Software-based Live Migration for RDMA

Xiaoyu Li[♥♣] Ran Shu[♣] Yongqiang Xiong[♣] Fengyuan Ren[♥]

[♥]Tsinghua University, [♣]Microsoft Research
Beijing, China

ABSTRACT

Live migration is critical to ensure services are not interrupted during host maintenance in data centers. On the other hand, RDMA has been widely adopted in data centers, and has attracted both academia and industry for years. However, live migration of 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 that does not rely on any extra hardware support. `MigrRDMA` provides a software indirection layer to achieve transparent switching to new RDMA communications. Unlike previous RDMA virtualization that provides sharing and isolation, `MigrRDMA`'s indirection layer focuses on keeping the RDMA states on the migration source and destination identical from the perspective of applications. We implemented `MigrRDMA` prototype over Mellanox RNICs. Our evaluation shows that `MigrRDMA` adds little downtime when migrating a container with live RDMA connections running at line rate. Besides, the `MigrRDMA` virtualization layer only adds 3% ~ 9% extra overheads in the data path. When migrating Hadoop tasks, `MigrRDMA` only incurs an extra 3-second job completion time.

CCS CONCEPTS

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

KEYWORDS

Data Center, RDMA, Live Migration, Virtualization

ACM Reference Format:
Xiaoyu Li, Ran Shu, Yongqiang Xiong, and Fengyuan Ren. 2025. Software-based Live Migration for RDMA. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3718958.3750487

1 INTRODUCTION

Live migration enables server upgrades, server maintenance, and load balancing without interrupting the running services. It is one of the key technologies behind modern data centers [35, 44]. Modern data centers have widely adopted live migration for virtual machines (VMs) for years [11, 44, 46, 58], and there are also solutions for live migration of containers [35] or even bare-metal instances [21, 22].

At the same time, data centers have seen widespread deployment of Remote Direct Memory Access (RDMA). 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 [15, 42], distributed storage [5, 16], databases [6, 56], etc. As RDMA is running in the clouds, RDMA live migration is now one of the critical technologies in modern data centers.

However, today's data centers do not support live migration of RDMA. 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 [41], 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.

Nevertheless, realizing software-based RDMA live migration is not straightforward due to the following challenges. First, setting up RDMA communication is slow [53]. To reduce the blackout time, it is necessary to adopt communication pre-setup during memory pre-copy. Pre-copy is a live migration phase when the migrated service is partially restored on the migration destination and is still running on the migration source. Nevertheless, for ease of restoring the memory pages, live migration tools may map the application's memory temporarily at a location other than the application's original address. As the memory structure during pre-copy is different, it is difficult to restore the registered memory regions during pre-copy. Second, during communication, applications need to use the states managed by RNICs to identify different RDMA resources and to verify memory access authorization. The workflow bypasses the kernel. Thus, it is challenging to hide the differences between the old and new RDMA communications with a software-based solution. Third, RNICs still continue processing the work requests during live migration. The software is unable to get the states of inflight work requests from RNICs. It is hard to keep the execution of the work requests consistent using an interposing layer.

In this paper, we provide `MigrRDMA`, a software-based live migration system for RDMA applications on commodity RNICs. `MigrRDMA` introduces a software indirection layer in the RDMA driver, and mainly provides three features to support RDMA live

SIGCOMM '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/25/09
https://doi.org/10.1145/3718958.3750487

migration. First, MigrRDMA directly maps the RDMA-related memory structures to the application's virtual memory addresses before the live migration system starts memory restoration, and keeps their mappings unchanged during the subsequent iterations of pre-copy. For RDMA memory newly registered by the migration source after the memory restoration begins, its restoration is delayed to the stop-and-copy phase. Second, MigrRDMA provides efficient virtualization of the RDMA communications' states. The RDMA driver manages the translation table of virtual to physical values, and shares the table with the RDMA library. The RDMA library does the translation inside the userspace. For translation efficiency, MigrRDMA assigns the virtual access keys one by one, and uses an array to maintain the translation table. As such, the RDMA library can translate the states with low overheads. Third, MigrRDMA proposes wait-before-stop to handle inflight work request consistency. MigrRDMA suspends the RDMA communication, and waits for the completion of inflight work requests. Although the communication is suspended, applications can still execute computation tasks. After restoration, the indirection layer acts as a virtual RNIC to let applications process all inflight CQ entries.

We implement MigrRDMA virtualization features in the Mellanox OFED driver and library. We also implement a CRIU [13] plugin and extend runc [4] to integrate our solution with the existing container live migration workflow. Evaluation results demonstrate that MigrRDMA adds little downtime when migrating a container with RDMA connections running at line rate, while achieving low virtualization overheads, only 3% ∼ 9% extra overheads in the data path. MigrRDMA only incurs an extra 3 seconds in the job completion time when migrating a Hadoop task. Our prototype is available at https://github.com/hittingnote/migrrdma.

In conclusion, the main contribution of this paper is as follows:

- Propose the first software-based system to support RDMA live migration in data centers, which enables RDMA pre-setup, provides efficient virtualization of RDMA communication states, and realizes wait-before-stop to support RDMA live migration purely in the software.
- Implement a prototype of MigrRDMA, which integrates well with the existing container live migration system.
- Conduct evaluation on MigrRDMA, and the results show the great benefit from RDMA pre-setup, correctness and negligible overheads of wait-before-stop, and low overheads introduced in the data path.

Ethic: This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

In this section, we provide a brief background on the live migration of virtual machines, containers, and bare-metal instances. Then, we state our motivation for proposing software-based RDMA live migration.

2.1 Live Migration

Live migration is the procedure of moving running services from one physical server to another with minimal disruption. It is one of the key technologies behind the compute engine products of modern data centers [44] to achieve server upgrades, data center maintenance, and load balancing during the service runtime. Live migration has been widely adopted for VMs [11, 44, 46, 58]. There are also trends of enabling live migration of containers [35, 37] and even bare-metal instances [21, 22]. For example, Google [35] has enabled container live migration in data centers.

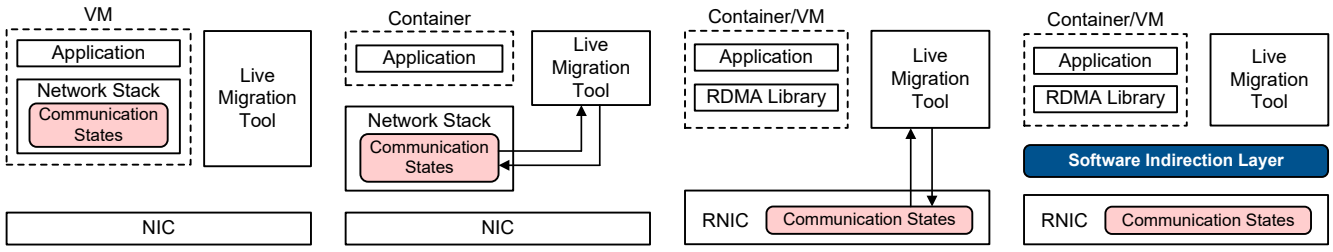
To keep transparent to the running services, live migration tools need to migrate and restore the internal states of the services. These states mainly include memory, storage, and networking. The memory/storage states contain two parts, i.e., virtual address tables and the contents of all pages. For VMs and containers, live migration tools map the same virtual addresses to new physical addresses, and write exactly the same contents to all the regions. For bare-metal instances, the guest OSs need to run lightweight hypervisors to introduce lightweight virtualization [10, 14] or even on-demand virtualization [21, 22] for live migration. The rest of the migration workflow is similar to that of VMs. The states of the memory/storage are typically very large. Therefore, directly adopting a stop-and-copy approach (stop the running services, copy the states, and restore the services) will incur too much blackout time. There are two approaches to reduce the blackout time, namely, pre-copy and post-copy.

For pre-copy [11], live migration tools copy the states of memory/storage to the migration destination and start restoring the services, while the services are still running on the migration source. The pre-copy is done iteratively. The first iteration copies all the memory/storage, and each subsequent iteration only copies the differences from the previous one. Upon each iteration, live migration tools on the migration destination merge the new differences with the previously restored states. The pre-copy terminates when the size of the modified pages is below a threshold, or when the bandwidth required to transfer the modified pages is so great that the running services may experience performance declines due to the network contention [11]. Then, live migration tools migrate and restore the rest pages during stop-and-copy.

For post-copy [19], live migration tools do not guarantee the consistency between the migration source and destination during stop-and-copy. Instead, they fetch all the memory/storage from the migration source after restoring the migrated instance. When applications on the migration destination need a region of memory/storage still absent on the migration destination, live migration tools fetch it from the source. Post-copy has limitations compared to the prior two approaches. First, applications' performance may be limited by the network when live migration tools handle the storms of page faults, or when the network is spotty. Second, the migration source needs to keep the memory/storage of the migrated services until the migration destination has fetched all of it. Application states may be lost if the network or source fails. Therefore, modern data centers typically adopt the combination of pre-copy and stop-and-copy to migrate the memory/storage [44].

Although reducing the blackout time, pre-copy and post-copy incur brownout time, during which services are available, but with performance declines. The major goal of live migration is to minimize the blackout time, and reduce the impacts of brownout.

The networking states mainly consist of two parts, namely, states of communication pipes created by applications (e.g., sockets for TCP/IP), and virtual networks. For the kernel-space network stack,



(a) TCP Migration (VM) (b) TCP Migration (Container) (c) MigrOS (d) MigrRDMA

Figure 1: MigrRDMA and Existing Solutions for TCP/RDMA Migration

migration of sockets is different in VM, bare-metal, and container environments. For VMs, all the socket states are maintained in the network stack of the guest OS. Therefore, as long as the live migration tools restore the guest OS, the sockets become alive again. A similar method is also applied for migrating bare-metal instances (Figure 1(a)). For containers, as they share the underlying OS, the host network stack needs to provide mechanisms that enable live migration of the sockets. TCP_REPAIR [12] was introduced to the Linux kernel starting from v3.5. It allows live migration tools 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 (Figure 1(b)). In terms of the userspace TCP/IP network stack (e.g., mTCP [24], Snap [36], etc.), the communication states are maintained in the migrated instance. Thus, the live migration of userspace TCP/IP stack is similar to Figure 1(a). For all cases, the inflight packets originating from or destined for the migrated services may be lost during live migration. The upper layers of the network stack will treat it as a normal packet loss event and perform the packet loss recovery. In terms of virtual networks, cloud providers need to ensure that packets destined for the migrated services will be forwarded to the migration destinations after live migration. To reduce the blackout time, operators store the new configuration in the network fabric during the pre-copy phase, and activate the new settings during the stop-and-copy step. Before the new configuration fully propagates across the network, operators make the migration source forward the packets it receives during the blackout to the migration destination [44].

2.2 RDMA Live Migration

RDMA has now been widely adopted in data centers to boost the performance of online services [7, 9, 16, 47, 52, 59] as it achieves extremely high throughput, ultra-low latency, and high CPU efficiency. Different from TCP which runs in the software, RDMA implements data transport in the hardware. That is the key reason why RDMA achieves those benefits.

However, modern data centers do not support live migration of RDMA. As RDMA offloads data transport to the hardware, RNICs manage most communication states. They are inaccessible from the software. MigrOS [41] introduces a mechanism similar to TCP_REPAIR inside the RNIC to enable extraction and injection of communication states (Figure 1(c))[1]. However, MigrOS requires modification to the hardware, which is not supported by any commodity RNICs.

Commodity RNICs do not provide sufficient network virtualization features inside the hardware to hide the differences between the migration source and destination. As a result, RDMA live migration requires explicitly notifying the communication partners that the migrated service has now changed to another location. For example, MigrOS extends the RoCEv2 protocol to enable notifying the partners without awareness of the software components. The partner suspends communication with the migration source during live migration, and finally restores communication with the migrated service running on the migration destination. Our work proposes communication suspension in the software to prevent further WRs from being posted on the wire during stop-and-copy (§3.4).

The motivation of this paper is to provide an RDMA live migration solution over commodity RNICs. As Figure 1(d) shows, we introduce an indirection layer in the kernel, which enables RDMA live migration purely in the software while keeping the RDMA performance benefits and low extra overhead of migration. However, software-based RDMA live migration is challenging in three folds:

***1)* Communication pre-setup during memory pre-copy**. Setting up an RDMA connection takes several milliseconds [53]. Setting up hundreds or even thousands of RDMA connections is common in data centers [16, 38]. Modern data centers typically require a short blackout time during migration, usually as short as tens of milliseconds [44]. Thus, establishing RDMA communication during stop-and-copy is unacceptable. RDMA pre-setup during the pre-copy phase is then an attractive solution. However, registering memory to RNICs is one of the critical steps to establish communications. During the pre-copy phase, memory restoration is complicated, and the structures of the partially restored memory are typically different from the memory structures of the migrated services. For example, when restoring memory in the pre-copy case, CRIU temporarily maps the applications' memory together to a consecutive region and does iterative restoration of memory contents. It remaps the memory region to the applications' virtual address when conducting the final iteration of memory restoration. Thus, it is difficult to register memory regions to RNICs using the applications' virtual memory addresses during partial restore (we will interchangeably use "pre-copy" and "partial restore" in the rest of the paper).

[1]MigrOS was originally designed for containers. This paper is trying to summarize the solutions from a higher view (live migration for all scenarios).

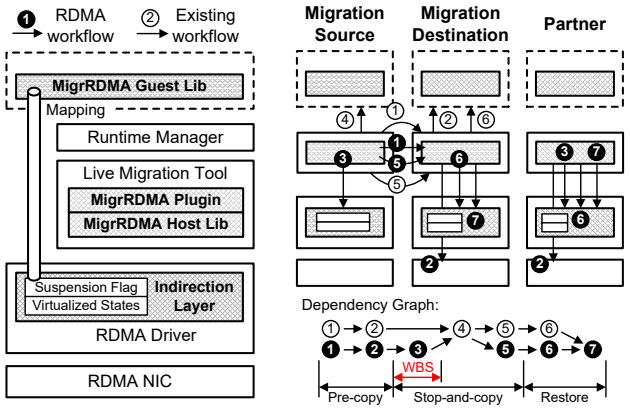
2) **Hiding differences between the original and new RDMA communications**. Many RDMA resources and their states are managed by RNICs. These states are exposed directly to the userspace. For example, after establishing communication, applications will get IDs of RDMA communications, and memory access keys, both allocated by RNICs. They are used at each time of data path operation, without the involvement of the OS. Live migration tools cannot control the management of access keys and IDs, therefore unable to keep the states the same between the original and new RDMA communication. Besides, for one-sided operations, applications also need to get the remote memory access keys of the remote side. It is even more challenging to hide the differences in memory access keys from the communication partners of the migrated services.

3) **Consistency of inflight work requests**. Even though live migration tools have frozen the services to be migrated from the software, the RDMA communications inside RNICs are still active. RNICs will still continue processing the inflight work requests (WRs) during the final dumping of the migrated services' states, raising consistency issues of inflight WRs. Live migration tools in the software are unable to get the states of inflight WRs directly from the hardware. Even worse, the migrated service is unaware of the one-sided operations issued by its communication partners. Therefore, it is hard for the software interposing layer to preserve the consistency of inflight operations – making their results consistent as if no migration occurred. It is possible to modify the communication to the initial states to make RNICs discard the inflight WRs, and replay these WRs after migration. However, stopping all the corresponding RDMA communications managed by RNIC is very slow, typically taking milliseconds or even seconds.

3 DESIGN

This paper proposes `MigrRDMA`, which mainly provides three features to support RDMA live migration:

- *Pre-setup of RDMA communications during partial restore.* During partial restore, `MigrRDMA` directly maps the RDMA-related memory structures to the application's virtual memory addresses. The indirection layer can use the original virtual memory addresses to register the memory. As such, we can set up all the RDMA communications during the partial restore. Like memory, `MigrRDMA` also guarantees the convergence of RDMA communication during live migration.
- *Efficient virtualization of the RDMA communications' states.* To hide the differences between the original and new communications, the software indirection layer virtualizes the states exposed to the userspace. These states fall into one of the two categories: they are either used by applications or just used inside the RDMA library. For the former category, `MigrRDMA` maintains the translation table in the kernel, and shares it with the RDMA library. `MigrRDMA` assigns the virtual access keys sequentially, and uses an array to maintain the translation table. For the latter category, `MigrRDMA` hooks inside live migration tools and updates the states after the memory restoration finishes, while keeping transparent to applications.



(a) Architecture (b) Workflow

Figure 2: `MigrRDMA` Overview

- *Wait-before-stop to preserve inflight WR consistency.* When the stop-and-copy starts, `MigrRDMA` suspends the RDMA communication to prevent further send WRs, and waits for all the inflight WRs to be completed. During this period, applications can still execute the computing tasks. `MigrRDMA` processes the completion notifications on behalf of the application during wait-before-stop. When restoring communication, `MigrRDMA` acts as a virtual RNIC to let the application process all the completion notifications previously processed by `MigrRDMA`.

3.1 Overview

Figure 2 illustrates our design overview. `MigrRDMA` adds three components to the original live migration system (Figure 2(a)). The indirection layer, residing in the RDMA driver, bookkeeps the minimal states enough to rebuild the RDMA resources. Besides, it shares the suspension flag of each QP, and the translation of memory access keys and communication IDs with the RDMA library through memory mapping. `MigrRDMA` Plugin calls the indirection layer to do RDMA checkpointing and restoration. We realize the `MigrRDMA` Plugin inside the live migration tool. `MigrRDMA` Host Lib and `MigrRDMA` Guest Lib are modified from the existing RDMA library. `MigrRDMA` Host Lib provides extra APIs for live migration tools to restore the RDMA states. `MigrRDMA` Guest Lib suspends communication over QPs, and translates the states based on what is shared by the indirection layer.

Figure 2(b) shows our live migration workflow. When pre-copy starts, the live migration tool pre-dumps and copies the states of memory to the migration destination (①). The live migration tool starts restoring the memory of the migrated services at the migration destination (②). Meanwhile, MigrRDMA Plugin calls the indirection layer to pre-dump the states of RDMA communication, and copies them to the migration destination (❶). MigrRDMA Plugins on the migration destination and partner call the indirection layer to create or destroy the RDMA communication (❷). The migration source needs to notify the partner through the network (omitted in the figure).

When the final stop-and-copy starts, MigrRDMA Plugin calls the indirection layer to raise the suspension flag (❸). When MigrRDMA Lib detects the suspension flag, it suspends communication over the corresponding QPs, and conducts wait-before-stop (WBS in the figure) to ensure no inflight WRs during the stop-and-copy. To keep RDMA's asynchronous semantics, during communication suspension, MigrRDMA Lib intercepts the WRs to be posted to RNICs, and pretends that they had been posted on the wire. Besides, the migration source needs to notify the partners to suspend communication and conduct wait-before-stop as well (omitted in the figure). The main logic is similar to that on the migration source. The only difference is that on the migration source, we suspend all the RDMA communications created by the applications, while on the partner side, we only suspend the RDMA communication destined for the migration source. The wait-before-stop terminates when all the inflight WRs are completed.

After the wait-before-stop has terminated on both the migration source and the partners, the live migration tool freezes the services to be migrated (④), and dumps and copies the differences of the memory states to the migration destination (⑤). On the migration destination, the live migration tool starts the final iteration of the memory restoration (⑥). Parallel with ⑤ and ⑥, MigrRDMA Plugin dumps and copies the differences of RDMA communication, plus additional information related to RDMA communication state virtualization (❺). It then calls the indirection layer to map the new RDMA resources into the restored processes (❻). After the memory restoration, MigrRDMA Lib on the migration destination and the partner post the intercepted WRs to RNICs, and replay on the new QPs the RECV WRs previously posted but without messages received yet (❼). Finally, the migration source reclaims all the resources of the services that have been migrated (omitted in the figure).

`MigrRDMA` supports concurrent migration of two services connected with each other. Specifically, the migration destinations of both services establish RDMA communication with each other. The migration destination of the first migrated service may have already set up connections with the service that starts to be migrated later. These connections will be closed.

`MigrRDMA` supports all the RDMA operations, which include SEND/RECV, READ, WRITE, and ATOMIC. Besides, `MigrRDMA` has considered all the features of ib_verbs APIs, including completion channels, on-chip memories, and memory windows.

3.2 RDMA Pre-setup during Partial Restore

Checkpointing the RDMA communication. An RDMA communication contains multiple resources, including protection domains (PDs), memory regions (MRs), completion queues (CQs), queue pairs (QPs), shared receive queues (SRQs), completion channels, on-chip memories, and memory windows, the latter four are optional. The completion channel is an abstraction over the RNIC's hardware mechanism that uses interrupts to notify the WR completion. The on-chip memory allows applications to use the memory of the RNIC for data transmission, reducing the overhead of PCIe transactions [50]. The memory window offers a supplementary layer over the MR to provide finer granularity of access authorization control [55]. Interested readers may refer to specifications [2] and manuals [48] for details.

As most of the RDMA communication states are maintained by the RNIC, it is impossible for us to dump those states. Instead, `MigrRDMA` maintains a copy of states in the indirection layer. These states are not the full set of states but the minimal states to rebuild RDMA communications on the migration destination. It is achieved by intercepting all control path calls and maintaining the roadmap of RDMA communication establishment. `MigrRDMA` not only maintains the information of each individual RDMA resource, but also the dependencies among the RDMA resources. To restore the states on the migration destination, `MigrRDMA` replays the control path calls. A resource may be created and later destroyed. To avoid unnecessary resource allocation and release during RDMA state restore, `MigrRDMA` deletes the corresponding resource creation log when the resource is destroyed.

During the partial restore, we not only pre-copy the memory image, but also pre-copy additional RDMA-related information for RDMA pre-setup. The live migration tool checkpoints the RDMA communication by interacting with the indirection layer. At the migration destination, `MigrRDMA` establishes RDMA communications equivalent to the original ones based on the checkpointed states of RDMA resources.

Registering MRs during partial restore. To restore the MRs on the migration destination, the indirection layer needs to use the applications' original virtual addresses for memory registration. The virtual address of the MR is maintained in the states of on-NIC live RDMA resources. However, the memory structure during partial restore is complicated, and the virtual address of the memory at this time is different from the application's virtual memory address. To tackle this issue, the live migration tool compares the address range of each MR recorded in the pre-dumped RDMA states with the memory table, and finds which memory structures contain the MRs to be registered. Then, it remaps these memory structures directly to the applications' original virtual addresses. Since the virtual address now equals the original virtual address, MigrRDMA Lib can call the indirection layer to register the MRs. The live migration tool checks whether each memory structures contain the MRs, and avoids remapping them to another virtual address.

At the start of the memory restoration, the live migration tool allocates the memory by itself to store the intermediate states, which will be released when it finishes memory restoration. To avoid the possible address conflict between the MRs and live migration tool's memories, we let the live migration tool restore the MR's memory structures before the memory restoration starts. Nevertheless, during memory pre-copy, the service running on the migration source may register new MRs. These MRs may conflict with the memory of the live migration tool. We restore the conflicting MRs at the end of stop-and-copy, after the release of the temporary memory allocated by the live migration tool. For non-conflicting MRs, we register them during the pre-copy phase. As applications typically register MRs at initialization, the new MRs registered during the pre-copy only constitute a small part of all the MRs in normal cases.

Mapping new RDMA resources into applications. During the stop-and-copy phase, `MigrRDMA` dumps additional RDMA information that is related to RDMA communication virtualization. Such information includes the virtual access keys of MRs and virtual

QPNs of QPs. During the final iteration of restore, MigrRDMA Plugin calls the indirection layer to update and virtualize all RDMA-related states to match the new ones.

Establishing new RDMA communication on partners. For connection-oriented communications, in addition to the migration destination, the communication partner also needs to establish a new RDMA communication with the migration destination for each QP connected with the migration source. One possible approach is to reuse the existing QPs for new connections leveraging the QP reset feature. When a QP is reset, it destroys the connection with the remote peer, and can re-establish a new connection with another QP. However, during pre-copy, we need to ensure the service to be migrated can communicate with its partners. If we reset QPs, the communication would be destroyed. Thus, QP resetting can only be done in the stop-and-copy phase. While, resetting QPs is as slow as setting up new connections, and thus would incur a long blackout time, against the goal of proposing RDMA pre-setup.

Our design choice is that the partner establishes new RDMA communications connected with the migration destination, while still continuing data transmission with the migration source. This requires all the partners to have enough spare QPs to set up new connections. Otherwise, RDMA pre-setup would fail. Nevertheless, modern RNICs have supported more than 10K QPs [51], while applications typically minimize the number of QPs they use to avoid performance scalability issue (e.g., 1,000 QPs [16]). Thus, the problem only happens at extreme cases.

RDMA-based applications usually use one CQ to get completions of multiple QPs. Those QPs may be connected to endpoints other than the migrated instance. Thus, for communication partners, just mapping the new QPs to another CQ would break the transparency. Instead, we let the old and new QPs share the same CQ on the partner. Other resources (e.g., PDs, SRQs if any) can all be reused on the partner. Thus, different from the migration destination that needs to create all the RDMA resources, the partner only needs to establish new QPs.

As the partner may have established connections with the servers other than the migration source, the partner needs to know for which QP it should establish a new one with the migration destination. To achieve this, `MigrRDMA` lets the live migration tool on the migration source notify all the partners. The notification message contains the network address of the migration destination and a list of physical QPNs of each partner (if any). For each physical QPN the partner receives, the partner establishes a new QP and communicates with the migration destination to exchange the new physical QPNs. To figure out which partner `MigrRDMA` needs to notify, and the list of physical QPNs to be sent to each partner, we add the fields of the destination physical QPN and the destination network address to the metadata of the connection-oriented QP. Communication pre-setup on the partner side does not incur blackout time, but there is still a brownout time. Finally, the communication partner also takes a step to virtualize the states to hide differences. Specifically, right before Step ❼ in Figure 2(b), the partner translates the original physical QPN to the virtual QPN (the related data structure will be introduced in §3.3), and maps the virtual QPN to the new QP.

3.3 Hiding Differences between Source and Destination

The newly established RDMA communications are different from the original ones in many aspects. `MigrRDMA` needs to provide a virtualization layer to ensure transparency for applications. We summarize all RDMA states that are different between the migration source and destination. These states fall into one of the four types according to their characteristics. We take different actions for each type to achieve efficient virtualization. Table 1 summarizes the states and their characteristics. We will discuss the detailed operations of hiding the differences one by one.

Local states that are hidden from applications. These states are managed in the RDMA library or driver. They include the registers, metadata, head and tail pointers, and the contents of the queues (i.e., QPs, CQs, and SRQs). For the register, metadata, and pointers, `MigrRDMA` only needs to update them to the new values. No virtualization is required for this type. For queues, `MigrRDMA` leverages the live migration tools to restore the queue contents.

Local states that have been virtualized. Some states are exposed to applications and have been virtualized for applications. The virtual address of the MR and on-chip memory belongs to this type. To hide the difference of physical addresses, `MigrRDMA` needs to guarantee that all the memory tables are correct. When registering an MR, the application needs to specify the previously allocated virtual memory address by itself. Thus, as long as the live migration tool restores the MR's memory structures (§3.2), we can restore the MRs by registering their original virtual memory addresses to the RNIC on the new location. In terms of the on-chip memory, the RDMA driver allocates the NIC's memory with the size specified by the application, then maps the NIC's memory into the application's virtual address space. Therefore, when restoring the on-chip memory on the new location, the live migration tool needs to allocate it with the same size, and remaps it to the original virtual memory address. The remapping can be done by calling the `mremap()` system call in Linux, which only changes the virtual memory address and keeps the physical address unchanged.

Local states that have not been virtualized. The third type is the local states that are used by applications, and the values are not virtualized. The local QP Number (QPN) and the local access key (*lkey*) of each MR belong to this category. `MigrRDMA` maintains translation tables from virtual to physical, or vice versa. Special considerations need to be taken for translation efficiency.

For local QPN, the translation happens only in the physical-to-virtual path. The indirection layer in the RDMA driver maintains the QPN translation table as an array. The index denotes the physical QPN, and the corresponding value is the virtual QPN. As the physical QPN is unique within each RNIC, each physical QPN has only one virtual QPN. The specification uses 24 bits to represent a QPN [2]. Thus, the array size is 2^{24}. This array is shared with MigrRDMA Lib of each process, which only has the read access to the array. When the application creates a new QP, the RNIC returns the local QPN in the QP context. `MigrRDMA` just sets the virtual QPN the same as the physical value. When the application processes the CQ entry, it needs to use the local QPN inside the CQ entry to identify the QP from which the current completed request is. `MigrRDMA` Lib translates the local QPN from physical to virtual

Table 1: Summary of States MigrRDMA Considers to Virtualize for Transparency

Exposed to App	Virtualized	Local or Remote	States	Solutions
No	N/A	Local	Queue register, queue metadata, queue pointer, queue content	• Leverage live migration tool to restore the content. • Live migration tool updates the register, metadata, and pointer after memory restoration.
Yes	Yes	Local	Virtual memory address of RDMA MR and on-chip memory	• For MR, restore the MR's memory structure (§3.2), and register the same virtual address to the RNIC on the new location. • For on-chip memory, remap it to the original virtual address after its allocation on the RNIC of the new location.
Yes	No	Local	Local QPN and local access key	• Manage virtual values and build translation tables.
Yes	No	Remote	Remote QPN and remote access key	• Fetch from the remote side and cache it locally.

when the application processes the CQ entry. The translation table is updated during RDMA pre-setup when the new physical QPN is assigned.

For *lkey*s, the translation happens from virtual to physical ones when applications post requests on QPs. To reduce table lookup overhead, MigrRDMA manages the virtual values in a dense way – the virtual values are assigned one by one. With this design, the indirection layer maintains the translation table as an array. The corresponding index is the virtual *lkey* exposed to the application. To avoid security issues raised by this simple key allocation method, MigrRDMA maintains the table at a per-process granularity. The process ID combined with virtualized *lkey* then becomes the key of translation table entries. An attacker is unable to forge a virtualized access key and attack an RDMA-based service under such a design. Admittedly, applications may register plenty of MRs, and thereby the array can be very large. Nevertheless, operators typically minimize the total number of the MRs (e.g., below one hundred [26, 34, 49, 52]) for performance concerns. In this case, the *lkey* translation does not incur massive overhead.

Remote states that have not been virtualized. The fourth type is the states that have not been virtualized like the previous type, but the values are maintained in applications running on remote nodes. Remote QPNs and remote access keys (*rkey*s, including both the MRs' and memory windows' *rkeys*) belong to this type. Applications usually use out-of-band channels to exchange remote QPNs and *rkey*s [18, 43]. The RDMA library or the RDMA driver is unaware of such an exchange. There are three cases that require MigrRDMA to translate remote QPNs or *rkeys*. 1) For remote QPN of connection-oriented communications, the translations are only required at connection setup, as the RNIC needs to use the physical QPN to connect with the other. The remote QPN will not be needed during the communication. 2) For remote QPN of datagram communications, translations are needed for each request. 3) For *rkey* of one-sided operations (i.e., READ, WRITE, ATOMIC), the translations are required for each request. For the latter two cases, MigrRDMA Lib caches the mapping of *rkey*s and remote QPNs to translate them locally. At first, the whole cache is invalidated. For the first time of translation from a particular *rkey* or remote QPN, MigrRDMA Lib fetches the corresponding physical one from the remote side. Then, the subsequent translation from those *rkey*s and remote QPNs is only done locally, without extra communication with a remote server. The overhead mainly comes from the remote fetching upon the first translation, which is amortized over all the subsequent translations. During live migration, the migration source invalidates all its communication partners' caches that store the *rkey*s and QPNs of the migrated service. After restoration, the partners need to fetch new physical *rkey*s and remote QPNs from the migration destination. To speed up the remote fetching, we can optimize it using pre-fetch or batch-fetch. This could be one of our future work to improve MigrRDMA.

3.4 Inflight Request Consistency

The design choice of MigrRDMA to handle inflight request consistency is wait-before-stop. When stop-and-copy starts, MigrRDMA Plugin calls the indirection layer to raise the suspension flag, and notifies the MigrRDMA Lib to conduct wait-before-stop. When all the inflight WRs are completed, the live migration tool does the final stop-and-copy. We have considered other methods such as drop-and-replay. However, we do not adopt it for two reasons: 1) replaying causes a similar time to wait-before-stop as their performance bottleneck is network throughput, and 2) discarding all the WRs on the QPs is time-consuming.

A critical issue to consider is how to conduct wait-before-stop while ensuring transparency to applications. The RDMA data path bypasses the kernel. MigrRDMA Lib is then the optimal point for wait-before-stop execution. MigrRDMA Lib is loaded within the process space of each application. We need to find a possible point to conduct wait-before-stop in the MigrRDMA Lib without affecting the applications. Our goal is to keep the asynchrony of RDMA operations during wait-before-stop, i.e., applications can still execute their computing tasks, instead of getting stuck by the communication suspension.

Communication suspension. To do this, MigrRDMA Lib spawns another thread (called wait-before-stop thread) when being loaded into each process space for the first time. This thread listens to the indirection layer, and keeps suspended until the indirection layer sends a signal. Thus, the wait-before-stop thread does not contend for CPU resources with the application's threads. When the indirection layer sends the pause signal, the wait-before-stop thread raises the suspension flags of the QPs. For the migrated service, all the QPs should be suspended. For the partners, only the QPs connected to the migrated service need to be suspended. When applications post WRs, MigrRDMA Lib will intercept the WRs and return without blocking if detecting the suspension flag of the QP.

The intercepted WRs are buffered in the memory. With this design, applications only perceive that they get the CQ entries slower than before, but they can deal with computation tasks as normal. Both sides of the RDMA communication will issue the intercepted WRs on the RNICs when the live migration tool restores the RDMA communication.

Wait-before-stop. Then, MigrRDMA Lib conducts wait-before-stop to ensure no inflight WRs during stop-and-copy. In the regular workflow of the CQ entry processing, for each CQ entry, the RDMA library identifies the QP the completed WR is from, and then updates the tail pointer of the SQ/RQ. The window capped by the head and tail pointers of the SQ/RQ is exactly the inflight WRs. Thus, the main idea is that the wait-before-stop thread keeps polling all the CQs until there are no inflight WRs of all the suspended QPs, and meanwhile application threads are still running. To make sure applications can consume the CQ entries as normal, we maintain a fake CQ for each real CQ. Upon each CQ entry, the wait-before-stop thread places the CQ entry to the fake CQ. If the application polls the CQ during wait-before-stop phase, MigrRDMA Lib will direct it to poll the fake CQ. Such a design reduces the communication blackout time, as applications can process the CQ entries during wait-before-stop.

For two-sided verbs, the receiver needs to post RECV WRs before the sender posts SEND WRs. The WRs in the receive queue of one side are more than those in the send queue of the other side. Therefore, for receive queues, we cannot use the same way to determine if all inflight requests are completed. Instead, there are no inflight receive WRs when and only when the number of two-sided verbs the peer has posted (denoted as *n_sent*) equals the number of completed receive WRs (denoted as *n_recv*). We add fields in the QP metadata to maintain the number of posted two-sided verbs and completed RECV WRs for each QP since its creation. If the number of posted two-sided verbs is non-zero, the wait-before-stop thread sends the *n_sent* to the other side. If receiving *n_sent* from the other side, the wait-before-stop thread compares *n_sent* and *n_recv*. For the inflight RECV WRs without the other side posting corresponding send operations, `MigrRDMA` will replay the RECV WRs on the migration destination and partners when restoring the communication.

Consistency of CQ entries not consumed by applications. For the migrated services, `MigrRDMA` creates new RDMA communications on the migration destination. Thus, the original CQ entries are lost. However, applications may still not consume all the CQ entries at the time of live migration. We need a mechanism to preserve the consistency of the CQ entries. Please note that we have introduced a fake CQ for each real CQ during wait-before-stop. `MigrRDMA` also migrates the fake CQ containing the unprocessed CQ entries. After restoration, MigrRDMA Lib first polls the fake CQ. When the fake CQ becomes empty, it polls the real CQ as normal and destroys the fake CQ. MigrRDMA Lib needs to translate the physical QPNs in the CQ entries from both the fake and real CQs back to the corresponding virtual QPNs. To do this, when wait-before-stop terminates, MigrRDMA Lib maintains a temporary translation table from the original physical QPN to virtual QPN for all the QPs. After restoration, if the CQ entry comes from a fake CQ, MigrRDMA Lib translates the QPN based on the temporary translation table. When finding a CQ entry from a real CQ, MigrRDMA Lib deletes the corresponding table entry (because there will no loger be CQ entries for the WRs of the old QPs). Then, it translates the QPN based on the QPN translation table managed by the indirection layer (mentioned in §3.3).

On the partners, the new QP share the same CQ with the old one. After the restoration of communication, the CQ contains the CQ entries for both the old QP and the new one. The QPN translation table of the indirection layer has recorded the entries for both the old and new QPs (i.e., old pQPN to vQPN when the application creates the QP, new pQPN to the same vQPN when the partner creates the new QP during partial restore). MigrRDMA Lib can naturally translate the QPNs recorded in all the CQ entries. When all completions belonging to old QPs are consumed, the old QPs and related QPN translation table entries are destroyed.

Consistency of CQ events. Handling CQ consistency in interrupt mode (using CQ events) is different from polling mode. As applications will process CQ entries immediately when notified by CQ events, there is no need to use the wait-before-stop thread to poll the CQ. What `MigrRDMA` does is to make sure all CQ entries of a CQ that uses events are processed before live migration. Thus, `MigrRDMA` tracks the number of unfinished CQ events in `MigrRDMA` Guest Lib. The number is increased when the application waits for an event, and decreased when an event is successfully handled. The absence of any unfinished CQ events (i.e., the number equals zero) is another necessary condition for the termination of wait-before-stop.

Handling buggy network situations. The live migration may happens in a spotty network. In this case, simply waiting for all the inflight WRs delivered would cause an unexpectedly long time during wait-before-stop. To handle this case, `MigrRDMA` sets an upper bound on the time elapsed for wait-before-stop. If the time expires and there are still WRs not completed, `MigrRDMA` directly starts stop-and-copy. After finishing restoring the applications, the migration destination and partners first replay the WRs previously posted but not completed, before replaying the WRs intercepted by MigrRDMA Lib.

4 IMPLEMENTATION

Our implementation is targeted for container live migration. Container live migration involves the following components: a CLI tool to manage the container runtime (e.g., runc [4]), and a live migration tool (e.g., CRIU [13]). The cloud manager calls runc to checkpoint and restore. Runc in turn calls CRIU to perform the corresponding commands.

The existing CRIU and runc do not offer the full ability for live migration. First, CRIU and runc do not support partial restore during the memory pre-copy. We add `PartialRestore` command in runc, which calls the restore command of CRIU. We also modify the restore logics of CRIU. We break the restore procedure of CRIU into two parts. The first part is partial restore, and the second part is full restore. Before the full restore starts, we add a logic inside CRIU to wait for the stop-and-copy signal. We extend runc with `FullRestore` command to signal CRIU through the UNIX socket. Second, runc does not support migrating all the processes of a container. Specifically, to start a container, users need to execute "`docker run`" or the combination of "`docker create`" and "`docker`

Table 2: Description of Extended Commands in Runc

Command	Description
`CheckpointRDMA`	Dump the container images containing the difference of memory and that of RDMA-related information.
`PartialRestore`	Execute the existing restore command of CRIU. The restore procedure of CRIU is broken into partial restore and full restore.
`FullRestore`	Signal CRIU to start full restore.
`Exec`	Extend the existing `Exec` command to support live migration feature.

Table 3: `MigrRDMA`'s APIs

Existing RDMA APIs	APIs for CRIU
`ibv_open_device()`	`ibv_restore_context()`
`ibv_alloc_pd()`	`ibv_restore_pd()`
`ibv_reg_mr()`	(Does not need extra API)
`ibv_create_cq()`	`ibv_restore_cq()`
`ibv_create_qp()`	`ibv_restore_qp()`
`ibv_modify_qp()`	(Does not need extra API)

`start`". For both cases, container starts an initial process. This initial process is eventually created by runc through its `Start` command. After the initialization, operators may start any number of non-initial processes in the container by executing "`docker exec`". They are created by runc through its `Exec` command. The current runc only provides a restoration option for `Start` command. Therefore, we extend `Exec` command to also support the restoration. To migrate all the processes of the container, we write a script to get the root process IDs of the initial and all the non-initial processes by reading the files maintained by Docker [3]. Then, we start a CRIU for each root process ID to do checkpointing and restoration.

To add support for RDMA live migration, we add `Checkpoint RDMA` command in runc. It dumps the container images containing the RDMA-related information. If a previous dump file exists when this function is called, it generates only the difference instead of the full set. RDMA pre-setup can only map memory at the beginning of restoration on the target. For the updated RDMA memory during pre-setup, its recovery is delayed to the end of full restore. Thus, we only need to dump RDMA states twice, one at the beginning of pre-copy, the other at the stop-and-copy. However, we still support multiple rounds of memory pre-copy. During partial restore, CRIU maps the RDMA-related memory structures to the original virtual memory addresses before the memory restoration starts, then establishes all the new RDMA communications. All the command extended in runc are listed in Table 2.

To evaluate the benefits of RDMA pre-setup, we implement another RDMA live migration workflow without communication pre-setup for comparison. For this case, we only do one dumping during stop-and-copy. The RDMA dumping logic is the same. In terms of restoring, we restore the RDMA after all the memory are restored.

We implement a prototype of `MigrRDMA` based on Mellanox OFED 5.4 driver, including ~4,500 lines of C code (LoC) added in the RDMA kernel-space driver, and ~11,000 LoC in the RDMA library. The kernel-space driver maintains all the required information to create equivalent RDMA communication. The MigrRDMA Lib in the host offers the extended APIs to CRIU (Table 3). The MigrRDMA Lib in the container translates QPNs and access keys based on the table shared by the RDMA driver, and handles inflight request consistency. We add around 3,200 LoC to CRIU, and around 1,100 LoC to runc. We do not modify any codes of Docker.

5 EVALUATION

This section conducts evaluations on our `MigrRDMA` prototype. The main conclusions from the evaluation results are:

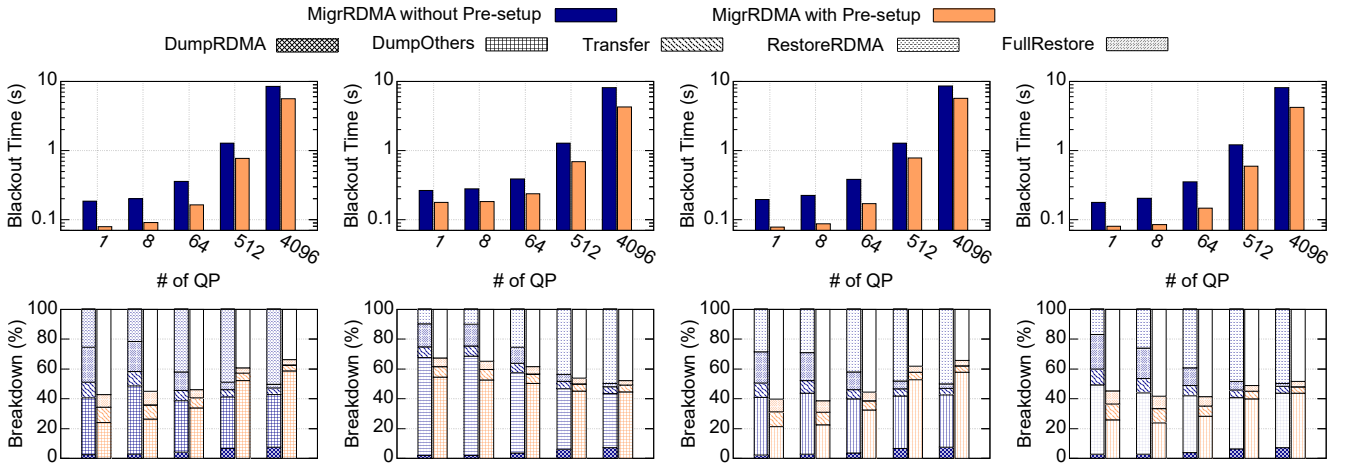
- The restoration of RDMA communication contributes significantly to the migration time. Adopting RDMA pre-setup reduces the blackout time by up to 58%.
- The overhead of wait-before-stop is relatively small compared to the total communication blackout time.
- The virtualization layer of `MigrRDMA` does not introduce massive overhead in the data path, only 3% ~ 9% extra CPU overhead for each operation, which means only 0.15 ~ 0.42 extra CPU cores to support 100 million RDMA operations per second.
- When migrating a running application, `MigrRDMA` incurs negligible performance implication experienced by the appliation. Although the total migration time may be long, the brownout has little effect on the performance.

5.1 Evaluation Setup

Our testbed consists of six servers, representing a migration source, a migration destination, and four communication partners, respectively. Each server has dual 16-core Intel Xeon CPU E5-2698 v3 CPUs, 256 GB memory, and a Mellanox ConnectX-5 100 Gbps RNIC. The RNICs are connected through an Arista 7260CX3-64 switch.

Our evaluation of `MigrRDMA` mainly consists of four aspects. The former three are performed under microbenchmark (i.e., `perftest` [1]), while the final one is for real-world applications. Firstly, we break down the blackout time of `MigrRDMA` under the case without and with partial restore. Secondly, we evaluate the correctness and overhead of wait-before-stop. We will show that the wait-before-stop has guaranteed the consistency of inflight WRs, and its overhead is small. Thirdly, we evaluate the performance implication of `MigrRDMA`. We will assess the overhead of the software indirection layer introduced by `MigrRDMA`, and then measure the real-time throughput when live migration occurs. Finally, we use `MigrRDMA` to migrate an RDMA-based Hadoop [28]. We use the binary code of RDMA-based Hadoop without any modification. Thus, we can demonstrate the transparency to applications.

To evaluate `MigrRDMA`, we extend the standard `perftest` in three evaluations for correctness checking, one-to-many communication pattern generation, and virtualization overhead measurement. It does not mean that `MigrRDMA` requires modification of applications.



(a) Migrated Service is Posting Out-bound WRITE

(b) Migrated Service is Serving In-bound WRITE

(c) Migrated Service is Posting SEND

(d) Migrated Service is Posting RECV

Figure 3: Breakdown of `MigrRDMA`'s Blackout Time

5.2 Benefits of RDMA Pre-setup

We implement `MigrRDMA` for the case without and with RDMA pre-setup, and break down their blackout time. The blackout time consists of five parts (if any): DumpRDMA, DumpOthers, Transfer, RestoreRDMA, and FullRestore. For the case with RDMA pre-setup, the blackout time only includes DumpOthers, Transfer, and FullRestore. In terms of the case without RDMA pre-setup, all the five parts are included in the blackout time.

Figure 3 shows the results of the breakdown. As the number of QPs increases, the elapsed time of RestoreRDMA grows significantly (when the number of QPs reaches 4,096, RestoreRDMA takes nearly 50% of the blackout time in the case without pre-setup). Adopting RDMA pre-setup can significantly reduce the blackout time. As the number of QPs increases, the time of DumpOthers increases significantly even for MigrRDMA with pre-setup. This is due to the inefficient CRIU implementation for large and complicated memory structures which has been reported before [41]. For `MigrRDMA` with pre-setup, the elapsed time of DumpOthers in the case of migrating the sender (Figure 3(a) and 3(c)) increases more rapidly than in the case of migrating the receiver (Figure 3(b) and 3(d)). We speculate that it is because the memory structure of the sender is more complicated than that of the receiver.

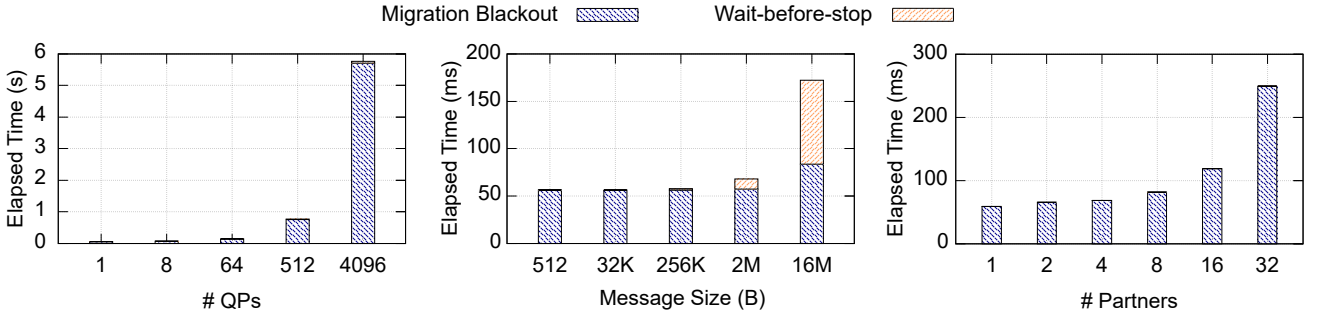
5.3 Correctness of RDMA Live Migration

We need to confirm that after migration, applications can get the completion of all the WRs in order, without duplication, lost, and content corruption. RDMA has provided a WR ID field in each WR that is populated by applications. Upon WR completion, the RNIC fills the the WR ID in corresponding CQ entry [2]. This allows applications to check whether each individual WR is completed or not. Therefore, we extend `perftest` to populate the WR ID with a sequence number we define for each QP. By checking the order of the WR ID returned from the CQ entries, we can identify whether all the WRs are completed in order, without duplication and lost. In terms of the content of the CQ entry, we check if any field is conflict with local inferred value. During the evaluation, there is no error information that we added printed out, indicating that we have guaranteed the consistency of inflight WRs.

5.4 Overhead of Wait-before-stop

In this set of experiments, we assess how much overhead wait-before-stop incurs. We change three factors to evaluate their effect on elapsed time of wait-before-stop: 1) the number of QPs, 2) the message size of inflight WRs, and 3) the number of partners. Besides, we compare the elapsed time of wait-before-stop to the migration blackout time under the varying three factors, separately. As Figure 4 shows, in most cases, wait-before-stop contributes little to the communication blackout time, and the elapsed time of wait-before-stop is less than that in theory[2]. The difference between the measured and theoretical values is because the RNIC has completed a part of the WRs by the time when MigrRDMA Lib enters wait-before-stop. In Figure 4(b), the measured elapsed time is 6× that in theory for the message size of 512 B. This is because the CPU overhead of the wait-before-stop thread's processing becomes dorminant when the total inflight bytes is small. In terms of wait-before-stop under different number of partners (Figure 4(c)), we extend `perftest` to support one-to-many communication pattern. The migrated container starts one `perftest` and creates n QPs, while each of the n partners starts one `perftest`, creates one QP, and connects it to the migrated container. Wait-before-stop on the partner sides and on all the RDMA-based processes of the migrated containers are in parallel, respectively. Thus, there are more WRs completed before MigrRDMA Lib enters wait-before-stop.

[2]The bandwidth test of `perftest` posts operations in best-effort mode. Thus, the elapsed time of wait-before-stop in theory is $inflight\_msgsize/link\_rate$. Taking Figure 4(a) as an example, the time equals $\#QP \times 4\ KB \times 64/100\ Gbps$.



Wait-before-stop (WBS) only (ms)

# QP	1	8	64	512	4096
WBS	0.02	0.09	0.91	9.42	70.91
Theory	0.02	0.17	1.34	10.74	85.90

(a) Wait-before-stop with Different # QPs (4 KB Message, 1 Partner)

Wait-before-stop (WBS) only (ms)

MsgSize	512	32K	256K	2M	16M
WBS	0.02	0.12	1.20	10.84	86.85
Theory	3 µs	0.17	1.34	10.74	85.90

(b) Wait-before-stop with Varying MsgSize (1 QP, 1 Partner)

Wait-before-stop (WBS) only (ms)

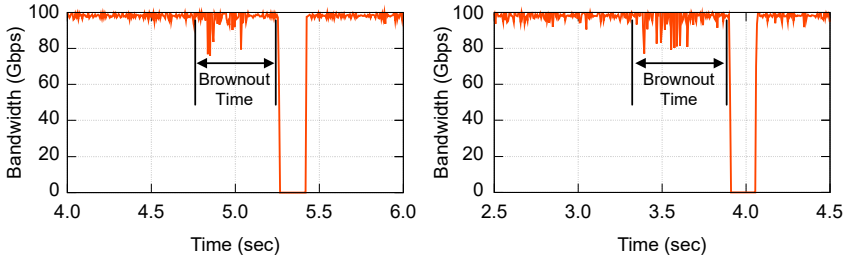
nPartner	1	2	4	8	16	32
WBS	0.02	0.02	0.02	0.03	0.04	0.10
Theory	0.02	0.04	0.08	0.17	0.34	0.67

(c) Wait-before-stop with Different # Partners (1 QP for each partner, 4 KB Message)

Figure 4: Overhead of Wait-before-stop (Queue Depth: 64)

Table 4: 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%



(a) Migrate Sender Side, Measure Receiver Side

(b) Migrate Receiver Side, Measure Sender Side

Figure 5: Performance Implication during Live Migration

5.5 Performance Implications

The performance implications come in two parts. First of all, no matter whether the container is migrated or not, the virtualization layer is always involved in the data path. We need to evaluate how much overhead `MigrRDMA` introduces in each kind of data path operation. Second, live migration impacts the performance of applications. We need to evaluate how much implication `MigrRDMA` causes during live migration.

5.5.1 Virtualization Overhead. To evaluate the overhead of the 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 are conducted under the 64 B message sizes for 5 seconds. As the CPU is the bottleneck under such message size, the uncontrollable CPU cycles for busy polling are excluded. Table 4 lists the results. Our virtualization only adds 4.6 ∼ 8.3 extra cycles in these operations, resulting in 3% ∼ 9% overheads in the data path. Assuming typical cloud servers with 2 ∼ 3 GHz CPU clock frequency, the per-WR overhead is about 1.5×10^{-9} to 4.2×10^{-9} CPU cores. Supporting 100 million RDMA operations per second (such throughput is just an extreme case) only requires 0.15 to 0.42 CPU cores.

5.5.2 Implications during Live Migration. To evaluate the performance implication on applications, we measure the application throughput during live migration. However, `perftest` does not support fine-grained throughput measurement. Thus, we leverage counters provided by Mellanox RNICs [39] that indicate how many bytes have been sent or received. By sampling the counters and the current timestamp repeatedly, the real-time throughput can be evaluated. The log we print shows 5-ms time granularity, which is enough to evaluate the performance implication of live migration. In this evaluation, we migrate a container running `perftest` that transmits 2-MB messages using one-sided verbs through 16 QPs. At the same time, we measure the real-time performance on the partner side. The cases of migrating the sender and receiver sides are both evaluated.

Figure 5 shows the real-time performance of the partner side. When migrating the sender side (Figure 5(a)), the data transmission on the partner side does not involve the CPU due to the characteristics of one-sided verbs. Thus, what the partner does during live migration is pre-establish new RDMA communications during partial restore. However, the partner side perceives slight performance

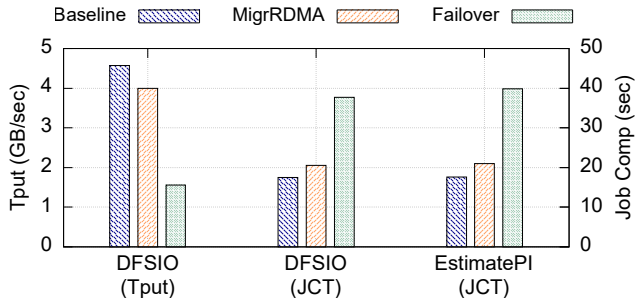


Figure 6: Migration of RDMA-based Hadoop

implications during the brownout time. This phenomenon is due to the resource contention on RNIC and was first reported by Kong et al. [27]. When full restore starts, the sender is down, thus the partner does not receive any messages. The blackout time lasts about 150 ms. When migrating the receiver side (Figure 5(b)), the partner needs to pre-establish new RDMA communications while transmitting data simultaneously, which incurs extra contentions on CPU cache and memory. As a result, the partner perceives slightly higher performance implications than in the case of migrating the sender side. When the sender side starts its wait-before-stop, the receiver side enters into a blackout. The blackout time is also about 150 ms.

In summary, the performance implication of the container during live migration is not significant. Besides, although the brownout time may be long as establishing all the RDMA communication is slow, the brownout has little impact on the application performance.

5.6 Migration of Real-world Applications

Finally, we use `MigrRDMA` to migrate a real-world RDMA-based application. The application we choose is Hadoop File System (HDFS) [17, 23]. We use the binary of the RDMA-based Hadoop [28] without any modification.

In the evaluation, we start a master and two slave containers on two servers, respectively. We submit a task on the master node, and the master node distributes tasks to one of the servers. During the runtime of the task, suppose the operator needs to restart the server running the slave container for maintenance. With `MigrRDMA`, operators can migrate the slave container to another server. However, without RDMA live migration scheme, operators leverage the Hadoop's native failover mechanism for the application's reliability. Specifically, the master node detects the loss of the slave node. It then re-distributes the tasks to a backup container on another server, and recovers their runtime on the migration destination based on the log of the tasks. In this evaluation, we compare the application-perceived throughput (Tput) and job completion time (JCT) of baseline, `MigrRDMA`, and failover.

Figure 6 shows the results of two kinds of tasks provided by Hadoop, TestDFSIO and EstimatePI. Compared to failover, applications experience much fewer performance declines during live migration. For the throughput of DFSIO, there is only a 12.5% throughput loss when using `MigrRDMA` to migrate the tasks. In contrast, for failover, the throughput loss is up to 65.8%. In terms of the JCT of DFSIO and EstimatePI, there are only the extra 3 seconds for RDMA live migration, versus the extra 20 seconds the failover incurs. EstimatePI does not calculate the throughput result. Thus, `MigrRDMA` provides benefits for RDMA-based applications when they need to be migrated to another physical machine.

6 DISCUSSION

`MigrRDMA` vs. MigrOS. Here we compare the performance of `MigrRDMA` vs. MigrOS. MigrOS [41] needs to modify the RNIC. The authors do not provide a hardware prototype but provides implementation over SoftRoCE [33] for features verification. As SoftRoCE has severe performance declines, the current MigrOS prototype is not a good choice for performance comparison. Thus, we instead examine the key differences between `MigrRDMA` and MigrOS, and provide a theoretical analysis. Although current MigrOS does not consider memory pre-copy [41], we suppose MigrOS had achieved it for a fair comparison. The performance of pre-copy is dominated by memory allocation and access characteristics. Therefore, the major differences between MigrOS and `MigrRDMA` lie in stop-and-copy.

Stop-and-copy consists of three steps: 1) waiting for the RDMA communication to a safe state for live migration, 2) transferring all the states to the new location and restoring at the final iteration, and 3) replaying what applications have previously posted but still not issued on the wire. Only the second step is the service blackout time, and all of the three steps constitute the communication blackout time. For the waiting and replaying steps, `MigrRDMA` waits for all inflight requests to be finished. While, MigrOS chooses to stop the communications on migration source and let RDMA packets naturally propagate the information to communication partners. During the replaying step, all non-acknowledged packets and pending WRs are transmitted. As the performance bottleneck is the amount of data that needs to be transmitted, `MigrRDMA` and MigrOS have almost the same performance for the total time of these two steps. In terms of the second step, MigrOS needs more time to extract and inject the states from and to the RNIC, respectively, and to modify the QP state to STOP. In contrast, `MigrRDMA` does not need to get/set the detailed states from/to the RNIC. As the metadata is stored in memory, `MigrRDMA` can leverage the existing memory live migration scheme to migrate the RDMA states. Thus, MigrOS takes longer time than `MigrRDMA` for the states transferring. In conclusion, the blackout time of MigrOS is longer than that of `MigrRDMA`.

`MigrRDMA` vs. other RDMA virtualization models. `MigrRDMA` provides a virtualization layer to ensure transparency of live migration. There are other models with the capability of ensuring such transparency, e.g., FreeFlow [25] and LubeRDMA [30]. FreeFlow provides virtual RNIC instances in the software, and naturally supports transparency. FreeFlow virtualizes the entire queue on the data path which causes high overhead [18]. While `MigrRDMA` only virtualizes the minimal required states with extremely low overhead. LubeRDMA aims at RDMA fault tolerance while keeping transparency to applications. It also has the ability to hide the differences between old and new RDMA communications by virtualizing the queue IDs, *lkey*, and *rkey*. The paper does not include design details of how LubeRDMA virtualizes them. According to the private communication with the authors, the major difference is

how to manage the virtual values and translate the virtual ones to physical ones. LubeRDMA uses a linked list to manage the virtual *lkey* and *rkey*. Searching a physical *lkey* and *rkey* needs to walk along the entire linked list. LubeRDMA optimizes it by moving the found *lkey*/*rkey* mapping to the head of the linked list. However, it still suffers from performance declines if the application accesses different MRs. MigrRDMA assigns the virtual *lkey*/*rkey* one by one and maintains the mappings as an array (§3.3). Thus, MigrRDMA incurs much lower overhead in the data path, and our solution can be adopted to further optimize LubeRDMA.

MigrRDMA in hybrid case. MigrRDMA does not support migrating the service with at least one of its partners running on non-MigrRDMA environments (i.e., the server installs the regular RDMA driver and library) because we cannot conduct wait-before-stop on those partners. Besides, communications between MigrRDMA and non-MigrRDMA environments cannot use one-sided operations without special design. This is because the non-MigrRDMA environment cannot exchange virtual to physical mapping of *rkey* with the MigrRDMA environment. To solve this, we add a negotiation mechanism in MigrRDMA Guest Lib to detect if the other side supports MigrRDMA. If not, MigrRDMA will exclude virtualization for this communication.

7 RELATED WORKS

RDMA virtualization. RDMA virtualization has gained enormous traction in both academia and industry [18, 25, 31, 32, 40, 45, 49, 57]. Their main focus is resource sharing, isolation, and network virtualization. For example, HyV [40] mainly provides virtualized RDMA I/O for VMs. FreeFlow [25] and MasQ [18] provide RDMA virtualization for virtual private clouds. Justitia [57] achieves fine-grained QoS policies over RDMA through queue virtualization. Other works also focus on sharing the limited on-NIC resources through virtualization to avoid performance degradations. LITE [49] provides a kernel-level indirection layer to share the underlying RDMA resources. ScaleRPC [8] designs a time-division multiplexing to share the underlying QPs and MRs. MigrRDMA's virtualization mainly focuses on hiding the changes in RDMA communications when migrating an RDMA-based application to a new location.

Migration on RDMA-enabled systems. Some other works provide fast migration features by leveraging RDMA. Huang et al. [20] and Wei et al. [54] leverage RDMA to transfer the VM's/container's states to reduce both the migration time and performance impact on hosted applications. MigrRDMA focuses on the live migration of applications running RDMA, but uses TCP to transfer the states. It is possible to integrate these works with MigrRDMA to achieve faster live migration. There is also research that provides communication abstraction over RDMA, and has considered live migration features. For example, SocksDirect [29] provides sockets over RDMA fabric. As SocksDirect builds an abstraction layer to translate RDMA interface to socket interface, it can naturally virtualize the states inside the abstraction to hide differences, and ensure the consistency of inflight packets. Achieving communication pre-setup is also straightforward – switching the socket to TCP during RDMA pre-setup, and switching it back to the new RDMA communication when it is ready. MigrRDMA focuses on migrating applications directly using the RDMA fabrics, where there are unique challenges to provide lightweight virtualization to hide the changes, ensure inflight consistency, and achieve communication pre-setup to reduce the blackout.

8 CONCLUSION

This paper proposes MigrRDMA, a readily deployable software-based system to enable RDMA live migration over commodity RNICs. MigrRDMA develops several novel mechanisms to tackle the unique challenges of RDMA live migration, namely, a separate RDMA-related memory restoration to enable RDMA pre-setup, lightweight virtualization to hide differences between migration source and destination, and efficient wait-before-stop solution to preserve inflight WR consistency. Evaluations show the great benefit from RDMA pre-setup (nearly 50% optimization compared to the case without RDMA pre-setup) and low overheads of wait-before-stop, and only 3% ∼ 9% extra overheads introduced in the data path. The evaluation for real-world applications demonstrates that MigrRDMA provides benefits for RDMA-based applications when they need to be migrated to another server, achieving only a 12.5% throughput loss and an extra 3-second JCT.

ACKNOWLEDGMENTS

We gratefully acknowledge the anonymous reviewers and our shepherd, Stewart Grant, for their constructive comments and suggestions. 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] 2007. InfiniBand Architecture Specification Volume 1. https://www.afs.enea.it/asantoro/V1r1_2_1.Release_12062007.pdf.
[3] 2024. Docker. https://www.docker.com/.
[4] 2024. runc. https://github.com/opencontainers/runc.
[5] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 49–67.
[6] 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
[7] 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
[8] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 14 pages. https://doi.org/10.1145/3302424.3303968
[9] 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
[10] Kevin Cheng, Spoorti Doddamani, Tzi-Cker Chiueh, Yongheng Li, and Kartik Gopalan. 2020. Directvisor: Virtualization for Bare-metal Cloud. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/3381052.3381317

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *2nd Symposium on Networked Systems Design & Implementation (NSDI 05)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi-05/live-migration-virtual-machines

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

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

[14] Takaaki Fukai, Takahiro Shinagawa, and Kazuhiko Kato. 2021. Live Migration in Bare-Metal Clouds. *IEEE Transactions on Cloud Computing* 9, 1 (2021), 226–239. https://doi.org/10.1109/TCC.2018.2848981

[15] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/3651890.3672233

[16] 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

[17] Apache Hadoop. 2021. https://hadoop.apache.org/.

[18] 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

[19] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy Live Migration of Virtual Machines. *SIGOPS Oper. Syst. Rev.* 43, 3 (July 2009), 14–26. https://doi.org/10.1145/1618525.1618528

[20] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K. Panda. 2007. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *2007 IEEE International Conference on Cluster Computing*. 11–20. https://doi.org/10.1109/CLUSTR.2007.4629212

[21] Jaeseong Im, Jongyul Kim, Jonguk Kim, Seongwook Jin, and Seungryoul Maeng. 2017. On-demand Virtualization for Live Migration in Bare Metal Cloud. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 378–389. https://doi.org/10.1145/3127479.3129254

[22] Jaeseong Im, Jongyul Kim, Youngjin Kwon, and Seungryoul Maeng. 2023. On-Demand Virtualization for Post-Copy OS Migration in Bare-Metal Cloud. *IEEE Transactions on Cloud Computing* 11, 2 (2023), 2028–2038. https://doi.org/10.1109/TCC.2022.3179485

[23] Nusrat Sharmin Islam, Dipti Shankar, Xiaoyi Lu, Md. Wasi-Ur-Rahman, and Dhabaleswar K. Panda. 2015. Accelerating I/O Performance of Big Data Analytics on HPC Clusters through RDMA-Based Key-Value Store. In *2015 44th International Conference on Parallel Processing*. 280–289. https://doi.org/10.1109/ICPP.2015.79

[24] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[25] 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

[26] 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

[27] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. 2023. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 31–48. https://www.usenix.org/conference/nsdi23/presentation/kong

[28] Network-Based Computing Laboratory. 2021. High-Performance Big Data (HiBD). http://hibd.cse.ohio-state.edu/.

[29] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. Socksdirect: Datacenter Sockets can be Fast and Compatible. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 90–103. https://doi.org/10.1145/3341302.3342071

[30] Shengkai Lin, Qinwei Yang, Zengyin Yang, Yuchuan Wang, and Shizhen Zhao. 2024. LubeRDMA: A Fail-safe Mechanism of RDMA. In *Proceedings of the 8th Asia-Pacific Workshop on Networking* (Sydney, Australia) *(APNet '24)*. Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3663408.3663411

[31] Liran Liss. 2015. Containing RDMA and High Performance Computing.

[32] Liran Liss. 2015. RDMA Container support. https://www.openfabrics.org/images/eventpresos/workshops2015/DevWorkshop/Tuesday/tuesday_08.pdf.

[33] Liran Liss. 2017. The Linux SoftRoCE Driver. https://www.openfabrics.org/images/eventpresos/2017presentations/205_SoftRoCE_LLiss.pdf.

[34] 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

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

[36] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413. https://doi.org/10.1145/3341301.3359657

[37] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. 2008. Containers Checkpointing and Live Migration. In *Proceedings of the Linux Symposium*, Vol. 2. 85–90.

[38] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. 2021. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 212–227. https://doi.org/10.1145/3477132.3483576

[39] NVIDIA. 2022. Understanding MLX5 Ethtool Counters. https://enterprise-support.nvidia.com/s/article/understanding-mlx5-ethtool-counters.

[40] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. 2015. A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey) *(VEE '15)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/2731186.2731200

[41] 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

[42] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 691–706. https://doi.org/10.1145/3651890.3672265

[43] 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

[44] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. 2018. VM Live Migration At Scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA) *(VEE '18)*. Association for Computing Machinery, New York, NY, USA, 45–56. https://doi.org/10.1145/3186411.3186415

[45] 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.

[46] Bin Shi, Haiying Shen, Bo Dong, and Qinghua Zheng. 2022. Memory/Disk Operation Aware Lightweight VM Live Migration. *IEEE/ACM Transactions on Networking* 30, 4 (2022), 1895–1910. https://doi.org/10.1109/TNET.2022.3155935

[47] 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

[48] Mellanox Technologies. 2016. Mellanox Adapters Programmer's Reference Manual (PRM). https://network.nvidia.com/files/doc-2020/ethernet-adapters-

programming-manual.pdf.

[49] 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

[50] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

[51] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. 2023. SRNIC: A Scalable Architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1–14. https://www.usenix.org/conference/nsdi23/presentation/wang-zilong

[52] 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

[53] 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

[54] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[55] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. 2023. Patronus: High-Performance and Protective Remote Memory. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 315–330. https://www.usenix.org/conference/fast23/presentation/yan

[56] 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

[57] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. 2022. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1307–1326. https://www.usenix.org/conference/nsdi22/presentation/zhang-yiwen

[58] Jiechen Zhao, Ran Shu, Lei Qu, Ziyue Yang, Natalie Enright Jerger, Derek Chiou, Peng Cheng, and Yongqiang Xiong. 2024. SmartNIC-Enabled Live Migration for Storage-Optimized VMs. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems* (Kyoto, Japan) *(APSys '24)*. Association for Computing Machinery, New York, NY, USA, 45–52. https://doi.org/10.1145/3678015.3680487

[59] 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