Software-based Live Migration for Containerized RDMA

Xiaoyu Li[1, 2, 3] Ran Shu[3] Yongqiang Xiong[3] Fengyuan Ren[1, 2]
[1]Tsinghua University, [2]Beijing National Research Center for Information Science and Technology (BNRist),
[3]Microsoft Research
Beijing, China

ABSTRACT

Container live migration is critical to ensure services are not interrupted during host maintenance in data centers. On the other hand, RDMA containerization has attracted both academia and industry for years. However, live migration for containerized RDMA is not supported in today's data centers. Although modifying RDMA NICs (RNICs) to be aware of live migration has been proposed for years, there is no sign of supporting it on commodity RNICs. This paper proposes `MigrRDMA`, a software-based RDMA live migration for containers, which does not rely on any extra hardware support. `MigrRDMA` provides a minimum virtualization layer inside the RDMA library loaded in applications, which achieves transparent switching to new RDMA communications. Unlike previous RDMA virtualization that provides sharing and isolation, `MigrRDMA`'s virtualization layer focuses on keeping the RDMA states on the migration source and destination the same from the perspective of applications. Our evaluation shows that `MigrRDMA` only adds 0.7∼12.1 ms downtime to migrate a container with live RDMA connections running at line rate. Besides, the `MigrRDMA` virtualization layer only adds 3% ∼ 9% overheads in the data path operations.

CCS CONCEPTS

• **Networks → Data center networks**; • **Software and its engineering → Operating systems**.

KEYWORDS

Containers, RDMA, Live Migration, Virtualization

ACM Reference Format:
Xiaoyu Li, Ran Shu, Yongqiang Xiong, and Fengyuan Ren. 2024. Software-based Live Migration for Containerized RDMA. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024), August 3–4, 2024, Sydney, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663408.3663416

1 INTRODUCTION

Nowadays, due to containers' features of fast provisioning and low memory footprint, they have become the *de facto* choice for data centers to deploy distributed online services (e.g., key-value stores [11], machine learning [26], etc.). Data center operators desire to enable server upgrades, data center maintenance, and load balancing, without interrupting the running services. As a result, container live migration is supported in major container platforms (e.g., Docker [10], LXC [30]) and is widely used in many data centers, such as Google [19] and Amazon [27].

Parallel with the popularity of containers is the widespread deployment of Remote Direct Memory Access (RDMA) in modern data center networks. RDMA offloads data transport into the RDMA NICs (RNICs), and provides kernel-bypassing, zero-copy interfaces, and abstraction of remote memory to applications. Thereby, it achieves extremely high throughput, ultra-low latency, and high CPU efficiency. Modern data centers have been widely deploying RDMA to accelerate distributed applications, such as machine learning [16, 26], distributed storage [13], databases [5, 33], etc.

However, live migration of containerized RDMA is not supported in today's data centers. The main reason is that the majority of RDMA states are maintained by RNICs due to the hardware offloading nature. These states are not available externally, thus unable to be migrated to another server's RNIC. A recent work, MigrOS [21], proposes an RDMA protocol extension to enable RDMA live migration. It requires modification of RNICs, thus not deployable in today's data centers. This paper aims to provide a software-based RDMA live migration solution that can be readily deployed over commodity RNICs. The vantage solution is to establish new pairs of RDMA communication between the migration destination and communication partners, and provide a virtualization layer to make the changes transparent to applications. Unlike the previous RDMA virtualization solutions [14, 16] that focus on providing sharing and isolation for RDMA I/O and network, RDMA live migration only requires the virtualization layer to hide the differences between migration source and destination (e.g., QP numbers, and access keys).

Nevertheless, realizing the solution is not straightforward due to the following challenges. First, it is challenging to minimize the blackout time during migration as RDMA communication setup time is quite long [32]. Second, virtualization is incurred during data transmission. Minimizing performance declines is not an easy task. Third, RNICs own inflight states of RDMA communications and directly interact with applications during data transmission. The RDMA driver cannot access these states. It is hard to handle the consistency of inflight states with an interposing layer efficiently.

In this paper, we provide `MigrRDMA`, a software-based live migration system for containerized RDMA applications on commodity RNICs. `MigrRDMA` adopts the pre-copy idea from memory migration of container live migration and uses it for RDMA migration – `MigrRDMA` pre-establishes new RDMA communications on the migration destination (and also communication partners for connection-oriented communications) before conducting the final stop-and-copy. Second, `MigrRDMA` lets the RDMA driver manage virtualization tables and maps the tables into the RDMA library in userspace. Applications use the mapped tables to translate the address and the translation is done in the RDMA library, which

APNet 2024, August 3–4, 2024, Sydney, Australia

ACM ISBN 979-8-4007-1758-1/24/08
https://doi.org/10.1145/3663408.3663416

ensures transparency. This design not only provides easy virtualization management but also provides high efficiency. Third, MigrRDMA adopts wait-before-stop approach – stop-and-copy starts only after all the inflight WRs are completed. During wait-before-stop, the MigrRDMA Lib processes completions of inflight requests on behalf of the application. During container restoring, the MigrRDMA Lib acts as a virtual RNIC to let applications process all inflight CQEs (Completion Queue Elements).

We implement MigrRDMA virtualization features in the RDMA driver and library, and also implement a CRIU [9] plugin to integrate MigrRDMA virtualization with this container live migration system. Evaluation results show that MigrRDMA only adds 0.7∼12.1 ms to the live migration blackout time and the application is not interrupted – only experiencing a minimal pause. Also, RDMA throughput is not affected during the brownout time. Besides, MigrRDMA virtualization layer only incurs 3% ∼ 9% additional CPU cycles in data transmission.

2 BACKGROUND

2.1 Container Live Migration

Container live migration is the procedure of moving containers from one server to another with minimal disruption to the applications running inside them. In modern data centers, it is one of the key technologies for server upgrades, data center maintenance, and load balancing, while ensuring that the migrated services continue running seamlessly. Major container platforms (e.g., Docker [10], runC [22]) have offered live migration support. Major cloud vendors (e.g., Google [19], Amazon [27], etc.) have enabled container migration in data centers.

Typically, live migration tools first pre-transfer some states. At this time, containers to be migrated are still running on the migration source. Then, the containers are paused and all the remaining necessary states are transferred to the migration destination. Live migration tools then restore the containers and re-activate them again. Some configurations are changed after containers continue running (e.g., network routing). Containers are completely unavailable when paused, and the duration between when they are paused and re-activated is called *blackout time*. During the pre-transfer and post-restore phase, although available, the migrated containers perceive performance declines because live migration tools are competing network bandwidth or CPU resources with running applications, or because changing configurations after restoring potentially causes performance jitter. This duration is called *brownout time*. The goals of live migration are to minimize the blackout time and to reduce the impacts of brownouts. The latter refers to not too long brownout time and minimized performance loss.

To ensure the migrated containers can continue seamlessly, one of the most important states that need migrating is the container's memory and storage. Pre-copy is the method most commonly used for memory migration [3, 4]. Before stopping the containers, live migration tools copy the memory pages iteratively and incrementally, and they conduct the final stop-and-copy when the amount of page modification is below a threshold. Pre-copy significantly reduces the blackout time when the migrated containers have a large amount of memory. In terms of storage migration, post-copy is also adopted [20]. Live migration tools only copy essential pages enough to restore the containers. When a page fault occurs during the runtime, the corresponding pages are fetched from the migration source.

Another critical state to be migrated is the container's network sockets – migrated containers can still communicate over their original sockets without explicitly closing and re-creating them. As containers share the underlying OS, the OS needs to provide mechanisms that enable live migration of the containers' sockets. TCP_REPAIR [8] was introduced to the Linux kernel starting from v3.5. It allows live migration tools (e.g., CRIU [9]) to extract the socket states (including send and receive queues, packet sequence number, etc.), and to inject these states into the sockets created on the migration destination. If inflight packets originating from or destined for the migrated containers are lost during live migration, the upper layers of the network stack will treat it as a normal packet loss event and do the packet loss recovery. As such, the migrated containers and their partners can continue their communication, although they may perceive minor packet drops.

2.2 Container Live Migration with RDMA

RDMA has now been widely adopted in data centers to boost the performance of online services [6, 7, 13, 25, 31, 34] as it achieves extremely high throughput, ultra-low latency, and high CPU efficiency. Different from TCP which runs in the host software stack, RDMA implements data transport in the hardware. That is the key reason why RDMA achieves those benefits. Supporting RDMA for containers in data centers has gained enormous traction [5, 12, 16, 24, 29].

However, modern data centers do not support live migration and RDMA for containers simultaneously. As data transport is offloaded to the hardware, RNICs manage most communication states. They are inaccessible from the software. MigrOS [21] proposes modifications to the RDMA protocol to enable live migration support on RNICs. The main idea is similar to TCP_REPAIR. However, as RDMA is running on the hardware, MigrOS requires RNIC modification, which is not supported by any commodity RNICs. The motivation of this paper is to provide an RDMA live migration solution over commodity RNICs while keeping the RDMA performance benefits and low extra overhead on live migration.

To avoid modification of the RNIC, we have to enable live migration purely in software. Due to the inability to extract internal states from commodity RNICs, the most suitable solution is to establish new RDMA communication between the migration destination and the communication partners. As one cannot migrate the hardware-resided states to the new communication, a software indirection layer is required to virtualize them to make their changes transparent to applications. Yet, we still face the following challenges:

***1)* Minimal blackout time**. Setting up an RDMA connection takes several milliseconds due to hardware limitations, which cannot be significantly reduced through batching [32]. Thus, the pre-establishment of new RDMA communications before the stop-and-copy phase is an attractive solution. However, an application's RDMA communications can only be set up by itself, instead of any components outside the process space. It is challenging to establish RDMA communication before the container is moved to the new location.

2) **Lightweight virtualization layer in the data path**. RDMA's data path bypasses the kernel. As a result, it is hard to introduce a virtualization layer in the kernel. Besides, with the software indirection layer, translation is incurred in the data path. It needs careful design to avoid serious performance declines in the data path.

3) **Preservation of inflight request consistency**. Once the migrated container is frozen, the original QPs still manage states of inflight work requests (WRs) that cannot be accessed by the software. It is hard for the live migration tool to preserve the consistency of inflight WRs – making the results of WR execution consistent as if no migration occurred.

3 DESIGN

3.1 Overview

To tackle the above challenges, this paper proposes the following design ideas:

- *RDMA information pre-copy* and *communication pre-setup*: Before the migrated containers are frozen, the live migration tools pre-copy the RDMA information to the migration destination. `MigrRDMA` advances the container restoring to the pre-copy phase and pre-establishes the new RDMA communication. When restoring the containers, live migration tools restore the rest of the container states.
- *Per-process mapping tables shared with userspace*: To minimize the changes in the data path, `MigrRDMA` manages the mapping tables in the kernel space and maps them to the RDMA library of each container. As such, the RDMA library can translate the related fields with low overheads. To make the translation efficient, the RDMA driver maintains per-process mapping tables.
- *Waiting for completion of inflight WRs* and *QP suspension in the software*: For the WRs on the flight at the time of suspension, live migration tools take over the operation on the QPs and wait for the completion of all inflight requests before conducting the final stop-and-copy. `MigrRDMA` on the communication partner suspends the QPs to prevent the communication partners from sending further WRs.

Figure 1 illustrates the workflow of our RDMA live migration. We extend the existing RDMA driver to provide a virtualization layer that virtualizes access keys of Memory Regions (MRs) registered to RNICs and Queue Pair Numbers (QPNs) of QPs. We also offer a plugin compatible with existing live migration tools to migrate RDMA. During the pre-copy phase, the live migration tool dumps and pre-copies the RDMA information (❶). Then, normal memory pre-copy is conducted (②). On the migration destination, the live migration tool pre-establishes the RDMA communication (❸ on the migration destination). If the communication is connection-oriented, the migration destination needs to connect the QP with the partner's. At this time, it is unable to connect to the partner's QP already established with the QP of the migration source. However, resetting the partner's QP directly will lead to a blackout, counteracting the benefits of RDMA pre-copy and pre-setup. Our design choice is to notify the live migration tool on the partner side to create a new QP, but the vQP is still mapped to the original QP (❸ on the partner). If the communication is connectionless, the partner side does

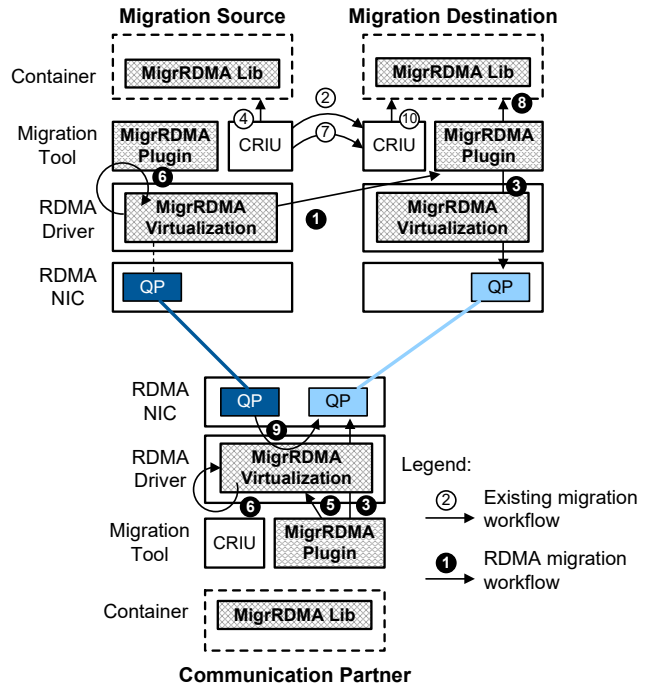

Figure 1: RDMA Live Migration Workflow

not need to create a new QP. When doing the final stop-and-copy, the live migration tool freezes the container to be migrated (④), and then notifies the partner side to suspend the corresponding QPs (❺). Any subsequent WRs posted on the QPs by the partners will be blocked inside the RDMA APIs until the migration destination has restored the containers and the RDMA communication can go on. Such design is to reduce the number of inflight WRs that `MigrRDMA` needs to wait for, and to prevent the exception on the partners caused by the errors of sending WRs to absent QPs. Both sides wait for the inflight WRs to be completed (❻). After that, the memory page modification in the final iteration is copied to the migration destination (⑦). The live migration tool updates the access key and QPN mapping of the migrated containers (❽), and notifies the partner to switch the vQP from the original QP to the new one and to restore the communication of the partner's QP (❾, only necessary for connection-oriented communication). Finally, the containers are restored (⑩).

3.2 RDMA Pre-copy and Pre-setup

To make new RDMA communication equivalent to the original one, the pre-copied information needs to reflect the roadmap of RDMA resource creation, including which MRs and QPs belong to the same Protection Domain (PD), and which CQs (Completion Queues) are attached to the send and receive queue of a QP. Besides, to ensure the processes can still use the original virtual memory addresses to access the DMA-mapped regions, the virtual memory addresses corresponding to the MRs, CQs, and QPs are also required in the RDMA pre-setup. Other information copied during the pre-copy phase includes the (virtual) access keys of MRs and QPNs of QPs on

the roadmap. Although they are only required in the restore phase, we also choose to copy them in the pre-copy phase for convenience. The total size of them is not large, so when to copy them does not affect the system.

After the necessary information is ready, the migration destination sets up the RDMA communication. When creating MRs, CQs, and QPs, the RDMA driver needs to do a page walk of the process's virtual memory regions and DMA-map the corresponding pages to the RNIC. Thus, to restore them, both the virtual memory addresses and physical pages need to be ready. As the container has not been launched during the pre-copy phase, the virtual memory addresses are invalid, and the physical pages are unknown. It is impossible to restore them firstly outside the process space and subsequently map them into the process. Therefore, `MigrRDMA` advances the container's restoring to the pre-copy phase – during pre-copy, only establish RDMA communication and restore necessary memory mappings, and during the restore phase, restore the rest.

In addition to the migration destination, the communication partner also needs to establish RDMA communication. To do this, the RDMA library registers a process signal handler when it is loaded. `MigrRDMA` sends a process signal to trigger the RDMA library to set up the QP. When switching the vQP, the RDMA driver updates the QPN mapping so that any subsequent data path operation on the vQP will be directed to the new QP.

3.3 Lightweight Virtualization

In the data path, what needs to be virtualized includes IDs/tokens allocated by RNICs and directly exposed to applications (e.g., QPNs, access keys). `MigrRDMA` needs an additional virtualization layer in the data path to hide the changes of QPNs and access keys after migration. This layer translates not only the QPNs and access keys assigned by the local RNICs, but also the ones of the other side. The QPs consist of connection-oriented QPs and datagram QPs. Communication over the datagram QPs requires applications to specify remote QPNs. For connection-oriented QP, the remote QPNs are only used in QP setup, which is in the control path. RDMA operations include two-sided and one-sided verbs. Two-sided verbs only require local access keys, while one-sided verbs also require remote access keys.

The goal of virtualization layer design is to make the translation of QPNs and access keys as fast as possible. For local QPNs, they are maintained in the QP data structure, and applications specify the pointer to the QP data structure in the RDMA APIs when transmitting data. As such, `MigrRDMA` slightly extends the QP data structure to make it contain both the virtual and physical QPN. The QP data structure contains generic fields and vendor-specific fields. The virtual QPN is stored in the generic fields, while the physical QPN is hidden in the vendor-specific fields. When applications post WRs, the MigrRDMA Lib tells the physical QPNs to the RNIC.

For access keys, however, each application needs to manage the values of access keys by itself, rather than specifying the pointer to the memory region's data structure. Therefore, `MigrRDMA` needs to maintain mapping tables of virtual-to-physical access keys. The access key translation needs to be added in the data path and be configurable by live migration tools for table updates. Directly spawning a background thread in the RDMA library to realize real-time updates would lead to changes in applications' behaviors. To minimize the changes to the applications' behavior, `MigrRDMA` maintains the mapping tables outside the applications and uses memory mapping to share the tables with applications. It is possible to realize the mapping tables in a userspace daemon, but the daemon may run into exception and causes the system failures. For robustness, we maintain the mapping tables inside the RDMA driver. To maintain the mapping tables efficiently, `MigrRDMA` maintains the mapping table in a per-process granularity. The RDMA library only queries the access keys and QPNs allocated to the process. As registering too many memory regions may reduce application performance, applications typically register huge memory regions and manage them into smaller chunks. Usually, the number of MRs per process does not exceed one hundred [17, 18, 28, 31]. So, the per-process mapping table is typically not large. Besides, `MigrRDMA` maintains the translation tables as arrays that store the physical access keys, and the corresponding index is the virtual access key exposed to the application. The access keys are virtualized within each process space. An attacker is unable to forge a virtualized access key and attack an RDMA-based service. Therefore, our virtualization does not raise extra security issues.

For translation of remote access keys, `MigrRDMA` caches the translation for each remote process. At first, arrays for remote access keys contain no entries. When MigrRDMA Lib translates remote access keys for the first time, it fetches the physical ones from a remote service. During migration, the migration source invalidates its all communication partners' remote access keys for the migrated container. For remote QPNs, the similar approach for remote access key translation can also be applied, and we omit the details here.

3.4 Inflight WR Consistency

To tackle the consistency issue induced by inflight WRs (the WRs already inside RNICs), `MigrRDMA` proposes a wait-before-stop approach. For the inflight WRs in the RNIC of the migration source, we leverage the mechanism of process signal handler offered by the current Linux kernel [15]. When the live migration tool stops processes of a container, the main workflow of the processes stops, except the signal handler which is called once the signal is caught by the process. Thus, when called after the process captures the stop signal, the signal handler traverses all the QPs, and keeps polling them until all the inflight WRs are completed. Whether a QP has inflight WRs can be identified by checking its head and tail pointers. The Linux kernel has offered several signals resulting in the process stopping, and when a `CONTINUE` signal is sent, the process can go on. `MigrRDMA` chooses `SIGTSTP` as the stop signal. For the inflight WRs in the RNICs of the partners, `MigrRDMA` introduces a flag for each QP. This flag is maintained in the RDMA driver and shared with the MigrRDMA Lib. The migration source notifies the partners to raise the flags of the corresponding QPs. When the MigrRDMA Lib on a partner detects a QP's flag is raised, it performs wait-before-stop, and responds to the migration source that all the inflight WRs are completed. After the wait-before-stop of both sides, the processes of the migrated container stop, and the live migration tool can copy all the states to the migration destination.

As the MigrRDMA Lib has taken over the data path operation, we need a mechanism to make sure applications can get the inflight WRs' CQEs (Completion Queue Elements) previously processed by the MigrRDMA Lib. `MigrRDMA` introduces a CQE-inflight buffer for each CQ, which is specifically used to store the CQEs of the inflight WRs. When the MigrRDMA Lib gets a CQE, it places the CQE into the CQE-inflight buffer. When applications poll the CQ, the RDMA library first checks whether the CQE-inflight buffer has CQEs. If so, it directly returns the earliest generated CQEs to the applications. Otherwise, the RDMA library polls the CQ as normal.

During the stop-and-copy phase, the communication partner may continue sending WRs to the migration source. The number of inflight WRs increases, and the waiting time becomes longer. To prevent partners' communication from the migration source, `MigrRDMA` introduces a suspension flag for each QP. When the stop-and-copy phase starts, `MigrRDMA` on the partner sets the flag of the QP. When the RDMA library finds that the flag is turned on, it blocks the WR posting inside the APIs. The RDMA library continues posting the WRs when it detects the flag is turned off again.

3.5 Discussion

3.5.1 Reducing virtualization overhead. The software indirection layer not only needs to translate the access keys and QPNs allocated by the local RNICs, but also requires to translate the ones from the remote side. As the RDMA access is managed by hardware, the access keys on the migration destination are likely to be different from those on the migration source. All the partners need to update their remote access key translation to access the migrated container. The current `MigrRDMA` design is to invalidate all access key translation on all partners. One possible optimization is that we keep tracking of rkey translation fetching on each node. Once a live migration happens, all fetched rkey translations are updated to all partners instead of a invalidation. Another optimization point of remote translation fetching is the fetching policy. Current `MigrRDMA` only fetches the key when it is requested. A prefetch policy can be added to further reduce the latency impact caused by fetching rkeys for one-sided operation.

The number of remote nodes that a container connects with can be very large [13]. Suspending all partners of a migrated container means notifying a large number of remote containers. As the suspension is during the stop-and-copy phase, the communication latency may dominate the migration blackout time. Furthermore, network reliability and performance variance make minimizing blackout time even more complicated. Similar to the access key virtualization, an approach similar to post-copy is also a possible solution. However, it is challenging since we need to identify whether a data path exception is due to container migration or the real exception in hardware or physical networks.

3.5.2 Optimizing Inflight Handling. In our current design, `MigrRDMA` needs to wait for the completion of all inflight WQEs before starting the final memory copy. This happens after the container stops. The blackout time can be very long if the inflight size is very large. The long blackout time may leads to SLA violation of applications. What's worse, applications may monitor instance status and treat the long blackout time as the abnormalies, resulting in breaking normal behaviors of applications. Thus, it is necessary to optimize inflight handling.

The bandwidth-delay product of data centers is usually not large. However, the number of bytes waiting in the RNIC can be much larger than the number of bytes on the wire. Thus, one possible approach to reduce blackout time is only waiting for the real inflight requests that have already been on the wire, and migrating the rest requests that have not been sent to the remote side. The problem of how to determine whether a request has been sent to the wire or not needs further investigation.

Another possible approach is to predict the container's stopping time and pre-suspend new request submissions. The goal is to let the migrated container and all partners have minimal inflight requests at the container stop time. This can also minimize the time waiting for inflight requests to finish.

3.5.3 Integration with Existing RDMA Virtualization Systems. Previous RDMA virtualization systems focus on sharing and isolation for multiple RNIC users. It is easy to integrate `MigrRDMA` with them. Software solutions like FreeFlow [16] use a software router to virtualize the RDMA request path. `MigrRDMA` can be easily added to the software router to enable live migration. For hardware solutions, integrating `MigrRDMA` is similar to the design proposed in this paper, the only difference is that the QP setup should be done through the virtual RNIC. For hybrid solutions like MasQ [14], we can easily add `MigrRDMA` software virtualization to the data path and the `MigrRDMA` connection management to the software control path.

3.5.4 Security Issue. `MigrRDMA` requires containers to use the provided RDMA library, which raises a security issue. An attacker may use a malicious RDMA library, which injects malicious codes in the application runtime to attack a normal RDMA-based service. Such an issue can be lethal. For example, attackers may leverage one-sided verbs to tamper the memory of the victims silently, but the victims are unable to detect that they are being attacked. Two mechanisms to mitigate the issue have been discussed in [23]. One is checking what RDMA library is being loaded to make sure only the MigrRDMA Lib can be loaded into the application. Another possible solution is leveraging binary code attestation supported as a feature of CPUs [2] to detect the malicious codes.

4 EVALUATION

4.1 Implementation and Evaluation Setup

Implementation. We implement a prototype of `MigrRDMA` based on Mellanox OFED 5.4 driver, including ~3,500 lines of C code (LoC) added in the RDMA kernel-space driver, ~5,000 LoC in the RDMA library, and ~2,500 LoC for the plugin of CRIU 3.18. The extended RDMA kernel-space driver constructs the roadmap that guides the live migration tool to establish new RDMA communication, and provides virtualization to hide the difference between the new communication and the old one. The RDMA library extends the existing RDMA APIs for RDMA restoring (e.g., `ibv_restore_cq`, an extension of `ibv_create_cq`, allows the process being restored to specify the original virtual memory address of the queue), and maintains the access key and QPN mapping shared with the RDMA driver. Preservation of inflight WR consistency is also realized in

Table 1: Migration Time of MigrRDMA (ms)

Test: send_bw Queue depth: 64 Message size: 16 KB

# of QPs	Baseline	MigrRDMA	
	Blackout Time	Blackout Time	Extra Downtime
1	67.5	68.6	1.1
2	69.5	70.5	1.0
4	70.7	71.4	0.7
8	71.9	72.9	1.0
16	72.5	74.3	1.8
32	73.8	79.8	6.0
64	74.6	85.1	10.5
128	77.5	89.6	12.1

the RDMA library. The CRIU plugin enables dumping of the RDMA roadmap and pre-establishment of RDMA communication.

Evaluation Setup. Our testbed consists of three servers, which represent a migration source, a migration destination, and a communication partner, respectively. Each server is equipped with dual 16-core Intel(R) Xeon(R) CPU E5-2698 v3 @2.30GHz CPUs, 256 GB memory, and a Mellanox ConnectX-5 100 Gbps RNIC. They are connected through an Arista 7260CX3-64 switch. Our evaluation consists of two parts. In the first evaluation, we migrate a container running an RDMA-based microbenchmark test (i.e., `perftest` [1]) and evaluate the blackout time during live migration. In the second evaluation, we evaluate the virtualization overhead by running `perftest` over `MigrRDMA` and the original RDMA and comparing their performance. Both evaluations use throughput test of the `perftest`, which tries its best to saturate the network. Thus, the send queues are almost always full during the tests.

4.2 Minimal Migration Overhead

To identify the baseline, we first prepare a modified `perftest` which only requests all the necessary memories and does not create any RDMA resources, and then migrate it using regular CRIU. Thus, the total size of the container is equivalent to the size of the container running a regular `perftest`. Then, we migrate the regular `perftest` using `MigrRDMA` and evaluate the blackout time. Table 1 shows the results of the migration. `MigrRDMA` only incurs 0.7∼12.1 ms extra blackout time. For the cases of large numbers of QPs (over 16 QPs), the extra blackout time is dominated by the waiting time of handling inflight work requests. We find that the blackout time is not that stable in the cases of small numbers of QPs. Identifying the major cause is one of our future work. Nevertheless, we observe no throughput declines during the RDMA pre-copy and pre-setup. Therefore, the brownout has nearly no effect on RDMA migration. The current implementation simply waits for completion of the WRs in the RNIC. As the QPs are almost always full in the throughput test, increasing the total queue depth will lead to more inflight WRs in the RNIC. As a result, the extra downtime increases with the number of QPs in our evaluation. There are many aspects to optimize the inflight WR handling (as mentioned in §3.5.2). We leave them as our future work.

Table 2: CPU Cycles of Each Operation

Operation	w/o virt	with virt	extra cycles	overheads
send	123.7	128.3	4.6	3.7%
recv	59.4	64.7	5.3	8.9%
write	125.0	133.3	8.3	6.6%
read	127.3	133.8	6.5	5.1%

4.3 Virtualization Overhead

To evaluate the overhead of virtualization layer, we measure and compare the CPU cycles of send, receive, write, and read of a single RC QP in the case with and without virtualization. We slightly modify the `perftest` to sample the CPU cycles of each invocation of these operations. The measurements of operations are all conducted under the message sizes ranging from 64 B to 1 KB separately, and each test runs for 5 seconds. We calculate the average of the sampled results during the last 4 seconds to find the steady-state CPU cycles. Therefore, the overheads of one-sided verbs do not take the remote access key fetching at the first translation into account. The CPU cycles under these message sizes have little difference, so we only show the CPU cycles under the message size of 64 B for each operation. Table 2 lists the results. Our virtualization only adds single-digit extra cycles in these operations, resulting in 3% ∼ 9% overheads in the data path.

5 CONCLUSION

This paper proposes `MigrRDMA`, a software-based live migration system to support RDMA live migration for containers readily deployable over commodity RNICs. Due to the hardware offloading features, RDMA live migration has unique challenges. `MigrRDMA` develops several novel mechanisms to tackle them, namely, pre-copy and pre-setup of RDMA, per-process mapping tables of access keys and QPNs shared with userspace, and inflight WR handling method to preserve the consistency. Evaluation shows 0.7∼12.1 ms extra blackout time during migration, and 3% ∼ 9% extra overheads in the data path.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the anonymous reviewers for their constructive comments. This work is supported in part by the National Key Research and Development Program of China (No. 2022YFB2901404), and by National Natural Science Foundation of China (NSFC) under Grant No. 62132007 and No. 62221003.

REFERENCES

[1] [n. d.]. GitHub - linux-rdma/perftest: Infiniband Verbs Performance Tests. https://github.com/linux-rdma/perftest.
[2] [n. d.]. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf.
[3] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2017. Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. 472–479. https://doi.org/10.1109/CLOUD.2017.67
[4] Mohamed Azab and Mohamed Eltoweissy. 2016. MIGRATE: Towards a Lightweight Moving-Target Defense Against Cloud Side-Channels. In *2016 IEEE Security and Privacy Workshops (SPW)*. 96–103. https://doi.org/10.1109/SPW.2016.28
[5] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong

Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 29–41. https://www.usenix.org/conference/fast20/presentation/cao-wei

[6] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2545–2558. https://doi.org/10.14778/3551793.3551813

[7] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/2901318.2901349

[8] Jonathan Corbet. 2012. TCP Connection Repair. https://lwn.net/Articles/495304/.

[9] CRIU. 2023. CRIU Main Page. https://criu.org/Main_Page.

[10] Docker. 2024. Docker Checkpoint. https://docs.docker.com/reference/cli/docker/checkpoint/.

[11] Docker. 2024. Redis – Docker Official Image. https://hub.docker.com/_/redis/.

[12] Parav Pandit Dror Goldenberg. 2019. Mellanox Container Journey. https://qnib.org/data/hpcw19/7_END_2_MellanoxJourney.pdf.

[13] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 519–533. https://www.usenix.org/conference/nsdi21/presentation/gao

[14] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3387514.3405849

[15] Michael Kerrisk. 2023. signal(7) – Linux Manual Page. https://man7.org/linux/man-pages/man7/signal.7.html.

[16] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 113–126. https://www.usenix.org/conference/nsdi19/presentation/kim

[17] Kwangwon Koh, Kangho Kim, Seunghyub Jeon, and Jaehyuk Huh. 2019. Disaggregated Cloud Memory with Elastic Block Management. *IEEE Trans. Comput.* 68, 1 (2019), 39–52. https://doi.org/10.1109/TC.2018.2851565

[18] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. 2024. Harmonic: Hardware-assisted RDMA Performance Isolation for Public Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1479–1496. https://www.usenix.org/conference/nsdi24/presentation/lou

[19] Victor Marmol and Andy Tucker. 2018. Task Migration at Scale using CRIU. https://www.slideshare.net/RohitJnagal/task-migration-using-criu.

[20] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. 2017. Voyager: Complete Container State Migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2137–2142. https://doi.org/10.1109/ICDCS.2017.91

[21] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. 2021. MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 47–63. https://www.usenix.org/conference/atc21/presentation/planeta

[22] Adrian Reber. 2016. Container Live Migration Using runC and CRIU. https://www.redhat.com/en/blog/container-live-migration-using-runc-and-criu.

[23] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 4277–4292. https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger

[24] Alibaba Container Service. 2019. Using RDMA on Container Service for Kubernetes. https://www.alibabacloud.com/blog/using-rdma-on-container-service-for-kubernetes_594462?spm=a2c41.12560487.0.0.

[25] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 317–332. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/shi

[26] TensorFlow. 2023. TensorFlow – Install Docker. https://www.tensorflow.org/install/docker.

[27] Luan Teylo, Rafaela C. Brum, Luciana Arantes, Pierre Sens, and Lúcia Maria de A. Drummond. 2020. Developing Checkpointing and Recovery Procedures with the Storage Services of Amazon Web Services. In *Workshop Proceedings of the 49th International Conference on Parallel Processing* (Edmonton, AB, Canada) *(ICPP Workshops '20)*. Association for Computing Machinery, New York, NY, USA, Article 17, 8 pages. https://doi.org/10.1145/3409390.3409407

[28] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 306–324. https://doi.org/10.1145/3132747.3132762

[29] Zhe Wang, Teng Ma, Linghe Kong, Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Guihai Chen, and Wei Cao. 2022. Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 639–654. https://www.usenix.org/conference/atc22/presentation/wang-zhe

[30] Ashton Webster, Ryan Eckenrod, and James Purtilo. 2018. Fast and Service-preserving Recovery from Malware Infections Using CRIU. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1199–1211. https://www.usenix.org/conference/usenixsecurity18/presentation/webster

[31] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[32] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. 2022. KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 121–136. https://www.usenix.org/conference/atc22/presentation/wei

[33] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proc. VLDB Endow.* 14, 10 (jun 2021), 1900–1912. https://doi.org/10.14778/3467861.3467877

[34] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. 2021. Octopus+: An RDMA-Enabled Distributed Persistent Memory File System. *ACM Trans. Storage* 17, 3, Article 19 (aug 2021), 25 pages. https://doi.org/10.1145/3448418