency of typical RDMA-enabled applications that require elasticity has reached to microsecond-scale (see Figure 1(a)), this high connection time may significantly decrease the application efficiency, e.g., increasing latency when expanding resources to handle load spikes. The cost is challenging to reduce because it not only includes software data structure initialization costs but also involves extensive hardware resource configurations, as RDMA offloads network processing to the network card (§2.3.1).

*Rong Chen is the corresponding author (rongchen@sjtu.edu.cn)

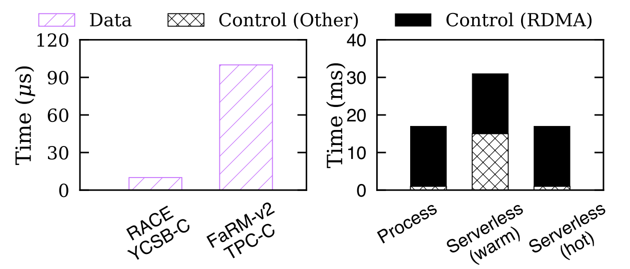

Fig. 1. (a) The execution time (**Data**) of typical elastic RDMA-enabled applications, and (b) the breakdown of control path costs. RACE [67] is a disaggregated key-value store. FaRM-v2 [46] is a database that can accelerate serverless transactions [63]. YCSB-C [11] and TPC-C [50] are representative benchmarks for each system. The serverless platform evaluated is Fn [43].

A common approach to avoiding the control path cost is to cache connections and share them with different applications. However, user-space RDMA connections can not be directly shared by different applications, because each app has its own exclusive driver data structure and dedicated hardware resources. Nevertheless, sharing a kernel-space RDMA connection is possible since applications share the same kernel (LITE [53]). However, LITE has performance and resource inefficiency issues (§2.3.2) in elastic computing, because it doesn't target this scenario. First, it still pays the initialization cost under cache misses. Second, caching all RDMA connections to all nodes is resource inefficient (e.g., taking several GBs of memory), especially when a production RDMA-capable cluster has reached a scale of more than 10,000 nodes [34]. Finally, sharing RDMA connections complicates the preservation of the low-level verbs interfaces, which is important to apply RDMA-aware optimizations [67, 55, 14, 57, 24, 25]. LITE only provides a high-level API.

We continue the line of reusing connections to boost the RDMA control path, and further overcome the issues mentioned above. We present **KRCORE**, a networking library with an ultra-fast control plane. KRCORE can establish a full-fledged RDMA-capable connection within 10 μs, only 0.05% and 0.22% of the verbs and LITE under cache misses, respectively. More importantly, KRCORE only needs a small amount of fixed-sized memory for the connection pool (e.g., 64MB), irrelevant to the cluster scale. Finally, KRCORE supports low-level RDMA interfaces compatible with existing RDMA-aware optimizations.

Supporting such a fast control plane seems to contradict our promise of a small fixed-sized connection pool. To achieve this, KRCORE makes a key innovation: we *retrofit* a less-studied yet widely supported advanced RDMA hardware feature—*dynamic connected transport* (DCT) [1]—*to the kernel*. DCT allows a single RDMA connection to communicate with different hosts. Its connection and re-connection are offloaded to the hardware and thus, are extremely fast (less than 1μs). Our observation is that when virtualizing an established kernel-space DCT connection to different applications, they no longer pay the control path cost and memory consumption of ordinary RDMA connections.

In designing KRCORE, we found virtualizing DCT with a low-level API brings several new challenges, and we propose several techniques to address them (§3.1). First, DCT requires querying a piece of metadata to establish a new connection. Using RPC can not achieve a stable and low latency. Further, RPC needs extra CPU resources to handle DCT-related queries. Observing the small memory footprint of DCT metadata, we propose an architecture that deploys RDMA-based key-value stores to offload the metadata queries to one-sided RDMA READ (§4.2). Second, DCT has a lower data path performance than normal RDMA transport (RC) due to its dynamic connecting feature. The performance is mostly affected when a node keeps a long-term communication with another. Therefore, we introduce a hybrid connection pool that retains a few RC connections connected to frequently communicated nodes to improve the overall performance. KRCORE further adopts a transfer protocol that can transparently switch a virtualized connection from DCT to RC (§4.6). Finally, we propose algorithms to safely virtualize a shared physical QP to multiple applications with a low-level API (§4.4).

We implement KRCORE as a loadable Linux kernel module in Rust. We also extended an existing kernel-space RDMA driver (mlnx-ofed-4.9) to bring DCT to the kernel. To the best of our knowledge, KRCORE is the first to achieve a microsecond-scale RDMA control plane. Although KRCORE is a general-purpose RDMA library, it really shines with elastic computing applications. Our experiments demonstrated that KRCORE can reduce the computing node startup time of a state-of-the-art production RDMA-enabled disaggregated key-value store (RACE [67]) by 83%, from 1.4s to 244ms (§5.3.1). For serverless computing—another popular elastic application, KRCORE can shorten the data transfer time over RDMA by 99%, from 33.3ms to 0.12μs (§5.3.2).

Our source code and experiments are available at `https://github.com/SJTU-IPADS/krcore-artifacts`.

2 Background and Motivation

2.1 The case for fast control path in elastic computing

KRCORE targets systems that require elasticity: the ability to automatically scale according to application demands. One such case is disaggregated storage systems where the computing nodes and storage nodes are separated and connected by the network [52, 16, 67]. Under high loads, the system can dynamically add computing nodes for better performance: and they need to establish connections to the storage nodes on-the-fly. Another important case is serverless computing [22] where the platforms instantaneously launch short-lived tasks with containers[1]. The launch time typically includes network connections [51].

Unlike long-running tasks (e.g., web servers), the control path (e.g., network creation) is typically on the critical path of elastic applications. For example, before executing the application code, a serverless function that issues database transactions must first establish network connections to remote storage nodes [63, 21]. With RDMA, the transaction latency has reached 10-100μs [14, 57]. Reducing the control path costs—including launching a container and creating network connections—is therefore vital to the end-to-end execution time or tail latency of elastic applications (see Figure 1).

Much research has focused on reducing other control path costs, e.g., the container launch time to about 10ms [40] and even sub-millisecond [15]. However, only a few considered accelerating network connection creation [51], especially for RDMA. The control path of RDMA is indeed several orders of magnitude slower than its data path (e.g., 22ms vs. 2μs in §2.3). It is also orders of magnitude slower than the execution time of common elastic RDMA-enabled applications, or other control path costs (see Figure 1).

2.2 RDMA and queue pair (QP)

RDMA is a high bandwidth and low latency networking feature widely adopted in modern datacenters [53, 19]. It has two well-known primitives: *two-sided* provides a message passing primitive while *one-sided* provides a remote memory abstraction—the RDMA-capable network card (RNIC) can directly read/write server memory in a CPU-bypassing way.

Although RDMA is commonly used in the user-space, the kernel adopts the same *verbs* API (verbs), which exposes network connections as queue pairs (QPs). Each QP has a send queue (*sq*), a completion queue (*comp_queue*), and a receive queue (*recv_queue*). Both primitives follow a similar execution flow. To send a request (or a batch of requests), the CPU uses `post_send` to post it (or them) to the *send queue*. If the request is marked as *signaled*, the completion can be polled from the *completion queue* via `poll_cq`. For two-sided primitive, the CPU can further receive messages with `poll_cq` over the *receive queue*. Before receiving, one should use `post_recv` to post message buffers to the QP. Note that the CPU needs to register memory through `reg_mr` to give RNIC memory access permissions.

QP has several kinds of transport each with different capabilities. We focus on improving the control path performance of *reliable connected* QP (RCQP), as it is the most commonly used one that supports both RDMA primitives and is reliable.

[1] Serverless platforms may use virtual machine (VM)s to run tasks, which is not the focus of our paper. §6 discusses how KRCORE can apply to VMs.

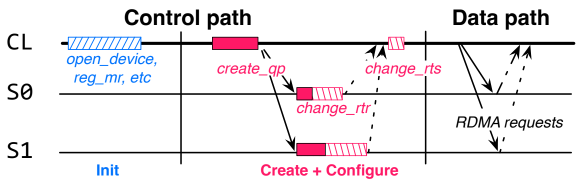

Fig. 2. The execution flow of a client (**CL**) communicating with two nodes (**S0** and **S1**) using user-space verbs. `change_rtr` changes the QP to ready to receive status while `change_rts` changes the QP to ready to send status.

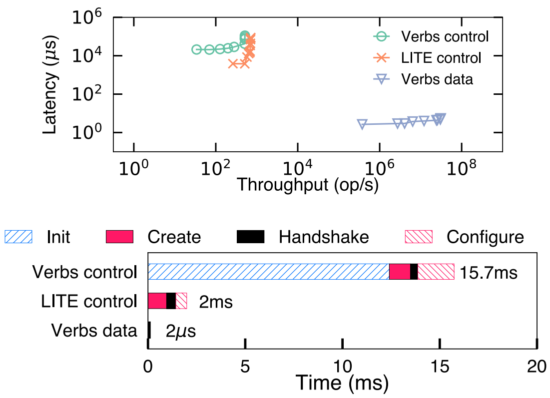

Fig. 3. (a) Huge performance gap btw. RDMA's control path and data path (issuing 8B READ) when connecting and communicating with one node. (b) A breakdown of RDMA control path time.

2.3 Analysis of RDMA control path costs

2.3.1 User-space control path costs

Consider the example in Figure 2 where a client sends RDMA requests to two nodes. The control path includes first initializing the driver context (**Init**)[2], creating the QPs (**Create**), exchanging the QP information to the remote peer with a **handshake** protocol and configuring the QPs to ready states (**Configure**). Figure 3(a) reports its latency, which is 7,850X higher than the data path (Verbs control vs. Verbs data).

Issue: High hardware setup cost. To quantify the costs in detail, Figure 3 breakdowns the control path time. We carefully optimize the connection handshake with RDMA's connectionless datagram [26], which is orders of magnitude faster than using TCP/UDP. Contradicting the common wisdom, exchanging the connection information through the network (**Handshake**) is **not** the dominant factor: **Handshake** only contributes 2.4% of the total time. The cost is dominated by communicating with the RNIC hardware for the connection setups. Consider the `create_qp` in **Create**: we found 87% of the `create_qp` time (361μs vs. 413μs) is waiting for the RNIC to create the hardware queues.

2.3.2 Existing kernel-space solution is insufficient

LITE [53] is the only kernel-space RDMA solution and is the closest to our work. It provides high-level remote memory *read*, *write* and *RPC* interfaces over the low-level verbs API (§2.2). LITE maintains an in-kernel connection pool that

[2]Including creating the protection domain and registering the memory.

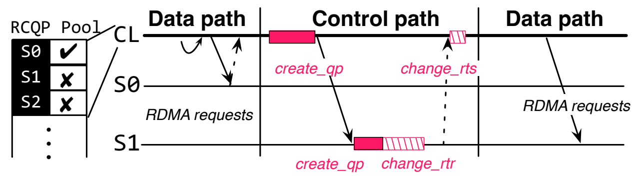

Fig. 4. The execution flow of a client (**CL**) communicating with two nodes (**S0** and **S1**) with the kernel-space RDMA assuming that **CL** has cached a QP to **S0** in its connection pool.

caches RCQPs connected to all nodes, which avoids the user-space **Init** (Figure 2) costs because applications share the same kernel-space driver data structures. However, it still has the following **issues** for elastic applications:

Issue#1: High cost connecting to a new node. If the RCQP of the target node is not cached, LITE must follow the same **Create** and **Configure** as user-space RDMA, e.g., S1 in Figure 4, which are non-trivial (2ms for each connection). Note that we have carefully optimized LITE's control path: LITE originally adopts a centralized cluster manager to create connections, which can only establish tens of QPs per second. We optimize it with a decentralized connection scheme using RDMA's connectionless datagram. The optimization achieves a 2ms per-connection latency and 712 QPs/second per node throughput (Figure 3), bottlenecked by the RNIC (see §2.3.1).

Issue#2: Huge memory consumption. Caching RCQPs connected to all other nodes can mitigate Issue#1. However, this strategy has huge per-machine memory consumption since the number of RCQPs needed scales linearly with the cluster size. In LITE, each QP consumes at least 159KB memory[3], excluding the message buffers and receive queues (may share between different QPs via shared receive queue). Therefore, LITE would consume at least 1.52 GB memory per node for fast connection on a modern RDMA-capable cluster with more than 10,000 nodes[17].

Issue#3: Inflexible interface. LITE exposes a high-level RDMA API (e.g., a synchronous remote memory read), which simplifies sharing the same QP to different applications. However, it is inflexible to apply RDMA-aware optimizations widely adopted in the literature [67, 55, 14, 57, 24, 25], e.g., sending different read/write requests within a batch asynchronously. To utilize these optimizations, applications need verbs low-level API (§2.2). Unfortunately, directly executing the low-level API on a shared QP can easily corrupt the QP states (see §3.1), and interrupt application running. We carefully design the QP virtualization algorithms to correctly virtualize a shared QP with verbs's low-level API (§4.4).

[3]It configures the QP with 292 *sq* and 257 *comp_queue* entries, a common setup in RDMA-based systems. Each *sq* entry takes 448B while *cq* takes 64B. The driver would further round queues to fit the hardware granularity.

3 Approach and Overview

Opportunity: *advanced RDMA transport (DCT).* Dynamically Connected Transport (DCT) [1] is an advanced RDMA feature widely supported in commodity RNICs (e.g., from Mellanox Connect-IB [37] to ConnectX-7 [35]). DCT preserves the functionalities of RC and further supports dynamic connecting: a DCT QP (DCQP) can communicate to different nodes without user-initiated connections: RNIC can create DCT connections on-the-fly by piggybacking control plane messages with data plane ones. Since the connections are only processed in the hardware, DCT re-connection is extremely fast: our measured overhead is less than 1μs. When using DCQPs, the host only needs to specify the target node's RDMA address and its DCT metadata (i.e., DCT number and DCT key) in each request.

Basic approach: *virtualized kernel-space DCQP.* The goal is to achieve an ultra-fast control plane for the applications. Our basic approach is to virtualize kernel-space DCQPs (as VQPs) to user-space applications. The observation is that DCT naturally addresses the costly creation overhead (**Issue#1**) and the huge memory consumption (**Issue#2**) of RCQPs (§2.3.2). A kernel-space solution further mitigates the user-space driver loading costs (§2.3.1).

VQP also supports low-level RDMA interfaces (e.g., `ibv_post_send`) with the necessary extended API suitable for elastic computing (§4.1). Therefore, users can flexibly apply existing RDMA-aware optimizations [24, 25, 57] (**Issue #3** in §2.3.2). Note that different VQPs can share the same physical QP in the kernel. Nevertheless, KRCORE provides an exclusively owned QP abstraction to the applications.

3.1 Challenges and solutions

C#1. Efficient DCT metadata query. DCQP needs to query the DCT metadata before sending requests. Specifically, to allow communicating with DCT, the server must first create a *DCT target* identified by a key and number (DCT metadata). Afterward, the clients can piggyback the metadata in their requests to communicate with the created target.

A viable solution is to send an RPC to the target node to query the metadata using RDMA's connectionless datagram (UD)[4], which prevents control plane costs as UD is connectionless. However, it is inefficient in performance and CPU usage. First, the latency of RPC may vibrate to tens of milliseconds due to the scheduling and queuing overhead of the CPU. Second, KRCORE must deploy extra kernel threads to handle the queries.

Solution: RDMA-based meta server. We replicate the DCT metadata at a few global meta servers backed by RDMA-enabled key-value stores (KVS) [67, 58, 55, 13], meaning each node can query it with one-sided RDMA bypassing the CPU. To support one-sided RDMA while preventing QP overprovisions, KRCORE only maintains a few RCQPs connected to nearby meta servers. Replicating the DCT metadata is practical because it is small: 12B is sufficient for one node to handle all requests from others.

[4]It only supports two-sided RDMA.

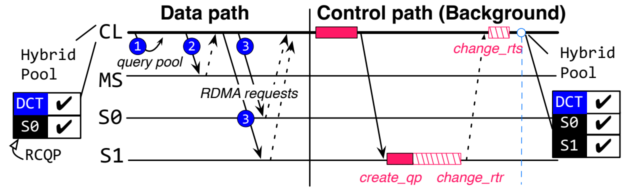

Fig. 5. The execution flow of a client (**CL**) communicating with two nodes (**S0** and **S1**) with KRCORE. **MS**: meta server. Note that KRCORE always put the hardware control path (i.e., creating RC-QPs) in the background.

C#2. Performance issues of DCT. DCT is slower than RC in peak throughput and may incur high tail latency due to reconnection (§5.2). The performance is mostly affected when a node frequently sends requests to the same node.

Solution: virtualized hybrid QP. KRCORE manages a hybrid QP pool that stores both RC and DC QPs. A VQP can transparently switch between DC and RCQP (§4.6), allowing us to create RCQPs in the background on-the-fly without exposing the creations overhead to the applications.

C#3. QP state protection. If we directly forward the VQP request (from `ibv_post_send`) from different applications to the (same) shared physical QP, QP's physical states can easily be corrupted due to malformed requests or queue overflow, because verbs API assumes an exclusively owned QP. Bringing the QP back to a normal state is costly because it requires reconfiguration (the **Configure** in Figure 3 (b)).

Solution: pre-check. KRCORE carefully checks the physical queue capacity and request integrity before forwarding the requests to the physical QP. The overhead of these checks is negligible as they only involve simple calculations. Thus, we can avoid QP corruption while preserving the RDMA-aware optimizations (§4.4) of using low-level interfaces.

3.2 Execution flow and architecture

Execution flow. Applications can use KRCORE to create RDMA-capable connections in a few microseconds. Figure 5 presents its execution flow when communicating to two nodes. First, we find available RCQPs in the hybrid pool (❶). If exists (S0), we directly virtualize it. Otherwise (S1), we choose a DCQP and fetch the target node's DCT metadata (❷) accordingly. Finally, we virtualize the selected QP so that the client can send RDMA requests with them (❸).

To increase the likelihood of hitting RCQPs, KRCORE analyzes the host's networking patterns and creates RCQPs in the background (e.g., to S1).

Architecture. Figure 6 presents the KRCORE library architecture. On each node, KRCORE is a loadable Linux

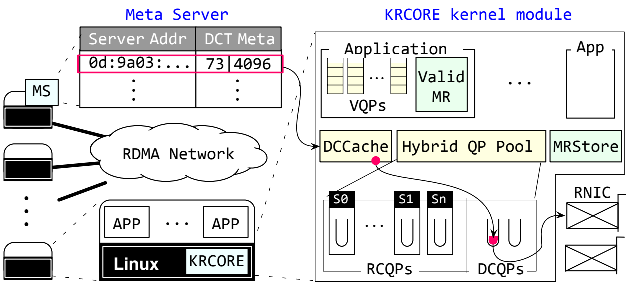

Fig. 6. An overview of KRCORE architecture.

KRCore's extended verb's control path API

```
int qbind(ibv_qp *qp, ibv_gid gid, int port); ## like POSIX bind
int qconnect(ibv_qp *qp, ibv_gid gid, int port); ## like POSIX connect
```

KRCore's extended verb's data path API

```
int qpop_msgs(ibv_qp *qp, int num_entries,    ## like POSIX(accept) +
              ibv_qp **src_qp, ibv_wc *wc);   ## verbs(ibv_poll_cq)
```

Example code: Client

```
ibv_qp_init_attr attr;
attr.qp_type = KRCORE_VQP;

qp = ibv_create_qp(..., &attr);

qconnect(qp, gid, port);

ibv_send_wr wr;
ibv_send_wr *bad_wr_ptr;
## send a message
ibv_post_send(qp, &wr, &bad_wr);
```

Example code: Server

```
ibv_qp_init_attr attr;
attr.qp_type = KRCORE_VQP;

qp = ibv_create_qp(..., &attr);

qbind(qp, gid, port);

ibv_qp *new_conn = NULL;
ibv_wc wc;
## receive a message
qpop_msg(qp, 1, &new_conn, &wc);
```

Fig. 7. The KRCORE extended API atop of verbs and a simplified use case. Lines in [green] and [pink] are extended code for the client and server, respectively. Applications can also use the verb's data path call (e.g., `ibv_post_send`) to issue RDMA requests with KRCORE.

kernel module hosting per-application (e.g., VQP) and per-node (e.g., Hybrid QP Pool) data structures (§4.2). KRCORE also deploys meta servers (MS) on a few nodes to facilitate DCT metadata lookup. These servers are backed by DrTM-KV [58]—a state-of-the-art RDMA-enabled KVS—to accelerate the metadata lookup. The metadata is broadcasted by each machine during its boot time.

4 Detailed Design

4.1 Programming interface of KRCORE

To simplify application development and porting, it is important to keep backward compatibility between KRCORE and verbs, the de facto standard for using RDMA. In principle, KRCORE can provide the same interface with verbs similar to existing work (i.e., Freeflow [30]). However, verbs is not designed for elastic computing and may bring inflexibility or under-utilization of KRCORE. Therefore, we propose an extended API based on verbs inspired by Demikernel [64], as shown in Figure 7. Specifically, KRCORE introduces a new type of QP (*VQP*) with the following new primitives:

`qconnect` and `qbind`. The verbs API has no method for 'connect' commonly found in networking libraries. Therefore, developers have to implement and optimize RDMA connection setups themselves. We provide a `qconnect` API to abstract the fast connection provided by KRCORE. Specifically, after calling `qconnect` on a VQP to a remote host (identified by the RDMA address (gid) and a port), the VQP can issue one-sided and two-sided requests to it. Note that remote end must bind to the address using `qbind` beforehand so that the sender can issue two-sided requests, similar to POSIX `bind`.

`qpop_msgs`. RCQPs are one-to-one connected—meaning the server must know how many clients may connect. This is unhandy for elastic applications because clients can dynamically connect to a server. Therefore, KRCORE VQP is many-to-one: after binding to an address, a VQP can dynamically accept new connections when receiving messages: `qpop_msgs` will return a list of $(src\_qp, message)$ pairs, where the src_qp is a VQP connected to the corresponding sender of the $message$.

Besides the extended API, KRCORE also supports common verbs data path API, e.g., `ibv_post_send`, `ibv_post_recv` and `ibv_poll_cq` (see §2.2). Figure 7 showcases a simplified code example of sending a message from a client to a server with VQP. At the client, it can use `KRCORE_VQP` as a marker to create a VQP. After successfully connecting the VQP with `qconnect`, the client can call `ibv_post_send` to send the message.

Note that the VQP has the semantic as RCQP—meaning that they have reliability guarantees and support all RDMA operations (with various low-level optimizations).

4.2 Data structures

Hybrid QP pool. Each VQP (§4.1) is backed by a kernel-space virtual QP that has an identifier, a reference to a physical QP and virtualized counterparts of RDMA queues (see §2.2). The physical QP is selected from a hybrid QP pool with both DCQPs and RCQPs. The DCQPs are statically initialized upon boot time and RCQPs are created on-the-fly.

In principle, the pool only needs one DCQP to handle all the RDMA requests of the host. However, only using one DCQP introduces extra latency when sending concurrent requests to different servers. Specifically, if two requests targeting different hosts go over the same DCQP, the second must wait for an additional reconnection before RNIC can process it. This can be mitigated by increasing the DCQP pool size since reconnections can run concurrently. Yet, the best choice of the pool size depends on the hardware setting (§5.2). On our platform, we choose 8 DCQPs in the pool.

To further prevent lock contention [26], we divide the pool on a per-CPU basis: Each VQP only virtualizes QPs from its local CPU's pool. This strategy is optimized for cases when each QP is exclusively used by one thread, a common pattern in RDMA applications [47, 55, 26, 17, 33]. In case of thread migrations, KRCore also re-virtualizes QPs in the background with a transparent QP transfer protocol (§4.6).

Meta Server. For steady and low-latency DCT metadata query, we replicate all the nodes' metadata at a few global meta servers backed by DrTM-KV [58], a state-of-the-art RDMA-enabled KVS. Note that replicating all the DCT meta at one server is practical because they are extremely small (e.g., 17KB for a 1,000-server cluster).

The meta server stores a mapping between the RDMA address (key) and its corresponding DCT number and key (value). These key-value pairs can be queried via DrTM-KV with a few one-sided RDMA READs. Since sending one-sided requests also requires RDMA connections, each node pre-connects to nearby meta servers (e.g., one in the same rack) with RCQPs during boot time and thus, it can find the DCT metadata of a given server in several microseconds even under high load.

Optimization: DCCache. Observing that the DCT metadata is extremely small (12B), each node further caches them locally to save network round-trips querying the meta server. The metadata is suitable for caching because they are only invalidated when the corresponding host is down.

ValidMR and MRStore. To safely virtualize a physical QP to multiple VQPs, KRCore additionally checks the validity of remote memory accesses to prevent QP state corruption (§4.4). These checks were originally done by the RNIC using the information stored in the NIC cache. Thus, we should also record them in KRCore. We additionally bookkeep the registered memory regions (MR)s in ValidMR, which is also implemented with DrTM-KV. After the bookkeeping, KRCore can query the local/remote ValidMRs to check the local/remote memory regions' validity.

Like DCCache, we also cache the checked remote MR locally (in MRStore) to avoid extra round-trips. However, caching remote MRs may introduce consistency problems: unlike long-lived DCT metadata, MRs are managed by the applications and can be de-registered on-the-fly. To this end, KRCore adopts a lease-based lightweight invalidation scheme: the cached MRs are periodically (e.g., 1 second) flushed. Upon de-registration, KRCore waits for this period before freeing the MR.

4.3 Control path operations

KRCore reuses initialized QPs upon VQP connection and creation, whose simplified pseudocode executed in the KRCore kernel is shown in Algorithm1.

`vqp_create` initializes the basic data structures of VQP—mainly allocating the software send and completion queues in the kernel. The physical QP assignment is delayed to the VQP connection (line 5) because we are unaware of the remote target during creation.

`vqp_connect` connects a VQP to a remote end by assigning a pre-initialized kernel-space QP (either RCQP or

Algorithm 1: VQP creation and connection

```
1:  Function vqp_create(Q):
2:      Q.id ← allocate a free identifier
3:      Q.comp_queue ← allocate a software queue
4:      Q.recv_queue ← allocate a software queue
5:      Q.qp ← NULL  ◁ Updated by qconnect
6:  Function vqp_connect(Q, addr):
7:      if Q.qp == NULL then
8:          if addr in HybridQPPool.RC then
9:              Q.qp ← select in HybridQPPool.RC[addr]
10:         else
11:             Q.qp ← select in HybridQPPool.DC
12:             if addr not in DCCache then
13:                 meta ← query nearby connected MetaServer
14:                 add meta to DCCache
15:                 Q.dct_meta ← meta
```

DCQP) to it. Given the remote $addr$, it first checks whether an RCQP is available in the HybridQPPool (line 8). If so, we choose an available QP and assign it to $Q.qp$ (line 9). Otherwise, we select a DCQP (line 11). Note that all DCQPs in the pool are available because KRCore can virtualize one physical QP to multiple VQPs (§4.4).

When assigning a DCQP to VQP, we need to fetch the remote end's DCT metadata (line 12−15) if the metadata is not cached in the $DCCache$. We issue one-sided RDMA READs to the $MetaServer$ to query it (line 13).

Background RCQP creations. To increase the likelihood of hitting an RCQP in the pool, KRCore maintains background routines to sample frequently communicated nodes, create RCQPs for frequently communicated ones in the $HybridQPPool$ and reclaim rarely used RCQPs. Currently, we choose a simple LRU strategy for the reclamation.

Other control path operations. Besides VQP creation and connection, other control path operations (e.g., memory registration, MR) have a straightforward implementation: we forward them to the corresponding verbs API and record the results in KRCore. If necessary, we will also return the virtual handler of the recorded results to the user. Due to space limitations, we omit a detailed description.

4.4 Data path operations

As we have mentioned in §3.1, a key challenge in virtualizing a physical QP to multiple VQPs is preventing shared QP state corruption. Specifically, we must consider:

1. **Detecting malformed request.** An incorrect operation code or an invalid memory reference would transit a QP into error states. Since an error states QP cannot handle any RDMA requests, we must filter out malformed requests before posting them to the physical QP.
2. **Preventing NIC queue overflow.** The physical QP has a limited queue capacity. If the user overflows a QP, the QP will also enter an error state. Preventing queue overflow is challenging under sharing because it can overflow even if all the shared users correctly avoid the queue overflows.

Algorithm 2: kernel handler of post_send and poll_cq

```
1:  Function post_send_virtualized(Q, wr_list):
        ◁ wr_list: the RDMA requests list
        ◁ Assumption: the size of wr_list is smaller
          than Q.qp.sq.max_depth and Q.qp.cq.max_depth
2:      while Q.qp.sq.max_depth - Q.qp.uncomp_cnt <
          wr_list.length do
3:          poll_inner(Q)
4:      unsignaled_cnt ← 0
5:      for req in wr_list do
6:          if req has invalid MR or invalid Op then
7:              return Error
8:          if req is signaled then
9:              Q.comp_queue.add(NotReady, req.wr_id)
10:             req.wr_id ← encode the pointer of Q and
                  (unsignaled_cnt + 1)
11:             unsignaled_cnt ← 0
12:         else
13:             unsignaled_cnt += 1
14:         Q.qp.uncomp_cnt += 1
15:     if last_req in wr_list is not signaled then
16:         mark last_req as signaled
17:         last_req.wr_id ← encode NULL and
              (unsignaled_cnt + 1)
18:     return post_send(Q.qp, wr_list)
19: Function poll_inner(Q):
20:     wc ← poll_cq(Q.qp.cq)
21:     if wc is ready then
22:         VQ, comp_cnt ← decode wc.wr_id
23:         Q.qp.uncomp_cnt −= comp_cnt
24:         if VQ is not NULL then
25:             VQ.comp_queue.head()[0] = Ready
26: Function poll_cq_virtualized(Q):
27:     poll_inner(Q)
28:     if Q.comp_queue.has_head() and
          Q.comp_queue.head()[0] is ready then
29:         user_wr_id ← Q.comp_queue.pop()[1]
30:         return READY, user_wr_id
31:     return NULL, 0
```

The queue can be cleared via explicit signaling and polling. Nevertheless, we should poll as little as possible because they have overheads [24].

3. **Dispatching completion events.** The polled results of a physical QP can be from different VQPs. Therefore, we must correctly dispatch them to the targets, i.e., software queues of VQPs.

To this end, KRCore will (1) check the request integrity before posting it to a shared QP; (2) inject necessary polls to the physical QP and (3) encode the VQP information in the request's wr_id—that will be returned upon request completion—to help the dispatch. Specifically, KRCore executes `post_send_virtualized` and `poll_cq_virtualized` after the user calls `ibv_post_send` and `ibv_poll_cq`, respectively. Algorithm 2 shows their simplified pseudocode. For simplicity, we assume the request list (wr_list) depth is smaller than the QP capacity, which can be achieved by segmenting the request list before posting it.

`post_send_virtualized`. It first clears the physical QP's send and completion queues to prevent overflows (line 2−3) via polling the physical completion queue (line 20). Polling is tricky when considering unsignaled requests—the requests that don't generate completion events. Their entries are freed until a later signaled request is polled. Thus, we must track how many requests a signaled one is responsible to clear (line 4 and line 13), and encode the number in wr_id (line 10). Therefore, after polling a completion we can determine the left spaces of queues (line 23). Further, if the last request is unsignaled, we signal it (line 15−17).

For each request, we also check whether it is malformed (line 6) and record the dispatch information for the signaled ones (line 9−10). Finally, we can safely post these requests to the physical QP (line 18).

For two-sided primitive, KRCore must additionally notify the receiver the sender information. Otherwise, the receiver cannot create proper connections in `qpop_msgs`. Hence, we piggyback the sender's address in the message header (omitted in the algorithm).

`poll_cq_virtualized`. It first calls `poll_inner` to poll the physical QP events and dispatch the events to the proper VQPs according to the information recorded in the wr_id (lines 22−25). After the dispatch, it can check whether the virtualized QP has a completion event. KRCore examines the head of the virtualized comp_queue and returns the head's wr_id to the application if the head exists.

Due to space reasons, we briefly describe other operations:

`ibv_post_recv`. This function registers the buffers to the VQP by recording them in the virtualized recv_queue.

`qpop_msgs`. It polls the physical QP's $recv\_queue$ and dispatches the received messages, similar to `poll_inner`. To hold in-coming messages, we pre-post message buffers to physical QP before virtualizing it to the applications. The challenge of pre-post is that the KRCore doesn't know the exact payloads of the incoming messages. For now, we assume the pre-posted buffers can always hold the incoming message. §4.5 will describe how we cope with out-of-bound messages in detail. After receiving a message, we will check its destination VQP and copy it the user-registered buffer (from `ibv_post_recv`).

Besides receiving messages, `qpop_msgs` also creates a VQP connected to the sender (§4.1). The creation and connection follow the control path operations discussed in §4.3. To prevent the DCT metadata query, we further piggyback the metadata in the message header. Thus, `qpop_msgs` doesn't involve additional networking requests.

4.5 Zero-copy protocol for two-sided operations

The basic `qpop_msgs` (§4.4) has two issues. First, it incurs extra memory copies. Though the copy overhead is negligible for small messages (e.g., less than 1KB), it is non-trivial for the large ones (e.g., see results in Figure 9 (b)). Second, it cannot receive messages with payloads larger than the pre-posted buffers.

To this end, we adopt a zero-copy protocol to overcome the above issues. Intuitively, for large or out-of-bound messages, the receiver will use one-sided RDMA READ to read them to the user-registered buffers, inspired by existing RDMA-enabled RPC frameworks [48, 17]. Specifically, if the payload is larger than the kernel's registered buffer, the sender will first send a small message containing the destination VQP ID, the source message address and its size. The receiver can then use one-sided RDMA READ to read the message directly to the user-registered buffer in a zero-copy way. The cost of sending an additional message is trivial for large messages because the network transfer will dominate the time.

4.6 Physical QP transfer protocol

KRCORE supports seamlessly changing the physical QP virtualized by a VQP to another. The challenge of doing so is how to preserve the RCQP's FIFO property [7] of the VQP during transfer, i.e., after a request completes, all its previous requests are finished.

To ensure FIFO, upon the transfer starts, we first post a fake signaled RDMA request to the source QP and wait for its completion before the change. Meanwhile, we also notify the remote peers to transfer their physical QP. Otherwise, the VQP can no longer receive the remote end's message. For correctness, we must wait for the remote acknowledgments before changing the physical QP at the sender.

5 Evaluation

We aim to answer the following questions during evaluations:

1. How fast is the KRCORE control plane (§5.1)?
2. What are the costs to the data plane (§5.2)?
3. How RDMA-aware applications that require elasticity can benefit from KRCORE (§5.3)?

Implementation. We implement KRCORE from scratch as a loadable Linux kernel (4.15) module, which has more than 10,000 LoC Rust code. It exports system calls via *ioctl* without modifying the kernel. To simplify user-kernel interactions, we further implement a 100 LoC C shim library atop ioctl to provide the interfaces described in §4.1. Finally, we port DCT to the kernel-space RDMA driver by adding around 250 LoC C code to the mlnx-ofed-4.9 driver: DCT is currently only implemented in the user-space RDMA drivers.

Testbed setup. We conduct experiments on a local rack-scale RDMA-capable cluster with ten nodes. Each node has two 12-core Intel Xeon E5-2650 v4 processors, 128GB DRAM and one ConnectX-4 MCX455A 100Gbps InfiniBand RNIC. All nodes are connected to a Mellanox SB7890 100Gbps InfiniBand Switch. Without explicit mention, we deploy one meta server for KRCORE.

Comparing targets. We compare KRCORE with user-space verbs (**verbs**) and LITE[5]. Original LITE has an unoptimized control plane: it uses a centralized cluster manager to

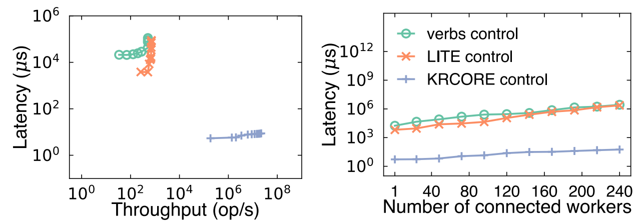

Fig. 8. The `qconnect` performance of KRCORE when using DCQP with DCT metadata uncached. (a) Connecting to a single server, and (b) establishing connections in a full-mesh fashion.

establish connections between servers and can only connect tens of RCQPs per second. Therefore, we further optimize it by enabling a decentralized QP connection scheme via RDMA's unreliable datagram (UD). Our optimized version can achieve an optimal kernel-space RDMA control plane performance—it is now only bottlenecked by the hardware limits. §5.1 will describe this in more detail. Note that our optimization leaves the LITE data plane unchanged.

5.1 Control path performance

The evaluations for the control path focus on creating and connecting RDMA connections. The costs of the other operations in KRCORE (and verbs) are typically much smaller. For example, registering 4MB memory only takes 1.4μs in KRCORE. Therefore, we omit their results.

We use two synthetic workloads (single and full-mesh connection establishment) to evaluate the control path performance. The connection pool and DCCache of KRCORE are cleared before the evaluations. Otherwise, KRCORE only has system call overheads and is extremely small (0.9μs).

Single-connection establishment performance. We first evaluated the latency and throughput of establishing a single RDMA-capable connection to one server w.r.t. the number of clients. Figure 8 (a) reports the throughput-latency graph when increasing the number of clients from 1 to 240. From the figure we can see that KRCORE can have several orders of magnitude better performance than verbs and LITE. At one client, KRCORE can establish a connection in 5.4μs, while verbs and LITE take 15.7ms and 2ms, respectively. The performance gain of KRCORE comes from replacing the costly RDMA control path operations (analyzed in §2.3.1 and §2.3.2 in detail) with fast RDMA data path operations, i.e., two one-sided RDMA READs to the meta server. For LITE, it saves the driver loading cost but still needs to create and configure QP on its control path. At 240 clients, KRCORE can handle 22 million (M) connections per second, while verbs and LITE can only establish 712 RCQPs per second. They are both bottlenecked by the server creating hardware resources, while KRCORE always reuses existing ones to prevent these overheads.

Full-mesh connection establishment performance. Besides establishing a single connection, creating full-mesh connections at a set of workers is common in elastic applica-

[5] https://github.com/WukLab/LITE

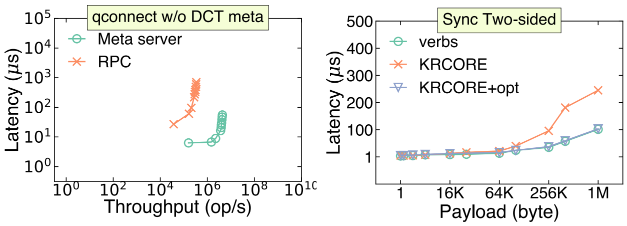

Fig. 9. (a) Performance comparisons of different DCT meta query methods, and (b) the effects of zero-copy protocol (KRCORE+opt) of KRCORE two-sided operations.

tions, e.g., burst-parallel serverless workloads [51]. Specifically, each worker should connect to the others and vice versa. Figure 8 (b) presents the full-mesh performance by varying the number of involved workers. In general, KRCORE can reduce 99% of the full-mesh creation time regardless of the worker number, thanks to the orders of magnitude faster single-connection establishment performance (see Figure 8 (a)). For example, KRCORE connected 240 workers in 81 μs, while verbs and LITE used 2.7 secs and 2.3 secs, respectively. These results suggest that KRCORE can handle complex control path operations well.

Benefit of the meta server. A key design choice of KRCORE is to use an RDMA-based meta server to store DCT meta. Figure 9 (a) illustrates the benefit of this design using the single-connection establishment workload of Figure 8 (a). The baseline (RPC) uses a kernel-space FaSST [26] RPC for the querying. FaSST is the state-of-the-art RDMA-based RPC that builds on RDMA's unreliable datagram. It also has no control plane overhead in the kernel because UD is connectionless. To save CPU resources, we only deploy one kernel thread to handle the queries. We can see that a meta server design achieves an 11.8X better throughput and up to 13X query latency compared with RPC. The RPC design is bottlenecked by the server CPU for handling DCT queries, while the RDMA-based meta server bypasses the CPU with one-sided RDMA.

5.2 Data path performance

KRCORE trades data path performance for a faster control plane. We first use a set of microbenchmarks to evaluate these overheads using two communication patterns: **sync** and **async**. In the sync mode, each client issues RDMA requests to one server in a run-to-completion way, aiming to achieve low latency [17, 47]. For async, each client posts requests in batches to achieve the peak throughput [57, 24, 25]. Without explicit mention, the workloads are inbound, i.e., multiple clients sending RDMA requests to one server. We reported the aggregated throughput of clients and their average latency.

One-sided operations. Figure 10 presents the one-sided data path performance of KRCORE when it virtualizes from DCQP (KRCORE(DC)) and RCQP (KRCORE(RC)), and compare them to verbs[6]. During the experiment, each client

[6]LITE's data path API is different so we compare to it separately.

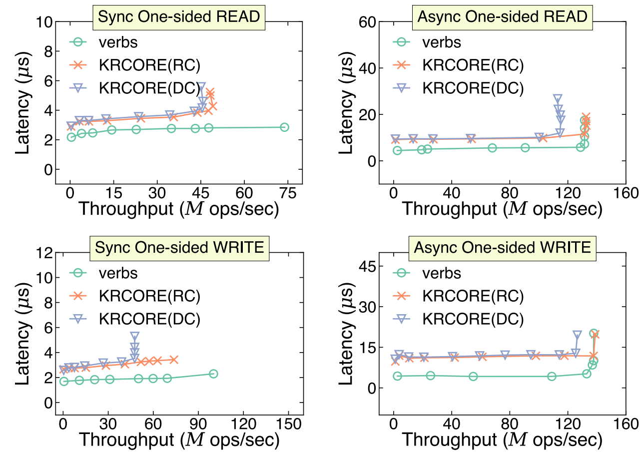

Fig. 10. The one-sided RDMA performance.

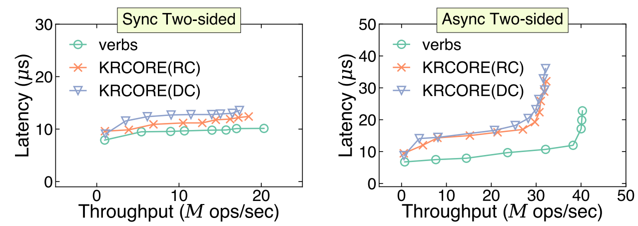

Fig. 11. The two-sided RDMA performance of KRCORE.

issued 8B random requests to the server, and we varied the number of clients from 1 to 240.

(1) Sync. For one-sided RDMA READ in Figure 10 (a), the latency of KRCORE (DC) and (RC) is 27%–46% and is 25%–41% higher than verbs. The additional latency of KRCORE under sync mode is dominated by the system call cost. On our hardware, we measure a $\sim$1μs overhead communicating with the kernel. For reference, when using one client, the latency of KRCORE (RC) is 3.15μs, and the verbs is 2.15μs. Another observation is that adopting DCQP has little latency overhead in the sync mode as DC reconnection is extremely fast. For example, the latency of KRCORE (DC) under one client is 3.24μs. The results of one-sided RDMA WRITE in Figure 10 (c) are similar to the READ.

(2) Async. For one-sided RDMA READ in Figure 10 (b), KRCORE (RC) can achieve a similar peak throughput as verbs (138M reqs/sec) when using 240 clients. With the same configuration, KRCORE (DC) is 14% slower (118 M reqs/sec). KRCORE (RC) and verbs are both bottlenecked by the server RNIC, while KRCORE (DC) is slower due to extra DCT processing at the RNIC. For one-sided RDMA WRITE in Figure 10 (d), the results are similar: KRCORE (RC) and verbs achieve a peak throughput of 145M reqs/sec while KRCORE (DC) is 8.9% lower (132M reqs/sec).

Two-sided operations. Figure 11 presents the two-sided throughput and latency of KRCORE w.r.t. to the number of clients (1 to 240). Each client sends an 8B request to the server in an echo fashion: after receiving a request, the server

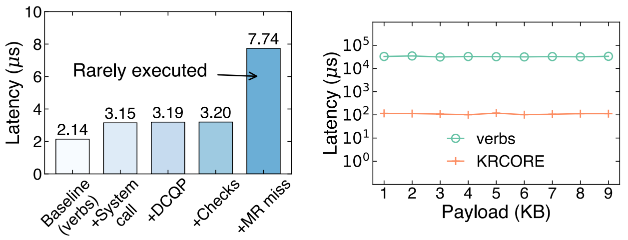

Fig. 12. (a) A factor analysis of the data path cost introduced by KRCore using one-sided RDMA READ. (b) The performance of KRCore in data transfer benchmark of serverless computing.

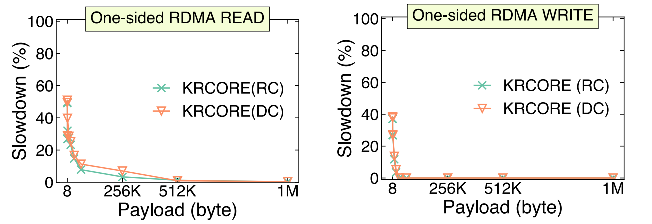

Fig. 13. The slowdown of KRCore compared to verbs on one-sided RDMA READ (a) and WRITE (b), respectively.

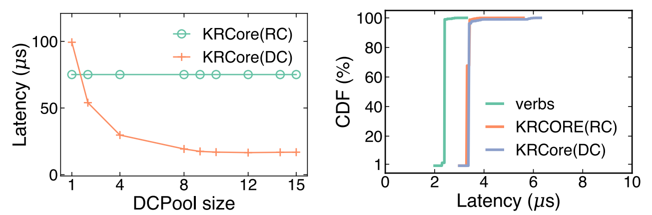

Fig. 14. (a) The impacts of DCQP pool size. (b) The CDF of latency of sending RDMA requests to different servers.

will send the request back, and the client will issue another request after getting the acknowledgment. The server utilizes all cores (24 threads) to handle these requests.

(1) Sync. In this mode, the performance comparisons are similar to one-sided RDMA: compared with verbs, KRCore (RC) and (DC) have 4–21% and 14–31% higher latency, respectively. The KRCore overheads added to two-sided RDMA are also dominated by the user-kernel interactions. For example, at one client, one KRCore (RC) echo takes 9.6μs while verbs takes 7.9μs. Compared to one-sided RDMA, the absolute latency gap is larger. KRCore two-sided has an additional system call overhead: the server needs to enter the kernel to receive a message.

(2) Async. Unlike one-sided RDMA, KRCore cannot achieve the same peak inbound throughput (when using 240 clients) as verbs for two-sided RDMA: it is 20% slower than verbs: which can only achieve 33.7M reqs/sec regardless of RC or DC. In comparison, verbs can achieve 42.3M reqs/sec. The extra bottleneck comes from CPU processing costs at the server due to user-kernel interactions. As a result, KRCore cannot saturate the RNIC's high performance. This also explains why KRCore has a similar performance when using RC and DC.

Effects of zero-copy optimization. We next examine the costs of memory copy—that KRCore uses to dispatch messages between virtual QPs—to the two-sided operations. We further demonstrate how we mitigate it with a zero-copy protocol (§4.5). Figure 9 (b) shows the two-sided echo latency when using one client to communicate with the server w.r.t. the payload size. We can see that the memory copy cost is negligible for small transfers (<=16KB) but is significant for large messages. Specifically, when transferring > 16KB messages, the latency of KRCore is 1.45–3.1X higher than verbs. To this end, the zero-copy optimization (KRCore+opt) reduces the overheads to 0.08-0.23X when transferring >= 16KB messages.

Factor analysis. Figure 12(a) conducts a factor analysis to show the detailed data path costs of KRCore in a sync one-sided RDMA READ request. The main observations are:

(1) The biggest cost to data path operations is additional RDMA requests to check the MR validity when the remote MR information is not cached locally (+MR miss, takes 4.5μs). Note the checks are rare because KRCore always caches the checked MR after a miss.

(2) For normal requests without MR checks, system call dominates the overheads (+System call), resulting in 1μs latency increase (3.15μs vs. 2.14μs). Other costs—including using DCQP (+DCQP) and KRCore check to prevent QP state corruptions (+Checks, see §4.4) are trivial (less than 0.5 μs).

Impacts of payload size to one-sided RDMA. The overhead of KRCore becomes smaller for one-sided RDMA with a larger payload, since transferring data through the network dominates the time. Figure 13 reports the slowdown compared to verbs on different request payloads. We measure the latency of sync one-sided RDMA with one client. For one-sided RDMA READ, the overhead is negligible for larger than 256KB reads (<7%). For WRITE, the overhead is negligible for larger than 8KB payloads.

Impacts of DCQP pool size. A larger DCQP pool is typically better for concurrently sending requests to different machines (§4.2). Figure 14 (a) reports the latency when sending a batch of 64 one-sided RDMA READs to different targets at one client with different pool sizes. The targets are randomly selected in 10 machines. We can see that when the pool only has one DCQP, KRCore (DC) has a 1.32X higher latency (99 vs. 75 μs) than KRCore (RC), since requests to the same QP are processed sequentially with reconnections. Increasing the pool size can significantly improve the latency. Interestingly, when the pool size is larger than 2, DC outperforms RC by 28–78%. RC needs 64 different connections to send these requests, and it has to do 63 additional polls than DC.

Tail latency. Figure 14 (b) reports the tail latency when using 50 clients sending sync one-sided RDMA READ to 5 servers. Under such a fan-out scenario, KRCore (DC) has a higher tail latency than the others due to extra round-trips caused by DC reconnections. The 99.9% latency of verbs,

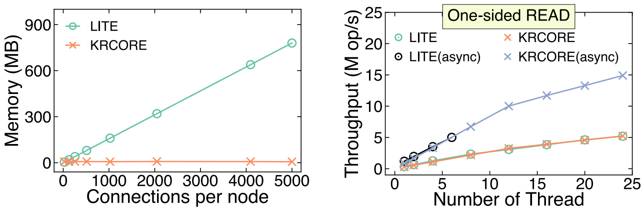

Fig. 15. (a) A comparison of memory usage on connections: KRCORE caches all DCT metadata, while LITE caches all RCQPs. (b) A comparison of data path performance when KRCORE uses DCQP.

KRCORE (RC) and KRCORE (DC) are 2.8μs, 3.8μs and 6μs, respectively.

Comparison to LITE. Finally, we show that KRCORE can achieve a similar (or better) data path performance than LITE with smaller memory usage.

(1) Memory. Figure 15 (a) shows the memory used for caching RDMA connections. In general, KRCORE consumes orders of magnitude smaller memory when supporting the same number of connections. For example, to maintain 5,000 connections, LITE consumes 780MB of memory, even without counting the memory of message queues (1.5GB if countered). In comparison, KRCORE only consumes 6.3MB of memory because it just maintains a (small) constant number of DCQP (48), and each DCT metadata only consumes 12B.

(2) Performance. Figure 15 (b) further compares the throughput when issuing 64B random one-sided RDMA READ from one node to others. We configure both systems to deploy a pool of 32 connections, preventing LITE from encountering RCQP scalability issues [26]. KRCORE uses DCQP for its connections. For sync, we can see that KRCORE is up to 20% slower than LITE due to performance issues of DCQP. On the other hand, KRCORE achieves a 3X higher peak async throughput (15.6M/sec vs. 5.2M/sec) in the async mode. LITE has a limited peak performance because it fails to run with more than 6 threads. LITE doesn't prevent QP queue overflows (see issue #3 in §2.3.2), so it will trigger QP errors for more than 6 threads. KRCORE handles overflows well (§4.4) and can thus, scale to more threads.

5.3 Application performance

5.3.1 Scaling RACE Hashing

Overview and setup. RACE hashing [67] is a production RDMA-enabled disaggregated key-value store. We chose it as our case study because it requires elastically—a demand not commonly found in existing RDMA-based key-value stores. At a high level, RACE separates the storage nodes and computing nodes by RDMA, where the computing nodes execute key-value store requests by issuing one-sided RDMA requests to the storage nodes. RACE further allocates computing nodes on-demand to cope with various workloads in a resource-efficient way, where the newly started nodes need dynamically establish RDMA connections to memory nodes. To improve performance, it embraces a set of low-level RDMA-aware optimizations—e.g., doorbell batching [25] that are tailed to RDMA's low-level verbs interface.

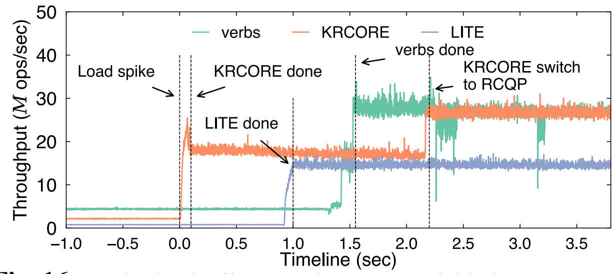

Fig. 16. Under load spikes, KRCORE can quickly bootstrap computing nodes for RACE Hashing [67].

Since RACE is not open-sourced, we implement a simplified version atop of verbs, LITE and KRCORE, respectively. We have calibrated that the performance is close to their reported ones. For example, RACE reports a peak 24M req/sec `Get` throughput on ConnectX-5 under YCSB-C [11]. Our (verbs) version can achieve 27M req/sec with more machines (8 vs. 5) on a similar RNIC (ConnectX-4).

Performance under load spikes. Our evaluating workload contains a load spike commonly found in real-world applications [8, 28, 2]. Under spikes, RACE allocates more computing processes to increase performance. During process startups, KRCORE can reduce its bootstrap time thanks to its fast control plane.

Figure 16 shows the timelines of RACE atop of verbs, LITE and KRCORE under load spikes, respectively. The spikes happen at time 0, and RACE forks 180 new processors to handle it. When using KRCORE, RACE can finish the startup in 244ms, 83% and 76% faster than verbs (1.4 seconds) and LITE (1 second), respectively. KRCORE is bottlenecked by OS creating worker processors. On the other hand, LITE and verbs are bottlenecked by RDMA's slow control path (§2.3). A fast boot further reduces the tail latency: during time 0-3, KRCORE has a 4.9X lower 99% latency than verbs.

Benefit of virtualizing a low-level RDMA API. KRCORE virtualizes a low-level RDMA (e.g., `ibv_post_send`), and thus, it can transparently apply existing RDMA-aware optimizations (see Issue #3 in §2.3.2). This leads to better performance of KRCORE on RACE compared to LITE: as shown in Figure 16, KRCORE has a 1.73X higher peak throughput (26M reqs/sec vs. 15M reqs/sec) than LITE after time 3.

Benefit of virtualizing hybrid QPs. As shown in Figure 16, using RCQP (e.g., after time 3) brought 1.4X (26M vs. 18M req/sec) throughput improvements to KRCORE, achieving a similar performance as verbs (26M reqs/sec). This is because RACE issues RDMA requests asynchronously, and KRCORE's RC async peak throughput is similar to verbs (see Figure 10 (b)). Further, we can see the overhead of switching from DCQP to RCQP is negligible (at time 2.2). However, there is a lag for detecting the switch because KRCORE needs time to collect the necessary information to decide which RC-

QPs to create.

5.3.2 Accelerating data transfer in serverless computing

Finally, we show that KRCORE can improve the communication performance between functions in serverless computing. We use an RDMA-version of data transfer testcase in ServerlessBench [62] (TestCase5), a state-of-the-art Serverless benchmark suite. This testcase measures the data transfer time between two serverless functions. The experiment runs on Fn [43], a popular open-source serverless platform.

Figure 12 (b) reports the time to pass a message w.r.t. the payload size when using verbs and KRCORE, respectively. The receiver function runs in a separate machine using a Docker container after the sender finishes execution. We use warm start to techniques [40] to reduce the control plane costs of starting containers. From the figure we can see KRCORE reduces the data transfer latency of verbs by 99% when transferring 1KB to 9KB bytes. The performance improvements are mainly due to the reduced RDMA control path of KRCORE, which we have extensively analyzed in §5.1.

6 Discussion

Trade-offs of a kernel-space solution. KRCORE chooses kernel-space RDMA for a microsecond-scale control plane (5,900X faster than verbs). Though it retains most benefits of RDMA (e.g., zero-copy), we sacrifice kernel-bypassing benefit and thus, result in a slower data path (up to 75% slowdown). We argue that such cost is acceptable to many elastic applications. First, the application usually issues a few networking requests. For example, the functions in ServerlessBench [62] and SeBS [12] only issue one request to read/write remote data on average. Second, the control path overhead (ms-scale) is commonly orders of magnitude higher than the cumulative data path overhead (μs-scale), see Figure 3. Finally, existing work (i.e., LITE [53]) also showed that kernel-space RDMA is efficient for many datacenter applications.

Other RNICs. Our analysis focuses on Mellanox ConnectX-4 Infiniband RNIC. Nevertheless, we argue the cost is unlikely to reduce due to hardware upgrades or different RDMA implementations (e.g., RoCE) since the cost is dominated by configuring the NIC resources. For example, we also evaluate the control path performance on ConnectX-6, where the user-space driver still takes 17ms for creating and connecting QP, similar to the ConnectX-4 we evaluated (15.7ms, see Figure 3).

KRCORE in virtualized environments. We currently focus on accelerating RDMA control plane with host networking mode. Using RDMA in virtual machines or virtualized RDMA network [30, 20] is also popular in the cloud. We believe the principles and methodologies of KRCORE are also applicable in these environments. For example, Freeflow [30] is an RDMA virtualization framework designed for containerized clouds. It leverages par-virtualization that intercepts virtualized RDMA requests to a software router. We can integrate our hybrid connection pool to the router to support a fast control plane atop of it. We plan to investigate applying KRCORE in virtualized environments in the future.

7 Related Work

RDMA libraries. Many user-space RDMA libraries exist [32, 4, 3, 36, 64], e.g., MPI, UCX [4], rsocket [3]. They can hardly provide a fast control plane because they are all based on verbs. LITE [53] is the only kernel-space RDMA library and is the closest to our work. We have extensively analyzed the issues when deploying LITE in elastic computing (§2.3.2) and how KRCORE addresses them (§3—§4).

DCT-aware and hybrid-transport systems. Several works used DCT to improve the performance and scalability of RDMA-enabled systems [49, 41]. Subramoni et al. [49] showed that DCT could provide comparable performance to RC while reducing memory consumption for MPI applications. Meanwhile, several works leveraged a hybrid-transport design to overcome the shortcoming of a single transport [31, 23]. For instance, Jose et al. [23] utilized UD to reduce the memory consumption of RC in Memcached.

RDMA-enabled applications. KRCORE continues the line of research on accelerating systems with RDMA, from key-value stores [38, 55, 67, 24, 13, 39], far-memory data structures [45, 6, 44], RPC frameworks [48, 26, 9, 27], replication systems [5, 42, 54, 29], distributed transactions [58, 46, 14, 10, 57, 65, 56], graphs [47, 59, 61, 18] and distributed file systems [66, 33, 60], just to name a few. Most of these systems do not target elastic computing, but we believe there are opportunities for applying them in such a setting. In such scenarios, they can benefit from KRCORE.

8 Conclusion

This paper presents KRCORE, a μs-scale RDMA control plane for RDMA-enabled applications that require elasticity. By retrofitting RDMA dynamic connected transport with kernel-space QP virtualization, we show that it is possible to eliminate most RDMA control path costs on commodity RNICs. Meanwhile, the data path costs introduced by KRCORE are acceptable for many elastic applications. Our experimental results confirm the efficacy of KRCORE.

9 Acknowledgment

We sincerely thank the anonymous shepherd and reviewers for their insightful suggestions. We also thank Dingji Li for discussing how to apply KRCORE to virtual machines, Xiating Xie for improving the figures and Sijie Shen, Zhiyuan Dong, Rongxin Chen and Yuhan Yang for their valuable feedback. This work was supported in part by the National Key Research & Development Program of China (No. 2020YFB2104100), the National Natural Science Foundation of China (No. 61732010, 61925206) and Shanghai AI Laboratory.

References

[1] Dynamically connected transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf, 2014.

[2] Daily Deals and Flash Sales: All the Stats You Need to Know. http://socialmarketingfella.com/daily-deals-flash-sales-stats-need-know/, 2016.

[3] rsocket(7) - Linux man page. https://linux.die.net/man/7/rsocket, 2021.

[4] Unified communication x. https://openucx.org, 2021.

[5] AGUILERA, M. K., BEN-DAVID, N., GUERRAOUI, R., MARATHE, V. J., XYGKIS, A., AND ZABLOTCHI, I. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 599–616.

[6] AGUILERA, M. K., KEETON, K., NOVAKOVIC, S., AND SINGHAL, S. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019* (2019), ACM, pp. 120–126.

[7] ASSOCIATION., I. T. Infiniband architecture specification. https://cw.infinibandta.org/document/dl/7859, 2015.

[8] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012* (2012), P. G. Harrison, M. F. Arlitt, and G. Casale, Eds., ACM, pp. 53–64.

[9] CHEN, Y., LU, Y., AND SHU, J. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 19:1–19:14.

[10] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., ACM, pp. 26:1–26:17.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010* (2010), J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds., ACM, pp. 143–154.

[12] COPIK, M., KWASNIEWSKI, G., BESTA, M., PODSTAWSKI, M., AND HOEFLER, T. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021* (2021), K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, Eds., ACM, pp. 64–78.

[13] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 401–414.

[14] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP'15, ACM, pp. 54–70.

[15] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 467–481.

[16] FACEBOOK. Introducing Bryce Canyon: Our next-generation storage platform. https://engineering.fb.com/2017/03/08/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/, 2017.

[17] GAO, Y., LI, Q., TANG, L., XI, Y., ZHANG, P., PENG, W., LI, B., WU, Y., LIU, S., YAN, L., FENG, F., ZHUANG, Y., LIU, F., LIU, P., LIU, X., WU, Z., WU, J., CAO, Z., TIAN, C., WU, J., ZHU, J., WANG, H., CAI, D., AND WU, J. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (2021), J. Mickens and R. Teixeira, Eds., USENIX Association, pp. 519–533.

[18] GUO, C., CHEN, H., ZHANG, F., AND LI, C. Distributed join algorithms on multi-cpu clusters with gpudirect RDMA. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019* (2019), ACM, pp. 65:1–65:10.

[19] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (2016), M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, pp. 202–215.

[20] HE, Z., WANG, D., FU, B., TAN, K., HUA, B., ZHANG, Z., AND ZHENG, K. Masq: RDMA for virtual private cloud. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (2020), H. Schulzrinne and V. Misra, Eds., ACM, pp. 1–14.

[21] JIA, Z., AND WITCHEL, E. Boki: Stateful serverless computing with shared logs. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 691–707.

[22] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C., KHANDELWAL, A., PU, Q., SHANKAR, V., CARREIRA, J., KRAUTH, K., YADWADKAR, N. J., GONZALEZ, J. E., POPA, R. A., STOICA, I., AND PATTERSON, D. A. Cloud programming simplified: A berkeley view on serverless computing. *CoRR abs/1902.03383* (2019).

[23] JOSE, J., SUBRAMONI, H., KANDALLA, K. C., WASI-UR-RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable memcached design for infiniband clusters using hybrid transports. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012* (2012), IEEE Computer Society, pp. 236–243.

[24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014* (2014), F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, Eds., ACM, pp. 295–306.

[25] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 437–450.

[26] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.

[27] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 1–16.

[28] KHANDELWAL, A., AGARWAL, R., AND STOICA, I. Blowfish: Dynamic storage-performance tradeoff in data stores. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016* (2016), K. J. Argyraki and R. Isaacs, Eds., USENIX Association, pp. 485–500.

[29] KIM, D., MEMARIPOUR, A. S., BADAM, A., ZHU, Y., LIU, H. H., PADHYE, J., RAINDEL, S., SWANSON, S., SEKAR, V., AND SESHAN, S. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (2018), S. Gorinsky and J. Tapolcai, Eds., ACM, pp. 297–312.

[30] KIM, D., YU, T., LIU, H. H., ZHU, Y., PADHYE, J., RAINDEL, S., GUO, C., SEKAR, V., AND SESHAN, S. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 113–126.

[31] KOOP, M. J., JONES, T., AND PANDA, D. K. Mvapich-aptus: Scalable high-performance multi-transport MPI over infiniband. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008* (2008), IEEE, pp. 1–12.

[32] LI, B., CUI, T., WANG, Z., BAI, W., AND ZHANG, L. Socksdirect: datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (2019), J. Wu and W. Hall, Eds., ACM, pp. 90–103.

[33] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (2017), D. D. Silva and B. Ford, Eds., USENIX Association, pp. 773–785.

[34] MA, T., MA, T., SONG, Z., LI, J., CHANG, H., CHEN, K., JIANG, H., AND WU, Y. X-RDMA: effective RDMA middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019* (2019), IEEE, pp. 1–12.

[35] MELLANOX. `https://www.mellanox.com/sites/default/files/doc-2020/pb-connectx-6-en-ic.pdf`, 2021.

[36] MELLANOX. `https://github.com/Mellanox/libvma`, 2021.

[37] MELLANOX. Connect-IB product brief. `https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Connect-IB.pdf`, 2021.

[38] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013* (2013), A. Birrell and E. G. Sirer, Eds., USENIX Association, pp. 103–114.

[39] MITCHELL, C., MONTGOMERY, K., NELSON, L., SEN, S., AND LI, J. Balancing CPU and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 451–464.

[40] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SOCK: rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 57–70.

[41] PARK, J., AND YEOM, H. Y. Design and implementation of software-based dynamically connected transport. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018* (2018), IEEE, pp. 58–64.

[42] POKE, M., AND HOEFLER, T. DARE: high-performance state machine replication on RDMA networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015* (2015), T. Kielmann, D. Hildebrand, and M. Taufer, Eds., ACM, pp. 107–118.

[43] PROJECT, F. https://fnproject.io, 2021.

[44] REDA, W., CANINI, M., KOSTIC, D., AND PETER, S. RDMA is turing complete, we just did not know it yet! *CoRR abs/2103.13351* (2021).

[45] RUAN, Z., SCHWARZKOPF, M., AGUILERA, M. K., AND BELAY, A. AIFM: high-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 315–332.

[46] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIC, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 433–448.

[47] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 317–332.

[48] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. RFP: when RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017* (2017), G. Alonso, R. Bianchini, and M. Vukolic, Eds., ACM, pp. 1–15.

[49] SUBRAMONI, H., HAMIDOUCHE, K., VENKATESH, A., CHAKRABORTY, S., AND PANDA, D. K. Designing MPI library with dynamic connected transport (DCT) of infiniband: Early experiences. In *Supercomputing - 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings* (2014), J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds., vol. 8488 of *Lecture Notes in Computer Science*, Springer, pp. 278–295.

[50] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[51] THOMAS, S., AO, L., VOELKER, G. M., AND PORTER, G. Particle: ephemeral endpoints for serverless networking. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 16–29.

[52] TSAI, S., SHAN, Y., AND ZHANG, Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 33–48.

[53] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.

[54] WANG, C., JIANG, J., CHEN, X., YI, N., AND CUI, H. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017* (2017), ACM, pp. 94–107.

[55] WEI, X., CHEN, R., AND CHEN, H. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 117–135.

[56] WEI, X., CHEN, R., CHEN, H., WANG, Z., GONG, Z., AND ZANG, B. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (2021), J. Mickens and R. Teixeira, Eds., USENIX Association, pp. 357–372.

[57] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 233–251.

[58] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 87–104.

[59] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving graph traversal with split live migration. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), D. Malkhi and D. Tsafrir, Eds., USENIX Association, pp. 723–738.

[60] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019* (2019), A. Merchant and H. Weatherspoon, Eds., USENIX Association, pp. 221–234.

[61] YAO, Z., CHEN, R., ZANG, B., AND CHEN, H. Wukong+g: Fast and concurrent RDF query processing using rdma-assisted GPU graph exploration. *IEEE Trans. Parallel Distributed Syst. 33*, 7 (2022), 1619–1635.

[62] YU, T., LIU, Q., DU, D., XIA, Y., ZANG, B., LU, Z., YANG, P., QIN, C., AND CHEN, H. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 30–44.

[63] ZHANG, H., CARDOZA, A., CHEN, P. B., ANGEL, S., AND LIU, V. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 1187–1204.

[64] ZHANG, I., RAYBUCK, A., PATEL, P., OLYNYK, K., NELSON, J., LEIJA, O. S. N., MARTINEZ, A., LIU, J., SIMPSON, A. K., JAYAKAR, S., PENNA, P. H., DEMOULIN, M., CHOUDHURY, P., AND BADAM, A. The demikernel datapath OS architecture for microsecond-scale datacenter systems. In

SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021 (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 195–211.

[65] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 51–68.

[66] ZHU, B., CHEN, Y., WANG, Q., LU, Y., AND SHU, J. Octopus$^+$: An rdma-enabled distributed persistent memory file system. *ACM Trans. Storage 17*, 3 (2021), 19:1–19:25.

[67] ZUO, P., SUN, J., YANG, L., ZHANG, S., AND HUA, Y. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 15–29.

A Artifact Appendix

Abstract. The artifact provides the source code and scripts to reproduce the experimental results from the USENIX ATC 2022 paper—–"KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing". KRCORE is a kernel-space RDMA solution that provides fast RDMA connection setups to user-space applications.

Scope. The artifact can be used to reproduce the evaluations in §5. It can also benefit the development of kernel-space RDMA-enabled applications.

Contents. The artifact contains the source code, the instructions for building and installation, and instructions for running the experiments in §5. All the above instructions can be found according to the steps in the `README.md` at the root directory of the artifact.

Hosting. The artifact is hosted on https://github.com/SJTU-IPADS/krcore-artifacts under the `main` branch with commit version 7ba3bf6.