MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications

Maksym Planeta and Jan Bierbaum, *TU Dresden;* Leo Sahaya Daphne Antony, *AMOLF;* Torsten Hoefler, *ETH Zürich;* Hermann Härtig, *TU Dresden*

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications

Maksym Planeta
TU Dresden

Jan Bierbaum
TU Dresden

Leo Sahaya Daphne Antony
AMOLF

Torsten Hoefler
ETH Zürich

Hermann Härtig
TU Dresden

Abstract

RDMA networks offload packet processing onto specialised circuitry of the network interface controllers (NICs) and bypass the OS to improve network latency and bandwidth. As a consequence, the OS forfeits control over active RDMA connections and loses the possibility to migrate RDMA applications transparently. This paper presents MigrOS, an OS-level architecture for transparent live migration of containerised RDMA applications. MigrOS shows that a set of minimal changes to the RDMA communication protocol reenables live migration without interposing the critical path operations. Our approach requires no changes to the user applications and maintains backwards compatibility at all levels of the network stack. Overall, MigrOS can achieve up to 33% lower network latency in comparison to software-only techniques.

1 Introduction

Major cloud providers increasingly offer RDMA network connectivity [1, 55] and high-performance network stacks [8, 18, 37, 53, 69, 85, 89] to the end-users. RDMA networks lower communication latency and increase network bandwidth by offloading packet processing to the RDMA NICs and by removing the OS from the communication critical path. To remove the OS, user applications employ specialised RDMA APIs, which access RDMA NICs directly, when sending and receiving messages [54]. These performance benefits made RDMA networks ubiquitous both in the HPC [15, 30, 43, 50, 77] and in the cloud settings [22, 59, 72].

At the same time, containers have become a popular tool for lightweight virtualisation. Containerised applications, being independent of the host's user space (libraries, applications, configuration files), greatly simplify distributed application deployment and administration. However, RDMA networks and containers follow contradicting architectural principles: Containerisation enforces stricter isolation between applications and the host, whereas RDMA networks try to bring applications and underlying hardware "closer" to each other.
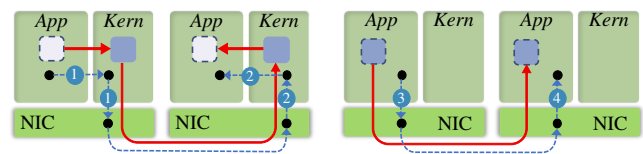

Figure 1: Traditional (left) and RDMA (right) network stacks. In traditional networks, the user application triggers the NIC via kernel (1). After receiving a packet, the NIC notifies the application back through the kernel (2). In RDMA networks the application communicates directly to the NICs (3) and vice-versa (4) without kernel intervention. Traditional networks require copy between application buffers (⬜) and NIC accessible kernel buffers (⬛). RDMA NICs (right) access message buffers in the application memory directly (🟦).

Container orchestration systems, like Kubernetes, stop containers and restart them on other hosts for the purpose of load balancing, resiliency, and administration. To maintain efficiency, containerised applications are expected to restart quickly [9]. Unfortunately, fast application restart requires in-application support and can be very costly, as RDMA applications tend to have large state. In contrast, *live migration* moves application state transparently and imposes no additional requirements for the application.

In the context of live migration, much of the container state resides in the host OS kernel. It can be extracted and recovered elsewhere later on. This recoverable state includes open TCP connections, shell sessions, file locks [13, 40]. However, as we elaborate in Section 3, *the state of RDMA communication channels is not recoverable by existing systems, and hence RDMA applications cannot be checkpointed or migrated.*

To outline the conceptual difficulties involved in saving the state of RDMA communication channels, we compare a traditional TCP/IP-based network stack and the IB verbs API[1] (see Figure 1). First, with a traditional network stack, the kernel fully controls when the communication happens:

---

[1]IB verbs is the most common low-level API for RDMA networks.

applications need to perform system calls to send or receive messages. In IB verbs, because of the direct communication between the NIC and the application, the OS cannot alter the connection state, except for tearing the connection down. Although the OS can stop a process from sending further messages, the NIC may still silently change the application state. Second, part of the connection state resides at the NIC and is inaccessible for the OS. Creating a consistent checkpoint transparently is impossible in this situation.

The contribution of our paper is a way to overcome the described limitations of current RDMA networks and the supporting operating systems with a novel architecture. The root of the problem to be solved is the impossibility for the OS to intercept packets to manipulate and repair disrupted connections, as it happens in transparent live application migration. The core point of the architecture is a small change in the RDMA protocol, that is usually implemented in RDMA NICs: we propose to add two new states and two new message types to RoCEv2, a popular RDMA communication protocol, while paying careful attention to backwards compatibility.

To make the proposed changes credible, we show that they are 1. *sufficient* for migrations, as an example for disrupted connections, and 2. indeed *small*. To prove the first point, we design and implement a transparent end-to-end live migration architecture for containerised RDMA applications. To prove the second point, and to enable the software architecture, we implement the new protocol by modifying SoftRoCE [48], a software implementation of the RoCEv2 protocol.

Providing a credible evaluation is hard for us, since we cannot modify the state machine of an RDMA NIC. Instead, we substantiate our claims by comparing the latency and bandwidth of the original and our modified protocol using SoftRoCE implementations and show that outside of the migration phase network performance is not affected. To justify the proposed protocol changes and the container-based software architecture, we evaluate software-level support for transparent live migration [2, 44]. We show that these approaches add significant overhead to the *normal* operation.

2 Background

This section gives a short introduction to containerisation and RDMA networking. We further outline live migration and how RDMA networking obstructs this process.

2.1 Containers

In Linux, processes can be logically separated from the rest of the system using *namespaces*. This way processes can have an isolated view on the file system, network devices, users, etc. Container runtimes leverage namespaces and other low-level kernel mechanisms [17, 47] to create a complete system view inside a *container* with one or multiple processes. Migration of a container moves all processes running inside
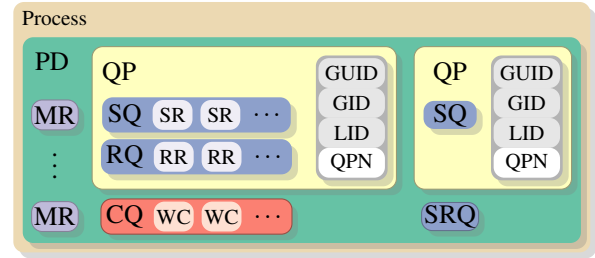

Figure 2: Primitives of the IB verbs library. Each QP comprises a send and a receive queue and has multiple IDs; node-global IDs (grey) are shared by all QPs on the same node.

a given container. A distributed application may comprise multiple containers across a network: a Spark application, for example, can isolate each *worker* in a container and an MPI [21] application can containerise each *rank*.

2.2 Infiniband verbs

The IB verbs API is a de-facto standard for high-performance RDMA communication today. RDMA applications achieve high throughput and low latency by accessing the NIC directly (*OS-bypass*), reducing memory movement (*zero-copy*), and delegating packet processing to the NIC (*offloading*).

Figure 2 shows the following IB verbs objects involved in communication. *Memory regions* (MRs) represent pinned memory shared between the application and the NIC. *Queue pairs* (QPs), comprising a send queue (SQ) and a receive queue (RQ), represent connections. To reduce the memory footprint, the individual RQs of multiple QPs can be replaced with a single *shared receive queue* (SRQ). *Completion queues* (CQs) inform the application about completed communication requests. A *protection domain* (PD) groups all these objects together and represents the process address space to the NIC.

To establish a connection, the IB verbs specification [54] requires the applications to exchange the following addressing information: *Memory protection keys* to enable access to remote MRs, the global vendor-assigned address (*GUID*), the routable address (*GID*), the non-routable address (*LID*), and the node-specific QP number (*QPN*). One way to perform this exchange is over an out-of-band network, e.g. Ethernet. During the connection setup, each QP is configured for a specific *type of service*. MigrOS supports only Reliable Connections (RC), which provide reliable in-order message delivery between two communication partners. Another popular type of service is Unreliable Datagram (UD), which does not provide these guarantees.

The application sends or receives messages by posting *send requests* (SR) or *receive requests* (RR) to a QP as *work queue entries* (WQE). These requests describe the message and refer to the memory buffers within previously created MRs. The application checks for the completion of outstanding work

requests by polling the CQ for *work completions* (WC).

There are various implementations of the IB verbs API for different hardware, including Infiniband [37], iWarp [23], and RoCE [35, 36]. InfiniBand is generally the fastest among these networks but requires specialised NICs and switches. RoCE and iWarp are easier to deploy, because they provide RDMA capabilities in Ethernet networks. They still require hardware support in the NIC, but do not depend on specialised switches. This work focuses on RoCEv2, an increasingly more popular version of the RoCE protocol [15, 60, 88]. At the same time, we change only parts of RoCEv2 that are defined in the same way for Infiniband [37] and RoCEv1 [35], hence MigrOS is also compatible with other RDMA protocols.

To enable RDMA-application migration, it is important to consider the following challenges:

1. User applications have to use physical network addresses (QPN, LID, GID, GUID) but the IB verbs API does not specify a way for virtualising these.
2. The NIC can write to any memory it shares with the application without the OS noticing.
3. The OS cannot instruct the NIC to pause communication, except for abruptly terminating it.
4. User applications do not handle changes in a connection destination address and will go into an erroneous state. As a result, the application will terminate abruptly.
5. Although the OS resides on the control path and is therefore aware of all IB verbs objects created by the application, the OS does not control the whole state of these objects, as the state partially resides on the NIC.

We address all of these challenges in Section 3.

2.3 CRIU

CRIU is a software framework for transparent checkpointing and restoring of Linux processes [13]. It enables live migration, snapshots, or accelerated start-up of processes and containers. To extract the user-space application state, CRIU uses conventional debugging mechanisms, like ptrace [73, 74]. However, to extract the state of process-specific kernel objects, CRIU depends on special Linux kernel interfaces.

During recovery, CRIU runs inside the target process and recreates all OS objects on behalf of the target. This way, CRIU utilises the available OS mechanisms to run most of the recovery without the need for significant kernel modifications. Finally, CRIU removes any traces of itself from the process.

CRIU can also restore the state of TCP connections, a crucial feature for live migration of distributed applications [40]. The Linux kernel introduced a new TCP connection state, `TCP_REPAIR`, for that purpose. In this state, a user-level process can modify the send and receive message queues, get and set the message sequence numbers and timestamps, or open and close a connection without notifying the other side. As of now, CRIU does not support saving and restoring IB verbs objects. Discarding IB verbs objects during migra-
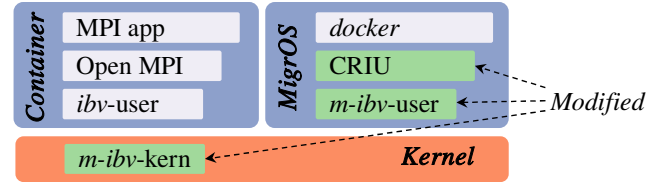

Figure 3: MigrOS architecture. Software inside the container, including the user-level driver (*ibv*-user, grey), is unmodified. The host runs CRIU, kernel- (*m-ibv*-kern) and user-level (*m-ibv*-user, green) drivers modified for migratability.

tion in the naive hope that the application will be able to recover is failure-prone: once an application runs into an erroneous IB verbs object, in most cases, the application will hang or crash. Thus, MigrOS provides explicit support for IB verbs objects in CRIU (see Section 3).

3 Design

MigrOS is based on modern container runtimes and reuses much of the existing infrastructure with minimal changes (see Section 3.1). Most importantly, we require no modification of the software running inside the container. Instead, MigrOS includes the following changes concerning RoCEv2: two new QP states to enable the creation of consistent checkpoints (see Section 3.2), a connection migration protocol (Section 3.3), and modifications to the packet-level RoCEv2 protocol (Section 3.4). Finally, Section 3.5 describes our modifications to the IB verbs API and how CRIU uses them. It is sufficient to only extend CRIU with IB verbs support, existing container runtimes use CRIU for their checkpoint/restore functionality [17, 47, 71, 76].

3.1 Software Stack

Typically, access to the RDMA network is hidden deep inside the software stack. Figure 3 gives an example of a containerised RDMA application. The container image comes with all library dependencies, like the libc, but not the kernel-level drivers. In this example, the application uses a stack of communication libraries, comprising Open MPI [21], Open UCX [77] (not shown), and IB verbs. Normally, to migrate, a container runtime would require the application inside the container to terminate and later recover all IB verbs objects. This removes transparency from live migration.

MigrOS runs alongside the container comprising a container runtime (e.g., Docker [17]), CRIU, and the IB verbs library. We modified CRIU to make it aware of IB verbs, so it can save IB verbs objects when traversing the kernel objects that belong to the container. We extend the IB verbs library (*m-ibv*-user and *m-ibv*-kern) to enable serialisation and deserialisation of the IB verbs objects. Importantly, the
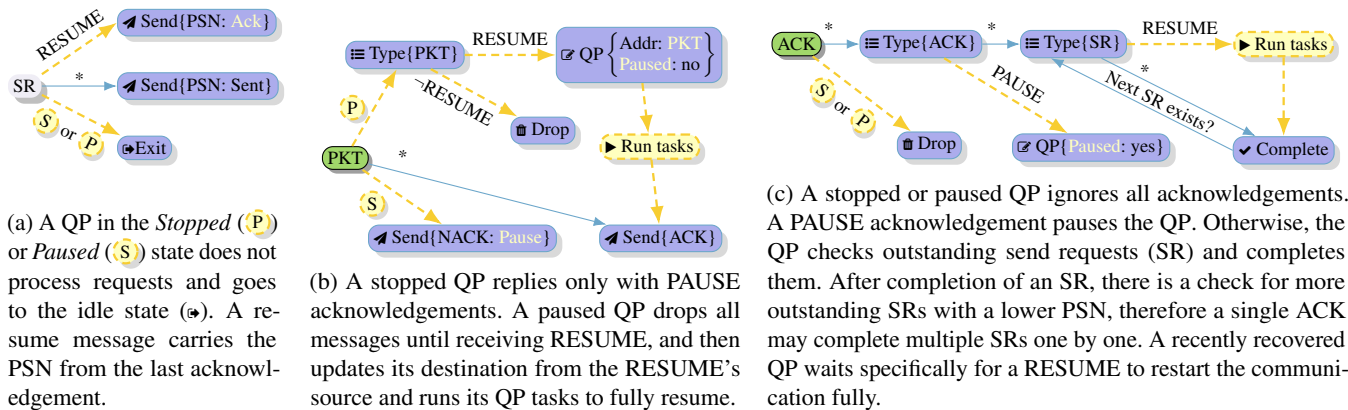
Figure 4: QP State Diagram. Normal states and state transitions (●, →) are controlled by the user application. A QP is put into error states (●, ⋯►) either by the OS or the NIC. New states (○, ⇢) are used for connection migration.

Figure 5: To migrate from host N_0 to host N_2, the state of the QP changes from RTS (R) to Stopped (S). Finally, the QP is destroyed (D). If the partner QP at host N_1 sends a message during migration, this QP gets paused (P). Both QPs resume normal operation once the migration is complete.

API extension is backwards compatible with the IB verbs library running inside the container. Thus, both *m-ibv*-user and *ibv*-user use the same kernel version of IB verbs. All of the IB verbs components (*ibv*-user, *m-ibv*-user, *m-ibv*-kern) comprise a generic and a device-specific part. MigrOS relies on modified kernel and user parts (*m-ibv*-user and *m-ibv*-kern), but *requires no modifications of the software inside the container*.

3.2 Queue Pair States

Before explaining QP migration, we first recapitulate how a QP functions in general. Communication can start, when an application establishes a connection by taking a QP through a sequence of states (depicted in Figure 4). Each newly-created QP is in the *Reset* (R) state. To send and receive messages, a QP must reach its final *Ready-to-Send* (RTS) state. Before reaching RTS, the QP traverses the *Init* and *Ready-to-Receive* (RTR) states. In case of an error, the QP goes into one of the error states; *Error* (E) or *Send Queue Error* (SQE). In the *Send Queue Drain* (SQD) state, a QP does not accept new send requests. Apart from that, SQD is equivalent to the RTS state and we do not describe it further in this paper.

To migrate connections safely, we add two new states invisible to the user application (see Figure 4): *Stopped* (S) and *Paused* (P). When the kernel checkpoints a process, all of its QPs go into the *Stopped* state. A stopped QP does not send or receive any messages. The QPs remain stopped until they are destroyed together with the checkpointed process.

A QP becomes *Paused* when it learns that its destination QP has transitioned to the *Stopped* state. A paused QP does not send messages, but also has no other QP to receive messages from. A QP remains paused, until the migrated destination QP restores at a new location and sends a message with the new location address. The paused QP retains the new location of the destination QP and returns to the RTS state. After that, the communication can continue.

3.3 Connection Migration

There are two considerations when migrating a connection. First, the communication partner of the migrating container must not confuse migration with a network failure. Second,

once the migration is complete, all partners of the migrated container need to learn its new address.

We address the first issue by extending RoCEv2 with a connection migration protocol. The connection migration protocol is active during and after migration (see Figure 5). This protocol is part of the low-level packet transmission protocol and is typically implemented entirely within the NIC. Also, we add a new negative acknowledgement type `NAK_STOPPED`. If a stopped QP receives a packet, it replies with `NAK_STOPPED` and drops the packet. When the partner QP receives this negative acknowledgement, it transitions to the *Paused* (P) state and refrains from sending further packets until receiving a message of the new *resume* type. If N_0 destroys the QP before N_2 sends the resume message, packets coming from N_1 will be dropped and no negative acknowledgement will be sent. It is possible to synchronise the destruction of old QPs and the transfer of resume messages, but we did not implement this feature in our prototype, because such packet loss does not impair correctness.

After migration completes, the new host of the migrated process restores all QPs to their original state and sends resume messages. Resume messages are sent unconditionally, even if the partner QP was not paused before. Any recipient of a resume message updates its QP's destination address to the source address of the resume message, i.e., to the new location of the migrated QP.

Each pause and resume message carries source and destination information. Thus, if multiple QPs migrate at the same time, there can be no confusion which QPs must be paused or resumed. On the other hand, MigrOS must prevent concurrent migration of QPs at both ends of the same connection, because the senders of resume messages may be aware only of old, outdated locations. If at any point the migration process fails, the paused QPs will remain stuck and will not resume communication. This scenario is completely analogous to a failure during a TCP-connection migration. In both cases, MigrOS will be responsible for cleaning up the resources.

3.4 Protocol Changes

Our protocol changes must be reflected in an RDMA NIC, because most packet-level RoCEv2 protocol implementations

(a) A QP in the *Stopped* (P) or *Paused* (S) state does not process requests and goes to the idle state (➡). A resume message carries the PSN from the last acknowledgement.

(b) A stopped QP replies only with PAUSE acknowledgements. A paused QP drops all messages until receiving RESUME, and then updates its destination from the RESUME's source and runs its QP tasks to fully resume.

(c) A stopped or paused QP ignores all acknowledgements. A PAUSE acknowledgement pauses the QP. Otherwise, the QP checks outstanding send requests (SR) and completes them. After completion of an SR, there is a check for more outstanding SRs with a lower PSN, therefore a single ACK may complete multiple SRs one by one. A recently recovered QP waits specifically for a RESUME to restart the communication fully.

Figure 6: MigrOS changes processing of incoming packets (PKT and ACK) and outgoing request (SR) by introducing new QP states (P and S), new messages (PAUSE and RESUME), new operations (⏸ and ▶), and new transitions (⇢). Solid arrows (→) represent previously existing transitions, "*" represents the "else" branch, ≣ Type checks the SR or packet type.

run in hardware. We proceed from the fact that a typical RDMA NIC does all packet-level work, including transmitting packets and acknowledgements by itself. As a consequence, the OS cannot itself compose and process arbitrary packets, including the packets that MigrOS introduces.

Therefore, for migration to work, we need to modify RDMA NICs. The NIC adds different packet headers, depending on the service (RC, UD, etc.) and message (read, send, etc.) type. Our changes touch only two headers: Base Transport Header (BTH) and ACK Extended Transport Header (AETH). BTH is the first RDMA-specific header, immediately following IP and UDP headers. Additionally, the NIC needs to maintain two new flags per QP for pause and resume states. The NIC must be able to transfer a QP into the *Stopped* or *Paused* state, process a send request with a resume message issued by the OS, and handle pause/resume packets.

Our changes (see Figure 6) cover three existing workflows in the underlying RoCEv2 protocol: 1. Processing a send request WQE (Figure 6a), 2. receiving a packet (Figure 6b), and 3. receiving an acknowledgement (Figure 6c). The NIC must consider the two new states of the QP in all these workflows. This change of logic is small and does not change the packet layout, as it is only part of the internal state of the QP.

The NIC must compose the new resume message or let the OS do so. In contrast to a normal message, resume takes the packet sequence number (PSN) from the last acknowledged message instead of the last sent message. A receiver recognises the resume message by a new *opcode* in the BTH header. Creating a new message type does not require changes in the message layout due to an abundance of unused opcodes.

Similarly, pause, sent as a new negative acknowledgement type, employs an unused value of the *syndrome* field in the AETH header. Therefore, the new pause NACK also requires no change to the existing packet layout.

The workflows in the Figure 6 run when the user triggers

the NIC through a doorbell register, when a message arrives, or by a timeout. Unless a QP is paused or stopped, the NIC will try to send or complete multiple messages at once (Figure 6a and Figure 6c). As part of resume (▶), the NIC also triggers these workflows.

Figure 6 does not include, for example, additional timeout reaction logic for the sake of brevity. Overall, the changes in logic are simple and mostly reuse existing functionality. Because we change only the AETH and BTH headers, our changes are equally applicable to other RDMA protocols (e.g. Infiniband or RoCEv1) that use these headers in the same way. We believe neither new logic nor new states incur prohibitive design or implementation cost.

A real RDMA NIC would need hardware support for the new QP states and follow the corresponding protocol. Full migration support would also require the NIC to extract the state of Infiniband objects and handle the new message types. We believe all of this can be implemented in the NIC's firmware[2]. Section 4 presents a proof-of-concept software implementation of the proposed changes.

3.5 Checkpoint/Restore API

To enable checkpoint/restore for processes and containers, we extend the IB verbs API with two new calls (see Listing 1): `ibv_dump_context` and `ibv_restore_object`. CRIU relies on the normal IB verbs API supplemented by the two new calls to save and restore the IB verbs state of applications.

The `ibv_dump_context` call returns a dump of all IB verbs objects within a specific IB verbs *context*, an object representing the connection between a process and an RDMA NIC. The creation of a dump runs almost entirely inside the kernel for two reasons: First, some links between the objects are

---

[2]The hardware/firmware-boundary will differ for FPGA and ASIC.

```
int ibv_dump_context(struct ibv_context *ctx,
        int *count, void *dump, size_t length);
int ibv_restore_object(struct ibv_context *ctx,
        void **object, int object_type, int cmd,
        void *args, size_t length);
```

Listing 1: Checkpoint/Restore extension for the IB verbs API. `ibv_dump_context` creates an image of the IB verbs context `ctx` with `count` objects and stores it in the caller-provided memory region `dump` of size `length`. `ibv_restore_object` executes the restore command `cmd` for an individual object (QP, CQ, etc.) of type `object_type`. The call expects a list of arguments specific to the object type and recovery command. `args` is an opaque pointer to the argument buffer of size `length`. A pointer to the restored object is returned via `object`.

only visible at the kernel level. Second, to get a consistent checkpoint it is crucial to ensure the dump is atomic.

Although the existing IB verbs API allows to create new objects, it is not expressive enough for *restoring* them. For example, when restoring a completion queue (CQ), the current API does not allow to specify the address of the shared memory region for this queue, instead this address is assigned by the kernel. It is also not possible to recreate a queue pair (QP) directly in its original state, like Ready-to-Send (RTS). Instead, the QP has to traverse all intermediate states to reach the desired state.

We introduce the fine-grained `ibv_restore_object` call to restore IB verbs objects one by one, for situations when the existing API is not sufficient. During recovery, CRIU reads the object dump and applies a specific recovery procedure for each object type. For example, to recover a QP, CRIU calls `ibv_restore_object` with the command CREATE and transitions the QP through the Init, RTR, and RTS states using `ibv_modify_qp`. The memory regions or QP buffers are recovered using the standard file and memory operations. Finally, when a QP reaches the RTS state (representing an active connection), the new host executes the REFILL command using the `ibv_restore_object` call. This command restores the driver-specific internal QP state and sends a *resume* message to the partner QP.

4 Implementation

To provide transparent live migration, MigrOS makes changes to CRIU, the IB verbs library, the RDMA-device driver (SoftRoCE), and the packet-level RoCEv2-protocol. To migrate an application, the container runtime invokes CRIU which checkpoints the target container. CRIU stops active RDMA connections and saves the state of IB verbs objects (see Section 4.1). SoftRoCE then pauses communication using our extensions to the packet-level protocol. After transfering the checkpoint to the destination node, the container runtime at that node invokes CRIU to recover the IB verbs objects and

restores the application. SoftRoCE then resumes all paused communication to complete the migration process.

SoftRoCE is a Linux kernel-level software implementation (not an emulation [48]) of the RoCEv2 protocol [36]. RoCEv2 runs RDMA communication by tunnelling Infiniband packets through a well-known UDP port. In contrast to other RDMA-device drivers, SoftRoCE allows the OS to inspect, modify, and control the state of IB verbs objects completely.

As a performance-critical component of RDMA communication, RoCEv2 usually runs inside the hardware and firmware of a NIC. We focus on minimising these protocol changes. The key part of MigrOS is the addition of connection migration capabilities to the existing RoCEv2 protocol (see Section 4.2).

4.1 State Extraction and Recovery

State extraction begins when CRIU discovers that its target process has opened an IB verbs device. We modified CRIU to use the API presented in Section 3.5 to extract the state of all available IB verbs objects. CRIU stores this state together with other process data in an image. Later, CRIU recovers the image on another node using the new API.

When CRIU recovers MRs and QPs of the migrated application, the recovered objects must maintain their original unique identifiers. These identifiers are system-global and assigned by the NIC (in our case the SoftRoCE driver) in a sequential manner. We augmented the SoftRoCE driver to expose the IDs of the last assigned MR and QP to MigrOS in userspace. These IDs are *memory region number* (MRN) and *queue pair number* (QPN) correspondingly. Before recreating an MR or QP, CRIU configures the last ID appropriately. If no other MR or QP occupies this ID, the newly created object will maintain its original ID. This approach is analogous to the way CRIU maintains the process ID of a restored process using the `ns_last_pid` mechanism of Linux, which exposes the last process ID assigned by the kernel.

It is possible for some other process to occupy an MRN or QPN, which CRIU wants to restore. Two processes cannot use the same MRN or QPN on the same node, resulting in a conflict. In the current scheme, we avoid these conflicts by partitioning QP and MR addresses globally among all nodes in the system before application startup. CRIU faces the very same problem with process ID collisions. This problem has only been solved with the introduction of process ID namespaces. To remedy the collision problem for IB verbs objects, a similar namespace-based mechanism, together with a virtual RDMA network [29], would be required. We leave this issue for future work.

Additionally, recovered MRs have to maintain their original memory protection keys. The protection keys are pseudo-random numbers [75] provided by the NIC and are used by a remote communication partner when sending a packet. An RDMA operation succeeds only if the provided key matches
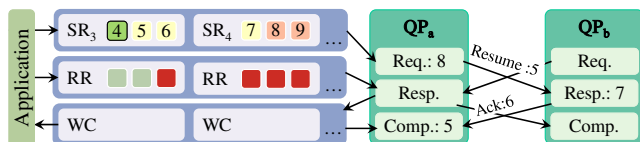
Figure 7: Resuming a connection in SoftRoCE. The figure depicts a snapshot of an immediate state, whereas the arrows indicate the flow of data over time. A send queue comprises multiple SRs, each expected to send multiple packets. Packets 8 and 9 (■) are to be processed by the requester. Packets 5 – 7 (□) are yet to be acknowledged. Packet 4 (■) is already acknowledged. A receive queue contains RRs with received (■) and not yet received (■) packets. QP_b expects the next packet to be 7. A resume packet has the PSN of the first unacknowledged packet (□). QP_b replies with an acknowledgement of the last received packet.

the expected key of a given MR. Other than that, the key's value does not carry any additional semantics. Thus, no collision problems exist for protection keys.

CRIU sets all protection keys to their original values before communication restarts by making an `ibv_restore_object` call with the `IBV_RESTORE_MR_KEYS` command.

4.2 Resuming Connections

The connection migration protocol ensures that connections are terminated gracefully and recovered to a consistent state. The implementation of this protocol is device- and driver-specific. In this work, we modify the SoftRoCE driver to make it compliant with the connection migration protocol (Section 3.3) by providing an implementation of the checkpoint/restore API (Section 3.5).

Figure 7 outlines the basic operation of the SoftRoCE driver, which creates three concurrent *tasks* for each QP: *requester*, *responder*, and *completer*. When an application posts send (SR) and receive (RR) work requests to a QP, they are processed by requester and responder correspondingly. A work request may be split into multiple packets, depending on the MTU size. When the whole work request is complete, responder or completer notify the application by posting a work completion to the completion queue.

The tasks process all requests packet by packet. Each task maintains the packet sequence number (PSN) of the next packet. A requester sends packets for processing at the responder of its partner QP. The responder replies with an acknowledgement sent to the completer. The completer generates a work completion (WC) after receiving an acknowledgement for the last packet in an SR. Similarly, the responder generates a WC after receiving all packets of an RR.

After migration, when the recovered QP_a is ready to communicate again, it sends a resume message to QP_b with the new address. Upon receiving the resume message, the re-

sponder of QP_b learns the new location of QP_a. Then, the responder replies with an acknowledgement of the last successfully received packet. If some packets were lost during the migration, the next PSN at the responder of QP_b is smaller than the next PSN at the requester of QP_a. The difference corresponds to the lost packets. Simultaneously, the requester of QP_b can already start sending messages. At this point, the connection between QP_a and QP_b is fully recovered.

The presented protocol ensures that both QPs recover the connection without losing packets irrecoverably. If packets were lost during migration, the QPs can determine which packets were lost and retransmit them, as part of the normal RoCEv2 protocol. The whole connection migration protocol runs transparently for the user applications.

5 Evaluation

We evaluate MigrOS from three main aspects. First, we analyse the implementation effort, with a specific focus on the magnitude of changes to the RoCEv2 protocol. Second, we study the overhead of adding migration capability, outside the migration phase. Third, we estimate the fine-grained cost of migration for individual IB verbs objects, as well as the full latency of migration in realistic RDMA applications.

For most experiments, we use a system with two machines: Each machine is equipped with an Intel i7-4790 CPU, 16 GiB RAM, an on-board Intel 1 Gb Ethernet adapter, a Mellanox ConnectX-3 VPI adapter, and a Mellanox Connect-IB 56 Gb adapter. The Mellanox VPI adapters are set to 40 Gb Ethernet mode. The SoftRoCE driver communicates over this adapter. The machines run Debian 11 with a custom Linux 5.7-based kernel. We refer to this setup as *local*. When comparing against DMTCP and FreeFlow, we use Ubuntu 14.04.

We conduct further measurements on a cluster comprising two-socket Intel E5-2680 v3 CPUs nodes with Connect-IB 56 Gb NICs deployed by Bull. We refer to this setup as *cluster*. Two nodes similar to those in the cluster were used in a local setup and equipped with Mellanox ConnectX-3 VPI NICs configured to 56 Gb InfiniBand mode.

5.1 Size of Changes

To quantify the changes MigrOS requires, we count the newly added or modified source lines of code (SLOC) in different components of the software stack. Out of around 4 k SLOC only around 10% apply to the kernel-level SoftRoCE driver. These changes mostly focus on saving and restoring the state of IB verbs objects. We separately counted changes to the requester, responder, and completer QP tasks responsible for the active phase of communication (see Figure 7). These tasks would be implemented in the NICs, for hardware-based RDMA implementations. Therefore, we keep the changes to QP tasks simple and small, as these changes must be reflected

Object	Features required	State (bytes)
PD	None	12
MR	Set memory keys and MRN	48
CQ	Restore ring buffer metadata	64
SRQ	Restore ring buffer metadata	68
QP	+ QP tasks state, set QPN	271
QP w/ SRQ	+ Current WQE state	823

Table 1: Additional features implemented in the kernel-level SoftRoCE driver to enable recovery of IB verbs objects. We provide the size that each object occupies in the dump.

	Latency, µs			Bandwidth, Gb/s		
Size, B	Unmod.	FF	DMTCP	Unmod.	FF	DMTCP
2^0	0.8*	1.2*	1.4*	0.09	0.02	0.01
2^4	0.8*	1.2*	1.4*	1.41	0.24	0.20
2^8	1.1*	1.6*	1.8*	22.31	3.95	3.25
2^{12}	2.3	2.7	2.9	36.50	36.57	36.49
2^{16}	15.8	16.2	16.5	36.59	36.59	36.59
2^{20}	230.8	231.2	231.4	36.59	36.59	36.59

Table 2: Performance of CX3/40. Comparing execution without modifications against DMTCP and FreeFlow. The variation over 30 runs was small, except,* when $0.05 < \sigma/\mu < 0.1$.

	Latency, $\mu \pm \sigma$ µs		
Size, B	Plain	Migratable	DMTCP
2^0	25.3 ± 0.2	25.0 ± 0.2	26.2 ± 0.6
2^4	25.3 ± 0.3	24.9 ± 0.4	25.5 ± 0.5
2^8	26.9 ± 0.6	25.7 ± 0.7	28.0 ± 0.5
2^{12}	35.7 ± 0.4	36.8 ± 0.7	35.5 ± 0.5
2^{16}	93.2 ± 1.9	93.8 ± 1.9	94.4 ± 1.9
2^{20}	793.9 ± 12.8	802.0 ± 11.3	802.1 ± 13.5

Table 3: Communication latency with SoftRoCE. Comparison of migratable and plain (non-migratable) SoftRoCE against DMTCP using plain SoftRoCE.

in firmware or hardware. In our implementation, changes to QP tasks accounted only for around 6% of overall changes.

Using gcov [20], we verified that most of the changes to the QP tasks do not affect the critical path of communication. During the active phase of communication, 1213 lines were touched out of 4808 lines of the SoftRoCE driver identified by gcov as executable. Among the touched lines only 28 lines correspond to code we added for migration, with 3 lines corresponding to variable assignments and the rest being jumps related to checking for the *Paused* or *Stopped* state. The remaining code changes to the QP tasks run only during the connection migration phase.

Besides additional logic in the QP tasks, saving and restoring IB verbs objects requires the manipulation of implementation-specific attributes. Some of these attributes cannot be set through the original IB verbs API. For example, recovering an MR requires the additional ability to restore the original values of memory keys and the MRN. Some other attributes are not visible in the original IB verbs API at all. The queues (CQ, SRQ, QP) implemented in SoftRoCE require the ability to save and restore metadata of ring buffers backing up the queues. If a QP uses a shared receive queue (SRQ), the dump of the QP additionally includes the full state of the current *work queue entry* (WQE). We identified all required attributes for SoftRoCE, calculated their memory footprint (see Table 1), and implemented all features required by these attributes.

The changes to RoCEv2 implemented in SoftRoCE are small and affect the critical path of communication only marginally outside of the migration phase. We believe that for RDMA NICs the same changes to RoCEv2 will remain equally small.

5.2 Overhead of Migratability

Just adding the capability for transparent container migration already may incur overhead even when no migration occurs. For example, DMTCP (see Section 6) intercepts all IB verbs library calls and rewrites both work requests and completions before forwarding them to the NIC. Both with DMTCP and FreeFlow, this interception happens persistently, even if

the process never migrates. In contrast to this, MigrOS does not intercept communication operations on the critical path, thereby introducing no measurable overhead. This subsection explores the overhead added for normal communication operations without migrations.

DMTCP [2] and FreeFlow [44] do not offer live migration. Nevertheless, they could be extended to provide it. Therefore, we start by estimating the cost of adding migration capability at the user level. We use the latency and bandwidth benchmarks from the perftest benchmark suite [68]. We ran each experiment 30 times with 10000 iterations each at the *local* setup, with CX3/40 NICs.

Both frameworks do additional processing for each IB verbs work request, which results into near-constant overhead to latency (see Table 2). Each work request corresponds to a single message, not a single packet, therefore the overhead diminishes for larger message sizes. Table 2 demonstrates that bandwidth is directly affected by the increased latency and thus is lower only for small messages. We expect such bandwidth reduction to be a minor disadvantage for realistic applications, whereas a near 50% increase in latency may be critical for many latency-sensitive applications [30, 78].

It is possible, that despite our best effort to minimise the changes to the NIC, changes proposed by MigrOS introduce measurable overhead. Therefore, we need to show that Mi-

Short	Full name	Location
SR	SoftRoCE	local
CX3/40	ConnectX-3 40 Gb Ethernet	local
CX3/56	ConnectX-3 56 Gb InfiniBand	cluster
CIB	ConnectIB	local
BIB	Bull Connect-IB	cluster

Table 4: RDMA-capable NICs used for the evaluation.

grOS adds no additional cycles during message transmission. For that, we compare the performance of migratable and non-migratable versions of the SoftRoCE driver[3] against DMTCP running with the non-migratable version of SoftRoCE. Running `ib_send_lat` [68] benchmarks in three different configurations shows (see Table 3) that migratability adds no visible overhead. Simultaneously, the latency increase for DMTCP is also minute, making it hard to distinguish all three configurations. The reason for such small difference between different configurations is the large communication overhead introduced by SoftRoCE. As a result, our experiment only shows that migratability does not add a large overhead, but does not allow to make a confident judgement regarding a potential microsecond-scale overhead. This situation is an unfortunate limitation of SoftRoCE. Considering how tiny we expect the overhead to be, even an FPGA-implementation may not be a precise reflection of a commercial RDMA NIC, because of the significant differences between an FPGA and an actual hardware implementation. We are convinced, due to the arguments given in Section 3, that MigrOS does not introduce even microsecond-scale overhead.

5.3 Migration Costs

With added support for migrating IB verbs objects, the container migration time will increase proportionally to the time required to recreate these objects. Our goal is to estimate the additional latency for migrating RDMA-enabled applications. This subsection shows the cost for migrating connections created by SoftRoCE, as well as the cost for connection creation with hardware-based IB verbs implementations.

Several IB verbs objects are required before a reliable connection (RC) can be established, see Section 2.2. Usually, an application creates a single PD, one or two CQs, multiple memory regions, and one QP per communication partner.

To measure the cost of creating individual IB verbs objects, we modified `ib_send_bw` [68] to create additional MR objects. We created one CQ, one PD, 64 QPs, and 64 1 MiB-sized MRs per run. Figure 8 shows the average time required to create

Figure 8: Object creation time for different RDMA devices (see Table 4). To send a message, a QP needs to be in the state RTS, which requires the traversal of three intermediate states (Reset, Init, RTR). Error bars show the interval of the standard deviation (σ) around the mean (μ), if $\sigma/\mu \geq 0.05$.

each object across 50 runs. Each tested NIC is represented by a bar. We draw two conclusions from this experiment. First, there is substantial variation for all operations across different NICs. Second, the time required for most operations is in the range of milliseconds.

The exact time required for migrating RDMA connections depends on two factors: the number of QPs and the total amount of memory assigned to MRs [56]. Both of these factors are application-specific and can vary greatly. Therefore, next, we show how the migration time is influenced by the application's usage of MRs and QPs.

Figure 9 shows the MR registration time, depending on the region's size. MR registration costs are split between the OS and the NIC: The OS pins the memory and the NIC learns about the virtual memory mapping of the registered region. SoftRoCE does not incur the "NIC-part" of the cost, so MR registration with SoftRoCE is faster than for RDMA-enabled NICs. For this experiment, we do not consider the cost of transferring the contents of the MRs during migration.

The number of QPs is the second variable influencing the migration time. Figure 10 shows the time for migrating a container running the `ib_send_bw` benchmark. This benchmark consists of two single-process containers running on two different nodes. Three seconds after communication starts, the container runtime migrates one of the containers to another node. The migration time is measured as the maximum message latency as seen by the container that did not move. The checkpoint is transferred over the same network link used by
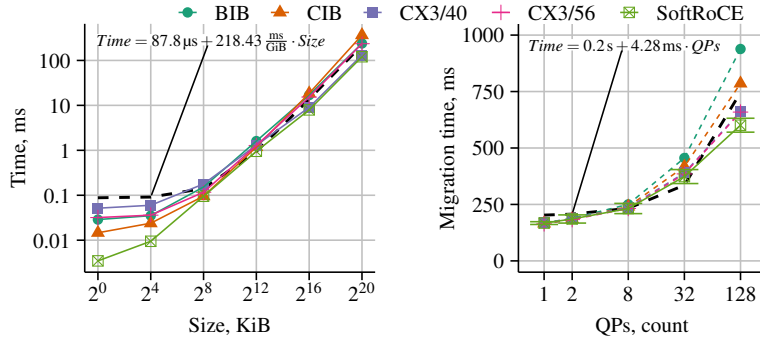
---

[3]Both versions are modified by us, because the original version (*vanilla kernel*) of the SoftRoCE driver is extremely unstable. It contained a multitude of concurrency bugs and could not be used in migration experiments. Unfortunately, fixing the race conditions required significant restructuring and resulted in a performance loss of around 20%.

Figure 9: MR registration time depending on the region size.

$Time = 87.8\,\mu s + 218.43\,\frac{ms}{GiB}\cdot Size$
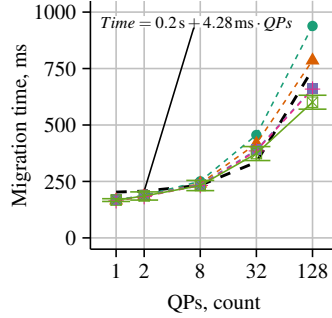
Figure 10: Migration speed with different numbers of QPs.

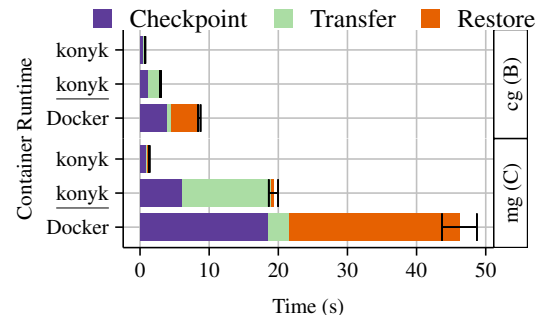$Time = 0.2\,s + 4.28\,ms\cdot QPs$

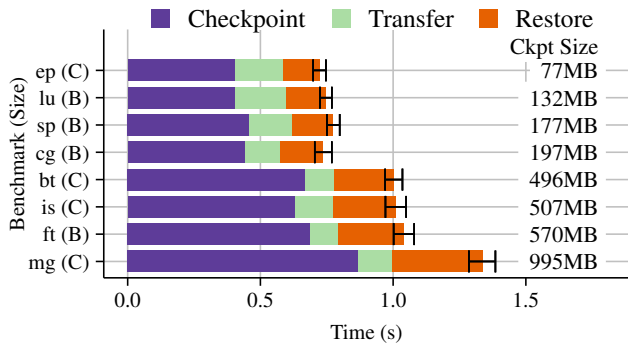Figure 11: Migration speed of Docker and konyk with optimisations (konyk) and without (k͟o͟n͟y͟k͟)

Figure 12: MPI application migration.

the benchmarks for communication. With a growing number of QPs, the benchmark consumes more memory, ranging from 8 MiB to 20 MiB. To put things into perspective, we estimated the migration time for real devices by calculating the time to recreate IB verbs object for RDMA-enabled NICs. We subtracted the time to create IB verbs objects with SoftRoCE from the measured migration time and added the time needed to create IB verbs objects with RDMA NICs (from Figure 8). We show our estimations with the dashed lines.

5.4 MPI Application Migration

For evaluating transparent live-migration of real-world applications, we chose to migrate NPB 3.4.1 [3], an MPI benchmark suite. The MPI applications run on top of Open MPI 4.0 [21], which in turn uses Open UCX 1.6.1 [77] for point-to-point communication. We configured UCX to use IB verbs communication over reliable connections (RC).

This setup corresponds to Figure 3. We containerised the applications using our self-developed runtime *konyk*, based on libcontainer [76]. Unlike Docker, our runtime facilitates faster live migration by sending the image directly to the destination node, instead of the local storage, during the checkpoint process. Additionally, konyk stores checkpoints in RAM, reducing migration latency even more. As any other container

runtime, konyk internally uses CRIU for checkpointing and restoring containers. Further description of our container runtime is out of scope of this paper.

To measure the application migration latency, we start each MPI application with four processes (*ranks*). Approximately in the middle of the application progress, one of the ranks migrates to another node. Each benchmark has a *size* (A to F) parameter. We chose the size such that all benchmarks run between 10 and 300 seconds. We excluded the "dt" benchmark, because it runs only for around a second. Figure 12 shows container migration latency and standard deviation around the mean, averaged over 20 runs of each benchmark.

We break down the migration latency into three parts: *checkpoint*, *transfer*, and *restore*. MigrOS first stops the target container and prepares the checkpoint. Almost immediately, and in parallel with checkpointing, MigrOS starts to transfer checkpoint data to the destination node. The transfer happens over the network link used by the benchmarks for communication. This overlap of checkpointing and data transfer minimises the time of exclusive data transfer. After the transfer is complete, MigrOS recovers the container at the destination node. Overall, the benchmarks experience a runtime delay proportional to the migration latency, which is proportional to the checkpoint size.

MPI applications (Figure 12) migrate slower than microbenchmarks (Figure 10), even after accounting for the checkpoint size, because of the difference in measurement methodology. For the microbenchmark, we calculate the migration time based on the maximum message latency observed by the non-migrating process. For the MPI benchmarks, we calculate the migration time from the increase in total execution time of the whole benchmark. This discrepancy indicates that the migration of parallel applications may cause a larger disruption to the application performance than the simple state transfer time can explain.

To show the interoperability of MigrOS with other container runtimes, we measured the migration costs when using Docker 19.03 (see Figure 11). We had to provide the end-to-

	Legion	Nomad	PS MPI	DMTCP	MOSIX-4	MOSIX-3	MigrOS
RDMA	✓	✓	✓	✓	✗	✗	✓
Overhead	N	N	N	Y	Y	Y	N
Runtime	✓	✓	✓				
User-OS					✓	✓	
Kernel-OS		✓				✓	✓
NIC							✓
Units	O	VM	P	P	P	P	C
Reference	[7]	[32]	[70]	[2]	[5]	[6]	Ours

Table 5: Selected checkpoint/restore systems handle either VMs, processes (P), containers (C), or application objects (O). Runtime-based systems naturally introduce no additional communication overhead for migration support.

end migration flow ourselves, because Docker features only checkpoint and restore. To our disappointment, Docker does not employ some important optimisations and requires much time to complete migration. To understand the performance difference better, we disabled some optimisations in konyk: saving checkpoints to main memory (instead of hard disk), sending checkpoints during dumping, and using an optimised way to transfer the checkpoint. All of this was not enough to match the performance of Docker. Further investigation revealed Docker unnecessarily moving checkpoint images across the file system, proving the importance of explicit live migration support within a container runtime. Nevertheless, we demonstrate the principle possibility of containerised RDMA-application migration using Docker.

6 Related Work

VMs Live migration of virtual machines (VMs) has a long usage history in cloud computing [11, 16, 31, 63, 67]. We expect live migration to become even more popular with the growth of new computing paradigms, like disaggregated and fog computing [12, 25, 65, 86]. Nevertheless, past techniques for live VM migration with RDMA NICs relied on migration-aware, paravirtualised drivers inside the VMs [32, 70]

Checkpoint/Restore Techniques Transparent live migration of processes [4, 57, 81], containers [51, 58, 62], or virtual machines [11, 16, 31, 63, 67] has long been a topic of active research. The key challenge of this technique lies in the checkpoint/restore operation. For processes and containers, this operation can be implemented at three levels: application runtime, user-level system, or kernel-level system. Table 5 compares a selection of existing checkpoint/restore systems.

Runtime-based systems expect the user application to access all external resources through the API of the runtime. This restriction resolves two important issues with resource migratability: First, the runtime system controls exactly when the underlying resource is used and can easily stop the application from doing so to serialise the state of the resource. Second, the runtime can maintain enough information about the state of the resource to facilitate resource serialisation and deserialisation. Such interception is cheap because it happens within the application's address space.

Almost all attempts to provide transparent live migration together with RDMA networks rely on modifications of the runtime system [2, 24, 26, 32, 41, 70]. Some runtimes operate on application-defined objects (tasks, agents, lightweight threads) for even more efficient state serialisation and deserialisation [7, 43, 87]. All runtime-based approaches bind the application to a particular runtime system.

Kernel OS-level checkpoint/restore systems [6, 28, 38, 42, 66] either do interposition at the kernel level or extract application state from the kernel's internal data structures. Although these systems support a wider spectrum of user applications, they incur a significantly higher maintenance burden. BLCR [28] has been abandoned eventually. CRIU [13], currently the most successful OS-level tool for checkpoint/restore, keeps necessary Linux kernel modification at a minimum and does not require interposition at the user-kernel API. We describe this tool in more detail in Section 2.3.

Finally, *user OS-level* systems interpose the user-kernel API, providing the same transparency and generality as kernel-based implementations. Such systems use the LD_PRELOAD mechanism to intercept system calls from applications and virtualise system resources, like file descriptors, process IDs, and sockets. In version 4, MOSIX has been redesigned to work entirely at the user level [5]. DMTCP [2] is a transparent fault-tolerance tool for distributed applications with support for IB verbs. To be able to extract the state of IB verbs objects, DMTCP maintains *shadow objects*, which act as proxies between a user process and the NIC [10]. In Section 5.2, we show that maintaining these shadow objects has non-negligible runtime overhead for RDMA networks.

Furthermore, live migration may employ RDMA networks to improve the speed of the checkpoint transfer [33, 39]. These techniques allow to reduce the downtime from migration and could be combined with our technique to improve the migration time of RDMA applications.

Network Virtualisation TCP/IP network virtualisation is an essential tool for isolating distributed applications from the underlying physical network topology. Even though network virtualisation enables live migration, it introduces overhead due to additional encapsulation of network packets [64, 89]. Several new approaches try to address these performance problems [8, 64, 69, 89]. However, these approaches do not consider RDMA networks.

RDMA-network virtualisation approaches focus on implementing connection control policies in software, but do not support live container migration [29, 44, 84]. As an exception, Nomad [32] uses InfiniBand address virtualisation for VM

migration, but implements the connection migration protocol inside an application-level runtime.

MigrOS uses traditional network virtualisation for TCP/IP networks, which is not on the performance-critical path for RDMA applications. However, MigrOS avoids unnecessary interception of RDMA communication. Instead, MigrOS silently replaces addressing information during migration.

RDMA Implementations SoftRoCE [48] and SoftiWarp [83] are open-source software implementations of RoCEv2 [36] and iWarp [23] respectively. Both provide no performance advantage over socket-based communication but are compatible with their hardware counterparts and facilitate the development and testing of RDMA-based applications. We chose to base our work on SoftRoCE because RoCEv2 found wider adoption than iWarp.

There are also open-source FPGA-based implementations of network stacks. NetFPGA [90] does not support RDMA communication. StRoM [79] provides a proof-of-concept RoCEv2 implementation. However, we found it unfit to run real-world applications (for example, MPI) without further significant implementation efforts.

7 Discussion and Conclusion

MigrOS is an OS-level architecture enabling transparent live container migration without sacrificing RDMA network performance. We are convinced the architecture of MigrOS can be useful for dynamic load balancing, efficient prepared fail-over, and live software updates in cloud or HPC settings.

Hardware Modifications and Software Implementation We believe that limited hardware changes are worthy of consideration and have already been proven feasible [14, 27, 45, 45, 61], even for RDMA protocols [34, 46, 49]. Nevertheless, propositions to modify hardware often meet criticism because they are hard to validate and evaluate for an OS designer. To overcome this difficulty, we have modified SoftRoCE. It turned out that adding only few states to the state machine and two new message types were necessary. As result, we enabled transparent live migration of containerised RDMA applications without affecting the critical path of the communication. Lesokhin et al. [46] have demonstrated that RDMA protocol changes can be achieved just through firmware updates, barring the need to replace NICs. Furthermore, our design maintains full backwards-compatibility with the existing RDMA network infrastructure at every level and can be adopted by other RDMA protocols (e.g. Infiniband and RoCEv1) verbatim.

Unreliable Datagram Communication MigrOS provides live migration for reliable communication (RC), but not for unreliable datagram (UD), because, first, every message received over UD exposes the address of its sender. When this sender migrates, its address will change and, currently, MigrOS cannot conceal the change from the receiver. Second, a UD QP does not know where to send resume messages after migration, because it can receive messages from anywhere. Consequently, to support UD, a NIC would need to maintain an additional simple table to translate between user-visible and actual addresses. We leave this issue for future work.

Security As of today, lack of authentication and integrity control is a general problem in RDMA networks [52, 75, 80, 82]. Therefore, modern RDMA networks rely on trusted NICs and hosts. In the context of this paper, we require the host OS to ensure that pause and resume messages may only be sent by authorised hosts. If either NICs or hosts cannot be trusted, additional protocols must prevent the sending of unauthorised (e.g. by spoofing) pause and resume messages. In this regard, we do not degrade security of the RDMA network.

White-box migration Previous live migration techniques require cooperation on the application's behalf, because they see RDMA NICs as *black boxes*. To our knowledge, our work is first to consider an RDMA NIC as a *white box* for the purpose of live migration. We categorise the device state as 1. *public state*, observable through the IB verbs API, 2. *device–driver-visible state*, visible by kernel- and user-level drivers, 3. *internal state*, invisible outside the device. The device must expose its internal state to the OS at the time of migration. We hope, these findings can be useful, when implementing live migration for other devices, e.g. GPUs.

Beyond migration We believe that the pause/resume protocol can find other uses, like efficient fail-over, congestion control, or load balancing. As an example, consider MasQ [29], a virtual RDMA network with firewall capabilities. MasQ can shut down RDMA connections, but unlike TCP/IP firewalls, cannot block them temporarily. Our protocol could return control over RDMA connections to the OS and replicate TCP/IP-like behaviour to RDMA firewalls as well.

Availability github.com/TUD-OS/migros-atc-2021.

Acknowledgments

References

[1] Amazon Web Services, Inc. Elastic Fabric Adapter. https://aws.amazon.com/hpc/efa/, 2020.

[2] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *2009 IEEE International Symposium on Parallel Distributed Processing*, IPDPS, pages 1–12, May 2009. doi:10/d62csg.

[3] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, September 1991. ISSN 0890-2720. doi:10/cgsfnm.

[4] Amnon Barak and Amnon Shiloh. A distributed load-balancing policy for a multicomputer. *Software: Practice and Experience*, 15(9):901–913, September 1985. ISSN 00380644, 1097024X. doi:10/c8r7m6.

[5] Amnon Barak and Shiloh, Amnon. The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster Private Clouds. Technical report, Springer-Verlag, 2016.

[6] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, Berlin, Heidelberg, 1993. ISBN 978-0-387-56663-4.

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–11, Salt Lake City, UT, November 2012. IEEE. ISBN 978-1-4673-0806-9. doi:10/gf9t42.

[8] Adam Belay, George Prekas, Christos Kozyrakis, Ana Klimovic, Samuel Grossman, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 49–65, October 2014. ISBN 978-1-931971-16-4.

[9] Brian Grant. Pod migration · Issue #3949 · kubernetes/kubernetes. https://github.com/kubernetes/kubernetes/issues/3949, January 2015.

[10] Jiajun Cao, Gregory Kerr, Kapil Arya, and Gene Cooperman. Transparent checkpoint-restart over Infiniband. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*, pages 13–24, Vancouver, BC, Canada, 2014. ACM Press. ISBN 978-1-4503-2749-7. doi:10/ggnfr4.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI '05, pages 273–286. USENIX Association, May 2005. doi:10.5555/1251203.1251223.

[12] Connor, Patrick, Hearn, James R., Dubal, Scott P., Herdrich, Andrew J., and Sood, Kapil. Techniques to migrate a virtual machine using disaggregated computing resources, 2018.

[13] CRIU. Checkpoint/Restore In Userspace. https://criu.org/Main_Page, 2011.

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Vivek Bhanu, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, NSDI'18, Berkeley, Calif, 2018. ISBN 978-1-931971-43-0.

[15] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An In-Depth Analysis of the Slingshot Interconnect. *arXiv:2008.08886 [cs]*, August 2020.

[16] Umesh Deshpande, Yang You, Danny Chan, Nilton Bila, and Kartik Gopalan. Fast Server Deprovisioning through Scatter-Gather Live Migration of Virtual Machines. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 376–383, Anchorage, AK, USA, June 2014. IEEE. ISBN 978-1-4799-5063-8. doi:10/ggnfmn.

[17] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, March 2014.

[18] DPDK. Data Plane Development Kit. https://www.dpdk.org/, 2013.

[19] FFMK. FFMK Website. https://ffmk.tudos.org, 2013.

[20] Free Software Foundation, Inc. gcov—a Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2021.

[21] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241, pages 97–104, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-30218-6. doi:10/bxhsv5.

[22] Peter X. Gao, Akshay Narayan, Rachit Agarwal, Sagar Karandikar, Sylvia Ratnasamy, Joao Carreira, Sangjin Han, and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 249–264, Savannah, GA, USA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. doi:10.5555/3026877.3026897.

[23] D. Garcia, P. Culley, R. Recio, J. Hilland, and B. Metzler. A Remote Direct Memory Access Protocol Specification. https://tools.ietf.org/html/rfc5040, October 2007.

[24] Rohan Garg, Gregory Price, and Gene Cooperman. MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '19*, HPDC, pages 49–60, Phoenix, AZ, USA, 2019. ACM Press. ISBN 978-1-4503-6670-0. doi:10/ggnd38.

[25] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI '17, page 21, Boston, MA, USA, 2017. USENIX Association. ISBN 978-1-931971-37-9. doi:10.5555/3154630.3154683.

[26] Wei Lin Guay, Sven-Arne Reinemo, Bjørn Dag Johnsen, Chien-Hua Yen, Tor Skeie, Olav Lysne, and Ola Tørud-bakken. Early experiences with live migration of SR-IOV enabled InfiniBand. *Journal of Parallel and Distributed Computing*, 78:39–52, April 2015. ISSN 07437315. doi:10/f68twd.

[27] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.

[28] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46:494–499, September 2006. ISSN 1742-6588, 1742-6596. doi:10/d33sc5.

[29] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 1–14, New York, NY, USA, July 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7. doi:10/gg9rjq.

[30] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *Journal of Chemical Theory and Computation*, 4 (3):435–447, March 2008. ISSN 1549-9618, 1549-9626. doi:10/b7nkp6.

[31] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS Operating Systems Review*, 43(3):14–26, July 2009. ISSN 0163-5980. doi:10/ccwrpt.

[32] Wei Huang, Jiuxing Liu, Matthew Koop, Bulent Abali, and Dhabaleswar Panda. Nomad: migrating OS-bypass networks in virtual machines. In *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07*, VEE'07, page 158, San Diego, California, USA, 2007. ACM Press. ISBN 978-1-59593-630-1. doi:10/frgqz4.

[33] Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, page 1, Seattle, Washington, 2011. ACM Press. ISBN 978-1-4503-0771-0. doi:10/bsgrkf.

[34] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: XRC*. InfiniBand Trade Association, February 2009.

[35] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: RoCE*, volume 1. InfiniBand Trade Association, 1.2.1 edition, 2010.

[36] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: RoCEv2*. InfiniBand Trade Association, September 2014.

[37] InfiniBand Trade Association. *InfiniBand Architecture Specification*, volume 1. InfiniBand Trade Association, 1.3 edition, March 2015.

[38] Jake Edge. Checkpoint/restart tries to head towards the mainline. https://lwn.net/Articles/320508/, February 2009.

[39] Joel Nider and Mike Rapoport. News from academia: FatELF, RDMA and CRIU. In *Linux Plumbers Conference*, Vancouver, BC, Canada, November 2018.

[40] Jonathan Corbet. TCP connection repair. https://lwn.net/Articles/495304/, May 2012.

[41] J. Jose, Mingzhe Li, Xiaoyi Lu, K. C. Kandalla, M. D. Arnold, and D. K. Panda. SR-IOV Support for Virtualization on InfiniBand Clusters: Early Experience. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Delft, May 2013. IEEE. ISBN 978-0-7695-4996-5. doi:10/ggm53b.

[42] Asim Kadav and Michael M. Swift. Live migration of direct-access devices. *ACM SIGOPS Operating Systems Review*, 43(3):95, July 2009. ISSN 01635980. doi:10/b9j36z.

[43] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. *ACM SIGPLAN Notices*, 28(10):91–108, October 1993. ISSN 0362-1340. doi:10/cgnqf7.

[44] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI, pages 113–125, Boston, MA, USA, 2019. ISBN 978-1-931971-49-2. doi:10.5555/3323234.3323245.

[45] Alec Kochevar-Cureton, Somesh Chaturmohta, Norman Lam, Sambhrama Mundkur, and Daniel Firestone. Remote direct memory access in computing systems, October 2019.

[46] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 449–466, Xi'an China, April 2017. ACM. ISBN 978-1-4503-4465-4. doi:10/ghm5x7.

[47] Linux Containers - LXC - Introduction. Linux Containers (LXC). https://linuxcontainers.org/lxc/, 2008.

[48] Liran Liss. The Linux SoftRoCE Driver, March 2017.

[49] Liran Liss. On Demand Paging for User-level Networking, 2013.

[50] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 295–304, San Francisco, CA, USA, June 2003. Association for Computing Machinery. ISBN 978-1-58113-733-0. doi:10/c4knj6.

[51] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via Docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 1–13, San Jose, California, 2017. ACM Press. ISBN 978-1-4503-5087-7. doi:10/gf9x9r.

[52] Manhee Lee, Eun Jung Kim, and M. Yousif. Security enhancement in InfiniBand architecture. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10 pp.–, April 2005. doi:10/c4xpq5.

[53] Mellanox. Messaging Accelerator (VMA) Documentation. Technical report, Mellanox, April 2019.

[54] Mellanox Technologies. RDMA Aware Networks Programming User Manual. Technical Report 1.7, Mellanox Technologies, 2015.

[55] Microsoft. High performance computing VM sizes. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-hpc, August 2020.

[56] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Euro-Par 2006 Parallel Processing*, volume 4128, pages 124–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-37783-2 978-3-540-37784-9. doi:10.1007/11823285_13.

[57] Dejan Milojičić, Frederick Douglis, and Richard Wheeler. *Mobility: processes, computers, and agents*. ACM Press/Addison-Wesley Publishing Co., USA, 1999. ISBN 978-0-201-37928-0.

[58] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Linux Symposium*, Ottawa, 2008.

[59] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013. ISBN 978-1-931971-01-0.

[60] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIG-COMM '18, pages 313–326, New York, NY, USA, August 2018. Association for Computing Machinery. ISBN 978-1-4503-5567-4. doi:10/ghf6mq.

[61] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 77–92, Santa Clara, CA, USA, February 2020. USENIX Association. ISBN 978-1-939133-13-7.

[62] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. Voyager: Complete Container State Migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142, Atlanta, GA, USA, June 2017. IEEE. ISBN 978-1-5386-1792-2. doi:10/ggnhq5.

[63] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, ATC '05, pages 391–394, Anaheim, CA, USA, April 2005. USENIX Association. doi:10.5555/1247360.1247385.

[64] Zhixiong Niu, Hong Xu, Peng Cheng, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making Network Stack Part of the Virtualized Infrastructure. *arXiv:1903.07119 [cs]*, March 2019.

[65] Opeyemi Osanaiye, Shuo Chen, Zheng Yan, Rongxing Lu, Kim-Kwang Raymond Choo, and Mqhele Dlodlo. From Cloud to Fog Computing: A Review and a Conceptual Live VM Migration Framework. *IEEE Access*, 5:8284–8300, 2017. ISSN 2169-3536. doi:10/ggnfkt.

[66] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: a system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, December 2003. ISSN 0163-5980. doi:10/fbg7vq.

[67] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. CompSC: live migration with pass-through devices. *ACM SIGPLAN Notices*, 47(7):109, September 2012. ISSN 03621340. doi:10/f3887q.

[68] perftest. perftest. https://github.com/linux-rdma/perftest, April 2020.

[69] Simon Peter, Jialin Li, Irene Zhang, Dan R K Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014*, OSDI '14, 2014.

[70] S. Pickartz, C. Clauss, S. Lankes, S. Krempel, T. Moschny, and A. Monti. Non-intrusive Migration of MPI Processes in OS-Bypass Networks. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1728–1735, May 2016. doi:10/ggscxh.

[71] podman.io. Podman: daemonless container engine. https://podman.io/, 2018.

[72] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '15*, pages 107–118, Portland, Oregon, USA, 2015. ACM Press. ISBN 978-1-4503-3550-8. doi:10/ggm3sf.

[73] proc - process information pseudo-filesystem. proc(5) - Linux manual page. http://man7.org/linux/man-pages/man5/proc.5.html, 2020.

[74] ptrace - process trace. ptrace(2) - Linux manual page. http://man7.org/linux/man-pages/man2/ptrace.2.html, 2020.

[75] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium*, page 16, Vancouver, B.C., 2021.

[76] runc. runc: CLI tool for spawning and running containers according to the OCI specification. https://github.com/opencontainers/runc, 2020.

[77] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, HotI, pages 40–43, August 2015. doi:10/ggmx8k.

[78] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, page 17, 2016.

[79] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *the Fifteenth European Conference on Computer Systems*, EuroSys '20, page 16, Heraklion, Greece, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387519.

[80] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud 20, page 8. USENIX Association, 2020.

[81] Jonathan M. Smith. A survey of process migration mechanisms. *ACM SIGOPS Operating Systems Review*, 22 (3):28–40, July 1988. ISSN 0163-5980. doi:10/bjp787.

[82] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 691–704, 2020. ISBN 978-1-939133-14-4.

[83] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A case for RDMA in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11*, page 1, Shanghai, China, 2011. ACM Press. ISBN 978-1-4503-1179-3. doi:10/fzv576.

[84] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*, pages 306–324, Shanghai, China, 2017. ACM Press. ISBN 978-1-4503-5085-3. doi:10/ggscxn.

[85] Dongyang Wang, Binzhang Fu, Gang Lu, Kun Tan, and Bei Hua. vSocket: virtual socket interface for RDMA in public clouds. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE, pages 179–192, Providence, RI, USA, 2019. ACM Press. ISBN 978-1-4503-6020-3. doi:10/ggscxg.

[86] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–16, Dresden, Germany, March 2019. Association for Computing Machinery. ISBN 978-1-4503-6281-8. doi:10/ggnq76.

[87] David Wong, Noemi Paciorek, and Dana Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):92–ff., March 1999. ISSN 0001-0782. doi:10/btg3k7.

[88] Yibo Zhu, Ming Zhang, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, and Mohamad Haj Yahia. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, pages 523–536, London, UK, 2015. ACM. ISBN 978-1-4503-3542-3. doi:10.1145/2785956.2787484.

[89] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Boston, MA, USA, February 2019. USENIX Association. ISBN 978-1-931971-49-2.

[90] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, September 2014. ISSN 1937-4143. doi:10/gg8qsd.