TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Chair of Operating Systems

Fine-grained OS Control over High-performance Networking

Maksym Planeta

Dissertation

to achieve the academic degree

Doktoringenieur (Dr.-Ing.)

Referee

Prof. Dr.-Ing. André Brinkmann

Fachreferent

Prof. Dr. Christof Fetzer

Supervising professor

Prof. Dr. rer. nat. Hermann Härtig

Submitted on: 26th April 2024
Defended on: 4th September 2024

To the memory of Rimma Govtva,
who passed away on the 13th of November, 2021.
May she rest in peace.

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Chair of Operating Systems

Abstract

In the evolving public cloud environment, RDMA (Remote Direct Memory Access) networks surpass traditional TCP/IP in performance but face integration hurdles across applications, hardware, and network stacks. This complexity has even prompted cloud providers to develop custom NICs and protocols for a better performance-usability balance. This thesis focuses on how operating system (OS) techniques can enhance RDMA network usability with minimal performance trade-offs.

A significant hurdle in the usability of RDMA networks is kernel bypass, which removes the OS from the communication dataplane. While effective in High-Performance Computing, kernel bypass is problematic in cloud environments, where diverse applications rely on OS support for resource sharing. Removing kernel bypass, as is done in TCP/IP, is not an option, as it would nullify the performance benefits of RDMA. Therefore, the challenge is to make the RDMA network dataplane more controllable with minimal changes to the existing RDMA network stack.

To address this, the OS can interpose on the RDMA dataplane continuously or intermittently to enhance overall control over RDMA communication. Continuous interposition redirects the dataplane through the OS kernel, offering control with minimal impact, enabling tasks like performance monitoring and rate limiting. Intermittent interposition, ideal for scenarios demanding low overhead, alternates communication between a high-performance bypass and an OS-managed manipulation phase, allowing for the implementation of, for example, transparent live migration for RDMA applications. These methods, continuous and intermittent interposition, significantly improve OS control over the RDMA dataplane, facilitating wider adoption and commoditization of RDMA networks in cloud infrastructures.

Acknowledgement

This work would not have been possible without the support of my supervisor, colleagues, friends, and family members. This support came in many forms, ranging from professional collaboration to personal encouragement. Despite this thesis being primarily a result of my work, to acknowledge all these contributions, I use "we" instead of "I" in this thesis.

First and foremost, I would like to thank my supervisor, Prof. Hermann Härtig, for his support throughout my PhD journey. The environment he created in the Operating Systems Group at TU Dresden, albeit challenging, has been designed to foster good systems researchers and engineers. I am grateful for the opportunity to be part of this group and to learn from him and my colleagues.

I would like to thank my colleagues at the Operating Systems Group for their support and collaboration. In particular, I would like to thank Michael Roitzsch, with whom I could always talk about my research, or anything else, and who has been a great mentor. Also, I would like to thank my office mate, Jan Bierbaum, for the technical and non-technical discussions, work collaboration, help with almost everything, and just for being a good friend. I wish him all the best in his future career. Furthermore, I would like to thank Hermann Härtig, Michael Roitzsch, Jan Bierbaum, and Martin Küttler for their feedback on this thesis.

I would like to thank my students, whom I had the pleasure to supervise. I learned a lot from them, and I hope they learned something from me. In particular, Viktor Reusch, Yaoxin Jing, and Philipp Schuster directly contributed to the work presented in this thesis.

I would like to thank my external collaborators, in particular, Prof. Torsten Hoefler and his group at ETH Zurich. I also would like to thank Richard Pitts and Mike Riley, with whom I had the pleasure of working as part of the Oracle for Research project.

I would like to thank my friends and family members for their support and encouragement. In particular, my parents have been role models for me and inspired me to pursue a career in computer science. Without their hard work, it would not have been possible. Finally, I would like to thank my fiancée, Daphne Antony, for her support and encouragement throughout all the years. I am grateful for all the debugging sessions, architecture discussions, and paper writing we shared.

Contents

List of Figures

List of Tables

Symbols and Acronyms

ACK	Acknowledgement
AETH	ACK Extended Transport Header
AH	Address Handle
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ATS	Address Translation Service
AWS	Amazon Web Services
BTH	Base Transport Header
CM	Connection Manager
CNP	Congestion Notification Packet
CoRD	Converged RDMA Dataplane
CPU	Central Processing Unit
CQ	Completion Queue
CQE	Completion Queue Entry
DCQCN	Data Center Quantized Congestion Notification
DMA	Direct Memory Access
DNAT	Destination Network Address Translation
DPA	Data Path Acceleration
DPDK	Data Plane Development Kit
DPU	Data Processing Unit (a more capable version of a SmartNIC)
DVFS	Dynamic Voltage and Frequency Scaling
eBPF	extended Berkeley Packet Filter
ECN	Explicit Congestion Notification
EFA	Elastic Fabric Adapter
EPT	Extended Page Table
FCP	Fastcall Provider
FCT	Fastcall Table
FPGA	Field Programmable Gate Array
GCP	Google Cloud Platform
GID	Global Identifier
GPU	Graphics Processing Unit
GUID	Globally Unique Identifier

GXF	Guarded Exception Levels
HPC	High Performance Computing
I/O	Input/Output
IOMMU	Input/Output Memory Management Unit
IP	Internet Protocol
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
ISA	Instruction Set Architecture
KPTI	Kernel Page Table Isolation
LID	Local Identifier
MDS	Microarchitectural Data Sampling
MMIO	Memory-mapped I/O
MMU	Memory Management Unit
MPI	Message Passing Interface
MPK	Memory Protection Keys
MR	Memory Region
MRN	Memory Region Number
MSR	Model Specific Register
MTT	Memory Translation Table
MTU	Maximum Transmission Unit
NACK	Negative Acknowledgement
NIC	Network Interface Controller
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access
ODP	On-Demand Paging
OS	Operating System
PD	Protection Domain
PFC	Priority Flow Control
POSIX	Portable Operating System Interface
PPR	Peripheral Page Request
PRI	Page Request Interface
PRS	Page Request Service
PRQ	Page Request Queue
PSN	Packet Sequence Number
QP	Queue Pair
QPN	Queue Pair Number
RC	Reliable Connection
RD	Reliable Datagram
RDMA	Remote Direct Memory Access
RNR	Receiver Not Ready
RoCE	RDMA over Converged Ethernet
RQ	Receive Queue
RR	Receive Work Request
RTR	Ready to Receive
RTS	Ready to Send
SLOC	Source Lines of Code
SoC	System on Chip

SQ	Send Queue
SQD	Send Queue Drain
SQE	Send Queue Error
SRQ	Shared Receive Queue
SR	Send Work Request
SRD	Scalable Reliable Datagram
SRQ	Shared Receive Queue
SR/IOV	Single Root I/O Virtualization
TCP	Transmission Control Protocol
UC	Unreliable Connection
UD	Unreliable Datagram
UDP	User Datagram Protocol
UTCB	User-level Thread Control Block
VF	Virtual Function
VM	Virtual Machine
VXLAN	Virtual Extensible LAN
WC	Work Completion
WQE	Work Queue Entry
XRC	Extended Reliable Connection
XDP	eXpress Data Path

1 Introduction

High Performance Computing (HPC) and Cloud architectures are converging: the former is advancing towards greater elasticity and higher resource utilization, while the latter is increasingly geared towards handling high-performance workloads. However, current HPC Cloud systems remain partitioned: vast sections lack high-performance networking support, with only a few segments offering such capabilities. This thesis studies how Operating Systems (OSs) can aid in HPC and Cloud convergence by making high-performance networks more accessible to cloud and data center applications. This approach can significantly enhance performance within Cloud environments.

The convergence of HPC and Cloud networks is challenging because these networks have very different architectures. High-performance networks explicitly separate communication into *control plane* and *data plane* operations and take extensive measures to optimize data plane operations. Control plane operations are responsible for setting up, managing, and tearing down connections. Although the control plane in high-performance networks looks quite different from the control plane in traditional networks, the OS still has significant control over the RDMA control plane.

The situation differs for data plane operations, which are responsible for sending and receiving messages over the established connections. The primary approach to optimizing the data plane is to delegate much of the traditional OS functionality to user applications and the hardware [Pet+14; DPD13; Inf15]. As a result, the OS does not participate in data plane operations and has little control over them.

Unfortunately, if the OS does not execute a data plane operation, it loses control over the data plane because it cannot change parameters of the operation for the purposes of resource allocation or virtualization. As a result, high-performance networks trade off *hard* properties, such as latency and bandwidth, for *soft* properties, such as deployability and usability. This thesis investigates how the aforementioned shift in performance balance has been addressed so far and proposes two novel OS architectures for high-performance systems that enhance OS control over high-performance network communication.

Upon closer inspection, modern high-performance network architectures, such as Remote Direct Memory Access (RDMA), utilize several techniques such as kernel-bypass, zero-copy, busy polling, offloading [1], and lossless flow control to enhance data plane performance. Kernel-bypass allows the application to access the Network Interface Controller (NIC) directly, avoiding overhead for crossing the user-kernel boundary. Zero-copy enables the NIC to access application memory directly, avoiding overhead for copying message content between kernel and user memory. With busy polling, the application continuously checks the NIC's message queues to avoid the overhead incurred by interrupt processing. Offloading allows the application to delegate some operations to the NIC, relieving precious CPU resources for application logic. Lossless flow control ensures that no message is lost, avoiding the overhead of retransmission. Together, these techniques impose on the user application a complicated programming model, forcing existing POSIX-based applications to be rewritten to benefit from high-performance networking [Enb+22].

However, these techniques circumvent "rich and robust" OS services [AMK22, p. 228], reducing the OS capabilities to manage applications and enforce policies [He+20]. This approach forces the OS to rely on the NIC to implement security and resource-sharing policies [Zha+22; Kim+19; He+20]. Generic hardware-based solutions that rely on IOMMU [Rot+21; Tar+20] or VXLAN [Sim+20] are inflexible and offer only a fraction of what an OS can do. Vendor-specific hardware-based solutions improve controllability for the OS [Bur21b; Fir+18], but are not portable and are not always available [Lin16]. In turn, to achieve high performance, the application must be rewritten in a new API, bring its own NIC drivers, and perform most of the scheduling and resource management itself [Yan+17; DPD13; Ope22]. As a result, the OS's role becomes almost insignificant for high-performance applications.

The OS community realizes that sometimes a do-nothing OS is the best choice [Ger+16; Jam+17; Gia+10; WLH19]. Still, many applications need high-performance communication *and* also expect the abstractions and services of a full-featured OS [Zhe+13; FSB19; Zha+19; Sad+21; Mar+19]. The conflict between the need for a classical OS architecture and the need for high-performance communication is fundamental, so one can only choose the right trade-off for a specific use case.

To regain OS control over high-performance networks, the OS can interpose on the data plane and do so either at the software level or with the assistance of the device. Such OS interposition can be either continuous [Kim+19] or intermittent [Les+17]. Through *continuous interposition*, the OS controls the state of the active connection and all messages sent and received over it. Through *intermittent interposition*, the OS is in control of the active connections only during selected phases of the connections' lifetime. Continuous interposition is more intrusive and resource-intensive, whereas intermittent interposition allows the OS to enforce fewer policies.

This thesis outlines and generalizes the existing landscape of OS-level interposition techniques for high-performance network data planes. Then, it studies hardware-supported intermittent interposition and software-level continuous interposition in the context of two separate real-life examples. First, software-level continuous interposition helps to implement several OS-level resource management policies. Second, hardware-supported intermittent interposition enables the OS to live migrate high-performance applications transparently with zero performance penalty to the application outside the migration phase. Although continuous interposition is more intrusive, we show that, in many cases, realistic high-performance applications experience

[1]*One-sided* or RDMA communication is a type of offloading.

only a very small overhead. Overall, both methods demonstrate that fine-grained OS control can be achieved with little impact on application performance and complexity.

1.1 Continuous interposition

The main idea behind continuous RDMA data plane interposition is to return the OS to the data plane, but without abandoning other high-performance techniques. In this regard, continuous interposition is the most straightforward approach to enable an OS-controllable data plane. Such interposition allows for arbitrary OS policies to be enforced throughout the entire application runtime.

The existing methods for continuous interposition typically offer novel OS and application architectures [Wei+21; TZ17]. To adopt any of these approaches, the application must be rewritten to use a new API. Given that these existing approaches adjust many “knobs” simultaneously, it is difficult to understand which aspect of the novel architecture is crucial for efficient continuous interposition. In this context, the question is, what modifications would minimally impact the application architecture and its performance?

To answer this question, we enabled continuous interposition by redirecting the RDMA data plane through the OS kernel, providing the OS with the interception point for all communication operations [Pla+23]. Although the resulting architecture added approximately 1.5 µs of overhead to the RDMA operations, we did not observe performance degradation above 1% in multiple high-performance applications. Considering that in some cases the overhead was much higher, we also proposed a new OS layer, which allows the OS to interpose on RDMA communication with less than 100 ns of added overhead [Mie+22]. Overall, we show that a careful implementation of the interposition mechanism minimally affects the performance of RDMA operations while providing flexible OS control over the RDMA connections.

1.2 Intermittent interposition

Intermittent interposition splits communication into two phases: *bypass* and *interposition*. During the bypass phase, the OS does not interpose on the communication, allowing the application to communicate directly with the NIC. During the interposition phase, the communication either pauses or is routed through the OS, enabling the OS to enforce policies or change the state of the connection. This approach allows the OS to control high-performance data plane communication with little to no overhead. On the other hand, not being able to inspect every message limits the OS’s capabilities.

To demonstrate that intermittent interposition can be effective in complex use cases, this thesis demonstrates an implementation of transparent live migration of RDMA applications. By interposing on the RDMA connections only during migration, the application does not experience runtime overhead. Transparency and zero runtime overhead are crucial to ensure that non-migrating applications are not penalized, but these properties have been traditionally challenging to achieve in high-performance networks [Pfe+15; Pic+16].

To ensure transparency, the application should not require modifications and should not get disrupted by migration. One of the key challenges for transparency is that the OS must prevent message loss during the migration operation. Such message loss can occur in two scenarios: during application checkpointing and immediately after migration. When the OS checkpoints an application, newly arriving messages may concurrently modify the application state but not be captured in the checkpoint. After migrating the application, the application's communication partners may send messages to the old location before learning of the new location. To maintain zero runtime overhead outside the migration phase, the OS may interpose on the RDMA connections only during the migration phase.

In this thesis, intermittent OS interposition achieves transparency and zero overhead through minimal modifications to the RoCE protocol, a popular low-level RDMA communication protocol [Pla+21]. These protocol modifications allow the OS to place the RDMA connections of the migrating application into a quiescent state, eliminating the risk of losing messages during migration. The prototype implementation of the proposed method includes modifications to a software implementation of the RoCE protocol and modified versions of kernel- and user-level RDMA device drivers. In our prototype, we added only a few instructions to the critical path of the RDMA data plane, which did not result in any measurable performance penalty. Furthermore, we integrated the prototype into the Docker container runtime to demonstrate that minimal modifications are sufficient for complex use cases.

1.3 Thesis Structure

This thesis is structured as follows: Chapter 2 provides the necessary background information about existing high-performance network architectures, specifically regarding RDMA communication. Chapter 3 outlines the landscape of existing continuous interposition techniques and introduces CoRD, a new continuous interposition technique focused on backward compatibility. Furthermore, Section 3.3 presents the fastcall architecture, which aims to reduce the overhead caused by continuous interposition. Chapter 4 presents intermittent interposition and introduces MigrOS, a new OS runtime that uses intermittent interposition to enable transparent RDMA application live migration. Finally, Chapter 5 concludes the thesis and outlines future work.

2 Background and Related Work

Among high-performance networks, Remote Direct Memory Access (RDMA) networks are the fastest and the most well-known. Therefore, in this thesis, we primarily focus on RDMA networks. This chapter outlines the key concepts of RDMA networks and the performance-improving techniques they employ.

In a narrow sense, Remote Direct Memory Access (RDMA) networks are networks that support RDMA *read* and *write* operations. These operations allow the application to access the memory of the remote host directly, without the need for the remote host's CPU to be involved in the data transfer. In practice, though, remote read and write operations are just one component of RDMA networks.

RDMA networks differentiate themselves from traditional TCP/IP-based networks at the application, OS, device, and network levels. For example, user-level RDMA applications may use an InfiniBand verbs-based communication API [Mel15], instead of the socket API [POS17]. And at the network protocol level, there is an independent InfiniBand-based communication protocol [Inf15], instead of Ethernet.

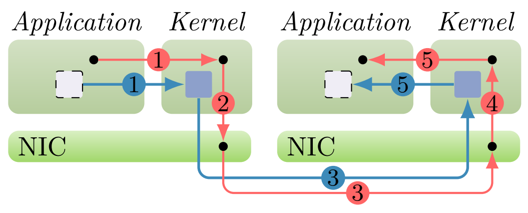
To achieve high performance, RDMA networks rely on a combination of performance-improving techniques, like zero-copy and kernel-bypass. Although these techniques are not exclusive to RDMA networks, they require special OS support and also put additional requirements on the OS, creating a lot of design challenges for OS developers. This chapter considers RDMA networks from the perspective of an OS developer.

Depending on the circumstances, some of the performance-improving techniques may not be used in a specific scenario. For example, applications often deliberately avoid RDMA read and write operations [DNC17; KKA16; KKA19]. On the other hand, traditional network technologies are often re-engineered to provide a performance similar to that of RDMA networks [Bel+14; DPD13; Pet+14; Zha+21]. Therefore, in a broader sense, we conflate and relate RDMA networks to all high-performance networks that rely on the aforementioned optimizations, regardless of whether they support RDMA operations or not.

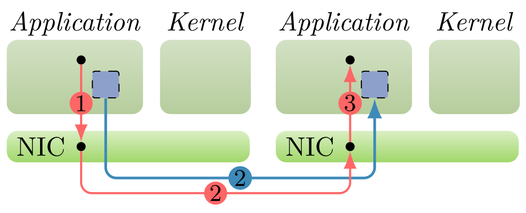
This chapter provides a general architectural overview of RDMA networks (Section 2.1), presents core RDMA techniques (Section 2.2), outlines the general technological landscape (Section 2.3), and finally introduces the software architecture and the programming model (Section 2.4).

2.1 RDMA Networks

This thesis identifies five main techniques that allow RDMA networks to achieve high performance: *zero-copy*, *kernel-bypass*, *busy-polling*, *CPU offloading*, and *lossless communication*. The first four techniques are directly reflected in the software architecture of the applications and the OS. Whereas the last one imposes additional design requirements for the software stack.



(a) When sending a message in a traditional network, both control and data planes go through the kernel.



(b) RDMA networks bypass the kernel when sending and receiving messages.

Figure 2.1: Comparison of traditional and RDMA networks. Bypassing the kernel opens up multiple opportunities for optimization.

To understand these main techniques, consider how they compare to commodity socket-based networks [POS17]. An example in Figure 2.1 depicts two applications running on two different nodes and connected over a network. The application on the left sends a message to the application on the right either over a traditional socket-based network (Figure 2.1a) or over an RDMA network (Figure 2.1b).

When sending a message over a socket-based network (see Figure 2.1a), the application prepares the message content in the application's virtual memory. Then, the application makes a system call (1) to instruct the kernel to send a message over a socket, representing an already existing connection. The OS kernel copies the message into the memory (1) eligible for device-initiated Direct Memory Accesses (DMAs) (e.g., memory must be pinned) and prepares the message for sending. For that, the kernel splits messages into packets, looks up routing tables to identify the right network interface, and schedules the packets for sending. Each packet may need to pass through a firewall and a packet scheduler [Hub01] to enforce security policies and control the network traffic. If hardware support is not available, the kernel composes corresponding packet headers, containing information like the destination address or the packet length. When the preparation is done, the kernel triggers the NIC by writing into a doorbell register to initiate the message transfer (2).

From that point on, the sender NIC transfers the message to the receiver NIC (3), which stores the message content inside the host system's memory (3). As soon as the transfer completes, the NIC notifies the OS about the received message by triggering an interrupt[1]. On the receiver side, the OS kernel performs many of the same operations as on the sender side, such as reassembling the message from the packets, looking up routing tables, and scheduling the message for delivery to the application.

For the message to finally reach the application, the application must also make a system call to the kernel requesting a message from a specific socket. If the application does not make

[1]For an interrupt to trigger, the kernel must *arm* the interrupt. A single interrupt can inform the OS about multiple messages at once.

the system call before the actual message arrival, the kernel keeps the message in the kernel memory until the application requests it. In either case, as soon as the kernel can match the destination buffer inside the application memory with the arrived message, the kernel copies the message (⑤) and returns the execution control to the application (⑤).

Moving the message between the kernel and user buffer is slow not only because of the additional memory movement but also because creating and destroying kernel buffers involves a memory allocator. Despite the developers' best efforts, memory allocation is inherently problematic for low-latency communication, for example, because the allocator may block to reclaim memory from other kernel subsystems. Moreover, the allocator is highly concurrent and can use all the CPU cores, adding additional contention.

RDMA networks improve communication performance as follows. First, the OS maps Memory-mapped I/O (MMIO) regions of the NIC directly to the application's address space. Now, the application does not need to enter the kernel to trigger message transmission (①). Second, when transferring the message over the network, the NIC takes the message content directly from the application memory (②). For this transfer to work, before actually receiving a message, the application on the receiving end makes the destination message buffer available to the receiving NIC. Third, as soon as the message fully reaches the destination (③), the destination NIC notifies the application by updating the message status through MMIO. Avoiding interrupts allows the receiving application to bypass the kernel when receiving a message.

These three techniques are correspondingly referred to as *zero-copy*, *kernel-bypass*, and *busy-polling*. Kernel-bypass allows the application to access the device directly, avoiding the overhead associated with crossing the user-kernel boundary. Zero-copy enables the NIC to access the application memory directly, avoiding the overhead of copying the message content between kernel and user memory as done with `read` and `write` system calls. With polling, the application continuously checks the NIC's message queues to avoid the overhead incurred by interrupt processing, as happens with the `epoll` system call. The use of these techniques is reflected in the API design employed by RDMA applications [Mel15; Zha+21; Cra19]. This API is very different from the traditional socket-based API, meaning that socket-based applications must be rewritten to fully benefit from high-performance networking. These techniques are referred to as *software techniques*.

Correspondingly, the other two performance-improving techniques are *hardware techniques*. The first technique is *lossless communication*, which relies on special network protocols to minimize the packet loss rate [IEE11; Zhu+15]. A low packet loss rate is important because retransmitting lost packets is extremely disruptive for high-performance networks. The second technique is *CPU offloading*. With CPU offloading, the NIC takes over work from the host system so that the host CPU can spend more time processing the application logic. In RDMA networks, one example of offloading is when, instead of the host CPU, a NIC splits messages into MTU-sized packets at the sender and reassembles the packets into a message at the receiver.

Another example of offloading is *one-sided* or RDMA communication. This technique further reduces the host CPU load by relieving a CPU at one side of the communication from participating in data transfer entirely. For that, the host enables certain memory regions to be accessible by the NIC, without the need to provide the NIC with destination buffers for each message. Then, the active side can initiate RDMA *Read* operations to receive data from the assigned memory region and RDMA *Write* to send data to the assigned memory region.

Together, these five techniques allow RDMA networks to achieve high performance. Employing these techniques comes with a cost, though, which we discuss in Section 2.2. For this reason, existing high-performance networks may not employ all of these techniques to the same extent. The choice of which techniques to use can also be made at the application or OS level.

2.2 RDMA Techniques

Despite their apparent advantages, RDMA network architectures remain a relatively niche technology even after decades of research and development [VIT98; Inf15]. The main reason for this is that each of the RDMA techniques comes with a set of requirements and limitations, which may not be met in a specific scenario. This section describes each of the RDMA techniques and their limitations.

2.2.1 Zero-copy

Zero-copy is a technique that allows the application to avoid copying the message content between the application and the kernel memory. Instead, the application prepares the message content in the application memory, and the NIC takes the message content directly from the application memory when sending the message. For this to work, the NIC must be able to access the message buffer both when sending and receiving the message.

Generally, application memory is not directly accessible to devices because application virtual memory does not have guaranteed physical memory assigned. Even if, at the moment when the application initiates the message transfer, the message buffer is backed by physical memory, the OS may page out this memory by the time the NIC tries to access it. In the unfortunate case, the NIC will send out memory that no longer belongs to the sender application. And in the case of receiving the message, the NIC will also overwrite memory that belongs to the OS or another application.

To avoid these problems, the application must request the OS to *pin* the message buffer in physical memory. By pinning, the OS promises that the message buffer will not be paged out or moved in physical memory, so that the NIC is guaranteed to access the correct memory. Pinning the memory limits the OS's ability to manage the memory, for example, for swapping, Non-Uniform Memory Access (NUMA) rebalancing [vRie14], memory compaction [Cor09], and memory defragmentation [PPG18].

Another challenge with zero-copy is that to access the message buffer, the NIC uses physical memory addresses[2], whereas the application only knows the virtual addresses of the messages it wants to send. An application can query the kernel for a specific virtual address to physical address mapping [DPD13; Bur19], but this approach is often unsuitable because exposing the physical memory layout to a user application may be a security risk from the OS perspective.

[2]The Linux kernel calls them *bus* addresses [MHJ20].

Instead, RDMA NICs maintain a Memory Translation Table (MTT) with per-process virtual-to-physical mapping that they use to translate from user-provided virtual addresses to physical addresses [Mel16].

Non-RDMA NICs do not have such a table but can rely on hardware virtualization instead: The OS can create a virtual device address space identical to the virtual application address space by configuring a virtualized NIC and the Input/Output Memory Management Unit (IOMMU) [Bel+14; Pet+14]. This way, a virtual address passed by the application to the NIC is automatically translated to the correct physical address by the IOMMU.

As a result, before the application can use certain memory as a zero-copy message buffer, it needs to *register* this memory with the OS. During the registration process, the OS pins the memory, creates a virtual memory mapping for the NIC, or passes the physical address to the application. Generally, applications try to keep the amount of registered memory as little as possible to reduce the amount of memory that needs to be pinned and the number of mappings in MTT. This could be achieved by registering the memory only when the application actually needs to send a message and deregistering the memory as soon as the message is sent. However, registering and deregistering memory is a slow operation [Mie+06; Pla+21], so high-performance RDMA applications try to find a balance between minimizing the amount of registered memory and the number of memory registration operations.

All the aforementioned considerations also apply to receive message buffers in the same way. In addition, the application must also make the receive message buffer available to the NIC before the message arrives [Mel15]. If the receive buffer is not available, the NIC will drop the message, which will be considered either as packet loss or a network error. In the latter case, the NIC will notify the OS about the error, terminating the connection and possibly causing the application to crash. The application must ensure this situation does not occur, for example, by synchronizing the receive buffer availability with the message sending. This requirement is also a result of lossless end-to-end flow control, explained in Section 2.2.5.

Although very advantageous for performance, the aforementioned requirements make zero-copy hard to use because they require the application to follow a quite complex programming model. There has been an attempt to make zero-copy available for socket-based API [Cor17b], but the approach is advantageous only for sending large messages and does not work for receiving at all. Section 4.2 discusses an extension to the RDMA API [Lis13], which trades performance [TDH21] in exchange for relaxing some of the zero-copy requirements.

2.2.2 Kernel-bypass

Kernel-bypass is a technique that allows applications to access devices directly, without the OS's involvement in the data transfer. This technique minimizes the time between an application preparing a message and the message being sent over the network. However, without the kernel's involvement, either the application or the NIC must assume the OS's responsibilities, such as preparing low-level device commands or managing concurrent access to the device.

The most common method to implement kernel-bypass is to map the NIC MMIO regions directly into the application's address space, enabling direct access to the NIC registers without entering the kernel. RDMA NICs explicitly support the creation of per-application MMIO regions [Mel16], whereas non-RDMA NICs are designed for only one active user at a time.

DPDK addresses the limitation with non-RDMA NICs by taking control of the NIC from the OS, allowing only a single DPDK application to use a specific NIC [DPD13]. SR/IOV-based hardware virtualization enables the OS to create multiple virtualized instances from the same physical NIC and assign them to different applications [PCI19; Bel+14; Pet+14]. The OS divides the physical NIC's resources among virtual NICs, with the physical NIC enforcing the resource allocations imposed by the OS. SR/IOV necessitates the OS allocating resources at device creation time. Alternatively, Intel proposed Scalable I/O Virtualization [Int20], allowing more dynamic resource reallocation, compared to SR/IOV, but this technology is not yet widely available.

Kernel bypass also removes another crucial role from the OS: that of a device driver that translates between the high-level application API and low-level device commands. Thus, high-performance applications must integrate user-level device drivers [Mel15], which execute device-specific data plane operations on behalf of the application. For security reasons, user-level drivers request kernel-level drivers to perform control plane operations: the kernel establishes a policy, and the NIC enforces it [Mel16].

Chapter 3 introduces *continuous dataplane interposition*, an architecture that allows OS control over the RDMA dataplane through either software stack modifications or the use of programmable NIC features. This thesis specifically discusses CoRD, a variant of continuous interposition architecture, which eliminates kernel bypass while maintaining zero-copy. CoRD aims to provide flexible data plane control while ensuring backward compatibility with existing RDMA applications (see Section 3.2).

It is also possible to implement kernel bypass without zero-copy. For instance, high-performance MPI libraries improve performance by copying small user-provided messages to and from pre-registered message buffers instead of registering user-provided memory [HSG06; Gab+04]. This method minimizes the overhead associated with memory registration and deregistration. Similarly, this optimization for small messages is used in InfiniBand verbs for *inline* messages [Mel15]. Other approaches, such as `io_uring` [HA20] and Shenango [Ous+19], seek to bypass system calls by transferring requests via shared memory, without fully bypassing the kernel's role.

In summary, excluding the OS from the data plane necessitates applications to incorporate their device drivers, while the NIC must enforce OS-specified rules and limits. The OS becomes more rigid, limited to direct execution only of control plane operations. Nonetheless, kernel bypass continues to be a significant technique for enhancing performance in latency-sensitive applications.

2.2.3 Busy-polling

Busy-polling is a technique used to minimize the time between when the message arrives at the NIC and when the application receives the message. Traditionally, the NIC would generate an interrupt when the message arrives, and the OS would handle the interrupt to deliver the message to the application. But for high-performance applications, the interrupt processing takes too much time [BPH22]. Instead, the application expecting a message continuously checks the NIC for new messages to arrive. As long as a message does not arrive, the application will continue polling and will not release the CPU to other applications. This technique works best in combination with kernel bypass because then, after learning about the new message, the application does not need to return from the kernel.

High-performance applications are typically allocated a set of CPU cores exclusively, and no other application can run on these cores. Moreover, any other activity happening on the pre-allocated cores is considered disruptive and is avoided [HSL09; Tsa+05; PKP03]. The expectation is that if a high-performance application receives as much CPU time as possible, its overall runtime will be shorter, so that overall, the CPUs will be used more efficiently.

Although it is a deliberate performance feature, busy polling is also wasteful and consumes a lot of CPU cycles. There are use cases when even high-performance applications must share CPU cores with other applications. For example, Goldrush [Zhe+13] coordinates Message Passing Interface (MPI) applications to share CPU cores with in-situ visualization applications while minimizing the impact on the MPI application's performance. By "harvesting" idle CPU cycles, it adds functionality, which normally is considered to be performance noise.

Another problem with busy polling is that besides wasting CPU time, it also wastes energy. Saving energy in a data center is important not only for economic and environmental reasons but also because the data center's power consumption is limited by the available power supply and cooling capacity [BHR18]. Even all cores on the same CPU cannot run at full speed simultaneously for prolonged periods of time [Hac+15]. This means that if some CPU core is busy polling, the energy used by this core may not be available for some other, more useful activity. So, it is important to minimize the time spent busy polling.

An alternative to busy polling is *blocking* by requesting the OS to notify the application about the new message upon the arrival of an interrupt. Venkatesh et al. [Ven+15] combine blocking and busy polling to simultaneously limit the disruption to the application and minimize the time spent in busy polling. As an alternative, Adagio [Rou+09] uses Dynamic Voltage and Frequency Scaling (DVFS) to reduce the frequency of specific CPU cores when the application is not doing important work, thereby reducing energy consumption during busy polling. Therefore, it is possible to minimize the negative impact of busy polling without impacting application performance.

2.2.4 CPU offloading

CPU offloading is a set of techniques that allow the NIC to take over work from the host system, so that the host CPU can spend more time processing the application logic. Offloading can come in the form of delegation, when the host system instructs the NIC to perform a specific task, for example, when the NIC forms packet headers when transmitting a message. Or, the NIC can fully take over the task, for example, when the NIC handles incoming RDMA read and write requests on behalf of the host system.

There are several reasons for CPU offloading being advantageous. First, some tasks can be done more efficiently by the specialized hardware of the NIC than by the general-purpose CPU. Second, the cycles on the CPU are more valuable than the cycles on the NIC. The cost difference can be explained partially because a NIC can have many more simpler compute units than a CPU. Finally, the NIC circuitry is closer to the data arriving on the network, so it can process the data earlier than the CPU.

In RDMA networks, offloading is used extensively [Inf15]. To send a message, the application passes a message descriptor to the NIC. The NIC splits messages into packets, forms packet headers, and schedules the packets for sending. Scheduling is an important step because multiple applications may request to send more messages than the network can handle at once, and the NIC must decide on the order of messages to maintain fairness and to avoid congestion. The NIC needs to keep track of the outstanding messages and retransmit lost packets.

On the receiver side, the NIC needs to authenticate the incoming packet to be a part of an existing connection, check if writing the packet content into the application memory is allowed, and drop the packet if not. If the packet is allowed, the NIC must write the packet content into the application memory and send out an acknowledgment to notify the sender about the successful delivery.

In the case of RDMA read requests, the responder NIC must send out the requested data from the application memory without the host CPU being involved. In the case of atomic operations, the NIC must implement the corresponding operation and send out the result to the sender. Furthermore, the NIC must monitor network congestion events and send out congestion notifications on the receiver side and adjust the sending rate on the sender side.

Offloading these operations allows relieving the CPU from most of the communication protocol implementation and to focus on the application logic. Other offload operations include rate limiting or performance counters management. In TCP-based networks, normally most, if not all, of the aforementioned operations are handled by the in-kernel network stack.

More advanced offloading features include packet forwarding [Mel18; HB11], collective communication acceleration [Gra+16; Pet+01], and network virtualization support [Mah+14; GGS20]. RDMA NICs can also be extended with programmable hardware, such as FPGA [NVI20; Fir+18] or general-purpose CPUs [Bur21b; Bro18], which allows arbitrary operations to be offloaded. Overall, there is a large spectrum of offloading features that can be used to improve performance in different ways.

Offloading is also an important aspect in TCP-based networks. The features range from simple checksum offloading and splitting the message into packets [DA23] to more advanced features, like offloading the entire TCP stack to the NIC [Lin16; Sid+15]. Traditionally, offloading in TCP-based networks has been less widespread than in RDMA networks because offloading restrains the OS in controlling communication, adds dependencies on expensive hardware, risks vendor lock-in, and complicates OS development [Lin16].

Over the past decade, the growth of network performance has outpaced the growth of CPU performance, so that offloading has become more important and widespread even in TCP-based networks. These offloadings come in the form of general-purpose programmable SmartNICs or Data Processing Unit s (DPUs) [Bur21b; Sin20; Nja+22], able to directly communicate with peripheral devices, like Graphics Processing Units (GPUs) or FPGAs, without the need to involve the host CPU [NVI24; Com22; PCI19]. A DPU typically also supports RDMA communication, specifically RDMA over Converged Ethernet (RoCE) [Inf14], so nowadays RDMA and traditional high-performance networks overlap even more.

This thesis does not cover these advanced offloading features because they are not directly related to the OS aspects of high-performance networking. However, most of the ideas presented in this thesis can be applied to DPU-based offloading as well.

Employing offloading needs to balance between multiple factors. Hardware capable of sophisticated offloading is expensive, offloading features from multiple vendors may not be compatible with each other, offloading involves work both at the host and the device, increasing the complexity of the system. Nevertheless, as RDMA networks demonstrate, carefully designed and standardized features can significantly improve performance while keeping the system manageable.

2.2.5 Lossless communication

Lossless communication relies on a combination of flow and congestion control mechanisms to avoid or minimize packet loss. To achieve this, it is crucial to prevent conditions where the network must drop packets due to a NIC or switch port lacking buffer space to store the incoming packet. Designing an RDMA network where *flow control* can guarantee congestion never leads to packet loss is possible [IEE11]. However, for many real-world RDMA networks, this is not sufficient, so they additionally employ *congestion control* mechanisms to minimize the negative effects of network congestion.

In the RDMA domain, there are link-level packet flow control and end-to-end message level flow control. The most reliable type of flow control is provided by InfiniBand networks, combining link-level with end-to-end flow control [Inf15]. At the link level, InfiniBand utilizes credit-based flow control, allowing the sender to transmit a packet only if it has sufficient credits. Credits are replenished by the receiver sending *link management packets* to the sender when there's enough buffer space for an incoming packet, ensuring the sender never sends more packets than the receiver can handle. Link-level flow control can be more finely grained by dividing link resources into multiple virtual lanes and service levels, implementing flow control for each lane and service level separately [Inf15].

RDMA end-to-end flow control ensures the receiver expects a message before sending one on the level of a connection. For this, NICs maintain a per-receive-queue credit count, which the sender decreases upon sending a message and the receiver increases when posting a receive request. The credit system doesn't control message size, so ensuring messages fit into the receive buffer is the application's responsibility.

Hardware-based end-to-end flow control simplifies lossless communication for RDMA applications but is only effective for limited connection types. If the application uses a connection type not supporting end-to-end flow control, it must implement its own flow control mechanism. Failure to synchronize sender and receiver may result in dropped and later retransmitted messages, disrupting application performance. In the worst case, the network may interpret dropped messages as a network error and terminate the connection [Gab+04; HSG06].

RoCE networks, implementing RDMA communication over Ethernet, lack credit-based link-level flow control, relying on PFC to prevent packet loss [IEE11]. However, PFC is coarse-grained and disruptive when reacting to network congestion [Zhu+15]. To address this, RoCE networks employ congestion control mechanisms like Data Center Quantized Congestion Notification (DCQCN) to prevent situations triggering PFC [Zhu+15].

Congestion can also occur in InfiniBand networks, not resulting in packet loss due to more efficient flow control mechanisms than PFC, but it can still lead to network underutilization.

To prevent this, InfiniBand NICs send Explicit Congestion Notifications (ECNs) requesting the NICs causing congestion to reduce their send rate [STJ03; Inf15]. Other available congestion control mechanisms for InfiniBand include adaptive routing and multi-path routing [DH16; Bes+21].

High-performance Ethernet-based networks have also adopted lossless communication, supporting most flow and congestion control mechanisms except for credit-based link-level flow control. Despite some differences in specific implementations (like DCTCP [Ali+10] and DCQCN [Zhu+15]), the general principles closely align with those in RDMA networks. RoCE, the most popular data center RDMA network, essentially represents a high-performance Ethernet network with RDMA support.

Lossless communication is a fundamental concept in RDMA networks. For correctness, it must be implemented at a minimum as message-based end-to-end flow control at the application level. Therefore, lossless communication introduces additional requirements to the application programming model, as the application must set up a separate connection to synchronize sender and receiver before utilizing RDMA communication. While other flow and congestion control mechanisms are optional, they are vital for ensuring high performance and low latency.

2.3 High-performance Network Architectures

Existing RDMA and non-RDMA high-performance networks have distinct design objectives and, to strike a balance between functional and non-functional goals, employ RDMA performance techniques differently [DPD13; Cra19; Mel15; Bar+17a; Mar+19]. It is clear that there is no "one size fits all" solution [TFH22]. To achieve an optimal balance in specific use cases, high-performance networks have sometimes set aside techniques such as polling [Ven+15; BPH22], zero-copy [HSG06; TFH22], lossless congestion control [Lis17; Gar+07], hardware offloading [Kau+19; Høi+18; Lin16; Pet+14], and one-sided operations [Su+17; KKA14; DNC17] to enhance flexibility and resource utilization. Therefore, even RDMA networks do not always utilize all RDMA techniques all the time.

A fine-grained connection interposition architecture must consider the specific implementation details and design objectives. For instance, if a network architecture emphasizes low latency, the connection interception should be tailored to minimize latency overhead. Conversely, if designed for operation in a shared environment, the connection interception must enable fine-grained resource sharing. Following the description of RDMA techniques and their limitations, this section outlines how these techniques are combined in existing high-performance network architectures.

Figure 2.2 categorizes high-performance network architectures into RDMA, data plane, socket, and memory-based architectures. RDMA architectures have historically developed from the use of RDMA techniques (see Section 2.2), although they now do not depend on all these techniques simultaneously. Data plane architectures redesign traditional Ethernet-based network stacks to integrate the mentioned RDMA techniques. Socket-based architectures aim to boost the performance of traditional socket-based APIs, while attempting to minimize the need for substantial application changes. Memory-based architectures extend intranode communication technologies, like PCIe, to facilitate communication among multiple compute nodes.

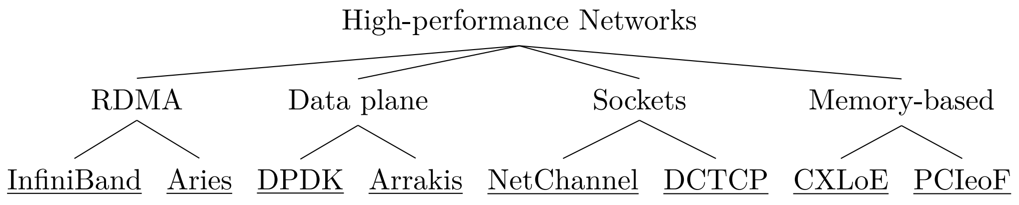


Figure 2.2: Classification of high-performance network architectures with some examples of concrete architectures.

Another dimension to these categories is whether the network is based on Ethernet or not. Non-Ethernet-based architectures have often been specifically designed for high performance, whereas Ethernet-based architectures attempt to upscale Ethernet for high-performance applications. Among Ethernet-based architectures, data plane architectures are built around a high-performance software stack, imposing few requirements on the computational and logical capabilities of the network devices. Such classification reflects historic development, although nowadays, there exists significant overlap between all the categories.

2.3.1 RDMA Networks

RDMA-based architectures are mostly represented by the InfiniBand protocol stack, which is supported by the corresponding network switches and cards and is accessible to the applications through the InfiniBand verbs API [Mel15]. InfiniBand networks offer the highest-available raw performance, with state-of-the-art NICs being capable of up to 400 Gibit/s throughput [NVI22]. Being designed for high performance from the inception, this network architecture employs all previously mentioned software and hardware RDMA techniques. On the other hand, InfiniBand is harder to manage than Ethernet, so it is common for a supercomputer site or a data center to deploy a low-performance Ethernet-based network in parallel, increasing the cost of InfiniBand deployment even further. Other examples of non-Ethernet networking include Cray Aries [Alv+12] and Intel OmniPath [Bir+15], and share many similarities with InfiniBand networks.

On a hardware level, the main difference between RDMA technologies is the raw performance, flow and congestion control mechanisms, available offloadings, and support for different network topologies. These differences are mostly transparent to the application because the application uses the same RDMA API to communicate with the network regardless of the underlying network technology.

As an exception, NVIDIA stands apart from other vendors in how many offloading capabilities its NICs have. For example, while kernel-bypass is a feature of most RDMA NICs, NVIDIA's *direct verbs* API goes one step further by exposing a user-level NVIDIA NIC driver API to the user application (normally, there is a level of generic InfiniBand verbs API for higher portability). This approach eliminates some indirection and locking, improving point-to-point latency even more. Similarly, whereas traditional zero-copy eliminates the need to copy data to and from NIC-accessible memory regions, NVIDIA makes an extra effort to improve performance for transferring non-continuous memory regions [Li+15].

Because of better performance, NVIDIA NICs also tend to be more expensive, so for economical and other reasons, major Cloud providers, like Amazon Web Services (AWS) [Ama20], Azure [MM23], and Google Cloud Platform (GCP) [LD23; Sin+20], have developed their own RDMA network solutions. These solutions are typically based on programmable NICs and are designed to be more flexible and easier to deploy and manage than traditional InfiniBand networks. As an example of using this flexibility, when AWS introduced Elastic Fabric Adapter (EFA) [Ama20], its RDMA NIC, it also introduced a new transport type, Scalable Reliable Datagram (SRD), which is designed to be more efficient than traditional RDMA transport types (see Section 2.4.2).

Nowadays, one of the most widespread RDMA-based networking solutions in data centers is RoCE, which encapsulates InfiniBand packets into UDP packets [Inf14]. Combining RDMA with Ethernet, RoCE still requires special support from the NIC to provide the RDMA techniques to the applications but does not need special support from the switches. Although, generally, InfiniBand-based networks are considered to be faster, the performance of RoCE has been catching up [NVI22]. Except data centers, RoCE maintains popularity as an HPC solution [TOP22]. Other examples of Ethernet-based RDMA networks include Intel iWARP [Gar+07] and Cray Slingshot [Sen+20].

Overall, the current development of RDMA networks goes in two directions. NVIDIA, as a high-end vendor, focuses on improving performance and adding offloading capabilities to its NICs. Cloud providers focus on cost-effective and flexible solutions, which improve performance compared to traditional Ethernet networks. In both cases, there seems to be a trend towards larger convergence with Ethernet networks.

2.3.2 Data Plane Architectures

Data plane architectures originate from the opposite direction: they aim to incorporate high-performance features into existing commodity Ethernet-based networks. Data plane architectures are designed around passing a high-performance NIC directly to the user application to eliminate the overheads associated with traditionally indirect device access. If passing the device directly to the application is not feasible, data plane architectures strive to minimize the frequency with which the application must interact with the OS to access the device.

Data Plane Development Kit (DPDK), a widely used data plane framework, facilitates the development of data plane applications for Linux. The framework addresses efficient memory allocation, low-noise scheduling, and batched packet processing [DPD13]. Utilizing the tools provided by DPDK, application developers can construct the entire network processing stack almost entirely in software, without requiring any special hardware features (though available hardware features can enhance performance). To run multiple DPDK applications on the same node, a NIC must either support hardware virtualization [PCI19], or the system must possess multiple NICs.

Arrakis [Pet+14] and IX [Bel+14] take a further step by reorganizing the OS architecture to focus on data plane applications. From the outset, they depend on hardware virtualization to grant each application direct access to the NIC. To ensure that the applications achieve the desired performance, these OSs provide an API similar to the RDMA API. Conversely, Arrakis also offers a POSIX layer, allowing traditional applications to run with minimal or no modifications, albeit at a lower performance.

Data plane architectures can also be relatively high-level. Several approaches provide library-level data plane primitives to the application, abstracting away the implementation specifics of the underlying hardware. For instance, Libfabric [Dre+13] is a library that offers data plane abstractions over multiple network technologies, including RDMA and uGNI [Cra19]. Additionally, Demikernel [Zha+21] is a library that provides data plane abstractions for various device types, including network and storage.

Data plane architectures often aim to avoid system calls as a source of overhead. To achieve this, they frequently set up a memory region shared with the kernel or another application and dedicate a CPU core to manage the data plane operations therein [Cor19; Boy+08; Ous+19; Mar+19]. Such interposition enables further abstraction from hardware details, although it may also detract from performance.

Data plane architectures are typically driven by the limitations of socket-based POSIX API [Enb+22], but do not necessarily require costly hardware features. Therefore, although data plane applications resemble RDMA applications, they can still operate over commodity Ethernet networks.

2.3.3 Socket-based Architectures

Socket-based architectures aim to improve the performance of traditional socket-based APIs without requiring significant application changes. Performance improvement may be completely transparent to the software level, for example, by employing a lossless flow control [IEE11], or a better congestion control mechanism [IEE10]. Alternatively, changes can be confined to the kernel-level network and remain transparent to the application, for instance, by scheduling network packets more efficiently [HB11], using multiple network links for the same connection [For+13; Cai+22], or by using hardware offloading [Lin16]. However, these methods employ RDMA techniques only to a very limited extent and can achieve relatively limited performance improvements.

It is possible to have more invasive modifications to the application communication while maintaining backward compatibility with the legacy socket-based POSIX API [Zhu+19]. For example, although Arrakis [Pet+14] is a data plane OS, it also offers a POSIX layer. Multiple approaches offer a socket wrapper around the InfiniBand verbs API to provide RDMA-level performance to socket-based applications [Mel19a; Li+19; Hef12; Wan+19]. All these approaches need to compromise part of the RDMA performance for compatibility but still can outperform traditional socket-based communication.

2.3.4 Memory-based Architectures

Memory-based high-performance network solutions extend intranode communication, like PCIe or CXL, to communicate among multiple compute nodes. These architectures aim to make communication over the network transparent, similarly to how access to local PCIe devices is transparent for the host CPU. GigaIO FabreX promises to provide point-to-point latency as low as 200 ns, which is several times lower than what internode-based high-performance network architectures provide [Gig21]. Other examples of these architectures include CXL-over-Ethernet [Wan+23] and PCIe-over-Fiber[Bur21a]. As of today, none of these architectures have gained popularity or demonstrated scalability beyond a single server rack. Therefore, this thesis does not discuss intranode-based architectures further.

2.4 InfiniBand Verbs

To communicate over an RDMA network, a distributed RDMA application creates a set of objects representing communication resources. These objects are created and configured using the InfiniBand verbs API, implemented by the InfiniBand verbs library. There are various implementations of the InfiniBand verbs API for different hardware, including InfiniBand [Inf15], iWarp [Gar+07], and RoCE [Inf10; Inf14]. Although alternative APIs exist, InfiniBand verbs is the de facto standard for high-performance RDMA communication today. This section describes the high-level concepts of the InfiniBand verbs API and the concepts it employs.

2.4.1 Objects

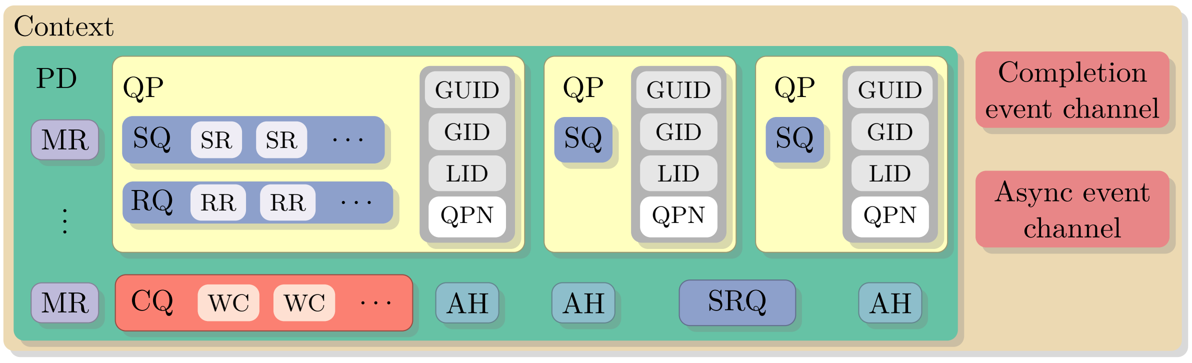


Figure 2.3: Primitives of the InfiniBand verbs library. Each QP comprises a send and a receive queue and has multiple IDs; node-global IDs (grey) are shared by all QPs on the same node.

A distributed RDMA application consists of many processes spawning over a set of compute nodes. Each process of the application starts by opening a device *context* (see Figure 2.3), which allows the process to access a NIC and to create and configure the InfiniBand verbs objects. A Protection Domain (PD) is the first object created. It groups all other objects together and represents the process's address space to the NIC. Objects that belong to the same PD can communicate with each other but not with objects from other PDs.

The main object required for communication is a Queue Pair (QP). A QP represents an endpoint of a connection between two or more communication partners. Each QP is associated with a Send Queue (SQ) and a Receive Queue (RQ), which are used to send and receive messages, respectively. The application sends or receives messages by posting Send Work Request (SR) or Receive Work Request (RR) to a QP as Work Queue Entry (WQE). These requests describe the message and refer to the memory buffers within previously created Memory Regions (MRs).

The NIC processes posted send requests to compose and send out message packets. Receive requests are used to determine where to store the incoming messages. To reduce the memory footprint, the individual RQs of multiple QPs can be replaced with a single Shared Receive Queue (SRQ).

A send or receive request can only point to memory that the NIC can access. For this, the process registers Memory Regions (MRs), so that the OS can pin the memory and configure the Memory Translation Table (MTT) of the NIC with a virtual-to-physical address mapping, enabling direct MR access by the NIC. The process can create and destroy MRs at any time during application runtime, as long as it ensures that the NIC only accesses the memory that is part of an existing MR.

When the NIC completes a send, receive, or an RDMA request, it updates the user process by posting a Work Completion (WC) to a Completion Queue (CQ). A CQ receives completions from multiple QPs that the process has associated with that specific CQ during QP creation. The application can *poll* the CQ for WCs to learn about the status of the communication requests. A WC contains information about the communication request, like the status, the number of bytes transferred, or request ID.

To establish a connection, a process needs to know the address of a remote QP. A QP address comprises a device port address and a Queue Pair Number (QPN). The QPN is a unique identifier of the QP within the device. A device port address includes multiple identifiers of the NIC port, unique within the network. A single NIC can have multiple physical ports, and in the case of hardware virtualization, also multiple virtual ports. Therefore, a single NIC can have multiple addresses.

A port address consists of a vendor-assigned Globally Unique Identifier (GUID), a routable Global Identifier (GID), and a non-routable Local Identifier (LID). Only GID and LID are used for communication. Each port can have multiple GIDs or LIDs, and the application can choose which one to use for communication. Having multiple addresses can be useful, for example, for multipath routing [Lu+18]. In InfiniBand networks, the use of GID may be optional, but in RoCE networks, the GID is mandatory.

When using atomic, RDMA read, and RDMA write operations, in addition to the QP address, the process needs to know the *memory protection key* to access remote MR. The remote NIC checks the memory key passed in atomic or RDMA operations to determine if the incoming message has permission to access the corresponding MR. Overall, processes need to exchange GID, LID, QPN, and memory keys to have full-featured RDMA communication.

There are two major ways to exchange this addressing information: either by passing the addresses over an out-of-band network, like Ethernet, or using RDMA Connection Manager (CM), a distributed service providing an interface for establishing RDMA connections [Mel15]. The latter approach offers more convenience to the application.

Once the remote process has the addressing information, it can create an Address Handle (AH) that represents a remote endpoint. When setting up a connection, a process passes the AH to the QP, instead of raw addressing information. Thus, each is an object AH representing a distinct remote destination QP.

Separately from the objects created inside a PD, there are also completion event channels and asynchronous event channels. A process can block on a completion event channel until a CQ has a new WC. This allows the process to avoid busy polling the CQ and to be notified about completions only when they occur. An asynchronous event channel is used to notify the process about asynchronous events, mostly errors.

2.4.2 Types of service

During the connection setup, each QP is configured for a specific *type of service*. Reliable Connection (RC), for example, provides reliable in-order message delivery of up to 2 GiB size. When configured, an RC QP can communicate only with a single remote QP, using send-receive, atomic, and RDMA operations. Another popular type of service, Unreliable Datagram (UD), does not provide reliable in-order delivery. On the other hand, UD QPs can communicate with multiple remote QPs using only send and receive messages that fit into a single packet. There are also other types of services, like Unreliable Connection (UC), Extended Reliable Connection (XRC), and SRD, each with its own set of features and limitations. This thesis primarily focuses on RC and UD QPs.

2.4.3 QP State Machine

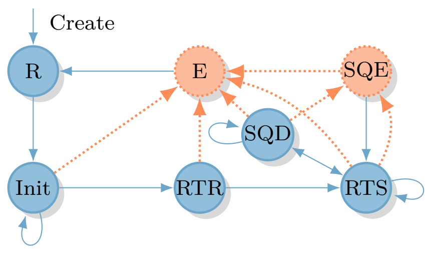


Figure 2.4: QP State Diagram. Normal states and state transitions (●, →) are controlled by the user application. A QP is put into error states (●, ⇢) either by the OS or the NIC.

During the creation of a QP, the process brings the QP through a sequence of states, as illustrated in Figure 2.4. At each step, the QP is configured with a specific set of parameters and linked to other InfiniBand verbs objects. Each newly-created QP begins in the *Reset* (R) state. To send and receive messages, a QP must reach its final Ready to Send (RTS) state, wherein it is connected to a remote partner QP. Before reaching the RTS state, the QP traverses the *Init* and Ready to Receive (RTR) states. In the event of an error, the QP transitions into one of the error states: *Error* (E) or Send Queue Error (SQE). In the Send Queue Drain (SQD) state, a QP refrains from accepting new send requests. Apart from this, SQD is analogous to the RTS state and is not further elaborated in this chapter.

When a QP in the RTR or RTS states receives an incoming packet, it must process it, update the QP state, and dispatch either a reply or an acknowledgment. For instance, a packet can constitute a segment of a send, read, write, or atomic operation message. Depending on the message type, the NIC is required to update internal counters (e.g., QP's Packet Sequence Number (PSN)), store the packet content into the receive buffer, and send an acknowledgment. If the packet is the last packet of the message, the NIC also needs to notify the application of the received message by updating the receive queue. Moreover, the NIC may need to update the Completion Queue (CQ) to notify the application of the completion of the receive request. This scenario exemplifies how the state of a QP and other related objects can be altered by an incoming packet.

Most of the operations required to establish a connection are control plane operations and end up in one or multiple system calls. Once a QP reaches the RTR or RTS state, the application can start posting send and receive requests to the QP using data plane operations. Data plane operations are optimized for performance and therefore avoid kernel interaction as much as possible.

3 Continuous Interposition

In high-performance computing, especially within RDMA networks, the conventional role of the OS in network communication has been limited. This chapter seeks to expand this role by examining the feasibility and benefits of applying continuous interposition in such environments and how to effectively balance high network performance with comprehensive OS control.

We re-examine continuous interposition, typically considered too resource-intensive for high-throughput low-latency applications, and explore its potential for providing control and security for data plane operations. With efficient continuous interposition, the OS can enforce policies, such as process-level rate limiting or enable transparent live migration, without hardware modifications. This approach involves rethinking the standard RDMA communication processes and strategically introducing modifications to reduce OS-induced overhead while maintaining high-speed data transmission.

Our investigation centers on software-level interposition, offering a flexible solution applicable across various systems. While this method consumes valuable CPU cycles, potentially impacting application performance, we examine its trade-offs and potential in high-performance environments. We contrast this with hardware-level interposition, which, although efficient, requires specialized hardware and comes with its own set of limitations.

Software-level continuous interposition is what traditional TCP/IP-based networks rely on, but they cannot reuse RDMA techniques (see Section 2.2) and, therefore, do not provide high performance. It would be too detrimental to network performance to take the same approach. Instead, we dissect the *RDMA datapath*, studying modifications that aim to optimize performance and control. This analysis is pivotal for understanding how software-level modifications can be fine-tuned for performance efficiency and enhanced OS management.

Through our research, we aim to identify the boundaries of practical applicability for continuous interposition. We implement and analyze several techniques to approximate the lowest achievable overhead for software-level continuous interposition. This analysis helps determine the range of high-performance applications that can benefit from this approach, as well as those for which it remains impractical.

Section 3.2 describes Converged RDMA Dataplane (CoRD), a new continuous interposition architecture, that adds approximately 1.5 µs of overhead point-to-point latency. But when measured with application benchmarks, the overall runtime overhead in most experiments remained within 1 to 2%, only reaching 30% in one benchmark. For scenarios where the overhead is excessive, we propose a novel OS layer in Section 3.3, capable of interposing RDMA communication at the cost of less than 0.1 µs of added overhead, thus facilitating more cost-effective OS-level control. In summary, this chapter contributes to a more nuanced understanding of continuous interposition in RDMA networks, paving the way for future advancements in high-performance network design.

3.1 Background

This section outlines the basic concepts of continuous interposition and reviews the state of the art in this area. Furthermore, we explain the conceptual problems of implementing continuous OS-level interposition for RDMA networks. As this chapter focuses on software-level continuous interposition, we do not delve into the details of hardware-level continuous interposition. Instead, we pay more attention to how a user application can efficiently pass a message to its OS, ensuring that the performance of RDMA communication remains as close as possible to the non-intercepted case.

3.1.1 Architectural Overview

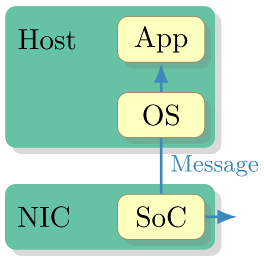


Figure 3.1: Interposition in the OS kernel

As outlined in the introduction, the most straightforward way to control each of the application's messages is to delegate device access operations to the OS kernel (see Figure 3.1). Once a user application decides to send or receive a message, it invokes a system call, transferring control to the OS kernel. This way, the OS kernel not only provides a convenient way to access the device but also allows enforcement of policies on the application's messages.

Such a generic method of continuous interposition enables the enforcement of very versatile policies, such as scheduling, security, and resource management. Moreover, the OS can modify the contents of the message, for example, to implement encryption or compression. For instance, to decide if a network packet can be sent, Linux checks if the destination IP address is allowed by the firewall [The23]. The same firewall mechanism can be used to dynamically redirect a packet to another destination for load balancing. To limit the amount of data sent over the network, Linux uses *traffic control* [Hub01] to enforce bandwidth limits. When a packet is initiated by a user application, it must traverse all the aforementioned subsystems before being handed over to the actual device driver.

However, this complex architecture is a significant obstacle for high-performance RDMA communication, as it requires each packet to traverse through many software layers. Additionally, each kernel network subsystem is designed for concurrent access from multiple user-level applications, adding further complexity. One possible remedy to this problem is to *offload* (delegate) typical network processing operations to a NIC, which can be programmed to enforce policies on the application's messages.

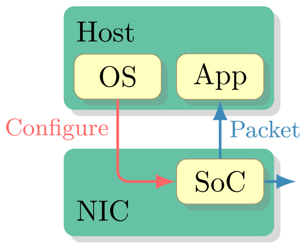


Figure 3.2: Interposition with specialized NIC offloading

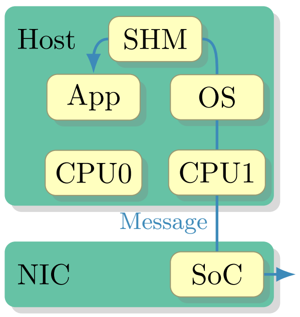


Figure 3.3: Interposition in a userspace OS service

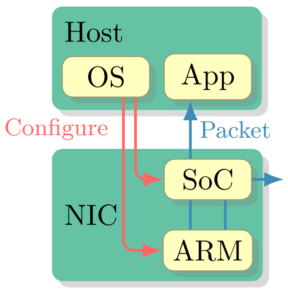


Figure 3.4: Interposition at a general-purpose DPU

Modern NICs, especially server-grade ones, are equipped with various hardware offloading operations. In the most basic case, a NIC computes and verifies checksums for TCP and User Datagram Protocol (UDP) packets. Other common functionality includes splitting and merging of network packets [DA23] or scheduling incoming packets to specific CPU cores [HB11]. More advanced NICs can even perform packet filtering and forwarding based on the contents of the packet header [NEX22; SE19]. In this setting, the OS is responsible for configuring the right policies on the NIC (see Figure 3.2) and keeping them synchronized with the host system.

Moving functionality from the host system to the NIC can be challenging [Mog03]. First, the OS must translate the interfaces it provides to the user applications into the interfaces supported by the NIC, potentially incurring performance penalties. Second, hardware-level protocol implementations generally offer less flexibility and are harder to change than software implementations. For example, an OS can maintain as many active connections as its RAM allows, but scaling the RAM size on a host system is much easier than on a dedicated NIC. Modern solutions, like *memory pooling*, may alleviate this problem, but are not yet widely deployed [Com22]. Third, identifying and fixing bugs in hardware implementations is much more challenging than in software implementations. Finally, different hardware implementations may not be compatible with each other, complicating the development of an architecture that incorporates all existing offloading implementations [Lin16]. Despite its controversy in the TCP/IP world [Cor05], hardware offloading is standard for RDMA networks, as it is the most straightforward way to achieve maximum performance [Inf15].

One way to combine the flexibility of software-level interposition with the performance of hardware-level interposition is to move the interposition logic to an OS service running on a dedicated CPU core (see Figure 3.3). Such an OS service can be implemented as a user-level application, further reducing development and maintenance costs. The important part of this architecture is that the OS service is still capable of employing the same performance techniques as traditional RDMA networks (see Section 2.2). For example, to avoid unnecessary movement of message content and context switches, the OS service can share a memory region (SHM) with the user application. To avoid interrupt latency, the OS service can poll the NIC and the shared memory region for new messages on a dedicated core. This technique is available for TCP/IP-based networks [Høi+18] as well as for RDMA networks [Kim+19; Zha+22]. Compared to offloading network processing to a System on Chip (SoC) of a NIC, this approach is more flexible but takes valuable resources from the host system and adds measurable latency to each message [Kim+19; Pla+21].

Considering that one of the key limitations of offloading continuous interposition to the NIC is the inflexibility of hardware implementations, several vendors have proposed the Data Processing Unit (DPU) architecture. The first variant of this architecture (see Figure 3.4) places an *off-path* general-purpose compute core (e. g., ARM) [Mel19b; Bro18; Sin20] into a DPU. Either the host or an off-path core can configure the SoC of the NIC to forward packets for further processing at the general-purpose NIC cores. This allows developers to write portable code and avoid vendor lock-in. Although the cost of a DPU is higher than that of a traditional NIC, the DPU can offer a more cost-efficient architecture overall. First, the general-purpose cores of a DPU can perform the same tasks as more expensive host cores equally fast and, second, the overall DPU architecture reduces data movement in the system [Sin20]. However, as a practical limitation, continuously interposing RDMA traffic may be either infeasible of very slow with some DPUs [Hoe+17].

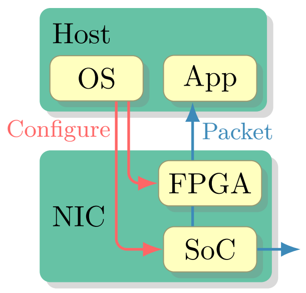


Figure 3.5: Interposition with a on-path FPGA

The off-path DPU architectures with general-purpose cores still require back-and-forth packet movement, compared to a traditional NIC (see Figure 3.2). Therefore, as an alternative, a DPU can include an *on-path* programmable chip (e. g., FPGA) to intercept all the traffic between the host system and the NIC (see Figure 3.5) [NVI20]. FPGAs are often more efficient with packet processing than general-purpose cores, although they are harder to program, because the OS needs to enforce policies on the host system, the NIC's SoC, and the NIC's FPGA. With the right abstractions, the software engineering aspect of this architecture can become manageable [KRA20] and has even been proven in practice [Fir+18]. Finally, one can combine the on-path DPU architecture with the off-path DPU architecture to achieve the best of both worlds [Bur21b].

The described architectures attempt to find a balance between flexibility, performance, and cost. A flexible and cheap option is to implement continuous interposition in the host system, but it is also the slowest. High performance at low cost can be achieved by offloading specific network processing operations to a NIC, but this approach lacks flexibility. Alternatively, a fully programmable DPU can offer both flexibility and high performance, but is typically more expensive and requires more development effort. Finally, continuous interposition can be offloaded to the network (e.g., switches), but this option is not considered in this thesis due to our focus on host-local OSs.

Further in this chapter, we concentrate on continuous interposition fully implemented in the host system. This is because hardware offloading is a way to achieve higher performance at a higher cost. Such a tradeoff is reasonable when the cost of OS interposition is too high for the application. However, this thesis studies OS-level continuous interposition, firstly, to understand the limits of software-level continuous interposition and, secondly, to discover what types of applications can benefit from it and which cannot use it at all. In this context, hardware offloading is a topic that is orthogonal to our research.

3.1.2 Software-level Interposition

As discussed in Section 3.1.1, continuous interposition can be achieved through a combination of hardware and software techniques. In this section, we focus on software-level continuous interposition because it provides valuable insights into how continuous interposition can be implemented and utilized in practice. Moreover, software-level interposition does not depend on specialized hardware and, therefore, much easier to deploy.

At a high level, software-level continuous interposition implies that once a user application prepares a message for network transmission, the message must be processed by the OS. To ensure safety and security, the OS must process the message in a compartment isolated from the application. The isolated compartment can be either the kernel or in a dedicated userspace service. Regardless of the location, the processing can occur either on the same or on a different CPU core. In the first case, the OS must context switch from the user application to the OS service (see Figure 3.1). In the second case, the OS must transfer the message to the OS service and notify it of the new message (see Figure 3.3).

In scenarios where a separate core is used, the notification to the OS service can be implemented as an Inter-Processor Interrupt (IPI) or as a message sent over a shared memory region. In the latter case, the OS service must poll the shared memory region for new messages, which results in a CPU core being permanently allocated to the busy-polling OS service. These two options allow for a trade-off between the latency of invoking the OS service and more efficient CPU core utilization.

The challenge of interposing the network data plane by the OS has also been studied in the context of high-performance TCP/IP-based networks. IX [Bel+14] and Arrakis [Pet+14] are two examples of data plane architectures that rely on OS-bypass, zero-copy, and polling to enable high-performance communication. Unlike RDMA networks, these OSs do not require the NIC to provide RDMA capabilities. Instead, they use hardware virtualization (e. g., Single Root I/O Virtualization (SR/IOV)) to provide each user application with a separate virtual NIC [PCI19]. To enforce OS-level policies, Arrakis fully relies on the available hardware offloading capabilities of the NIC. If certain functionalities are not available, Arrakis reduces the degree of continuous interposition and opts for providing the highest possible performance. This approach aligns with the philosophy of traditional RDMA networks (see Figure 3.2).

In contrast to Arrakis, IX relies on system calls from the outset, albeit with several techniques implemented to minimize system call overhead. This approach allows IX to implement a more flexible continuous interposition architecture, but at the cost of lower performance.

The Linux community has also recognized the shortcomings of the traditional POSIX API in the context of high-performance networking. A Linux application can use `io_uring` to benefit from a zero-copy API [HA20]. In interrupt-driven mode, `io_uring` reduces the number of context switches between the application and the OS by batching multiple Input/Output (I/O) operations (see Figure 3.1). In kernel polling mode, `io_uring` avoids context switches altogether but requires a kernel thread to busy-poll on the request queue shared between the application and the OS (see Figure 3.3).

Furthermore, Linux provides eXpress Data Path (XDP) [Høi+18], a framework for implementing high-performance packet processing in the kernel. With XDP, a user-level application can load an extended Berkeley Packet Filter (eBPF) program [Tig23] into the kernel, which is then executed for each packet received by the NIC. Compared to `io_uring`, XDP offers a lower-level but higher-performing API. XDP programs can be used to implement a firewall, load balancer, or packet filter.

RDMA networks have been evolving towards a similar architecture from the opposite direction. For example, FreeFlow [Kim+19] exemplifies an RDMA network, enabling continuous dataplane interposition by creating a proxy OS service to inspect and modify messages before passing them to the NIC. To reduce the overhead of context switches, the FreeFlow service runs on a separate CPU core and communicates with the user application through a shared memory region for incoming and outgoing messages. Thus, FreeFlow can inspect communication on a per-message basis and implement policies such as firewalls or load balancing. Google has implemented a very similar architecture for TCP/IP-based networks [Mar+19].

To reduce overhead from an intermediate OS service for small messages, Justitia [Zha+22] proposes sending small messages over an uninterrupted RDMA plane. Whereas for large messages, Justitia uses a separate OS service, which allows improvement in resource utilization and congestion control at the software level. Although this separation improves performance, the application is expected to be cooperative and not circumvent the OS policies.

To minimize the overhead of continuous interposition in a separate OS service, the OS must run this service on a dedicated CPU core. Moreover, this service must busy-poll the message queues shared with the user application. This architecture diverts precious resources from the user application, which could otherwise be used for computation. Alternatively, the OS can implement continuous interposition inside the kernel. Then, to send a message, the application invokes a system call.

Several architectures enable continuous interposition by moving the RDMA dataplane into the kernel. Among these, LITE [TZ17] and KRCore [Wei+21] focus on resource sharing, which even allows for improved communication performance in some common use cases. The concept of continuous interposition in this thesis aligns closely with these approaches. Section 3.2 presents CoRD, a continuous interposition architecture, which, unlike LITE and KRCore, focuses on backward compatibility with existing RDMA applications. Although not explicitly designed for improving performance, CoRD can aid in congestion control and improve application performance in some cases.

Overall, continuous data plane interposition is already seen to be useful for high-performance data center networks. However, there is no good understanding of how to integrate such interposition into the existing software stack with minimal performance impact, high deployability, and backward compatibility. Therefore, this thesis specifically focuses on these aspects.

3.1.3 System Calls

For most continuous interposition architectures, control transfer from the user application to the OS occurs through a system call. Although it is possible to bypass the system call layer, as in FreeFlow [Kim+19], which uses a busy-polling OS service and does not require system calls, in most other cases, the overhead introduced by system calls directly influences the communication data plane. Therefore, the exact mechanism of system call invocation is a key component of the continuous interposition architecture.

Conceptually, a system call is a straightforward operation, akin to a function call, but in reality, the process of transitioning from user space to kernel space is complex. A system call involves changing privilege levels, optionally switching address spaces, setting up the kernel stack, sanitizing system call arguments, dispatching the correct system call handler, and potentially many other steps. After processing the system call, most of these steps must be reversed. However, in the case of a complex system call, all these steps constitute only a small fraction of the overall work done by the kernel. For example, a message sent over a socket will traverse several kernel subsystems, which communicate with each other through asynchronous channels. Although the system call layer provides a very convenient instrument for continuous interposition for both the OS and applications, RDMA networks typically avoid them for data plane operations.

System Call Performance

In TCP/IP-based networks, the user application can invoke a `read` or `write` system call to send or receive a message. Architectures like LITE [TZ17], CoRD [Pla+23], and KRCore [Wei+21] employ this straightforward approach. However, the downside of being straightforward is that even stripped-down Linux system calls may not provide the lowest possible overhead. This section provides an overview of the existing system call mechanisms .

There are several reasons why system calls in Linux are costly. Firstly, depending on the CPU model, the cost of transitioning to the kernel can vary. Secondly, for every system call invocation, the kernel must set up the environment for the system call handler. This process includes, but is not limited to, saving the user application's registers, setting up the kernel stack, and dispatching the call to the appropriate device driver [Kul23]. Thirdly, almost every modern CPU is vulnerable to at least some side-channel attacks [Lip+20; Koc+19a], necessitating mitigations by the kernel. Fourthly, passing arguments between the kernel and user space may involve serialization and deserialization of data structures, which can be resource-intensive. Finally, dispatching system call arguments that reference specific kernel objects may also entail additional work.

The challenge of reducing the cost of system calls has been specifically explored in the context of microkernel-based OSs. In these systems, Inter-Process Communication (IPC) performance is crucial. Microkernels provide only a few system calls, with most system services being provided by other user-level processes through IPC invocations. The scope of IPC functionality is limited to the bare minimum. For instance, the Fiasco microkernel offers only synchronous IPC with a limited maximum message size [Här+97; EH13]. The architecture of microkernels demonstrates the feasibility of reducing system call costs through structural modifications.

Besides changing the software architecture, existing approaches also try to use Instruction Set Architecture (ISA) of a CPU more efficiently. One such approach is Simurgh [Mot+21], which proposes the concept of *protected user space functions*. These functions allow the user application to securely execute privileged operations without entering the kernel or changing the address space [MSB22]. Protected user space functions, provided by the kernel, can be safely invoked by the application using two new CPU instructions proposed by Simurgh. This mechanism is similar to the protected control transfer instructions in Exokernel [EKO95]. Simurgh has demonstrated the viability of this method by implementing an in-memory file system that allows avoiding much of the kernel overhead typical for file system interactions.

To evaluate the performance characteristics of privileged user functions, Simurgh implemented the new instructions in the gem5 simulator [Bin+11]. Conceptually similar, albeit more complex, *call gates* in Intel's `x86` [Int22] and Itanium [Gra+05] architectures have shown mixed performance results and were later discontinued [Gra+05; SB05].

The basic assumption of Simurgh is that the existing system call mechanism is inherently slow and proposes the use of new, faster instructions. We challenged this basic assumption by proposing a new system call mechanism that leverages the existing CPU instructions but changes the low-level system call implementation. Specifically, we proposed the *fastcall* architecture, which separates system calls into fast and slow paths [Mie+22]. The slow path is the traditional system call mechanism, whereas the fast path branches out into specialized *fastcall handlers*, which execute privileged operations on behalf of the user application. Fastcall handlers are conceptually similar to application-specific safe handlers in Exokernel [EKO95]. We discuss the fastcall architecture in more detail in Section 3.3.

There are other techniques that can make the exact mechanism of transitioning into the kernel or invoking an IPC faster. For example, SkyBridge [Mi+19] uses the `VMFUNC` instruction to reduce IPC latency by 85% for several L4Re Microkernel [Här+97] instances and by 60% for seL4 [Kle+09]. In absolute numbers, SkyBridge achieved an IPC latency of 396 cycles independent on the kernel, which would take less than 100 ns on the CPU used for evaluation. Such a dramatic improvement has been possible because `VMFUNC` allows bypassing the kernel and switching directly to the target of the IPC call.

Unfortunately, SkyBridge comes with a set of deployability, usability, and security limitations, preventing it from fully replacing the traditional system call mechanism. For example, unlike the `SYSCALL` instruction, `VMFUNC` allows the calling process to jump into an arbitrary location in the target address space, enabling a malicious caller to bypass any security checks a callee might have. SkyBridge proposes the use of software verification to prevent malicious IPC invocations.

Even more extreme ideas for reducing the degree of isolation have been tried with unikernels. A unikernel is an OS architecture designed for virtual environments and implements one of the most extreme interpretations of microservice architecture by compiling the entire application stack into a single address space [Kue+21]. The resulting process has everything it needs to run, ranging from device drivers up to user application logic[1]. When everything the application needs runs inside the same address space, no system calls or IPC are needed. In this case, a function call would be equivalent to a system call and would take only a couple of cycles.

Unfortunately, by compiling the entire software stack into a single address space, unikernels increase the attack surface of the applications exposed to potentially malicious inputs. Even ignoring the security aspect, removing boundaries between software components makes it easier for failures to propagate. To alleviate this problem, FlexOS [Lef+21] attempted to reintroduce boundaries between software components of a unikernel application using Intel Memory Protection Keys (MPK) [Par+19] technology. MPK allows parts of the application address space to be isolated into separate compartments, which can only be accessed by code running in the same compartment. The transition between the compartments is faster than an address space switch, but the security properties are similar to those of SkyBridge [Mi+19].

[1]The host OS can provide device access to a VM through hardware virtualization or paravirtualization, resulting in a different balance between isolation and performance properties.

SkyBridge [Mi+19] and FlexOS [Lef+21] create “soft” isolation boundaries and rely on verification to enforce them. Another direction of research has considered creating isolation boundaries without strict guarantees. KLOS [VYC05] leveraged `i386` segmentation registers to separate address spaces from each other within the same physical address space. Similarly to SkyBridge, KLOS cannot enforce fixed entry points into other applications' address spaces. Instead, KLOS relies on the need for a malicious application to guess the correct index in the segment descriptor table. Unfortunately, the size of this table is relatively small, and the likelihood of successful guessing is high.

As part of the fastcall architecture, we built upon the ideas of non-strict isolation guarantees by employing new CPU features. This approach allowed us to significantly decrease the probability of a malicious application guessing the secret. We explain this approach in more detail in Section 3.3.3.

In summary, system call performance can be improved either with a different software architecture or by leveraging new CPU features. In both cases, the performance improvement comes at the cost of reduced isolation or fewer available features. In this chapter, we explore both new software architectures and new system call mechanisms.

System Calls in RDMA Dataplane

Consider a case of sending a message from node A to node B (see Figure 3.6). The process of sending a message from node A to node B consists of three data plane operations: posting a receive request (recv), sending a message (send), and checking the completion queue for incoming messages (poll). In the most optimistic case, the critical path of the message exchange consists of send and poll operations. If each of these operations needs to make system calls, then the message exchange will incur two system calls.

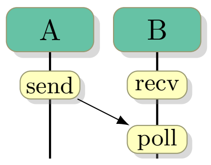


Figure 3.6: Example of a roundtrip message exchange

Just the transition from the user application to the kernel and back may take up to 300 ns [Mie+22]. Considering that the system call overhead needs to be accounted for multiple times per message and the fact that RDMA latency can be as low as 1 µs [Pla+23], the system call overhead can be significant.

The aforementioned numbers may create the impression that the system call overhead exhibits prohibitively high overhead, but a closer look reveals that this is not the case for modern systems. For example, a 300 ns overhead has been observed for a CPU which is vulnerable to the Meltdown attack [Lip+20], and therefore had an expensive Kernel Page Table Isolation (KPTI) [Cor17a] mitigation enabled. Modern CPUs are not vulnerable to Meltdown and do not require KPTI mitigation, which means that the system call overhead can be less than 100 ns [Mie+22].

On the other hand, a 1 µs latency has been measured for 2-byte messages between RDMA NICs connected back-to-back [Pla+23]. This setup, of course, does not reflect the real usage of RDMA networks, because in most cases there are multiple hops between two NICs and the messages are larger. For example, some HPC networks at AWS and GCP have demonstrated point-to-point latency in the order of tens of microseconds [Sen+22].

Such discrepancy between worst-case and real-case numbers makes it hard to draw definite conclusions from microbenchmarks about the feasibility of continuous interposition. Therefore, in this thesis, we go in two directions simultaneously. First, we try to minimize the system call overhead as much as possible, which we discuss in more detail in Section 3.3. Second, we try to understand what types of applications would not experience significant performance degradation from continuous interposition, which we discuss in Section 3.2.

3.2 Converged RDMA Data Plane

The existing RDMA dataplane bypasses the OS kernel and therefore cannot be interposed by the OS. There also exists a kernel-level RDMA dataplane, which is functionally almost identical to the user-level RDMA dataplane. The kernel-level RDMA dataplane has mostly been used for high-performance access to storage devices [KN14], but it could also be used for other forms of communication. Therefore, the simplest way to put the OS on the user-level data plane is to redirect the user-level requests to the existing kernel-level RDMA stack. In this section, we describe the design of Converged RDMA Dataplane (CoRD), which unifies user-level and kernel-level RDMA data planes to enable OS-level continuous interposition.

CoRD augments the existing RDMA architecture, while maintaining compatibility with existing RDMA applications. This allows providing OS functionality to the RDMA dataplane without the need for application modifications. Although our initial motivation for CoRD was to have a software-level architecture that would allow the live migration of RDMA applications, we show that CoRD can be useful for other purposes as well.

3.2.1 Design

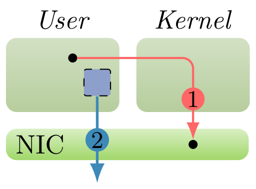


Figure 3.7: CoRD dataplane.

Figure 3.7 shows the CoRD dataplane, which resembles the classical RDMA architecture (see Figure 2.1b). The control plane of CoRD works in the same way as with traditional RDMA applications, including connection lifecycle and memory management. In contrast to traditional RDMA, CoRD requires an application to make a system call to execute RDMA dataplane operations.

For example, to send a message, the application invokes the kernel-level NIC driver (①) by making a system call. Once invoked, the driver triggers the NIC to perform the corresponding operation. To access the message content, the NIC accesses the application memory (②), because the message content still resides in the pinned memory (■), as was the case in traditional RDMA. Other operations function in a similar way.

To maintain high performance, CoRD requires *OS functions* implementing kernel-level message processing to be lightweight and non-blocking. We believe this requirement is easily achievable, as the OS can leverage more powerful RDMA offloading capabilities, compared to those in TCP/IP-based networks. Nevertheless, being non-blocking is not stringent, as we demonstrate in an example of a rate limiter (see Section 3.2.4).

OS functions are sufficiently capable to implement QoS, security, and isolation, similarly to other dataplane interception techniques [Kim+19; TZ17; Wei+21]. There is no prescribed location for implementing a function, but we believe the most natural place is within one of the core RDMA drivers, such as `ib_core`. Functions can be directly implemented by modifying driver logic in the corresponding loadable kernel module. Alternatively, the kernel may establish hooks to dynamically install functions using eBPF [Tig23].

The overhead from enforcing OS functions varies based on the specifics of the implemented interposition functionality. For a simple profiling operation, the overhead might be in the order of tens of nanoseconds. However, due to user-kernel switching, CoRD inevitably introduces additional per-message latency which is fixed and difficult to avoid. Therefore, when implementing OS functions, it is important to consider the total overhead.

3.2.2 Implementation

The RDMA API [Mel15] defines three main dataplane operations: sending a message, posting a receive buffer, and polling for incoming messages. To implement CoRD, we modified the user-level NIC driver to forward these operations to the kernel-level RDMA stack. For this purpose, we reused the existing private API of the RDMA stack, which is normally used for control plane operations. We also modified the kernel-level driver to add support for kernel-level objects used from user space. For our implementation, we specifically modified the `mlx5` device driver for NVIDIA's ConnectX-series NICs.

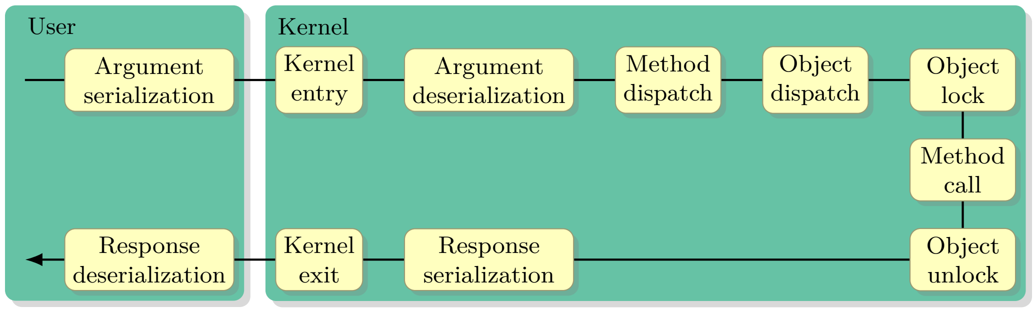


Figure 3.8: The flow of a dataplane operation in CoRD.

In CoRD, there are several additional steps for each of the dataplane operations compared to traditional RDMA [Mei23]. Figure 3.8 shows these additional steps, starting from the moment when the application invokes the user-level NIC driver.

First, to make a system call, the user-level driver needs to serialize the system call arguments into a dedicated buffer. This step is required because generally a kernel driver needs to copy the arguments into kernel memory [Cor+05], and the easiest way to do this is when all the arguments are already in a contiguous memory region. An alternative would be to pass the parameters for the RDMA operations through the CPU registers. However, this is not possible because the Linux system call interface allows only six arguments to be passed in the registers, which is insufficient for the RDMA operations.

After preparing the arguments, the user-level driver invokes the kernel using a `syscall` instruction. This instruction transfers execution to the kernel entry point. The entry point switches the execution context from the user application to the kernel and forwards the execution to the corresponding system call handler. In our case, the system call handler is the `ioctl` callback of the `ib_uverbs` driver.

The `ib_uverbs` driver is a part of the kernel-level RDMA stack. It is responsible for deserializing the arguments and dispatching user-initiated operations to the corresponding RDMA kernel modules. In particular, `ib_uverbs` copies the arguments from user memory into kernel memory. Then, it extracts the method and object identifiers from the arguments and dispatches the operation to the corresponding kernel module. Interestingly, method and object dispatch can be relatively expensive because the kernel needs to perform a lookup in a tree-like data structure. Although this method has been optimized for fast lookup, especially in the case of a large number of objects, on the scale of RDMA networks, the dispatch overhead is still non-negligible. Having all the required arguments, `ib_uverbs` calls into the device-specific driver, which, in our case, is `mlx5`.

The `mlx5` driver is responsible for executing the actual RDMA operation. It begins by acquiring the lock on the corresponding object (a QP or a CQ). This action, however, is optional and decided at compile time in the user-space driver because the driver can assume that the application will not modify the object concurrently. Then, the `mlx5` driver performs the actual operation, such as sending a message or posting a receive buffer. The implementation of the operation is essentially identical to that in the user-space driver. Finally, the `mlx5` driver releases the lock and returns control to the `ib_uverbs` driver.

The `ib_uverbs` driver is responsible for copying the results of the operation back to user space. In our case, this is relevant for the CQ polling operation, which returns work completion objects. For send and receive operations, the results are passed back to user space as a return value of the system call. The kernel exit routine switches the execution context back to user space. The user-level driver deserializes the results of the operation and returns control to the application.

3.2.3 Evaluation

Our goal is to evaluate how the CoRD architecture impacts both raw communication performance and end-to-end application performance. Additionally, we aim to identify potential avenues for reducing the overhead associated with CoRD communication.

We conduct experiments on two cloud systems. The first system comprises 8 bare-metal BM.Optimized3.36 nodes in the Oracle cloud. Each node is equipped with two hyperthreading-enabled 18-core Intel 6354 CPUs, operating at 3 GHz, and 100 Gbit/s NVIDIA ConnectX-5 Ex RoCE NICs. The system runs vanilla 6.2-rc7 Linux, either with or without our patch to support CoRD communication in the `mlx5` driver. Side channel attack mitigations are set to their default values[2], which, in particular, means that KPTI [Cor17a] is disabled. We also disable Turbo Boost, pin all benchmark processes to dedicated cores, and set the CPU power governor to the highest performance mode.

The second system comprises two virtualized HB120 nodes deployed in the Azure cloud. Each node has two 64-core AMD EPYC 7V73X CPUs, with only 120 cores passed to the VM, and virtualized 200 Gbit/s NVIDIA ConnectX-6 InfiniBand NICs. Similar to the Oracle system,

[2]The Linux kernel enables only those mitigations, which protect against vulnerabilities a specific CPU actually has.

we keep the default side channel attack mitigations (KPTI is disabled) and use core pinning. However, due to the cloud provider's policy, hyperthreading is disabled, and dynamic frequency scaling is enabled. We employed the second system because, in contrast to Oracle, it features an InfiniBand network.

Microbenchmarks

We begin by evaluating the performance of CoRD communication in isolation. Our objective is to understand the overhead that CoRD communication imposes on individual RDMA operations. We employ the perftest 4.5 benchmark [per20] to measure the latency and throughput of one-sided (RDMA read and write) and two-sided (send/receive) operations.

Table 3.1: Latency of point-to-point operations in µs on the Oracle system when lossless flow is enabled.

Message size	Baseline kernel				CoRD kernel			
	RC			UD	RC			UD
	Read	Write	Send	Send	Read	Write	Send	Send
1 B	3.1	1.7	1.7	1.7	4.5	2.8	3.4	3.2
8 B	3.1	1.7	1.7	1.7	4.6	2.8	3.4	3.3
64 B	3.1	1.7	1.7	1.7	4.6	2.8	3.6	3.6
512 B	3.3	2.3	2.3	2.3	4.6	3.4	4.0	3.8
4 KiB	3.9	3.0	3.0	3.0	5.4	4.0	4.7	4.5
32 KiB	7.0	6.1	6.1	—	8.3	7.2	7.8	—
256 KiB	25.7	24.9	24.8	—	27.1	25.9	26.5	—
1 MiB	89.9	89.2	88.9	—	91.6	90.3	90.6	—

Table 3.1 displays the latency of point-to-point operations on the Oracle cloud with both the baseline and CoRD kernels. For UD transport, the table does not include results for large messages and one-sided operations, because UD supports only send operations and messages only up to 4 KiB. Overall, it is evident that, regardless of message size, the CoRD kernel adds approximately 1 µs to 1.9 µs of latency to the baseline kernel.

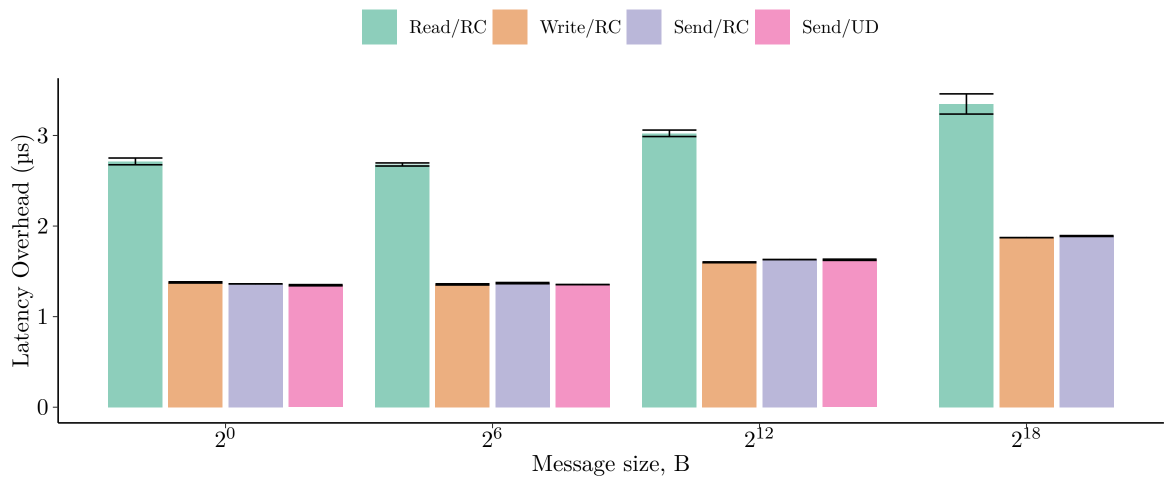


Figure 3.9: Latency overhead on the Oracle system when lossless flow control is disabled.

To understand how CoRD interacts with other RDMA performance techniques, we conduct some experiments in this chapter with deactivated lossless flow control by disabling Priority Flow Control (PFC) [IEE11] for RDMA traffic on the NICs. PFC helps to avoid packet loss in case of network congestion in Ethernet-based networks, which is crucial for the lossless operation of RoCE. The exact configuration of PFC-controlled flows has been set up by Oracle [BV22] and is based on DCQCN [Zhu+15]. Figure 3.9 shows that disabling PFC also adds 1 µs to 2 µs to point-to-point latency, except for the RDMA Read operation, which adds around 3 µs. We observe this behavior even when the network is not congested.

Another contributing factor to higher point-to-point latency is the location of communicating applications in the network topology. In our experiments, a 1.7 µs latency is only possible because the communication partners reside on the same rack and share a Top-of-the-Rack switch. If communication needs to go over multiple switches, the latency increases. Brar and Vincent report that for larger setups, the latency can be as high as 10 µs [BV22]. Therefore, we also disable PFC to simulate a situation where the communication partners are located in different parts of the data center and are subject to congestion caused by unrelated network flows. Such a situation can often occur in a data center setting.

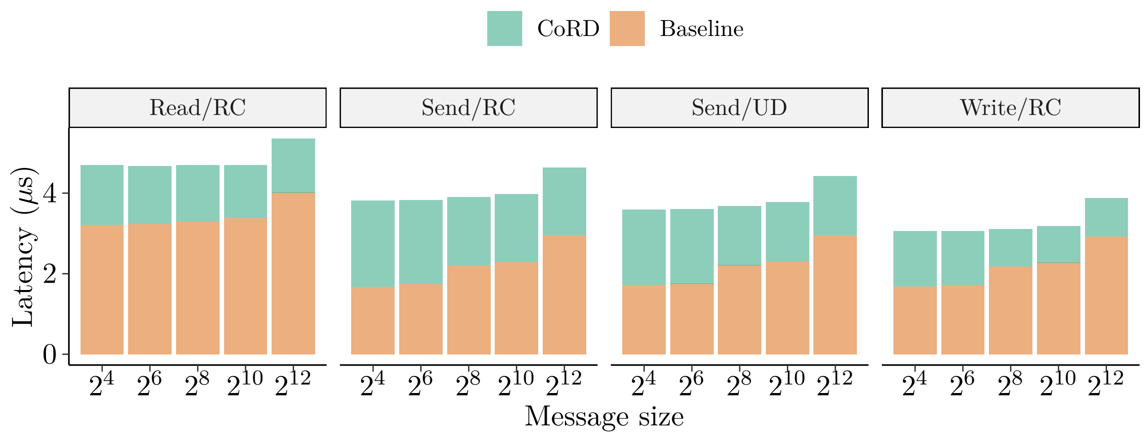


Figure 3.10: Small-message point-to-point communication latency on the Azure system when communicating over different transports (RC/UD) using one-sided (Read/Write) or two-sided (Send) communication.

To understand if our result reproduce with a different CPU and network, we repeated the same experiment on a different cloud provider. Figure 3.10 shows the latency overhead on the Azure cloud for small messages. This system has different CPUs (AMD instead of Intel) and an RDMA protocol (InfiniBand instead of RoCEv2). We do not have influence over the flow control settings on this system. On this system, the baseline latency is lower than that on the Oracle cloud. For very small messages (< 64 B), the per-message overhead from CoRD is slightly higher, because the version of CoRD we used for this experiment did not support *inline* messages, an optimization that saves the NIC from accessing a separate message buffer. Otherwise, the per-message overhead is similar to that on the Oracle cloud and is around 1.5 µs.

Overall, depending on the system, point-to-point latency can range from 1.5 µs to 10 µs for small messages. Whereas the overhead from CoRD is approximately 1.5 µs. In relative terms, the overhead ranges from 15% to 100%, which, for some applications, can be unacceptable.

For larger messages, the relative overhead significantly decreases. For example, we measured 30 % to 50 % overhead for 4 KiB messages and < 2% overhead for 1 MiB messages. So, to understand if CoRD is applicable, we need to know the latency requirements of the application and the baseline latency a particular system offers.

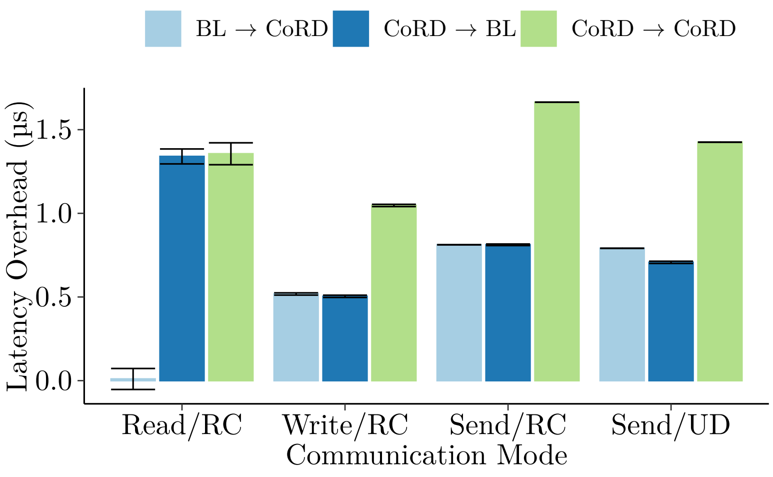


Figure 3.11: Latency overhead on the Oracle system when communicating over different transports (RC/UD) using one-sided (Read/Write) or two-sided (Send) communication. Client and server can independently run baseline (BL) or CoRD kernels, "→" indicates the direction of communication (from client to server).

To alleviate part of the overhead, the CoRD architecture can be enabled independently on either the sender or the receiver. In Figure 3.11, we examine how each side contributes to the absolute latency overhead compared to baseline-to-baseline communication, measured with a message size of 4 KiB. When the client does not run CoRD, there is no overhead for RDMA read operations, as the CoRD data plane is never triggered on the server side. With RDMA write operations, each side contributes almost equally to the overhead because perftest utilizes two writes to exchange data: One from the client to the server for synchronization, and another from the server to send the data to the client. During the send operation, both sides contribute equally as well because both the sender and the receiver execute a corresponding data plane operation.

Constant per-message overhead results in lower maximum throughput when messages are small and the message rate is high. Figure 3.12 corroborates this statement because, with larger messages, bandwidth degradation becomes insignificant (UD supports only up to 4 KiB messages). For example, baseline kernels exchange approximately 3 million 4 KiB messages per second using send/receive, and we observed a 55% bandwidth degradation. Whereas, for 32 KiB messages exchanged using send/receive, perftest measured approximately 370k messages per second with only a 1% bandwidth degradation. This behavior is similar for all types of communication (RC/UD, Send/Read/Write) as the per-message overhead is similar. These results indicate that CoRD performs worst when an application sends a large number of small messages.

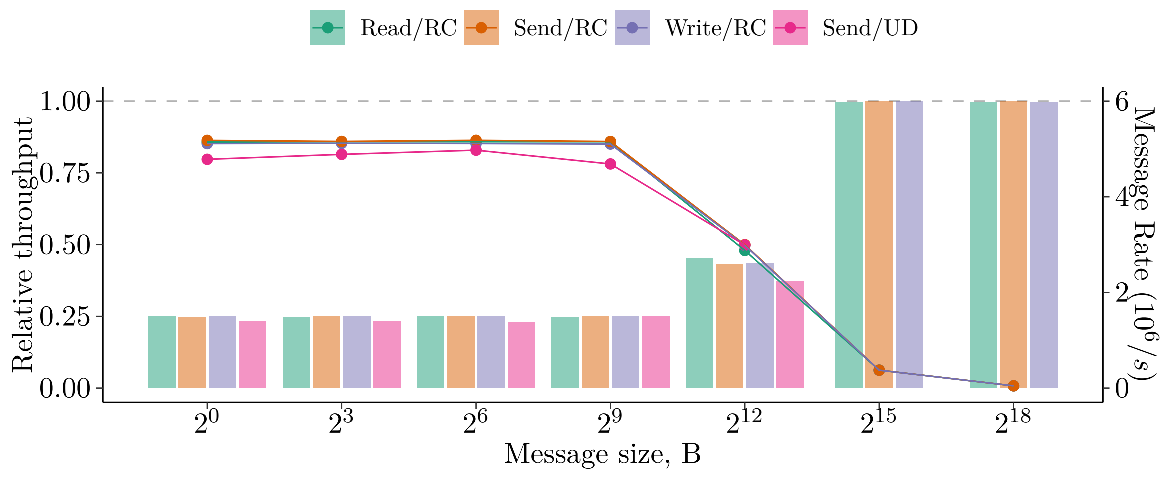


Figure 3.12: Throughput on the Oracle system with CoRD communication over RC or UD using one-sided (Read/Write) or two-sided (Send) communication relative to baseline communication. The overlayed lines show the message rate (right axis) in the baseline configuration.

MPI Microbenchmarks

To better understand the impact of CoRD on real applications, we scaled up the experiment to include the evaluation of *collective communication operations* [3]. Collective operations enable efficient data exchange between multiple processes. For example, the MPI standard [MPI15] defines several dozen collective operations and their variations, including *Broadcast*, *Scatter*, *Gather*, *Reduce*, *All-to-all*, and others. These operations frequently appear in real distributed applications and are often targets of intensive optimization efforts.

We tested collective operations using the OSU Micro-Benchmarks [Pan23] `7.1-1` to measure the latency of individual MPI collective operations. Each run of the benchmark executed an MPI operation with a specific message size over 1000 iterations with 100 warm-up iterations. We report the average latency across 3000 iterations measured over 3 distinct runs. To report variation, we also show a region of standard deviation measured across 3000 iterations for each data point.

We linked the benchmark against the OpenMPI [Gab+04] version `4.1.5rc4`, a popular open-source MPI implementation. Normally, OpenMPI communicates with the processes running on the same node using shared memory. To highlight the performance impact of CoRD, we disabled shared memory communication. We ran each benchmark on eight nodes of the Oracle system.

Collective operations are typically implemented using point-to-point communication. In most cases, the processes organize themselves into one or multiple logical *trees* and exchange data along the edges of these trees during multiple steps of a single collective operation [TRG05]. For the purpose of this experiment, we selected five collective operations with varying communication intensity:

Reduce operation reduces data from all processes using a predefined function (e.g., sum) to a single process called *root*. In the end, the root process holds the result of the reduction

[3] Also known as group communication operations.

operation (e.g., the sum of all values). To make the data exchange more efficient, non-leaf processes perform a partial reduction and send the result to their parent.

All-reduce operation is similar to reduce, but the end result is known to all processes. The need to send the result to all processes makes this operation more communication-intensive.

Gather operation collects data from all processes to the root process. Unlike reduce, in each subsequent step of the operation, a process passes on all the data it has received in the previous step from multiple processes. Therefore, each subsequent step of the collective operation is more data-intensive.

All-gather operation is similar to gather, but the end result is known to all processes.

All-to-all operation exchanges data between all processes. Unlike Gather, each process sends different data to every other process, making this operation the most data-intensive.

Figure 3.13 compares the latency of individual collective operations with the baseline and CoRD Linux kernels. We tested the five aforementioned collectives executed with different operand sizes. The figure shows the results of a run with 288 processes, where hyperthreads were not used. The results for the run with hyperthreads are similar, except that the benchmark omitted the largest operand size due to high memory consumption.

To explain the results, we categorize operand sizes into three groups: small ($\leq 128\,\text{B}$), medium ($\leq 8\,\text{KiB}$), and large ($> 8\,\text{KiB}$). Recall that for large point-to-point message throughput, CoRD was able to achieve near-baseline performance. The same is true for Reduce and All-reduce operations. For example, a reduction operation with a 16-byte operand takes 3.7 µs with the baseline kernel and 7.8 µs with CoRD, constituting a $\approx 2\times$ overhead. However, for a 16 KiB operand, the baseline kernel achieves 71 µs and CoRD 79 µs, which is only a 11% overhead. The overhead for small operands is low relative to single point-to-point message latency because the MPI library can mask the overhead by sending many small messages in parallel.

Counterintuitively, CoRD sometimes outperforms the baseline kernel for certain operand sizes. This happens because kernel bypass makes it easier for multiple processes to create a large message burst and congest the network. On the other hand, CoRD adds overhead to communication, making it harder to congest the network. We expect this effect to be stronger for RoCE-based networks, compared to InfiniBand networks, because they do not have link-level congestion control [Inf15; Zhu+15]. Disabling PFC-based flow control, makes this effect even stronger, as we elaborate later in this section.

The All-reduce operation has a more complex communication pattern compared to Reduce and Gather operations. As a result, latency masking becomes less effective: for 16 B operands, the baseline kernel achieves 22 µs, whereas CoRD achieves 55 µs, which is a $2.5\times$ overhead. However, for large operands, CoRD can sometimes even outperform the baseline kernel, because of the aforementioned network congestion effect. For instance, for 16 KiB operands, the baseline kernel achieves 307 µs, whereas CoRD achieves 297 µs.

When comparing collective operations with medium-sized operands, we observe that CoRD catches up with the baseline kernel much earlier for Gather and All-gather operations than for Reduce. This is due to the actual size of the messages exchanged during Gather and All-gather operations increasing with each step of the operation. For example, for 256 B operands, the baseline kernel achieves 142 µs for the All-gather operation, whereas CoRD achieves 153 µs.

If a communication pattern is complex, as in the All-to-all operation, the difference between baseline and CoRD almost disappears. For example, with 16 B operands, the baseline kernel

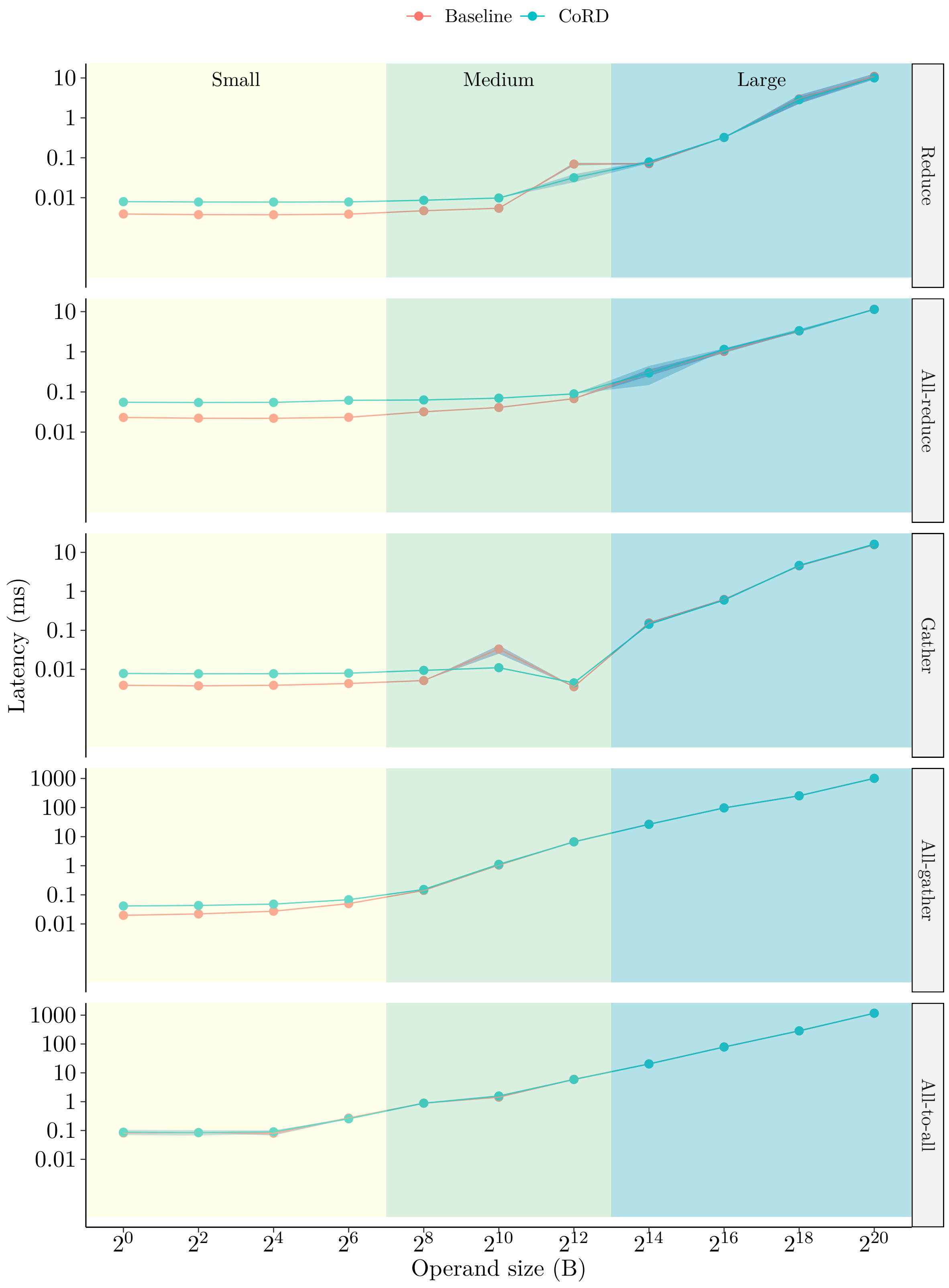


Figure 3.13: Basic collective operations on Oracle system. Lines show the average value, shades show the region of standard deviation around the mean.

achieves 81 µs, while CoRD achieves 90 µs. However, with 16 KiB operands, both CoRD and the baseline kernel achieve 20 ms latency. These experiments demonstrate that, that CoRD's disadvantage becomes less significant with more complex the communication patterns.

Overall, the experiments with MPI collective operations show that the baseline RDMA communication is faster or on par with CoRD. To our surprise, in some cases, CoRD outperforms the baseline RDMA communication by a significant margin that cannot be explained by experimental variability. Two examples in Figure 3.13 are the reduction operation with 4 KiB messages and the Gather operation with 1 KiB messages. Similar effects appear systematically across various runs and different compute node allocations.

Host and Network Configuration

When analyzing collective operations, we observed their performance could degrade due to two factors: network configuration and host configuration. First, consider the impact of network configuration. Figure 3.14 illustrates the latency of a 1 KiB-Gather operation with PFC-based flow control enabled and disabled. In this specific case, although lossless communication does not lose packets, it still suffers from congestion more than lossy communication, and CoRD offers even lower latency compared to both baseline configurations. Our interpretation of this performance anomaly is the following. Besides the protocol itself, the different network configurations also change the size of receive buffers allocated to each port on the switches. Lossy traffic has larger receive buffers but no guarantee of buffer space being available. Whereas lossless traffic has guaranteed receive buffer space, the size of the buffers is smaller. Unfortunately, we do not have access to the exact configuration of the network switches to confirm this hypothesis.

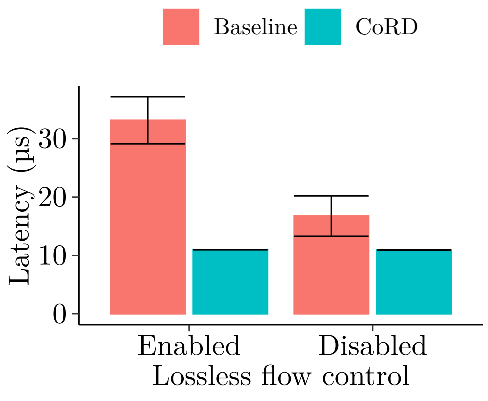
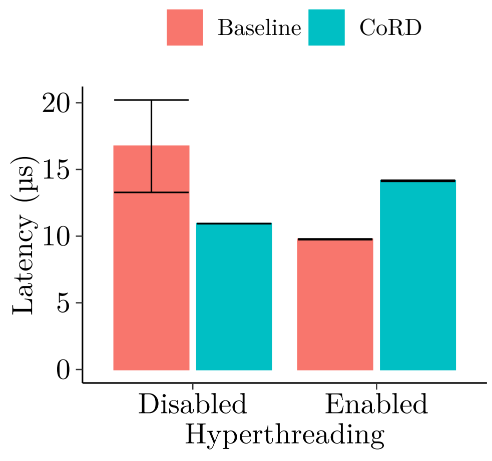


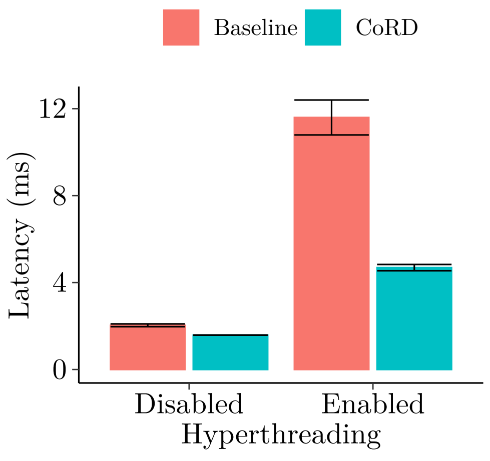
Figure 3.14: PFC influence on latency

Host side configuration can also impact latency, for example, by using hyperthreads. Figure 3.15 shows the latency of 1 KiB-sized Gather and All-to-All operations with and without hyperthreading. We ran the benchmark with 288 processes with PFC disabled on 8 nodes without hyperthreads and on 4 nodes with hyperthreads. Considering the fact that the benchmark does not use shared memory, hyperthreading should increase the latency because each NIC handles more processes, and processes on the sibling hyperthreads share the same core resources. This is what happened with the all-to-all operation (see Figure 3.15b) for both the baseline and CoRD kernels. On the other hand, the Gather operation (see Figure 3.15a) shows the opposite effect: hyperthreading decreases the latency on the baseline kernel.

Considering our focus on the OS aspect of RDMA communication, we did not investigate these effects further. Nevertheless, these results demonstrate that the performance of RDMA communication is influenced by many factors, and the performance of specific operations can change in unexpected ways. In this context, CoRD, despite adding to the latency of point-to-point operations, may not have a proportional impact on the performance of larger



(a) Hyperthreading decreases latency for Gather operation.



(b) Hyperthreading increases latency for All-to-all operation.

Figure 3.15: Latency influenced by hyperthreading

operations. Overall, the less predictable the system is, the less likely it is for CoRD to be the bottleneck.

Impact of Network Congestion

To understand the effect of network congestion on collective operations, we conducted an experiment where we modified the baseline user-space RDMA library to artificially delay the posting of new send requests. Such a delay reduces the rate at which the application can post messages and in a way emulates the behavior of CoRD.

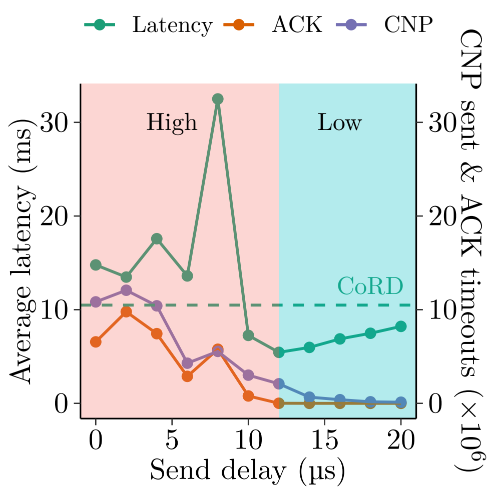


Figure 3.16: Timeouts increase average latency.

The benchmark ran all-to-all operation with 4 KiB operands over 288 MPI processes, varying the delay from 0 to 20 µs. To make the results more pronounced, we disabled PFC-based prioritization of RDMA traffic on the NICs. During the experiment, we first ran 2000 warm-up iterations to congest the network, and then we measured the latency of the operation over 100 iterations. We repeated each experiment 5 times and calculated the average of the results. Additionally, we added up the values of the performance counters reported by the NICs across all the nodes to quantify network congestion.

Theoretically, adding delays should have an adverse effect on performance, but that is not what we observed. Figure 3.16 shows that until a certain point, increasing the send delay *decreases* the average all-to-all latency. This period of high latency coincides with a large number of timeouts triggered by missing Acknowledgement (ACK) packets. The ACK packets go missing due to network congestion, which is also indicated by a high number of Congestion Notification Packets (CNPs) sent by the NICs. As soon as the number of timeouts reaches zero, the average latency reaches its minimum at a send delay of 12 µs. We identify the period with a send delay below 12 µs as the period of high network congestion, and the period with a send delay above 12 µs as the period of low network congestion.

For comparison, CoRD achieves an average latency of 10.5 ms for the same benchmark. At this latency, CoRD results in 20 million CNP packets sent by all the NICs, but only 0.1 million timeouts. The number of timeouts with CoRD is much lower than the number observed during the period of high network congestion with the baseline kernel, but the number of CNP packets is higher with CoRD. Nevertheless, CoRD is still faster than the baseline kernel without send delays, though it is higher than the minimal latency. This demonstrates that congestion is the primary source of latency overhead in our experiments.

Earlier in this section, we measured the latency overhead of CoRD in point-to-point operations to be around 1.5 µs, which is significantly lower than the optimal send delay of 12 µs. On the other hand, CoRD adds latency in several places, whereas send delay only affects the latency of the send operation. Therefore, these two numbers are not directly comparable.

High network congestion has an even more adverse effect on maximum latency, which is shown in Figure 3.17. In contrast to average latency, maximum latency correlates more with the number of CNP packets handled by the NICs. The difference between sent and received CNP packets comes from the fact that, in addition to dedicated CNP packets, a NIC can report congestion by setting the ECN bit in the header of regular packets. Therefore, the observed performance improvements in CoRD are just an unintended side effect of adding latency. The optimum send delay depends on the number of processes, the size of the operands, the exact benchmark, and even differs from allocation to allocation. Nevertheless, we always observed that sending packets at the highest possible rate is reproducibly detrimental to the performance of some specific operations.

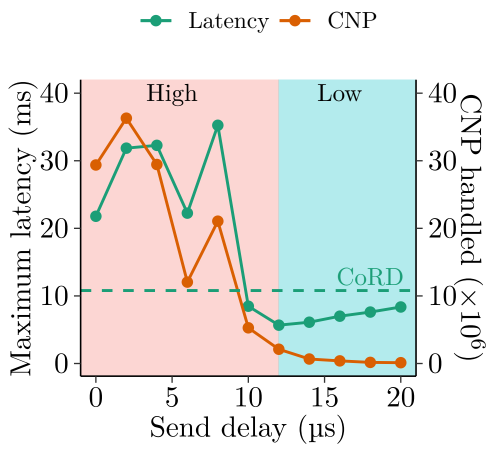


Figure 3.17: High congestion impacts maximum latency.

Although less frequent, congestion remains a problem, even in lossless networks (i.e. when PFC is enabled). Figure 3.18 shows the average latency of the Gather operation with a 1 KiB operand over 576 MPI processes using hyperthreads with PFC enabled. Similarly to previous experiments, we artificially delayed the posting of new send requests with the baseline RDMA communications, increasing the delay up to 14 µs. Increasing send delay reduces the average latency up to a certain point, after which the average latency rapidly increases.

In this experiment, we did not observe acknowledgment timeouts at all, indicating that the network indeed did not drop packets. Nevertheless, we observed a large number of ECN-marked packets, which indicates that the network was congested. Explicit Congestion Notification (ECN) is a mechanism that allows switches to notify endpoints about congestion without dropping packets by setting the ECN bit in the headers of packets passing through the congested switch ports [Zhu+15]. The ECN bit is set when the switch's buffer utilization exceeds a certain threshold. In response to receiving an ECN-marked packet, the NIC should send a CNP packet to the sender, which should then reduce the rate of sending packets. Correspondingly, the number of CNP packets is proportional to the number of ECN-marked packets.

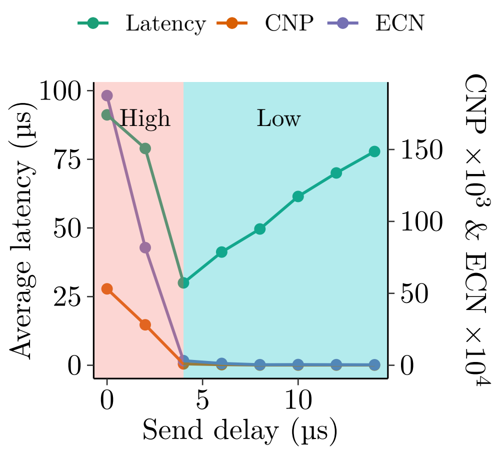


Figure 3.18: Lossless flow control is also subject to congestion.

In contrast to PFC-disabled flow control, with lossless flow control, the congestion is characterized by the number of ECN-marked packets, not by the number of timeouts. Nevertheless, in both cases, send delay emulates per-flow rate limiting. Similar rate limiting is also implemented by hardware-level congestion control algorithms, like DCQCN [Zhu+15]. Curiously, congestion control algorithms are designed to react to network congestion events on the scale of dozens of microseconds [Zhu+15], otherwise the algorithm may be prone to oscillations. Such time scales are also manageable for host-side software, particularly the OS. Therefore, experiments in this section indicate that it may be possible to implement RDMA congestion control at the OS and host level.

Application Benchmarks

To estimate how CoRD performs with real-world applications, we measured the performance of several MPI applications. First, we measured the performance of the popular NAS Parallel Benchmarks (NPB) suite [Bai+94] of size class D. Then, we looked at the performance of Gromacs [Hes+08], a popular molecular dynamics application.

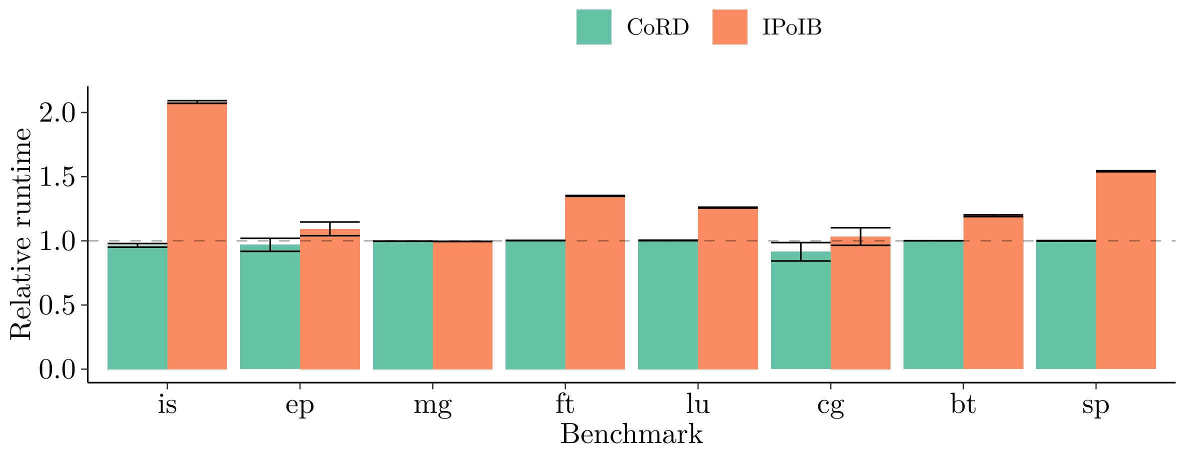


Figure 3.19: Relative runtime of the NPB benchmarks on Azure system.

We begin by evaluating the performance of the NPB benchmarks on the Azure system [4]. We compare communication over the bypass network, the CoRD network, and the IPoIB network. We selected IPoIB for comparison because it communicates over InfiniBand NIC and offers fine-grained control over data-plane operations, making it a functionally equivalent competitor to our CoRD architecture. At the same time, we did not assess the performance over the TCP protocol, as it is unavailable on InfiniBand networks. Each benchmark has limitations on the number of processes allowed for a run, which in our case ranged from 128 to 240 for different benchmarks.

For all benchmarks, the CoRD architecture incurs nearly zero overhead over baseline kernel-bypass communication, whereas IPoIB is up to 2× slower. IPoIB performs slowest with the IS (integer sorting) and SP (matrix factorization) benchmarks, which are simultaneously data-intensive, each process sends 72 Gbit/s and 34 Gbit/s, respectively, and message-intensive, both send around 1300 messages/second per process. EP (embarrassingly parallel), which communicates very little, and CG (conjugate gradient), which communicates using a few large messages, run slightly faster with CoRD communication. This behavior suggests that congestion control is also a factor in InfiniBand networks, albeit less pronounced than in RoCE networks.

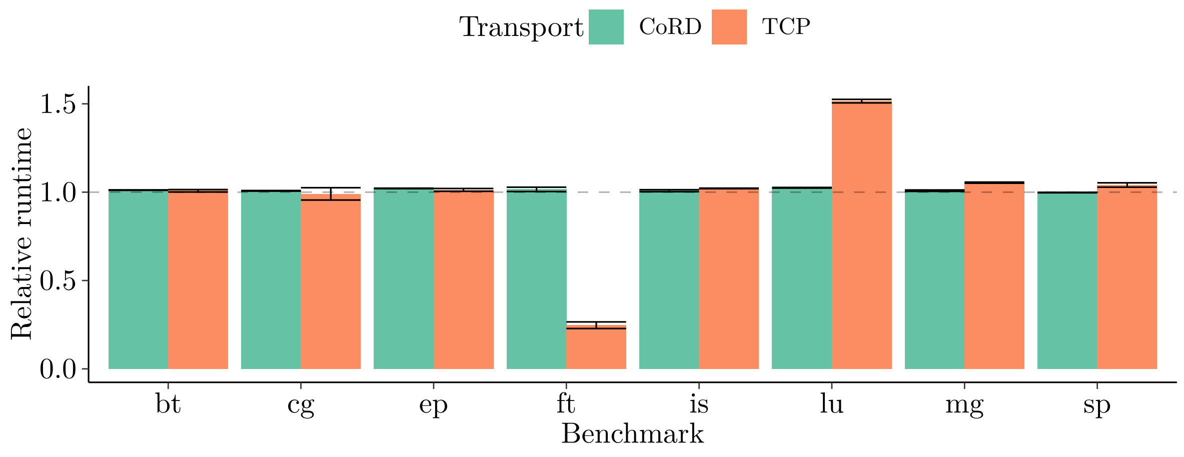


Figure 3.20: Relative runtime of NPB suite on Oracle system.

Figure 3.20 illustrates the relative runtime of the NPB benchmarks on the Oracle system with PFC-enabled flow control for CoRD and TCP-based communication[5], compared to the kernel-bypass version. The benchmarks were conducted on 8 nodes, utilizing hyperthreads, which yielded higher performance for all benchmarks except for FT. The number of processes used for each benchmark varied between 512 and 576 to accommodate the limitations of the respective benchmarks.

Similar to the Azure system, CoRD exhibited very low overhead compared to kernel-bypass communication. The most significant slowdown observed was 2.5% for the LU benchmark. Surprisingly, TCP-based communication demonstrated considerable performance variability. In the worst case, the LU benchmark ran 52% slower over TCP than over kernel-bypass communication. Conversely, in the best case, the FT benchmark ran 4× faster over TCP

[4]Our available resources permitted experiments only with two nodes and solely with NPB benchmarks.
[5]IPoIB is not available on RoCE networks.

than over kernel-bypass communication. While it is evident why TCP-based communication is slower than kernel-bypass communication for some benchmarks, the unexpected performance improvement warrants an explanation.

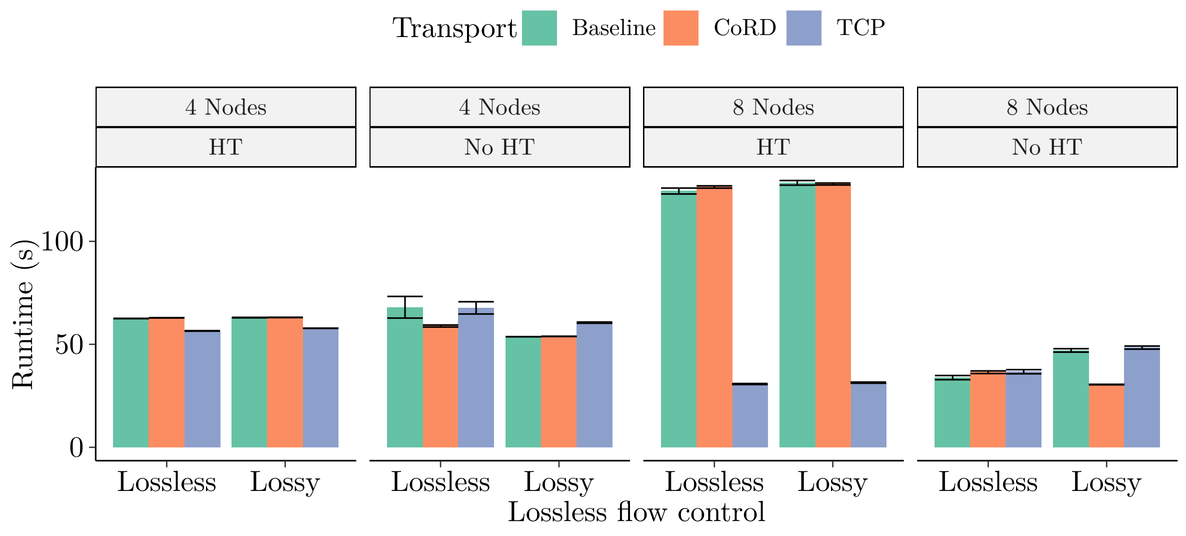


Figure 3.21: Runtime of NPB FT benchmark on Oracle system. Whiskers show the range of standard deviation centered around the mean.

There are two reasons why the FT benchmark ran faster over TCP-based communication. First, loopback communication over TCP operates entirely within the kernel, whereas loopback communication over an RDMA network passes through the NIC and necessitates more data movement. Second, when the FT benchmark runs with hyperthreads, it experiences congestion, resulting in overall performance degradation. Figure 3.21 displays the runtime of the FT benchmark with 4 and 8 nodes, with and without hyperthreads, with and without PFC-enabled flow control.

It is evident that the FT benchmark suffers from congestion, because it shows the worst runtime, when allocated the most resources: 8 nodes with hyperthreads. Among all configurations, the FT benchmark ran the fastest (30.5 s) with CoRD on 8 nodes without hyperthreads communicating over a lossy network. The second and third best configurations were TCP-based communication on 8 nodes with hyperthreads, with or without PFC-enabled flow control, which ran for 30.7 s and 31.4 s, respectively. Only then did baseline lossless RDMA communication on 8 nodes without hyperthreads follow, running for 33.8 s. CoRD achieves such good performance because it is less susceptible to congestion than the baseline kernel, as shown in the previous section. TCP-based communication is fast because it can bypass the physical network for loopback communication.

Finally, we measured the performance of Gromacs [Hes+08]. For this experiment, we utilized 8 nodes with hyperthreads and lossless communication, as this configuration delivered higher performance for all benchmarks, except benchRIB. Figure 3.22 presents the average runtime across 5 runs for Gromacs with four different data sets. Given that Gromacs is notably network-intensive, TCP-based communication is up to 4× slower than kernel-bypass communication. CoRD communication was also slower than kernel-bypass communication, but the disparity was considerably less.

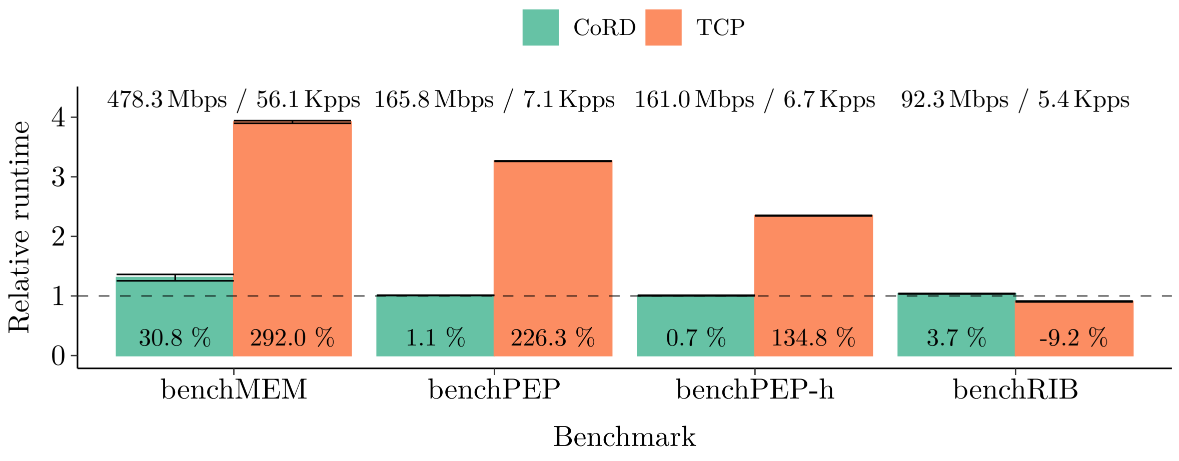


Figure 3.22: Runtime of Gromacs benchmarks on Oracle system. Annotations show average per-process throughput and packet rate of the baseline version, and the runtime overhead of CoRD and TCP communication, compared to the baseline RDMA communication (relative runtime is 1).

CoRD experienced the most significant slowdown of 30% for the benchMEM dataset, which is the shortest of the four datasets: it ran for 18.3 s with kernel-bypass communication and 23.9 s with CoRD communication. Whereas, other benchmarks took at least 200 s to complete, and the slowdown ranged from 0.7% to 3.7%. While being the smallest problem, benchMEM is also the most message-intensive, sending 478 Mibit s^{-1} and 56 kpacket s^{-1} per process. Together, all processes on one node send 33.6 Gibit s^{-1}, which is a third of the NIC's line rate. The other benchmarks reached only around a third of benchMEM's the data rate, and up to one-eighth of the packet rate.

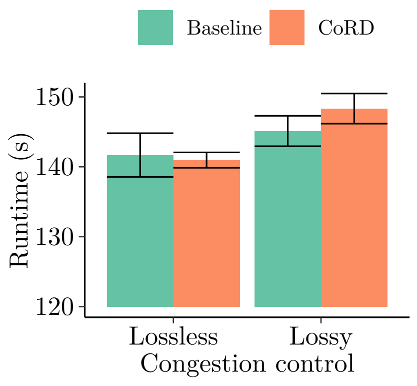


Figure 3.23: Congestion control impact on benchRIB. The Y-axis does not start at zero.

As we observed with other benchmarks, hyperthreads can have detrimental effects on the application performance. Specifically, for benchRIB, when hyperthreads are enabled, kernel-bypass ran for 240 s and CoRD ran for 249 s. Disabling hyperthreads reduced the runtime to 142 s and 141 s for kernel-bypass and CoRD communication, respectively. TCP-based communication also improves with hyperthreads disabled, but not as much as RDMA-based communication, running for 217 s and 193 s with and without hyperthreads, respectively. Disabling lossless flow control reduces performance for all configurations; however, the impact is relatively small (see Figure 3.23). This benchmark is similar to NPB FT (see Figure 3.21), in the way it reacts to configuration changes. Overall, when comparing the best available configurations for benchRIB, the performance of CoRD remains on par with the baseline kernel, and is significantly better than TCP-based communication.

The only application benchmark that showed a significant performance degradation with CoRD was benchMEM. The reason for this is that benchMEM is the most message-intensive benchmark and benefits from lossless networks, as well as from additional CPU resources. If

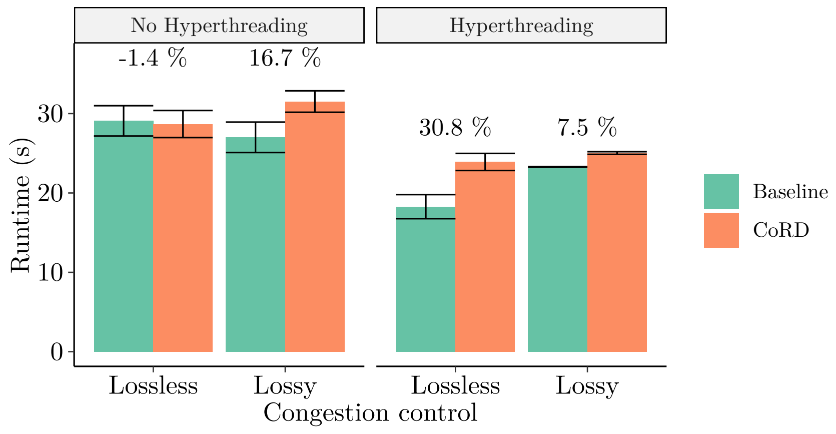


Figure 3.24: Performance of benchMEM with varying configurations. Annotations show runtime overhead of CoRD, compared to the baseline RDMA communication. Whiskers show standard deviation centered around the mean.

the configuration is not optimal, the performance gap between the baseline and CoRD shrinks dramatically.

Overall, our results show that in most circumstances, CoRD communication incurs a small overhead compared to kernel-bypass communication. In some cases, CoRD communication even outperforms kernel-bypass communication. We also observed the case when CoRD has been at a significant disadvantage compared to the optimal configuration of the baseline kernel, but finding such a configuration in the real world is not straightforward because it depends on the application, its communication patterns, host and network configuration, and other factors.

3.2.4 Use Cases

To demonstrate the usefulness of CoRD, we implement three use cases. The first use case is a tool for monitoring per-process RDMA traffic. The second use case is a rate-limiting mechanism for RDMA communication. The third use case demonstrates how CoRD can be used to implement congestion control for RDMA communication. All the use cases are implemented in the OS kernel and provide functionality not available in the baseline kernel-bypass implementation.

Traffic Monitoring

In traditional RDMA networks, the NIC is responsible for sending/receiving packets and reporting statistics about the traffic. However, a NIC typically reports only aggregated statistics for all processes running on the same node. This limitation makes it difficult to understand the traffic patterns of individual processes.

Some per-process traffic statistics are available through the iproute2 tools [Hem+23]. Unfortunately, these statistics do not include the number of sent bytes or packets [6]. Applications may use the `ibv_read_counters` API to query their own traffic statistics [Ros17]. However, this API is designed for use by the application itself, and the OS does not have easy access to these

[6]The set of available counters is defined in `drivers/infiniband/hw/mlx5/counters.c`.

statistics. Moreover, both approaches are available only for advanced NVIDIA NICs and not for NICs from other vendors.

To address this shortcoming, we added a Linux kernel tracepoint [Ros11] to the `ib_core` driver right before it invokes a device-specific function to send a message. A tracepoint allows to attach a user-defined function to a specific place in the Linux kernel, but when the tracepoint is not in use, the overhead is virtually zero. In our case, the user-defined function is provided by a user-level system service that records the number of bytes and packets sent by each process. We developed a simple user-level service that uses the Aya [Dec+21] framework to attach an eBPF [Tig23] program to the newly defined tracepoint.

An eBPF program can introduce significant overhead to a microsecond-scale operation. To reduce this overhead, we accumulated the data for each send request in an eBPF map and only shared the aggregated data with the user-level service. The user-level service then checks the map for new data every 1 s and reports the results to the user. With such a careful implementation, we ensured that the impact on application performance was negligible.

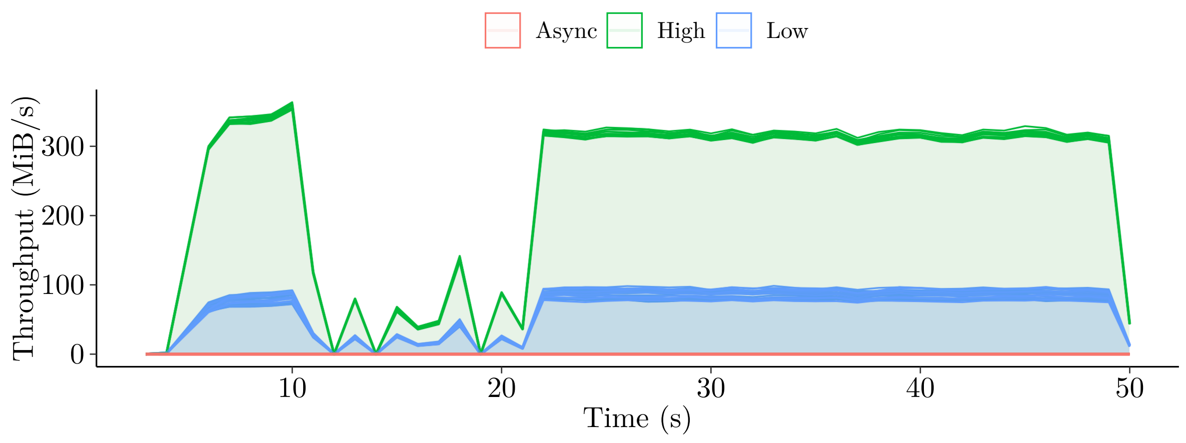


Figure 3.25: Monitoring per-process traffic of Gromacs.

Figure 3.25 shows the monitoring results from one of the four nodes running the Gromacs benchmark with the benchMEM data set and dynamic load balancing enabled. In total, our tool observed 72 threads belonging to 36 different MPI ranks. Each MPI rank has two threads: one for the main thread and one for the asynchronous helper thread. From the figure, we can see that there are two groups of main threads: one group with 12 high-traffic threads and another group with 24 low-traffic threads. This separation corresponds to the Gromacs architecture, which assigns threads to two dedicated roles [Hes+08]. A sudden drop in traffic from 11 s to 21 s corresponds to work rebalancing among the threads.

Compared to hardware-offloaded monitoring, the advantage of our approach is that we can deploy arbitrary monitoring logic without modifying the application. If necessary, we can also create new tracepoints or attach to other probe points. CoRD-based monitoring is aware of high-level OS concepts, like processes and threads, making it more flexible than hardware-offloaded monitoring. Finally, CoRD is portable because it does not require any special hardware support.

Rate Limiting

When multiple applications share the same node, to ensure fairness it can be desirable to enforce a limit on the amount of traffic each process can send. Existing RDMA solutions offer two main approaches to rate limiting. In the first approach, the application itself implements rate limiting by relying on the InfiniBand verbs API [Mel15] to set a per-flow packet rate. In the second approach, the OS employs hardware virtualization to enforce rate limits [NVI23].

The disadvantage of the first approach is that the OS has no influence over what the application sets as the rate limit. Therefore, it is hard for the OS to manage and coordinate multiple applications running on the same node. The disadvantage of the second approach is that it is coarse-grained because it allows setting only a single limit per Virtual Machine (VM) or application instance. Moreover, similar to the traffic monitoring case, some rate-limiting features are only available for advanced NVIDIA NICs [RW18].

In contrast, CoRD allows for a fine-grained, vendor-agnostic rate limiting implementation. To demonstrate this, we implemented a simple rate-limiting mechanism as a CoRD function. Our rate limiter enforces a specific throughput limit at the level of Linux cgroups [Heo15]. Such an interface allows the OS to enforce a limit on the amount of traffic each process can send.

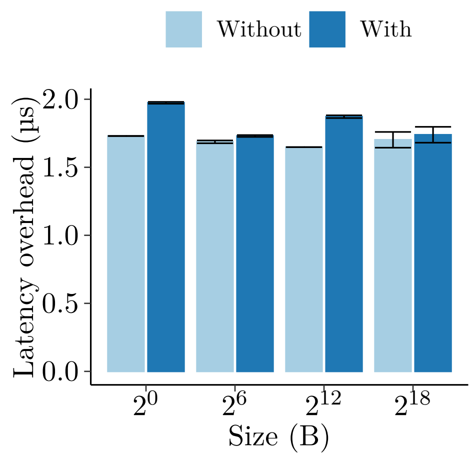


Figure 3.26: CoRD overhead with and without rate limiting support.

For each cgroup, the rate limiter maintains a counter of the number of bytes sent by all the processes belonging to the cgroup within a 200 ms sliding window. If the process attempting to send a new message is about to exceed the limit allocated for its cgroup, the rate limiter puts the process to sleep for a short period of time. The sleep time is calculated based on the current throughput of the cgroup and the amount of data the process is trying to send. The cgroup interface allows for the creation of hierarchical resource limits, so the rate limiter needs to ensure that the limit is enforced for every cgroup in the hierarchy.

Previously, we stated that OS functions must have minimal overhead for CoRD to be useful. Figure 3.26 shows the overhead of CoRD with and without the rate limiter. In contrast to traffic monitoring, the rate limiter has a latency overhead of around 300 ns, even when rate limiting is not enforced. We attribute this overhead mostly to the need to grab a mutex to access the cgroup data structure. We believe that using a faster mutual exclusion mechanism would reduce this overhead, but from our point of view, the convenience of the cgroup interface outweighs the overhead.

To demonstrate how our rate limiter works, we conducted an experiment on two Oracle Cloud nodes. Each node was running 18 instances of the `ib_send_bw` benchmark, split into two groups of 9 processes. The benchmarks on one node were sending 64 KiB messages to the benchmarks on the other node. A process in the first group (group A) sends messages over a single QP, whereas a process in the second group (group B) sends messages over 64 QPs at once. Every

10 s, each process reports its observed throughput. Approximately every 180 s, we change the configuration of the rate limiter to observe how the benchmarks react to the change.

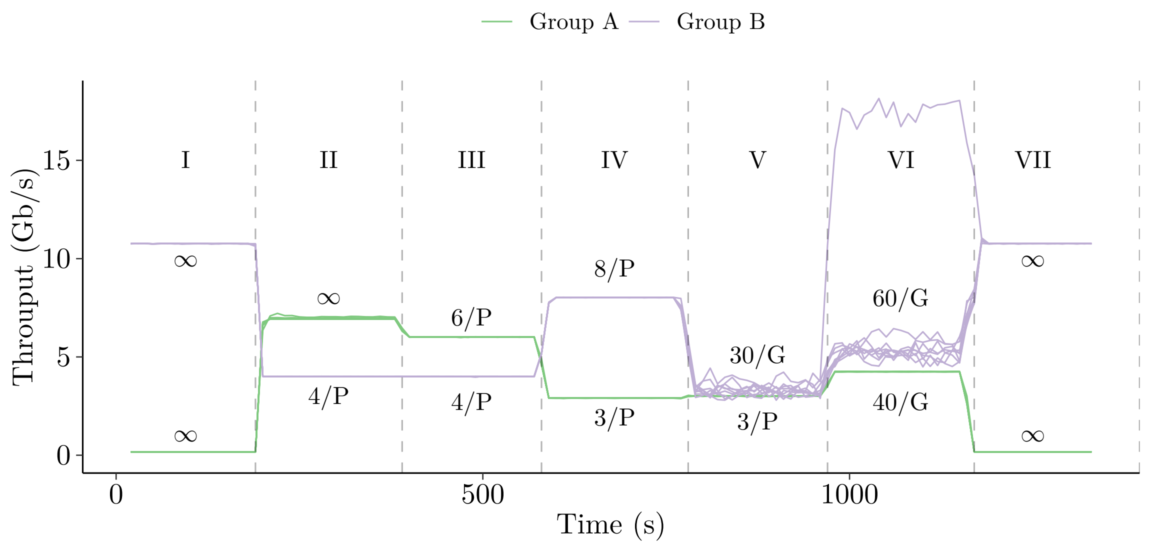


Figure 3.27: Throughput of rate-limited processes. The group has either no limit (∞), a per-process limit (e.g. 4/P), or an aggregate limit (e.g. 40/G).

Figure 3.27 shows the throughput observed by each individual process. We changed the rate limiter configuration 6 times, creating 7 periods with different configurations. The benchmarks start by sending at full speed (phase I) without any intervention from the rate limiter. Processes in group B receive a disproportionate share of the bandwidth because the NIC implements load balancing on a per-QP basis. The aggregate throughput of all the processes is 98.4 Gbit/s (see Figure 3.28), which is just a little short of the theoretical maximum of 100 Gbit/s.

Next, we attempt to prioritize processes in group A over the processes in group B. For that, we set the rate limit to 4 Gbit/s for each process in group B, so that group A can claim the remaining available bandwidth. To set the rate limit for each process in group B precisely, we created an individual cgroup for each process. As a result, in phase II, processes in group A receive approximately 6.9 Gbit/s throughput, and the aggregate throughput of all the processes remains the same as in phase I.

In phase III, we set the rate limit for each process in group A to 6 Gbit/s. Considering that the aggregate throughput of all the processes is limited to 90 Gbit/s, the observed aggregate throughput also drops (see Figure 3.28). In phase IV, we set the rate limit for each process in group B to 8 Gbit/s and for each process in group A to 3 Gbit/s. In all these cases, all the processes observe the expected throughput.

In phase V, we moved all the group B processes to a single cgroup and set the rate limit for this cgroup to 30 Gbit/s. Figure 3.28 shows that the aggregate throughput of all the processes in group B is indeed 30 Gbit/s, but Figure 3.27 shows significant variation between the processes within the group. Although we correctly implemented the rate limiting, our algorithm does not guarantee fairness between the processes within the same cgroup. As a result, some processes in the cgroup are blocked more often by the rate limiter than others. A better algorithm would also provide fairness [Rad+14], but rate limiting algorithms are beyond the scope of this thesis.

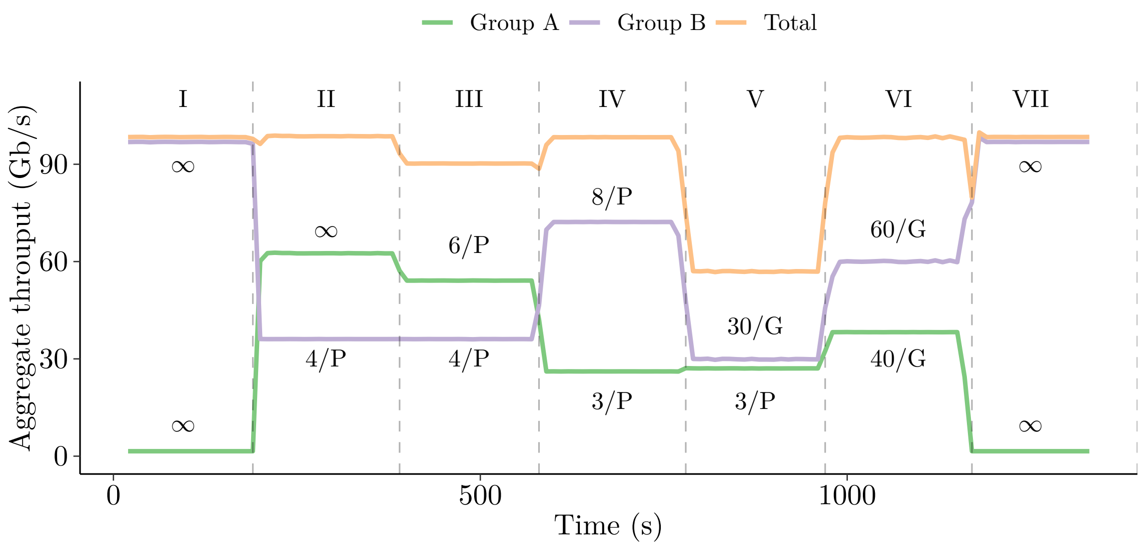


Figure 3.28: Aggregate throughput with rate limiting. The group has either no limit (∞), a per-process limit (e.g. 4/P), or an aggregate limit (e.g. 40/G).

In phase VI, we set the aggregate rate limit for group A to 40 Gbit/s and for group B to 60 Gbit/s. In this case, most of the processes within group A observe equal throughput, but there is even more variation in group B. Again, as in the previous case, we do not guarantee fairness between the processes within the same cgroup. Finally, in phase VII, we remove the rate limit for all the processes, and the bandwidth distribution returns to the state of phase I.

Congestion Control

The final use case demonstrates how CoRD can be used to implement congestion control for RDMA communication. We expand on the congestion control experiment from Section 3.2.3, where we added a send delay to see how rate limiting impacts the performance of MPI collectives. In this experiment, we add send delay as an OS function and study its impact on the performance of Gromacs.

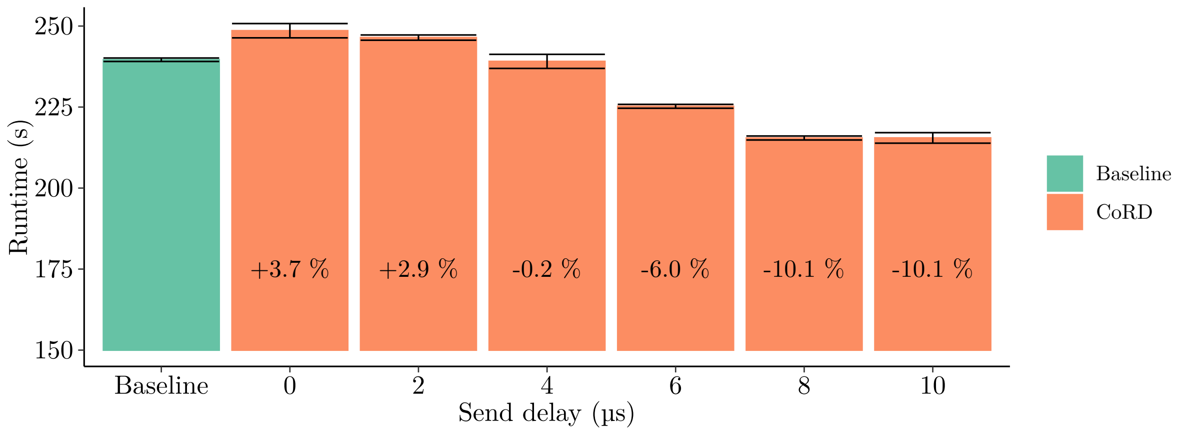


Figure 3.29: Reducing network congestion with CoRD

Figure 3.29 shows the run of Gromacs with the benchRIB dataset with hyperthreads and lossless flow control. Although the baseline RDMA configuration suffers from congestion, CoRD still runs slower. However, when we add a send delay to CoRD, the performance of the application improves by up to 10%. This result suggests that CoRD can be used to implement congestion control for real-world applications.

When running Gromacs with the benchPEP and benchPEP-h datasets, we observed no significant changes in the performance of the application when adding a send delay. On the other hand, when running Gromacs with the benchMEM dataset, adding any send delay increased CoRD overhead even further, in addition to the already existing 30% overhead (see Figure 3.24).

Our demonstration shows that the OS has the potential to control congestion in RDMA networks. Typical criticism of employing the OS for such a task is that it cannot react to congestion events in a timely manner [Bar+17b], therefore this task is better left to NICs and switches. Our results suggest that there are high-level sources of congestion, which can be controlled by the OS quickly enough. This role can be even more important if applications running on the same host do not coordinate their communication patterns, and therefore the NIC and switches cannot help alleviate the congestion.

3.2.5 Summary

CoRD is a novel continuous RDMA dataplane interposition architecture emphasizing interoperability with existing RDMA applications. Despite a measured overhead of 1.5 µs in latency microbenchmarks, we demonstrate that this overhead is very small for real-world applications. Benchmarking of MPI collectives shows that, despite adding overhead to communication, CoRD can still use the network very efficiently overall. In particular, CoRD has been able to achieve near-baseline performance for many operations because the actual communication bottleneck was not the CPU overhead, but the network congestion. However, it is clear that CoRD has room for improvement in reducing network congestion on the host side by coordinating all RDMA processes running on the same node. So far, the focus of our network congestion control study has been on the RoCE protocol, leaving a similar study with the InfiniBand network for future work. We also demonstrate that CoRD can be employed to implement a range of functions without necessitating modifications to the application code.

3.3 Fast System Calls

Although CoRD demonstrates that the overhead of system calls is not necessarily a bottleneck for high-performance applications, there is still room for improvement. To understand how much room for improvement there is, we take a bottom-up approach by constructing a minimalistic continuous interposition mechanism. This mechanism strips down the traditional system call mechanism to the bare minimum. Such an approach allows us to understand the fundamental limitations of software-level RDMA dataplane interposition.

The main idea of the minimalistic approach is to remove all the unnecessary overhead from the system call mechanism. Unfortunately, naively removing instructions from the Linux kernel system call implementation would result in a system being insecure, unsafe, and dysfunctional. To address this challenge, we propose *fastcalls*, a fast path system call implementation.

Fastcalls reside in *fastcall space*, a software layer that logically lies between user and kernel space. Fastcalls are guaranteed to have very low overhead, compared to traditional system calls, and can be employed for high-performance applications with extremely high performance demands. Fastcalls build on the ideas of the Exokernel [EKO95] architecture, but apply them to modern systems in the context of contemporary security threats.

Among the main concerns for the fastcall architecture are side-channel attacks, like Meltdown [Lip+20] and Spectre [Koc+19a]. These attacks allow leaking sensitive information from the kernel to a malicious user space application, without triggering any security checks. Different modern CPUs are vulnerable to various side-channel attacks, but the fastcall architecture requires a guaranteed level of protection for all of them.

3.3.1 Fastcall Architecture

Fastcalls provide a low-latency alternative to system calls by minimizing the transition overhead between user space and fastcall space. To achieve this low overhead, each fastcall implements only the fast path of an application-specific use case, like sending a network packet of a predefined protocol type and with a fixed destination address. Fastcalls trade off the generality and some of the security of system calls for a faster implementation of privileged operations.

The first design goal is to incur minimal latency overhead when invoking a fastcall function. To this end, the code of a fastcall function is very simple and highly application-specific. To enable such a requirement, each user-space process has its own private *fastcall space*.

The second goal is to enable secure enforcement of OS policies. For that, a user application must not be able to manipulate a fastcall's code or data. In modern CPUs, such isolation is achieved by making the fastcall code accessible only from within a privileged CPU mode. Then, to access the required service, similarly to system calls, the user application enters privileged mode through fixed entry points defined by the OS.

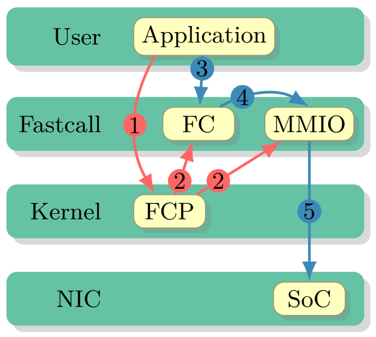


Figure 3.30: Fastcall architecture.

To better understand the fastcall architecture, consider an example where an application aims to send packets over the network using fastcalls (see Figure 3.30). We create a fastcall space by allocating a portion of the application's virtual address space. When a process spawns, its fastcall space is initially empty, meaning the application has no access to any fastcalls. The application begins by requesting (1) the *fastcall provider* (FCP) to register a fastcall (FC) function for the application process. The fastcall provider, a kernel component, is responsible for creating and managing fastcalls. The provider may contact user-level OS services to make policy decisions or compose fastcall code.

After the creation of the fastcall code, it includes OS-defined policies, such as filtering the destination address for outgoing packets or conditionally switching to the kernel for more complex processing on selected packets. Fastcalls are strictly controlled by the fastcall provider, preventing the application from bypassing or removing these checks. Finally, the provider maps the fastcall into the application's fastcall space (2), along with the resources necessary for the fastcall to fulfill its task. In this example, such a resource could be an MMIO *doorbell* register, which is used to trigger the NIC data transmission.

Now, if the application seeks to use the NIC for sending a packet, it directly invokes the fastcall function (3). The fastcall interacts with the MMIO region (4) of the NIC to initiate the corresponding I/O operation (5). Should the application invoke a fastcall with parameters that violate the OS policies established for that fastcall (such as sending a network packet to an unauthorized address), the fastcall function will reject the operation and return to user space. The application can then either revise its request or resort to the standard OS stack by making a system call.

The kernel is responsible for ensuring that the fastcall is self-contained in terms of memory accesses and the code it executes. This stipulation implies that the fastcall must neither conduct memory operations outside its dedicated region nor execute any code not explicitly part of it, such as external functions. There are two reasons for these restrictions: First, to facilitate maximum performance and straightforward implementation, the fastcall should not trigger CPU exceptions like page faults. Second, to enable the fastcall provider to verify the fastcall's logic, all potential execution paths of a fastcall must be observable. While this thesis does not delve into fastcall verification, we anticipate that fastcalls would be automatically verified in a manner akin to eBPF [Tig23] functions.

To ensure that a fastcall does not cause page faults, each fastcall is allocated a block of *scratchpad* memory for use as a stack. Additionally, the kernel sets up a pinned memory region shared between the fastcall and the application. Part of this shared region is writable by the application, while another part is read-only. The application can transfer data to a fastcall only through CPU registers or via this shared memory region.

To reduce the latency of fastcall invocation, the fastcall space does not incorporate software-based side-channel-attack mitigations. Specifically, the fastcall space exists within the same virtual address space as the application and is not safeguarded by measures like KPTI [Cor17a]. Therefore, although applications cannot alter data within the fastcall space, such data should not be considered confidential in the event of side-channel vulnerabilities like Meltdown [Lip+20]. This implies that the fastcall space should be designed as though the user application had read access to its own fastcall space. Consequently, the fastcall space must exclude any information that must remain confidential from a user application.

Furthermore, fastcall functions do not interact with the OS kernel in any way. A fastcall provider must verify that the code of a fastcall neither accesses kernel data nor calls any kernel functions; therefore, fastcalls must be self-contained (for security reasons, calling into user functions is also forbidden). As a side effect, this means that on systems running their kernel in a separate address space, there is strong isolation between fastcall space and kernel space.

For our primary implementation of the fastcall framework (see Section 3.3.2), which we call *privileged*, the CPU mode in which the fastcall space resides is equal to the standard kernel mode. However, this is not fixed by design. For example, Section 3.3.3 explores an *unprivileged* fastcall implementation, keeping the fastcall space in user space to further reduce the fastcall invocation latency.

Privileged fastcalls require the kernel to keep KPTI enabled, even if the CPU is not vulnerable to a Meltdown [Lip+20] attack. The reason for this is that from the CPU's perspective, fastcalls run in the same CPU mode and the same address space as the kernel code. Existing hardware- and software-based side-channel mitigations must be employed at the boundary between trusted and untrusted code. Fastcalls remove this boundary, so the kernel must ensure that the fastcall space is not used to leak sensitive information. One way to achieve this is to treat fastcall code as untrusted and keep fastcalls and kernel in different address spaces. As a result, privileged fastcalls slow down the performance of traditional system calls.

3.3.2 Privileged Fastcalls

Privileged fastcalls are the primary implementation of the fastcall architecture in this thesis. This implementation resembles a significantly stripped-down version of the traditional system call mechanism. Compared to unprivileged fastcalls (see Section 3.3.3), privileged fastcalls have a different invocation mechanism and run inside the kernel. This section describes the implementation of the creation and invocation of privileged fastcalls for the Linux kernel on the `x86-64` architecture[7].

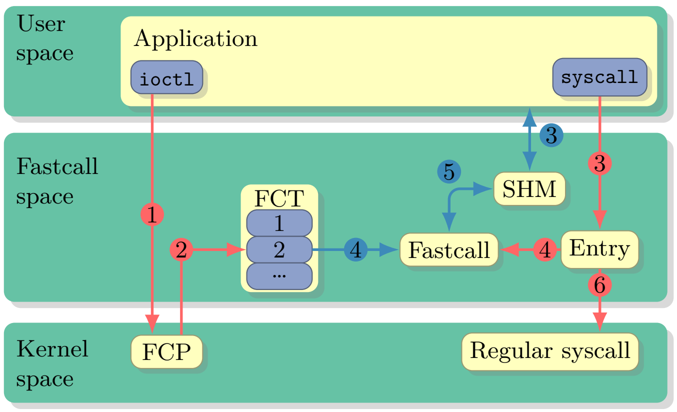


Figure 3.31: Registration (left) and invocation (right) of privileged fastcalls in `x86-64` implementation.

To add a new fastcall, the application makes a request to the Fastcall Provider (FCP) via the `ioctl` system call (1 in Figure 3.31). In our implementation, FCPs are loadable kernel modules that interact with the fastcall infrastructure built into the kernel. If the application is authorized to use the requested fastcall, the fastcall provider sets up the fastcall handler and inserts it into the application's per-process fastcall space (2). The per-process fastcall space hosts the data structures required for the fastcalls' runtime environment.

Each fastcall space contains a Fastcall Table (FCT), which includes entries for the metadata required to execute individual fastcalls. This fastcall metadata contains a pointer to the code of a fastcall function, configuration parameters, or pointers to memory regions associated with the fastcall. Additionally, the fastcall space hosts a memory region shared between the user application and the fastcalls for exchanging fastcall arguments, which are too large to fit into CPU registers. This shared region is similar to User-level Thread Control Block (UTCB) in the L4Re Microkernel [Lac04]. Depending on the requirements of specific fastcalls, the fastcall space also includes MMIO mappings and fastcall-private memory pages (e.g. for locks and counters) required by specific fastcalls. Finally, to support fastcall functions written in C,

[7]The source code is available at `https://github.com/planeta/linux/tree/mplaneta/fastcall/dev`.

fastcalls require a stack. To avoid concurrency issues, there is one dedicated stack per CPU, and interrupts are kept disabled during fastcall execution.

When the application invokes a fastcall, it first puts the fastcall arguments into the shared memory (3), and then executes the `syscall` instruction with a special value in the `rdi` register (3). This instruction transitions the CPU into privileged mode (ring 0), where the entry point handler dispatches the execution either to the *fastcall dispatcher* (4) or to the standard system call handler (6). Distinguishing between fastcalls and standard system calls right at the kernel entry allows for minimizing fastcall overhead. The fastcall dispatcher locates the requested fastcall function in the fastcall table (4) and executes it with the arguments passed either through CPU registers or through the shared memory region (5).

Figure 3.32 shows the code of the fastcall entry point called `entry_SYSCALL_64` in the Linux kernel. Line 3 checks if the system call number corresponds to a fastcall, and line 5 jumps to the fastcall dispatcher if that is the case. The `%rdi` register holds the index of the invoked fastcall. Lines 12 to 16 compute the address of the fastcall function using the fastcall number as an index into the fastcall table. Line 18 jumps to the fastcall function. Finally, the fastcall returns to the application via `sysret`.

```
 1 // kernel entry point
 2 entry_SYSCALL_64:
 3   cmpq $NR_fastcall, %rax
 4   // jump to fastcall dispatcher
 5   je fastcall
 6   /* original kernel entry
 7    * sequence [...] */
 8
 9 // fastcall dispatcher
10 fastcall:
11   cmpq $NR_TABLE_ENTRIES, %rdi
12   // table index out of bounds
13   jae error
14   movq $TABLE_ADDR, %rax
15   imulq $TABLE_ENTRY_SIZE, %rdi
16   addq %rdi, %rax
17   // &some_fastcall_function
18   jmpq *(%rax)
19
20 // example fastcall
21 some_fastcall_function:
22   /* fastcall function body
23    * [...] */
24   movq <RETVAL>, %rax
25   // return to user space
26   sysretq
```

Figure 3.32: Fastcall entry and dispatch.

Fastcalls do not execute operations common to system call entry and exit. For example, to avoid potentially unnecessary saving of processor state, we assume all registers to be callee-saved (i. e., saved by the fastcall function). On `x86-64`, fastcalls avoid setting up and tearing down the kernel environment (e. g., `swapgs`) and mitigating side-channel attacks like Spectre [Koc+19a], Meltdown, and Microarchitectural Data Sampling (MDS) [Can+19; Van+19]. Fastcalls also avoid storing and restoring general-purpose registers, the system call table lookup, and consistency checks unnecessary for the fastcall environment. These savings enable fastcalls to have an overall lower latency than system calls.

During execution, fastcalls have privileges similar to those of the kernel and, except for the shared region, fastcall space memory is not accessible from user mode. However, because KPTI [Cor17a] must be enabled, fastcalls cannot access kernel memory, thus they can neither read nor write kernel data, nor call kernel functions.

To simplify implementation, the kernel resets the fastcall space after `fork` [POS17], so no special care is needed to handle fastcall data structures, including actively used locks. The handling of forked processes is a known, albeit solvable, problem for RDMA applications [Mel15], which is beyond the scope of this thesis.

In essence, the presented design enables fastcalls to perform privileged operations without fully entering the kernel, which results in reduced latency compared to system calls. Even in the presence of currently known side-channel attacks, an application cannot interfere with or actively manipulate memory in the fastcall space. Therefore, although not confidential, fastcall space is safe even in the presence of malicious user applications.

3.3.3 Unprivileged Fastcalls

Despite removing most of the expensive operations, the privileged fastcall implementation described in the previous section still relies on a relatively slow `syscall` instruction to enter the kernel. In this section, we consider what would be required to remove this instruction and make fastcalls even faster. For that, we explore an implementation of a fastcall transition mechanism, which relies on the relatively recent ability of several modern CPUs to create executable non-readable pages [Jin21]. Our goal is to devise a mechanism that would allow invoking fastcalls almost as fast as a regular function call.

On a high level, executable non-readable pages work as follows. The OS maps an executable non-readable page into the address space of an application. The application can execute instructions located inside the non-readable page by jumping or calling into an arbitrary location in the non-readable page. When the control flow is already inside such a page, the CPU can also proceed to the next instruction without causing a CPU exception. Even if an instruction inside the non-readable page has an immediate value stored as an opcode, this operation still succeeds. On the other hand, data movement instructions, where one of the operands points to the executable non-readable page, would result in a CPU exception. We use this property to store secrets as immediate values in the instruction stream without the user application being able to read them directly. Then, a fastcall can be designed in such a way that its execution path depends on knowing the secret value.

For this architecture, we introduce two new page types, which are mapped into the fastcall space. The first page is a *protected page* with a secret, which is mapped as executable non-readable. The second page is a *hidden page*, which is mapped to a randomized location in the fastcall address space. A hidden page is readable and writable for the user application, but the user application does not know the address of the hidden page. Our goal is to design a fastcall in such a way that only the code inside the fastcall can access the hidden page, without leaking the address of the hidden page to the user application.

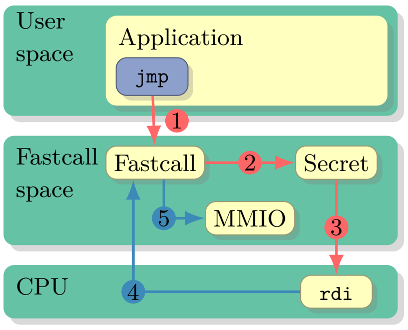


Figure 3.33: Successful invocation of an unprivileged fastcall.

Figure 3.33 shows the successful invocation of an unprivileged fastcall. The user application starts by invoking a fastcall (①). The fastcall first jumps into the protected region (②) to load the secret into a CPU register (③). After setting the secret for fastcall use, the code in the protected region jumps unconditionally back into the fastcall. This way, the secret is only accessible to the fastcall. Having the secret, the fastcall can check if the request from the user application follows the OS functions. If the request is permissible, the fastcall uses the secret (④) as an address to the hidden page (⑤), thereby executing a privileged operation. During its execution, the fastcall is not allowed to write the secret into the memory accessible by the user application. Finally, before returning to the user application, the fastcall erases the register containing the secret and then returns to the user application.

The application could try to bypass the OS functions in a fastcall and jump into the part of the fastcall code that accesses the protected page (1 in Figure 3.34). Such an attempt would fail, because the fastcall code does not know the address of the protected page (?). Alternatively, the application may jump into the protected page (2) in the hope of extracting the secret. Then, after loading the secret into a CPU register, the code in the protected region will still jump into the fastcall (3), which will erase the secret before returning to the user.

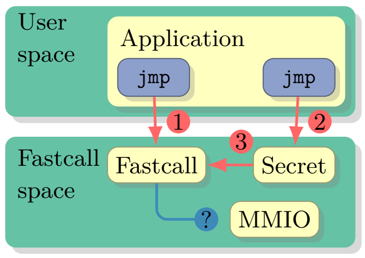


Figure 3.34: Malicious invocation of an unprivileged fastcall.

In addition to the measures described above, it is important that the Linux kernel does not leak the address of the hidden MMIO region through some kernel API. In our implementation, we randomize the addresses of the hidden pages inside the fastcall space and remove hidden page mapping from application-accessible process metadata, like the `/proc/self/maps` file. Otherwise, the application could circumvent the trampoline routine and access protected memory directly.

As we mentioned before, protected pages can be implemented in several ways. On Intel `x86-64` CPUs, this feature can be implemented using either the Intel MPK [Par+19] extension or the Intel Extended Page Table (EPT) [Int22, Section 29]. Apple M1 CPUs allow for protected pages through the Guarded Exception Levels (GXF) feature [Sve21]. Despite the available hardware, for the purpose of this thesis, we assumed the protected page to be executable non-readable, but we did not employ any of these techniques. The reason for such simplification is that our main focus has been the performance evaluation of unprivileged fastcalls and not the exact implementation details[8].

Figure 3.35 shows the code of the unprivileged fastcall entry point. When a user application invokes the example fastcall, the fastcall dispatcher (similar to the one described in Section 3.3.2) eventually jumps to Line 2. The fastcall starts by jumping to the protected page (Line 14), which, in our implementation, is located immediately after the fastcall code page. After loading the secret (Line 17), the control returns back (Line 18) to the fastcall (Line 4) and continues similarly to privileged fastcalls. The secret has been written by the OS into the protected page as an immediate argument of the `mov` instruction during the fastcall creation. Before returning, the fastcall erases the secret (Line 8) and sets the return value (Line 10).

```
 1  // example fastcall
 2  some_fastcall_function:
 3    jmp load_secret_1
 4  return_from_secret_1:
 5    /* fastcall function body
 6     * [...] */
 7    // erase secret
 8    xor %rdi, %rdi
 9    // set return value
10    movq <RETVAL>, %rax
11    ret
12
13  // Starts from the next page
14  load_secret_1:
15    /* load the address of
16     * the hidden region */
17    movq <SECRET_1>, %rdi
18    jmp return_from_secret_1
```

Figure 3.35: Unprivileged fastcall entry.

To ensure the confidentiality of the secrets, we implemented unprivileged fastcalls in Assembly language because the C language does not allow marking certain

[8]We tried to implement unprivileged fastcalls for M1 CPUs, but at the time, Linux kernel support for the M1 architecture had been limited, and we did not complete our prototype.

registers as containing a secret. Overall, unprivileged fastcalls offer very low overhead for privilege invocation in exchange for secret-based isolation. We study how secure such secret-based isolation on modern CPUs is in Section 3.3.4.

3.3.4 Evaluation

Fastcall space allows the OS to provide user applications with very low-overhead access to privileged operations. Considering that overhead is a priority, the goal of this section is to provide a quantitative performance comparison with other similar mechanisms. Additionally, we will provide a quantitative analysis of the security properties of unprivileged fastcalls.

The overall time required to execute a privileged operation consists of three parts. The first part is the time required to transition to and from the privileged mode. The second part is the time required to sanitize the arguments of the privileged operation and conduct necessary security checks. The final part is the time required to execute the privileged operation itself. The last two parts are very use-case-specific, and therefore, it is hard to provide a general estimate of their runtime. Therefore, we focus only on the overhead coming from the transition mechanism.

To evaluate fastcalls, we chose several comparison points. The fastest way to implement a kernel-supplied function is the vDSO library that is mapped into each user application [Fry23]. Effectively being normal function calls without any mode transitions, vDSO functions serve as a lower bound for the overhead introduced by any of the mechanisms used in our experiments. Special-purpose system calls or `ioctl`-based handlers do run in privileged mode, so they serve as the upper bound for fastcalls.

Table 3.2: CPU models used for latency measurements.

CPU Model	Short Name	Clock Rate	Cores	Released	Meltdown
Intel Core i7-4790	Intel 4790	3.6 GHz	4	2014	Vulnerable
Intel Xeon Platinum 8252C	Intel 8252C	3.8 GHz	12	2019	Vulnerable
AMD Ryzen 3700X	AMD 3700X	3.6 GHz	8	2019	Secure
AWS Graviton2 (ARM)	Graviton2	2.5 GHz	64	2020	Secure
Intel Xeon Gold 6354	Intel 6354	3.0 GHz	18	2021	Secure
Intel Xeon Platinum 8375C	Intel 8375C	2.9 GHz	32	2021	Secure

Table 3.2 lists the CPU models on which we conducted our experiments. The selection covers the ISAs prevalent in today's data centers — Intel/AMD `x86-64` and ARM `aarch64`. For the benchmarks on ARM, we ported our privileged fastcall implementation to that architecture. Two of the systems are vulnerable to the Meltdown [Lip+20] attack and therefore require KPTI [Cor17a] to be enabled. Intel has been releasing such systems until 2019. Comparing the performance of fastcalls on vulnerable and secure systems allows us to understand the cost of privilege transitions better.

We used a Linux kernel of version 5.11 for all experiments in this section and, if needed, modified it to support fastcalls. However, to prevent the in-kernel fastcall implementation from skewing the results of vDSO, system calls, and `ioctl`, these measurements used a vanilla kernel. Moreover, simultaneous multithreading ("Hyperthreading") and dynamic overclocking ("Turbo Boost") were switched off. Any CPU frequency scaling has been disabled by selecting the “performance” CPU governor. The source code used for running the experiments described herein is available online[9].

Fastcall Performance

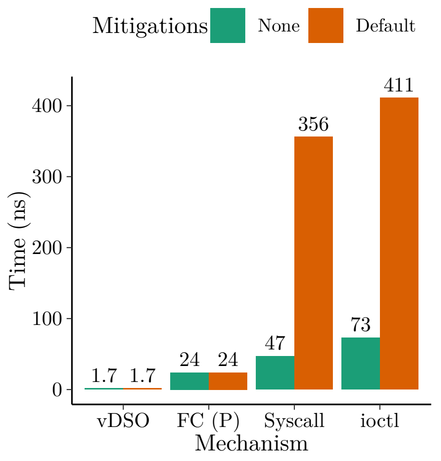


Figure 3.36: Fastcall performance on a Meltdown-vulnerable system.

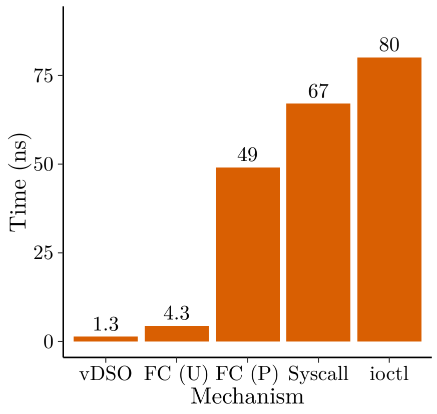


Figure 3.37: Fastcall performance on a Meltdown-secure system.

Initially, fastcalls were motivated by the need to reduce system call overhead in the presence of KPTI. Nowadays, most CPUs have hardware mitigations for Meltdown [Lip+20] and Spectre [Koc+19a] attacks, rendering KPTI unnecessary. Nevertheless, if there is a new attack of a similar nature, KPTI remains the most secure and generic way to isolate sensitive kernel data from a malicious user application. Therefore, we start the evaluation with a Meltdown-vulnerable system.

Figure 3.36 shows the latency of invoking an empty function using different mechanisms on an Intel 4790 with side-channel mitigations enabled and disabled. When mitigations are enabled, the kernel chooses only those interventions that mitigate vulnerabilities known for a specific CPU model. To gather the data, we used the Google microbenchmark support library [Fis+23], which, in particular, automatically runs as many iterations as required to achieve a statistically significant result.

The invocation latency of a vDSO function is less than 2 ns, comparable to the overhead of a function call and representing the fastest possible way to invoke a kernel-supplied function. A privileged fastcall requires 24 ns to execute, which is much faster than a dedicated system call or an `ioctl` handler, even when side-channel mitigations are disabled. Both vDSO and fastcall are not affected by enabled side-channel mitigations, as they return to the application before potential mitigations are applied. In contrast, system calls and `ioctl` handlers become much slower with the mitigations enabled. In the most optimistic case, the overhead of a privileged fastcall is 17× lower than that of an `ioctl` handler and 14.8× lower than the overhead of a system call.

When running the same benchmark on a Meltdown-secure system, the difference between fastcalls and system calls becomes less pronounced. Similar to the previous experiment,

[9]`https://github.com/planetA/fastcall-benchmark-results/tree/thesis`

Figure 3.37 shows the latency of invoking an empty function using different mechanisms on an Intel 6354 CPU. This CPU yielded identical results when mitigations were either enabled or disabled, so the figure only shows the case with enabled mitigations. The latency of a privileged fastcall is only 1.36× lower than the latency of a dedicated system call, indicating that only a few low-overhead mitigations are enabled for this system. On the other hand, by not switching privilege modes, unprivileged fastcalls are able to achieve more than 10× lower latency than privileged fastcalls and are very close to a vDSO function call.

Table 3.3: Median latency for invoking an empty function with default side channel mitigations.

CPU Model	vDSO	Fastcall Unprivileged	Fastcall Privileged	Syscall	`ioctl`
	[ns / cycles]	[ns / cycles]	[ns / cycles]	[ns / cycles]	[ns / cycles]
Intel 4790	1.7 / 6	–[11]	24 / 86	356 / 1281	411 / 1478
Intel 8252C	1.6 / 6	–[11]	35 / 132	56 / 215	97 / 368
AMD 3700X	1.4 / 5	–[11]	26 / 92	55 / 198	72 / 260
AWS Graviton2[10]	3.2 / 9	–[11]	36 / 97	95 / 258	129 / 349
Intel 6354	1.3 / 4	–[11]	49 / 147	67 / 202	80 / 239
Intel 8375C	1.7 / 5	4.3 / 13	51 / 147	87 / 251	98 / 284

Table 3.3 summarizes the average latency of empty, kernel-supplied functions across several CPUs available to us. We left mitigations at their default values. We make several observations. First, the privileged fastcall latency on Meltdown-secure systems is higher than on Meltdown-vulnerable systems. Second, even after disabling mitigations on an ARM system, the privileged fastcall latency offers a 2.5× performance improvement compared to system calls. This improvement is greater than on any Meltdown-mitigated `x86-64` systems, where the privileged fastcall latency is only 2.1× lower than the syscall latency on an AMD CPU and 1.7× lower on Intel CPUs. Compared to a dedicated system call, `ioctl` adds between 13% and 70% overhead, which might not be significant for some applications.

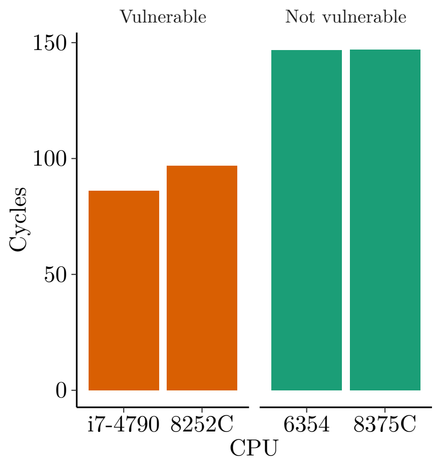


Figure 3.38: Privileged fastcalls on different Intel CPUs.

When we focus solely on Intel CPUs, curiously, we observe a degradation in fastcall latency when comparing Meltdown-vulnerable and Meltdown-secure systems. Two of the systems shown in Figure 3.38 are vulnerable to the Meltdown [Lip+20] attack, whereas the other two are not. The fastcall latency is almost double on Meltdown-secure systems compared to Meltdown-vulnerable systems. For this experiment, we specifically disabled all side-channel attack mitigations and measured the latency in CPU cycles to account for the differences in CPU base frequency among various CPU models.

Fastcalls are implemented such that the main contributor to their latency is the transition to and from privileged mode, triggered by the `syscall` and `sysret` instructions. Therefore, we conclude that the performance

[10]We disabled mitigations for this CPU.
[11]Measurements for this system are not available.

difference arises from these two instructions, which have apparently been hardened to prevent side-channel attacks on the OS kernel. Such hardening is not required for fastcalls, but unfortunately, there is no opt-out option.

As part of our research, we also attempted to use the GXF feature [Sve21] of the Apple M1 ARM-based CPU. This feature allows the setting up of executable, non-readable pages, which can be used to implement unprivileged fastcalls. To estimate their performance, we measured the overhead of the associated mode transition instruction. Our early measurements found that the round-trip latency for using the alternative privilege modes of GXF is only 69 cycles, compared to the 87 cycles required for `svc`/`eret`, ARM's equivalent of the `syscall`/`sysret` instructions. Given that unprivileged fastcalls would allow us to save only 18 cycles when transitioning to fastcall space, we did not pursue this implementation further. Such a small performance gain is most likely explained by the side-channel attack mitigations embedded in the mode transition instructions.

Our measurements show that fastcalls can be implemented with low overhead on most modern CPUs. However, the overhead of the privileged fastcalls on newer systems is higher compared to that on older, Meltdown-vulnerable systems. On the one hand, this indicates that fastcalls offer little room for performance improvement. On the other hand, it demonstrates that the performance of raw system calls is close to what is theoretically possible on modern CPUs.

Fastcall Space Control Plane

The implementation of fastcalls requires several changes to other kernel subsystems, such as memory management and the `fork` handler. Hence, we need to ensure that introducing fastcalls does not adversely affect the overall performance of the application. In this section, we examine the fastcall registration and deregistration costs, as they are the main fastcall control plane operations. We also investigate how the existence of the fastcall space influences process manipulation operations, like `fork`.

In our first experiment, we measure the registration and deregistration latency for privileged and unprivileged empty fastcalls. Through registration and deregistration operations, the application requests a kernel driver to correspondingly map and unmap a fastcall inside the process' fastcall space. The application may pass additional arguments to the kernel driver, which would allow the driver to specialize the fastcall for the application. In this specific experiment, empty fastcalls do not require any additional specialization.

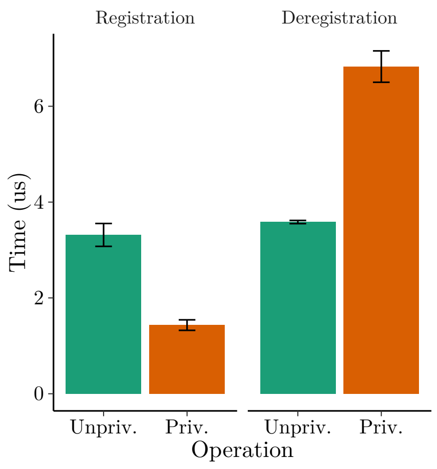


Figure 3.39: Fastcall registration and deregistration.

Figure 3.39 shows the median latency across 10 000 measurements for the registration and deregistration operations. To reduce the effect of OS noise (e. g., context switches), we filter out 1% of the slowest measurements. When registering a fastcall, the unprivileged fastcall registration latency is almost 2 µs higher than that of the privileged fastcall registration. We attribute this difference to the fact that unprivileged fastcall needs to perform more work. The unprivileged fastcall driver maps the fastcall into the fastcall space and allocates a hidden page, which is protected from user-level access. Additionally, the unprivileged fastcall needs to allocate and populate a secret page, which allows the fastcall to access the hidden page. When creating a secret page, the driver needs to generate several random numbers, which adds to the registration latency. On the other hand, the most time-consuming part of privileged fastcall registration is the mapping of the fastcall page into the fastcall space. Other operations take very little time.

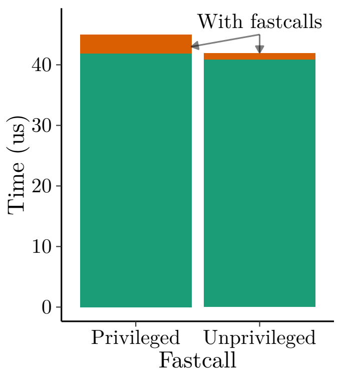


Figure 3.40: Fork operation.

Deregistration of privileged fastcalls takes 3 µs longer than deregistration of unprivileged fastcalls. This contradicts our expectations because both operations are very similar in nature. We believe that the difference is merely an implementation detail, as both operations were developed independently. However, considering that such a difference should not affect the overall application performance, we did not investigate this discrepancy further.

The second control plane operation we examine is process creation. Adding fastcall space changes the memory layout of the process, which may adversely affect `fork` performance. If a process has no fastcalls registered, the `fork` latency changes negligibly compared to a vanilla kernel. Figure 3.40 shows the impact of fastcalls on `fork` performance when a process registers 100 fastcalls. In this case, the fork latency increases by 7% and 2% for privileged and unprivileged fastcalls, respectively, because the memory mappings involved in the fastcall mechanism must be reset when spawning a new process. Note that the mitigation settings have no significant impact on `fork` performance. Overall, we conclude that fastcall control plane operations have a negligible impact on application performance.

Security of Unprivileged Fastcalls

The security of unprivileged fastcalls relies on two assumptions. First, it is assumed that accessing the data in the secret page from user space is impossible. Second, the secret address must be sufficiently difficult for an attacker to guess. We begin our discussion by evaluating the second assumption first.

To estimate whether an attacker can guess the secret address, we need to know the size of the key, how many guesses the attacker can make, and how long it takes to check a guess. The key's length is determined by the hardware architecture, while the other two parameters are dependent on the specific system design. For further analysis, we will present several techniques an OS can employ to make the attack more challenging.

The key protecting the location of the hidden page is its 64-bit virtual address. Unfortunately, not all 64 bits of the address are used as part of the key. First, the minimum size of the hidden region is a 4 KiB page, so the attacker would need to guess only a page-granular address, instead of a byte-granular one. Therefore, the lowest 12 bits of the address are not part of the secret. Similarly, some of the most significant bits of the address are also unused. For example, on an `x86_64` CPU with 5-level paging, the top 7 bits must have the same value as the 57th bit. Additionally, we map the fastcalls only in the user space, losing another bit, because in Linux the user space is limited to the lower half of the virtual address space. Accounting for all the aforementioned parts of the virtual address, the effective key size is reduced to 44 bits. In this case, if k is the key size, the probability of a single successful guess $P(1)$ is

$$P(1) = 1 - \frac{2^k - 1}{2^k} = \frac{1}{2^k}$$

Now, consider the scenario where the attacker makes n guesses in succession. If the attacker can assume that the secret address remains unchanged between the guesses, then the probability of making at least one successful guess $P_s(n)$ out of n independent guesses is:

$$P_s(n) = 1 - \frac{2^k - 1}{2^k} \cdot \ldots \cdot \frac{2^k - n}{2^k - n + 1} = 1 - \frac{2^k - n}{2^k} = \frac{n}{2^k}$$

The OS could make the attack more challenging by periodically changing the secret address, forcing the attacker to make repeated guesses. It is unlikely that the OS can remap the hidden page after every potential guess, as this would be too resource-intensive. However, we can estimate a lower bound for such a probability. Then, the probability of making at least one successful guess out of n independent guesses $P_i(n)$ becomes:

$$1 - \left(\frac{2^k - 1}{2^k}\right)^n \leq P_i(n) < P_s(n)$$

To understand how secure these probabilities can be in practice, consider the following possible attack. The attacker process sets up a signal handler for the `SIGSEGV` signal, which is triggered when the process tries to access an unmapped memory location. Then, the attacker probes random addresses in the fastcall region as long as the signal handler is triggered. A successful guess means that the attacker has found a valid hidden region. The OS could protect against such an attack by killing the process after access to the fastcall region because the user space should never access the unmapped part of the fastcall space.

The attacker could potentially work around this protection by employing a fork-bomb [Ray03]. The attacker process forks itself, and the child process tries to access the fastcall region, while the parent process waits for the child to terminate. The OS could protect against this attack by introducing a small delay into the creation of the fastcall space, so that each fork operation takes longer. This way, the attack can be slowed down to the point where it becomes too long to be practical.

Considering that these measures do not completely prevent the attack, there remains a possibility of a successful guess by the attacker. In other words, the attacker can calculate the amount of time to reach certain target probabilities of a successful attack. The longer the attacker tries to guess the secret address, the higher the probability of a successful attack becomes. On the other hand, the OS can also estimate the time required to reach a target probability by the attacker and use this estimation for further mitigations. Figure 3.41 shows the time required to reach a certain target probability of a successful attack, depending on how long a single guessing attempt takes.

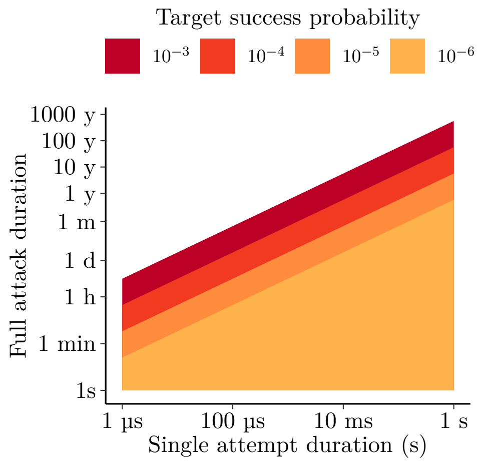


Figure 3.41: Attack on unprivileged fastcall.

For example, assume a situation where the OS wants to ensure that the attacker has less than a 10^{-6} probability of guessing the secret address during an attack. Then, if the attacker can make one guess per 1 s, the attacker will reach the target probability roughly after a month of continuous trying. If the attacks can happen more often, the attacker can reach the target probability in minutes or even seconds. Therefore, the OS should ensure as low a rate of attack attempts as possible. If the OS regularly remaps the hidden region, the attacker can exploit access to the hidden region only for a limited time. Combined with OS-level suspicious activity monitoring, the above-mentioned setup might be an acceptable trade-off.

Now consider our first assumption, where the attacker cannot leak the key from the secret page. In our current implementation, this is unfortunately not the case, because unprivileged fastcall space is vulnerable to Spectre-like attacks even in modern CPUs. We believe that this limitation can be overcome by using a different mechanism for storing secrets, such as Model Specific Registers (MSRs), which are not subject to speculative execution.

3.3.5 Summary

The fastcall space is a new concept that allows the OS to provide user applications with very low-overhead access to privileged operations. Unlike traditional kernel-level device drivers, fastcalls are not designed to provide full driver functionality. Instead, a fastcall is a small, specialized function that can be called by a user-level driver to give the OS an opportunity to enforce specific conduct on behalf of the application. We did not follow up on the fastcall architecture with an implementation of a full-fledged use case, like we did for CoRD (see Section 3.2.4), because it turned out that for the current CPU architectures, privileged fastcalls offer only a marginal performance improvement over system calls.

The main contribution of the fastcall architecture we have demonstrated so far is that it provides a lower bound estimate for the system call overhead. We found that existing privilege-switching mechanisms are designed purely for security and do not attempt to minimize mode transition overhead. Of course, we do not propose building less secure systems. Instead, what we believe our research shows is that there is a need for more than two privilege levels in the CPU to accommodate the needs of modern high-performance applications.

In contrast to the old privilege levels of the `x86-32` architecture [Int22], the new privilege levels should differ not only in the level of access to memory and instructions but also in the level of side-channel protection. Then, the OS can make a trade-off between security against side-channel attacks and the performance of system calls.

Historically, privilege-switching mechanisms did not have stable performance characteristics, and their performance degraded over time. In our opinion, this happened because there was no clear motivation for maintaining such high performance. The fastcall architecture provides motivation for having fast mode switching and the basic framework on how to use it.

3.4 Summary

For RDMA networks to become more widespread in the Cloud environment, their design priorities must prioritize sharing and multi-tenancy. These properties are hard to achieve without active and dynamic OS-level involvement to enforce security and resource management policies. Therefore, the OS must be able to have full control over application communication. In this chapter, we present several software techniques for OS-level interception of traditional RDMA network dataplane. We show that despite such interception, the end-to-end overhead for the user applications is negligible.

In fact, depending on the exact use case, the OS may choose which interception mechanism to use. In Section 3.2, we presented CoRD, which builds upon the existing RDMA network dataplane. CoRD intercepts the network traffic at the NIC driver level and allows for manipulating it in a flexible way. CoRD is straightforward to implement and easy to use. Despite up to 1.5 µs overhead per message, we show that CoRD's overhead is negligible, even for applications known for being latency-sensitive.

If the CoRD overhead is still too large for some applications, we propose the use of fastcalls, which are presented in Section 3.3. Fastcalls can easily reduce the per-message overhead to less than 100 ns, but have a more complex programming model. We pushed the fastcall concept to the extreme with unprivileged fastcalls, which allow for providing privileged operations to user applications without an expensive privilege mode switch. Ultimately, unprivileged fastcalls are almost as fast as function calls.

Both privileged and unprivileged fastcalls require certain compromises regarding security assumptions and cannot be used as drop-in replacements for traditional system calls. On the other hand, fastcalls demonstrate the need for further development of CPU architectures, specifically, the need for new privileged transition mechanisms. Such new modes could be used to provide user applications with privileged operations, but have a thinner isolation layer compared to the kernel mode. For example, Roitzsch et al. proposed hardware modifications, enabling software-defined CPU modes, which could be used to tune the exact behavior of the CPU mode switch [Roi+23]. Then, a mode switch could be configured to tune the isolation level suitable for a specific use case.

We propose an alternative to the seemingly inevitable trend of offloading every possible operation to the NIC. Naturally, the more functions are offloaded to the NIC, the more complex, expensive, and slow the NIC becomes. Therefore, additional offloading risks moving past the point of diminishing returns, especially because the less loaded the CPUs are, the more likely the network is to become a bottleneck.

Instead, we aim to keep the door open for OS-level continuous dataplane interposition, independent of the exact interception mechanism. Continuous interposition can be used for on-demand resource allocation, network virtualization, resource sharing, and other use cases. All these use cases are not intended to eliminate CPU offloading but rather to complement it.

Statement of Contributions

The work presented in this chapter is based on the following publications:

- Maksym Planeta, Jan Bierbaum, Michael Roitzsch, and Hermann Härtig. *CoRD: Converged RDMA Dataplane for High-Performance Clouds*. Sept. 2, 2023. DOI: `10.48550/arXiv.2309.00898`. arXiv: `2309.00898 [cs]`. preprint
- Ilya Meignan–Masson. “Bridging the Performance Gap Between Converged RDMA Dataplane and Kernel-Bypass”. MA thesis. 2023
- Till Miemietz, Maksym Planeta, Viktor Reusch, Jan Bierbaum, Michael Roitzsch, and Hermann Härtig. “Fast Privileged Function Calls”. In: *Systems for Post-Moore Architectures*. The 11th Workshop on Systems for Post-Moore Architectures. Rennes, France, 2022. URL: `https://www.barkhauseninstitut.org/fileadmin/user_upload/Publikationen/2022/202204_Miemietz_SPMA_FastCalls.pdf`
- Yaoxin Jing. “Mechanisms for Fast System Calls”. Studienarbeit. Dresden, Germany: TU Dresden, Oct. 2021

Some of the text in this chapter has being taken verbatim from the publications listed above. The author of this thesis contributed to the design, implementation, and evaluation of the CoRD [Pla+23; Mei23] and fastcall mechanisms [Mie+22; Jin21]. The author also contributed to the writing of the publications. The author also conducted the security analysis of the unprivileged fastcall mechanism.

4 Intermittent Interposition

For RDMA applications, any form of communication interception is typically undesirable due to the additional latency and disruption it can introduce. In this context, intermittent control strikes a balance between the need for peak performance in latency-sensitive applications and the necessity for the operating system to control its applications' communication. Exerted only when necessary, intermittent control offers an efficient solution to avoid the additional overhead associated with constant control. This chapter generalizes the basic principles of intermittent RDMA dataplane interposition and examines its specific applications, which can enable, for example, the transparent live migration of containerized applications.

Intermittent interposition divides the connection state into two distinct phases: bypass and manipulation (see Figure 4.1). In the *bypass phase*, the application uses the network without any interruption or added overhead, ensuring optimal performance. In the *manipulation phase*, the connection state is under full control of the OS. The OS *intercepts* the connection to move it into the manipulation phase, or it can configure the NIC to intercept the connection when necessary. As soon as the manipulation phase is over, the connection *resumes* normal operation, returning to the bypass phase.

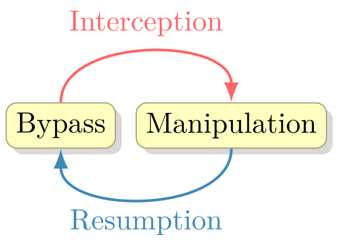


Figure 4.1: Phases of intermittent interposition

Implementing and effectively using intermittent interposition comprises three major steps. The first step is *connection interception*, which ensures a smooth transition between the bypass and manipulation phases without hampering performance. The second step is *connection manipulation*, where the OS changes the state of the intercepted connection. Finally, *connection resumption* moves the connection into the bypass phase and notifies the remote communication partners about the change.

The contributions of this chapter are as follows: Generalized interposition architecture described in Section 4.1; Design and implementation of a pause-resume protocol (see Section 4.3), extending the RoCE protocol with the ability to intercept and resume RDMA connections; Design and implementation of MigrOS (see Section 4.4), enabling transparent live migration for containerized RDMA applications; And, finally, a discussion of the inherent limitations of intermittent interposition with the example of a virtual RDMA network in Section 4.5.

Additionally, Section 4.2 discusses the general applicability of intermittent control with the example of known use cases.

4.1 Architecture

While being the key operations in intermittent interposition, connection interception and resumption have the same general structure for existing use cases. These operations enable a smooth transition between the high-performance bypass phase and the controllable manipulation phase, and back. From the design perspective, this transition must create minimal disruption to maintain optimum application performance. In this section, we describe our generalization of the connection interposition architecture.

In contrast to interception and resumption operations, connection manipulation is use-case specific. The manipulation phase may either halt communication completely or reroute it over a slower path that is under the operating system's full control. In both cases, the manipulation phase must be short and infrequent to minimize disruption to the application. Moreover, in the manipulation phase, the OS may need access to connection information stored within the device, potentially requiring certain use-case-specific device modifications.

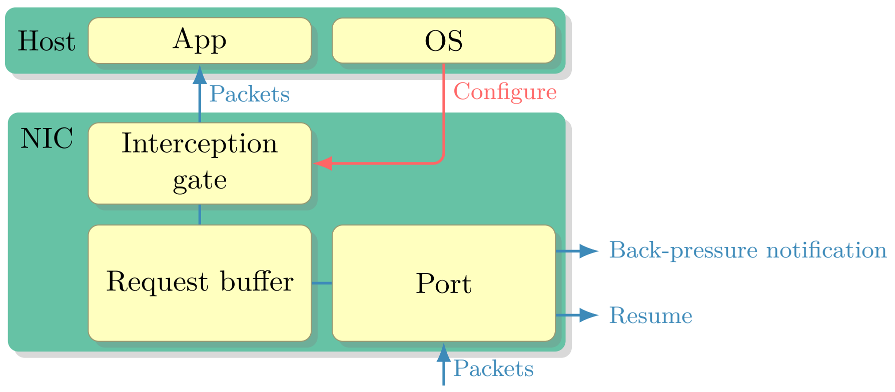


Figure 4.2: Key components of intermittent interposition architecture.

The process of intercepting and resuming the connection consists of multiple steps (see Figure 4.2). It begins with the activation of an *interception gate*, a hardware component that suspends send and receive operations on the intercepted connection transparently for the user applications. Subsequently, new in-flight messages are unable to reach the application, and the application is prevented from sending any new messages until the OS concludes the manipulation phase. The interception gate is configured by the OS to intercept the connection either immediately or when certain conditions are met. Once the interception gate is activated, the connection moves into the manipulation phase.

During the manipulation phase, the NIC may receive new messages intended for the manipulated connection. If possible, the in-flight messages should not be dropped because retransmission may be costly, delaying the resumption of communication. Therefore, the NIC needs to have a *request buffer* to store incoming messages temporarily. After the OS completes the manipulation phase, the NIC can replay the messages from the request buffer to the application.

To reduce network noise and prevent request buffer overflow, the manipulated connection can send *back-pressure* notifications in response to newly arriving messages. Back-pressure notifications signal the sender's NIC to temporarily halt communication with the suspended connection. Additionally, this notification ensures that the sender does not transition into an error state due to perceived connection issues. Once the connection transitions back to the bypass phase, the OS dispatches *resume notifications* to inform remote parties about resumed communication.

The exact implementation of these connection interception and resumption components depends on the use case and the underlying hardware. The remainder of this section describes each of these components in more detail. Section 4.2 discusses how this architecture enables several real-world use cases. Section 4.3 describes our specific implementation of the connection interception and resumption protocol.

4.1.1 Interception Gate

An interception gate prevents communication over a specific connection based on a condition configured by the OS. The interception gate can be either part of the NIC (as Figure 4.2 shows) or part of the host system. In the latter case, some IOMMUs can serve as interception gates. In this section, we discuss both scenarios.

Depending on the use case, the interception operation can be initiated by the OS or by the NIC. For instance, in the case of full virtual memory support, the OS configures a specific component in InfiniBand NICs known as Memory Translation Table (MTT) [Mel16] (see Figure 4.3). MTT is a part of the NIC SoC and is responsible for translating virtual addresses to physical addresses, similarly to how a Memory Management Unit (MMU) in the host system translates host virtual memory addresses to physical. In a situation where the mapping for a specific virtual address does not exist, the NIC intercepts the communication and sends a page fault interrupt to the OS. The OS then resolves the page fault, updates the configuration of MTT, and resumes communication.

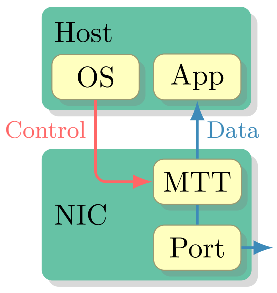


Figure 4.3: Interception gate in the MTT of the NIC

During live migration, the application state should not change until the entire application has been moved to another compute node. To guarantee this constraint, the NIC should not inadvertently deliver new data into the RDMA application. When starting live migration, the OS requests the NIC to intercept all connections belonging to the migrating application. So, instead of configuring an interception condition at the NIC, the OS simply requests the NIC to intercept a set of connections unconditionally.

The host system can also be responsible for connection interception by employing the IOMMU as an interception gate (see Figure 4.4). The IOMMU essentially shares the same role as the MTT within a NIC's SoC, as they both translate from virtual to physical addresses and back [Mel16]. In the event of a missing corresponding mapping, the IOMMU triggers a page fault interrupt, which is then handled by the OS through the allocation of a new physical page and the creation of a new mapping.

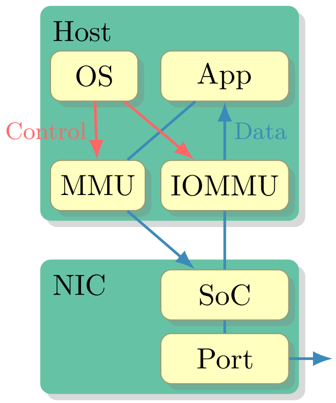


Figure 4.4: IOMMU as an interception gate

Unfortunately, not all IOMMUs support this workflow, so an IOMMU page fault may become an irrecoverable error for the NIC. To recover from the page fault transparently from the application, the IOMMU needs to support the Page Request Service (PRS) feature on Intel CPUs [Int23] and Peripheral Page Request (PPR) on AMD [AMD21]. These features implement PCIe Address Translation Service (ATS) and Page Request Interface (PRI) extensions that enable recoverable page faults for PCIe devices [PCI19].

In contrast to a device and vendor-specific SoC-based approach, the IOMMU offers a more portable interception gate. Moreover, IOMMU-based interception is easier to map to devices other than RDMA NICs. However, it is important to note that utilizing the IOMMU might introduce overhead to system performance [PCI19, Section 10.1.2], as ATS complicates memory coherency protocols. Additionally, the address now needs to undergo translation not once (at the MTT), but twice (at both the MTT and the IOMMU).

In summary, both the IOMMU and SoC provide a similar interception gate mechanism. They both shield the application from changes in the connection state, enabling the OS to manipulate the application or connection state. On the other hand, the activation of an interception gate can lead to dropped messages, which must not result in the failure of user applications. Therefore, while the interception gate is necessary for intermittent control, it is not a sufficient condition on its own.

4.1.2 Request Buffer

During the connection manipulation phase, a NIC may receive new messages intended for the suspended connections. One way to handle these messages and reduce network disruption is to store them temporarily in a buffer for later processing. Once the connection resumes normal operation during the bypass phase, the buffered messages can speed up communication recovery.

The request buffer might store either the entire messages or merely the metadata, thereby conserving the resources required for intermittent control. For instance, to improve virtual memory support, Lesokhin et al. [Les+17] chose not to store entire messages, as doing so would necessitate approximately 125 MiB of buffer space per NIC port. Instead, they changed the NIC to retain only the address translation requests, which the OS can then process in batches. After resolving the page faults, the NIC depends on the retransmission protocol to recover the missing messages.

Psistakis et al. [Psi+22] utilized the IOMMU to implement request buffer functionality in the form of a Page Request Queue (PRQ). The PRQ is a buffer in the host memory, allocated by the OS, where the IOMMU stores page translation requests arising from page faults [Int23]. In other respects, an IOMMU-based request buffer parallels its SoC-based counterpart.

Certain RDMA communication protocols, like UD, lack a retransmission protocol to resume communication. To address this, Lesokhin et al. modified the NIC to store complete incoming messages in a backup buffer located in pinned host memory. After a page fault is resolved, the OS moves messages from the backup buffer to the original memory destination. Despite the additional resource demands, this approach still does not guarantee that the NIC will not drop packets, because the OS may take so much time that the backup buffer still overflows.

The inability to store all incoming messages in the request buffer is an inherent limitation, as the OS intrinsically cannot process page faults at the same rate the NIC receives messages. In the context of live migration, we did not even use a request buffer (see Section 4.4), because it did not offer a substantial reduction in the message loss rate. The live migration process lasts much longer than, for example, page fault handling, so that message dropping is inevitable. Furthermore, the presence of a request buffer merely increases the state data that needs to be transferred to the destination host [Pla+21; Han+21]. Hence, we regard the request buffer as an optional component of connection interception.

4.1.3 Back-Pressure Notification

During the connection manipulation phase, new incoming messages must either be temporarily stored in the request buffer (see Section 4.1.2) or dropped. Given that the manipulation phase is relatively slow compared to the bypass phase, it is ultimately not feasible to store all unprocessed in-flight messages. Otherwise, the request buffer would become impractical. However, RDMA protocols, which are designed for lossless networks, struggle to handle large volumes of dropped messages effectively. In the worst case, the sender may interpret the dropped messages as a sign of network failure and terminate the connection, while the receiver still remains in the manipulation phase. Therefore, it is important to minimize the message drop rate.

Simply dropping messages that cannot be processed during the connection manipulation phase is an unsophisticated and insufficient approach for several reasons. Firstly, if the sender on the remote end remains uninformed about the paused state of the connection, they will continue to dispatch more messages. This exacerbates the load on the host OS, with a compounding number of entries being added to the request buffer. Secondly, this uninformed communication results in unnecessary network traffic, which is destined to be dropped. Lastly, the sender may misinterpret the dropped messages as an indication of a network failure, and consequently, flag an unrecoverable error to the application.

To overcome this problem, current connection interception methods [Les+17; Psi+22; Pla+21] cleverly repurpose the existing Negative Acknowledgement (NACK) mechanism. Upon the generation of a page fault interrupt by the NIC, a NACK of type Receiver Not Ready (RNR) is dispatched to the sender. This NACK informs the sender to pause sending further messages either for a predefined period [Psi+22] or until further notice [Pla+21], thereby reducing the likelihood of message drop occurrences. Although at a surface level back-pressure notification appears to only reduce the number of dropped messages, it is also crucial to maintain failure-free execution of user applications.

4.1.4 Resume Notification

Upon completion of the connection manipulation phase, the OS may decide to notify the sender to resume communication [Psi+22]. This proactive approach accelerates the restoration of communication, making it more efficient than simply waiting for a Receiver Not Ready (RNR) Negative Acknowledgement (NACK) timeout to expire. Additionally, it relieves the OS from the burden of setting an optimal RNR timeout.

A user application may also set the timeout to infinity, making sending a resume notification mandatory [Pla+21]. Another reason for having an explicit resume notification is to inform the remote NIC about additional information required to continue the communication, such as the updated location of a migrated application.

4.1.5 Connection Manipulation

The connection manipulation phase begins after intercepting a connection. During this phase, the OS changes the connection in a consistent manner, as it has exclusive access to its state. Here, the OS holds the power to retrieve or modify the connection state or even replace it with another.

Nonetheless, the manipulation capabilities may be limited due to the inaccessibility of certain connection state data from the host. This limitation can be mitigated by modifying the hardware or firmware of the NIC to expose the necessary state. Often, elements of the connection state are not initially present due to a lack of use cases, and these can be made available with minor hardware modifications. Alternatively, if the NIC incorporates a programmable component, such as an FPGA, modifications to the RDMA NIC can be much simpler [Psi+22].

The range of possible connection manipulations is not boundless. While the OS can intercept and process incoming messages during the manipulation phase, processing them at line rate is not feasible. Thus, the manipulation phase primarily offers a snapshot of the connection state, and it does not equip the OS with the ability to maintain communication on behalf of the application. As a result, most approaches aim to complete the connection manipulation swiftly to resume bypass communication as quickly as possible.

In the context of full virtual memory support, the connection state that the OS needs to manipulate involves the mapping between virtual and physical addresses of RDMA message buffers. When handling page faults, the OS assigns a free physical memory page to the respective RDMA buffer. Subsequently, the OS configures the MMU, MTT of the NIC, and IOMMU to make the mapping accessible to the user application, the NIC, and to translate between device virtual and host physical addresses, respectively. This comprehensive procedure constitutes the connection manipulation phase.

The process of connection manipulation becomes more intricate in the case of live migration. The primary goal for the OS here is to preserve the connection state and later reproduce it in a manner that allows seamless communication, without notifying the remote communication partners of any alterations. To accomplish this, the OS must extract all necessary connection state information from the NIC and mask any changes induced by the migration from the user application.

The connection state encompasses the application-visible state, OS-visible state, and hardware-internal state (see Figure 4.5). For instance, the application-visible state includes messages that have already been written into the receive buffers of the application. It is crucial that changes in physical resources do not affect the application-visible state. Some connection state attributes, such as the destination address of the remote connection, can only be retrieved by the application through queries to the OS. Changes in these attributes are more easily concealed from the application. The hardware-internal state includes those aspects that are typically inaccessible to the OS, like the state of the retransmission engine. If a use case necessitates access to the hardware-internal state, the NIC must be adapted.

Additionally, some of the connection state is unique to a specific device where the connection is established. In InfiniBand networks, such state usually consists of physical identifiers in the network, like the GUID of a device and physical ports. For instance, after live migration, the OS must ensure that the migrated application does not detect changes in these physical identifiers, otherwise, the application may enter an undefined state.

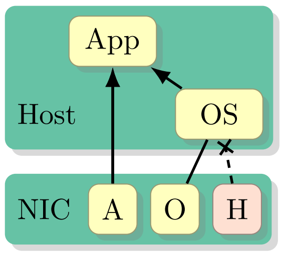


Figure 4.5: Connection state can be directly accessible by the application (A), accessible through the OS (OS), and application-inaccessible hardware internal state (H).

In summary, connection manipulation is a crucial step in intermittent control. During this phase, the OS has exclusive access to the connection state, allowing it to modify the state in a controlled manner. The specific processes within this phase can vary significantly depending on the use case requirements. The primary challenge is to adjust the state in such a way that neither the application nor the remote communication partners perceive any changes. Section 4.4 details how connection manipulation can facilitate live migration. Conversely, Section 4.5 discusses a use case of a virtual RDMA network where connection manipulation alone is not sufficient.

4.2 Use Cases

The two primary use cases for intermittent control are full virtual memory support and live migration. In both scenarios, the OS needs to manipulate the connection state, but the motivation and the exact operations differ. In the case of live migration, the OS must save and restore the complete state of a connection. In the case of virtual memory support, the OS needs to update the memory map so that the NIC can access the application's memory. This section elaborates on both use cases in more detail.

Virtual Memory Support

Simple virtual memory support has been available in RDMA networks since their inception [Inf15]. When an application sends or receives a message over an RDMA network, it must provide the NIC with an address to the corresponding memory buffers in the virtual address space of the application. As such, the NIC must maintain a virtual-to-physical address mapping for each user application. On the other hand, if the mapping is not present when the NIC accesses the application's memory, the NIC cannot continue communication on the connection where a page fault occurred. Additionally, the NIC cannot afford to wait for the OS to handle the page fault because more messages may arrive over the same connection, and the NIC cannot buffer them. Therefore, traditional OS-level on-demand page fault handling is not feasible for RDMA networks. Instead, to prevent this potentially irrecoverable error, the application must pin the memory buffers accessible by the NIC, so no faults occur.

The requirement to pin all the memory accessible by the NIC disables many virtual memory features, such as memory overcommitment, memory swapping, copy-on-write, or transparent huge pages support [Les+17]. Some RDMA NICs and the underlying communication protocol have been updated to enable efficient support for page faults caused by missing memory mappings [Les+17; Lis13]. With this support, the host OS has time to resolve a page fault without the underlying network connection being considered irrecoverably broken. For user-level RDMA applications, Liss [Lis13] implemented a feature, known as On-Demand Paging (ODP), which allows the applications to avoid pinning NIC-accessible memory.

In terms used by this thesis, ODP employs intermittent interposition to enable full virtual memory support. First, the NIC generates a page fault and moves into the manipulation phase. Then, the OS manipulates the connection state by creating a new memory mapping to resolve the page fault. Finally, the OS returns to the bypass phase, resuming communication on the connection.

Live Migration

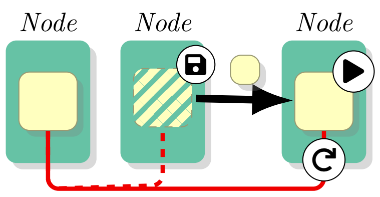


Figure 4.6: Live migration consists of saving the application state (💾), restoring it at the destination node (▶), and reconfiguring the network (↻).

The objective of live migration is to transfer a running application from one host to another without disrupting the application. Furthermore, the migration should be transparent to both the application and its communication partners. The process of live migration (see Figure 4.6) includes saving the state of the application, restoring it on the destination host, and reconfiguring the network, enabling the communication partners of the application to locate it on the destination host. Intermittent interposition grants the OS the ability to live migrate applications without compromising normal communication performance.

Intermittent interposition aids the OS in pausing communication without terminating it. This characteristic is vital, as it renders live migration transparent to RDMA applications. When the connection is in the manipulation phase, the OS can capture the state of the connection for saving and later restoring it on the destination host. Additionally, the OS can manipulate the state of the connection to refresh the network configuration, ensuring that the communication partners of the application can locate it on the destination host. After the migration concludes, the connections of the migrated application can revert to the bypass phase, resuming communication.

Summary

In summary, connection interposition is a sophisticated process that requires meticulous coordination between the OS and the NIC. Triggered by the interception gate, this process facilitates the transition from the bypass to the connection manipulation phase. During the manipulation phase, a request buffer temporarily holds incoming messages, while a back-pressure notification mechanism assists in managing the communication flow, preventing network disruptions, and safeguarding against possible application failures. Following the completion of the manipulation phase, a resume notification is sent, accelerating the restoration of communication. Overall, the connection interception process strives to concurrently minimize the impact on application performance and the resource demands within the system architecture.

4.3 Pause-Resume Protocol

This section introduces a pause-resume protocol, which augments the RoCE RDMA communication protocol and enables intermittent control over RDMA connections. The pause-resume protocol encompasses a back-pressure notification mechanism and a resume notification. Collectively, these mechanisms empower the OS to temporarily halt network communication on specific connections. The pause-resume protocol has been showcased for transparent live migration of RDMA applications (see Section 4.4), but its utility extends to other use cases as well. Notably, analogous protocols have been utilized for enhancing virtual memory support for RDMA NICs [Psi+22] and for the live migration of virtual machines in VMWare vMotion [Han+21].

An RDMA connection is represented by a Queue Pair (QP). Consequently, the pause-resume protocol is integrated into the QP state machine (see Section 4.3.1). The pause-resume protocol introduces two new QP states: *Paused* (P) and *Stopped* (S), which influence the NIC's transmission protocol (see Section 4.3.2). A specific implementation of the pause-resume protocol is detailed in Section 4.3.4.

InfiniBand networks support multiple transport types [1], but this discussion primarily focuses on RC connections, as they, along with UD, constitute the most prevalent types of connections. Differing from UD, RC connections, akin to other connection-oriented RDMA transport types (e.g., UC and XRC), incorporate a retransmission protocol, which forms the basis of our pause-resume protocol. The adaptation of the pause-resume protocol for datagram-oriented transport types, like UD, is addressed in Section 4.6.

4.3.1 Queue Pair States

The QP state machine (see Section 2.4) governs the progression of a connection through various states, each of which represents a distinct phase of the connection's lifecycle. To safeguard against these undesired changes during the manipulation phase, we introduce two new QP states, invisible to the user application (see Figure 4.7): *Stopped* (S) and *Paused* (P). For instance, when the OS is preparing for the live migration of an RDMA application, the OS initiates the transition of the application's QPs into the *Stopped* state. In the event of a page fault, the NIC itself moves the QP into the *Stopped* state. A stopped QP neither receives nor sends any messages, except for back-pressure notifications, and remains stopped until either the OS destroys it or transitions it back into the previous state.

[1]These include RC, UD, RD, UC, and XRC.

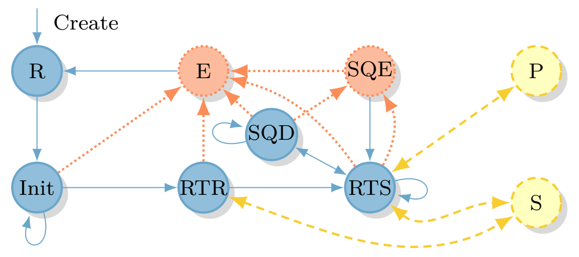


Figure 4.7: QP State Diagram. Normal states and state transitions (●, →) are controlled by the user application. A QP is put into error states (●, ⇢) either by the OS or the NIC. New states (●, ⇢) are used for intermittent connection control.

An active QP transitions into the *Paused* state after receiving a back-pressure notification from a stopped QP, acknowledging that its remote destination has been stopped. A paused QP neither sends nor receives messages, as its sole communication partner is stopped. A QP remains paused until it receives a resume notification, either from the original QP which has returned to the RTS state or from a new QP at a new location. The latter scenario occurs during live migration, necessitating that the paused QP also records the new address of the restored QP. After receiving the resume notification, the paused QP reverts to the RTS state, and communication can resume. For the prototype implementation detailed in Section 4.3.4, we permit transitions to the *Stopped* or *Paused* states exclusively from the RTS state.

4.3.2 Pause and Resume Notifications

When a connection is in a manipulation phase, the remote communication partner must not confuse paused communication with a network failure. Moreover, in the case of live migration, the partner of the migrated connection must learn its new physical address. For this, we implement back-pressure and resume notifications at the level of the RoCE protocol.

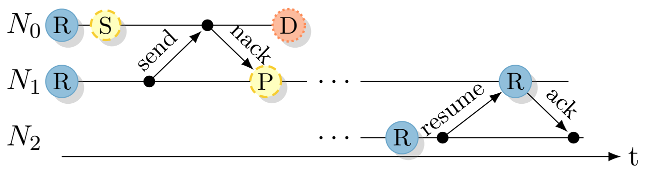


Figure 4.8: To migrate from host N_0 to host N_2, the state of the QP changes from RTS (R) to Stopped (S). Finally, the QP is destroyed (D). If the partner QP at host N_1 sends a message during migration, this QP gets paused (P). Both QPs resume normal operation once the migration is complete.

Figure 4.8 illustrates how the new notifications help to transparently migrate a connection from one compute node (N_0) to another (N_2). As soon as the OS decides it is time to make a checkpoint of an application, it moves all of the application's QPs into the *Stopped* state, thereby initiating the manipulation phase. While the OS is saving the application state, the NIC drops all messages coming to the stopped QPs and instead replies with a new negative acknowledgement type, `NAK_STOPPED`. This acknowledgement is passed in the ACK Extended Transport Header (AETH) along with the sequence number of the last packet completed before the QP transitioned to the paused state. This protocol is part of the low-level packet transmission protocol and is typically implemented entirely within the NIC. When the partner QP receives this negative acknowledgement, it transitions to the *Paused* (P) state and refrains from sending further packets until receiving a message of the new *resume* type.

After migration is complete, the new host of the migrated process restores all QPs to their original state and sends resume messages. Resume messages are sent unconditionally, even if the partner QP has not been paused previously. For the *resume* notification, we defined a new opcode in the Base Transport Header (BTH), a RoCE header. Any recipient of a resume message updates its QP's destination address to the source address of the resume message, i.e., to the new location of the migrated QP.

Each pause and resume message carries source and destination information. Thus, if multiple QPs migrate simultaneously, there can be no confusion about which QPs must be paused or resumed. However, if a source and destination QPs migrate concurrently, their corresponding resume messages may contain old and outdated addresses of their communication partners. If at any point the migration process fails, the paused QPs will remain stuck and will not resume communication. This scenario is not unique to RDMA connection migration and is completely analogous to a failure during TCP connection migration. In both cases, the host OS is responsible for coordinating the migration process and cleaning up the resources.

When handing over a connection to a new host, there is also a possibility that the remote communication partner sends a message to the old location of the migrated QP. If the QP at the old location has already been destroyed, the NIC will silently drop the message. InfiniBand networks allow the user to configure a QP in a way that any missing positive or negative acknowledgement will be interpreted as a network failure after a timeout. This way a user application may become aware of ongoing live migration.

It is possible to avoid this issue by destroying the old QPs only after sending a resume message from the new location. Alternatively, negative acknowledgements can be sent preemptively from the old location. However, in the prototype for our case study (Section 4.4), to simplify the implementation we neither synchronized QP destruction nor preemptively sent acknowledgements. Such simplification allows communication to resume faster and in most cases does not compromise correctness. Nonetheless, interpreting a single packet loss as a network failure is not recommended for production systems anyway [Bar13].

The pause-resume protocol is much simpler in the case of full virtual memory support because in this scenario the QP does not change its physical location. After a page fault, the NIC puts the QP into the *Stopped* state and sends a negative acknowledgement to the remote partner. After the OS resolves the page fault, it sends a resume message to the remote partner, and communication can resume.

4.3.3 Transmission Engine Changes

The pause-resume protocol must be implemented in an RDMA NIC because most packet-level RoCE protocol implementations are executed in hardware. A typical RDMA NIC independently performs all packet-level operations, including transmitting packets and acknowledgements. As a result, the OS is not able to independently compose and process arbitrary packets, including those introduced by the pause-resume protocol. An RDMA NIC running the RoCE protocol is responsible for adding and removing packet headers. These headers are not exposed at the software level unless the full protocol is implemented in software.

Therefore, to enable the pause-resume protocol, modifications to RDMA NICs are necessary. The NIC appends different packet headers based on the service (RC, UD, etc.) and the type of message (RDMA read, send, etc.). Our modifications only affect two headers: BTH and AETH. The BTH is the initial RDMA-specific header, coming right after the IP and UDP headers. Additionally, the NIC needs to manage two new flags for each QP to indicate pause and resume states. The NIC must be capable of transitioning a QP into the *Stopped* or *Paused* state, processing a send request with a resume message issued by the OS, and managing pause and resume packets.

Our changes affect three existing workflows in the underlying RoCE protocol: 1. Processing a send request WQE (Figure 4.9), 2. receiving a packet (Figure 4.10), and 3. receiving an acknowledgement (Figure 4.11). The NIC must account for the two new states of the QP in all these workflows. This alteration of logic is minor and does not modify the packet layout, as it only pertains to the internal state of the QP. The workflows mentioned above are activated when the user triggers the NIC through a doorbell register, when a message arrives, or due to a timeout.

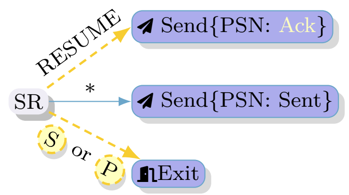


Figure 4.9: A QP in the *Stopped* (P) or *Paused* (S) state does not process requests and goes to the idle state (). A resume message carries the PSN from the last acknowledgement. "*" represents "else" branch.

The NIC processes a send request when the user application posts a new Work Queue Entry (WQE) (Figure 4.9) to the QP. For this, the application's user-level RDMA driver adds a new entry to the send queue and triggers the NIC by writing to the doorbell register. Under normal circumstances, the NIC handles the send request by dispatching a series of packets to the remote QP. One of the fields the NIC sets for the outgoing packets is the PSN field, which is used to maintain the sequence of the packets.

We suggest the following modifications to the processing of send requests. First, the NIC should either construct the new resume message or allow the OS to do so. Unlike a typical message, the resume message derives the PSN from the last acknowledged message rather than the last sent message. A receiver identifies the resume message by a new *opcode* in the BTH header. Introducing a new message type does not necessitate changes in the message structure thanks to the availability of unused opcodes. Second, if the QP is in the *Stopped* or *Paused* state, the NIC should not process the send request and should instead enter the idle state.

The rationale for incorporating the resume message into the BTH header is that the retransmission protocol is executed by RoCE at the BTH packet level. This integration enables us to make resume messages reliable with minimal additional effort.

Similarly, the back-pressure notification, transmitted as a new negative acknowledgement of the PAUSE type, utilizes an unused value of the *syndrome* field in the AETH header. Consequently, the new pause NACK also does not require alterations to the current packet structure.

Normally, the NIC receives packets and processes them by updating the QP state and sending back an acknowledgement (Figure 4.10). However, if a QP is in the *Stopped* or *Paused* state, this procedure is modified. A QP in the *Stopped* state responds solely with a PAUSE NACK, utilizing a previously unused value in the AETH packet's syndrome field. On the other hand, a QP in the *Paused* state disregards all messages until it receives a RESUME message. Upon receipt of a RESUME message, the NIC updates the QP's destination address based on the RESUME message's source field and processes pending tasks, such as outstanding send requests, to fully resume communication.

The final workflow we need to modify is the processing of acknowledgements (Figure 4.11). Normally, the NIC receives acknowledgements and processes them by updating the outstanding

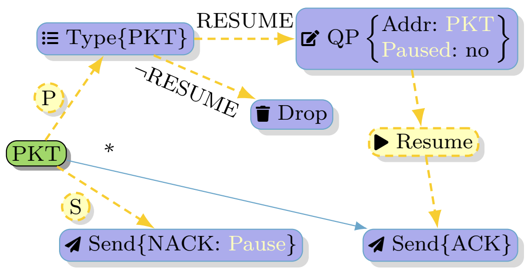


Figure 4.10: Pause-resume protocol changes packet processing logic. "*" represents "else" branch. ☰ Type checks the SR type.

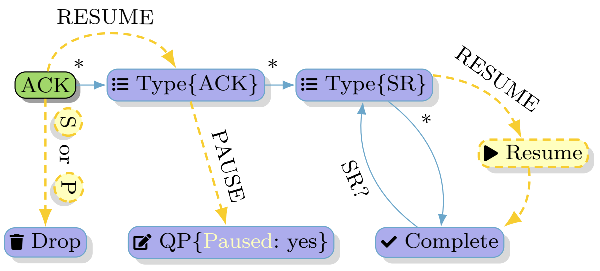


Figure 4.11: Handling acknowledgements with pause-resume protocol. "*" represents "else" branch. ☰ Type checks packet type.

send requests and the QP state. An outstanding send request is a Send Work Request (SR) for which the NIC sent out messages, but has not yet received acknowledgements. Once a send request is completed, the NIC notifies the application by updating the CQ and removes the send request from the send queue. After completing a single send request, the NIC checks for more outstanding send requests with a lower PSN; therefore, a single ACK may complete multiple send requests one by one.

With the addition of the pause-resume protocol, if a QP is in the *Stopped* or *Paused* state, it ignores all acknowledgements until receiving a RESUME message. The reason a paused QP may receive acknowledgements is that a QP may have sent such acknowledgements before being stopped. Dropping such late acknowledgements is necessary because they may have a PSN higher than that of the upcoming RESUME message. The RoCE protocol dictates that old messages must be ignored to avoid duplicating messages, which may result in RESUME messages being ignored. Therefore, a QP with an outstanding RESUME send request does not process more send requests until it receives an acknowledgement for the RESUME messages. Only then does a resumed QP fully resume communication. Receiving a PAUSE acknowledgement is much simpler: the NIC puts the QP into a paused state.

For the sake of brevity, our description omits some details, such as additional timeout reaction logic. Overall, the changes in logic are straightforward and largely reuse existing functionality. Because we change only the AETH and BTH headers, our modifications are equally applicable to other RDMA protocols (e. g., InfiniBand) that use these headers in the same manner. We believe that neither the new logic nor the new states incur prohibitive design or implementation costs.

4.3.4 Implementation

The exact implementation of the pause-resume protocol is device- and driver-specific. In this work, we chose to avoid direct hardware modifications because existing open-source FPGA-based RDMA NICs [Psi+22; Sid+20] offer limited support for high-level applications. Considering that our goal is to provide services for high-level applications, we opted to implement the pause-resume protocol in a software implementation of the RDMA communication protocol, called SoftRoCE.

SoftRoCE is a Linux kernel-level software implementation (not an emulation [Lis17]) of the RoCE protocol [Inf14]. RoCE facilitates RDMA communication by tunnelling InfiniBand packets through a well-known UDP port. The SoftRoCE driver implements this functionality atop a kernel-level UDP socket. Its capabilities include splitting a message into packets, formatting packet headers, and ensuring message order and reliable delivery. Unlike other RDMA-device drivers, SoftRoCE allows the OS to inspect, modify, and fully control the state of InfiniBand verbs objects.

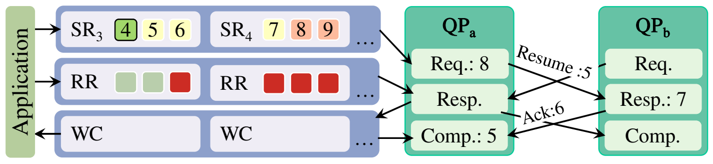


Figure 4.12: Resuming a connection in SoftRoCE. The figure depicts a snapshot of an immediate state, whereas the arrows indicate the flow of data over time. A send queue comprises multiple SRs, each expected to send multiple packets. Packets 8 and 9 (🟧) are to be processed by the requester. Packets 5 – 7 (🟨) are yet to be acknowledged. Packet 4 (🟩) is already acknowledged. A receive queue contains RRs with received (▨) and not yet received (🟥) packets. QP$_b$ expects the next packet to be 7. A resume packet has the PSN of the first unacknowledged packet (🟨). QP$_b$ replies with an acknowledgement of the last received packet.

Figure 4.12 outlines the basic operation of the SoftRoCE driver, which creates three concurrent *tasks* for each QP: *requester*, *responder*, and *completer*. When an application posts a Send Work Request (SR) or a Receive Work Request (RR) to a QP, it is processed by the requester and responder correspondingly. A work request may be split into multiple packets, depending on the Maximum Transmission Unit (MTU) size. When the entire work request is complete, the responder or completer notifies the application by posting a work completion to the completion queue.

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

When transferring QP$_a$ back into the bypass phase, the NIC sends a resume message to QP$_b$ with the new address. Upon receiving the resume message, the responder of QP$_b$ learns the new location of QP$_a$. Then, the responder replies with an acknowledgement of the last successfully received packet. If some packets were lost during the connection manipulation phase, the next PSN at the responder of QP$_b$ is smaller than the next PSN at the requester of QP$_a$. The difference corresponds to the lost packets. Simultaneously, the requester of QP$_b$ can already start sending messages. At this point, the connection between QP$_a$ and QP$_b$ is fully recovered.

The presented protocol ensures that both QPs recover the connection without irrecoverably losing packets. If packets were lost during the connection manipulation, the QP can determine which packets were lost and retransmit them as part of the normal RoCE protocol. As part of the SoftRoCE implementation, the pause-resume protocol runs transparently for the user applications.

4.3.5 Evaluation

We implemented the pause-resume protocol in SoftRoCE as part of the live migration support, which we describe in more detail in Section 4.4.3. The aim of our evaluation is to estimate the cost of adding the pause-resume protocol to a NIC and to compare it with the cost of continuous interposition. In this section, we evaluate only the performance of the bypass phase, while in Section 4.4.5, we assess the performance of the manipulation phase as part of the live migration operation.

For all the experiments in this section, we use the following two-node system: Each machine is equipped with an Intel i7-4790 CPU, 16 GiB RAM, an on-board Intel 1 Gb Ethernet adapter, a Mellanox ConnectX-3 VPI adapter, and a Mellanox Connect-IB 56 Gb adapter. We exclusively used Mellanox VPI adapters, which are set to 40 Gb Ethernet mode. The SoftRoCE driver also communicates over this adapter. The machines run Debian 11 with a custom Linux 5.7-based kernel. When comparing against DMTCP [Cao+14] and FreeFlow [Kim+19], we use Ubuntu 14.04.

To estimate the effect of the pause-resume protocol on the potential performance of the bypass phase, we used `gprof` to record the coverage of pause-resume support code outside the migration phase. Out of all the changes made to the QP tasks, only 28 lines were affected while application communication was active. Among these, 3 lines are variable assignments, one is an unconditional jump, and the rest are newly introduced if-else conditions checking a QP state that occur at most once per packet sent or received. Additionally, we modified the QP state to include binary flags to encode paused and stopped QP states. The rest of the code changes to the QP task run only during the connection manipulation phase and do not affect the performance of the bypass phase.

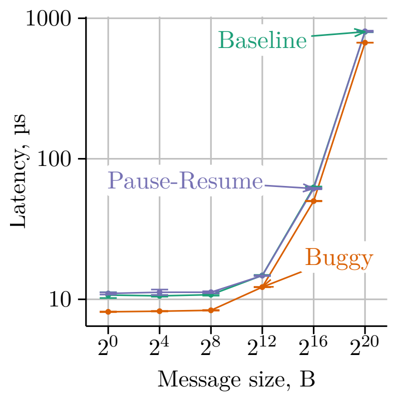


Figure 4.13: Communication latency over SoftRoCE

We anticipate that such minor changes will not have any measurable effect on the performance of the bypass phase. To validate this hypothesis, we measured the performance of SoftRoCE's bypass phase before and after integrating the pause-resume protocol. Unfortunately, the original version of the SoftRoCE driver (*vanilla kernel*, without any modifications on our part) proved to be notoriously unstable[2]. The original driver contained numerous concurrency bugs and necessitated significant restructuring. After rectifying these issues, we ended up with three versions of the driver: the original *buggy* version, a baseline (fixed) version, and a baseline version enhanced with the pause-resume protocol. Consequently, fixing the race conditions led to significant restructuring,

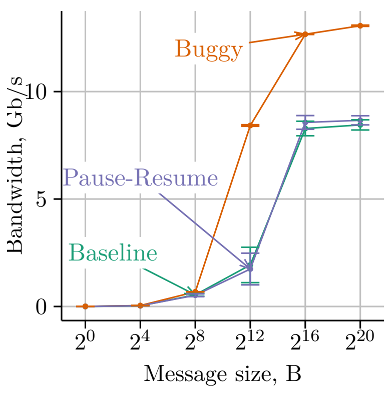


Figure 4.14: Communication throughput over SoftRoCE

[2] `SIGINT` to a user-level RDMA application caused the kernel to panic.

resulting in approximately 20% higher latency (see Figure 4.13) and an even more pronounced degradation in throughput (see Figure 4.14). Nevertheless, for the remainder of this chapter, we had to rely on a fixed version.

Fortunately, when comparing the baseline with the version incorporating the pause-resume protocol, we observed no significant difference in latency or throughput. Although, compared to traditional InfiniBand implementations, the substantial communication overhead introduced by SoftRoCE may obscure the potential performance impact of the pause-resume protocol. On the other hand, given that we expect the overhead to be minimal, even an FPGA implementation might not accurately reflect the performance of a commercial RDMA NIC, due to the significant differences between an FPGA and a real hardware implementation. Therefore, considering the scale of the changes and the measured results, we conclude that the pause-resume protocol introduces no runtime overhead during the bypass phase.

It is also essential to note that the pause-resume protocol offers an advantage over software-level continuous interposition methods, which are easier to deploy. For comparison, we utilized two existing solutions [3]: DMTCP, a checkpoint/restart library for MPI applications [AAC09; Cao+14], and FreeFlow, a software-level RDMA virtual network [Kim+19]. These approaches intercept all InfiniBand verbs library calls and rewrite both work requests and completions before forwarding them to the NIC. Both DMTCP and FreeFlow persistently intercept RDMA dataplane, even if the connection never enters the manipulation phase. In contrast, the pause-resume protocol does not intercept communication operations on the critical path, thereby introducing no measurable overhead.

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

Table 4.1: Comparing execution without modifications against DMTCP and FreeFlow. The variation over 30 runs was small, except,* when $0.05 < \sigma/\mu < 0.1$.

We expect that continuous interposition will incur significant overhead because it intercepts all communication operations. To test this assumption, we used the latency and bandwidth benchmarks from the perftest benchmark suite [per20]. We ran each experiment 30 times with 10 000 iterations each with ConnectX-3 40 Gibit NICs.

Both frameworks perform additional processing for each InfiniBand verbs work request, resulting in near-constant overhead on latency (see Table 4.1). Each work request corresponds to a single message, not a single packet; therefore, the overhead diminishes for larger message sizes. Table 4.1 demonstrates that bandwidth is directly affected by the increased latency, and thus it is lower only for small messages. We expect such a reduction in bandwidth to be a minor disadvantage for realistic applications, whereas a near 50% increase in latency may be

[3] At the time of conducting these experiments, CoRD had not been available.

critical for some latency-sensitive applications [Shi+16]. Generally, these results align with our expectations and are similar to those described in Section 3.2.

4.4 Live Migration of RDMA Applications

The primary use case for intermittent control in this thesis is the live migration of RDMA applications. With the aid of the pause-resume protocol (Section 4.3), we can transparently migrate RDMA connections from one node to another. To migrate the entire application, we must also save and restore the state of each RDMA connection, as well as the entire application. This section details how we integrate the modifications to the low-level RoCE protocol into the high-level software stack to enable fast and transparent live migration of RDMA applications.

To begin, we anticipate that end users run RDMA applications containerized, using tools such as Docker [Mer14] or Singularity [KSB17]. These tools have already become widespread in both cloud and HPC environments. Therefore, we build upon modern container runtimes and repurpose much of the existing infrastructure for live migration with minimal alterations. This approach allows us to focus predominantly on migration support for RDMA networks.

4.4.1 CRIU

For the live migration of containerized applications, we depend on CRIU [Eme+11]. CRIU is a software framework designed for transparently checkpointing and restoring the state of Linux processes. It facilitates live migration, snapshots, and remote debugging of processes, process trees, and containers. To extract the user-space application state, CRIU employs conventional debugging mechanisms [Ker20b; Ker20a]. However, CRIU relies on specialized Linux kernel interfaces to extract the state of process-specific kernel objects.

To restore a process, CRIU initializes a new process that initially runs the CRIU executable. This executable reads the image of the target process and reconstructs all OS objects on its behalf. This method enables CRIU to utilize the existing OS mechanisms to conduct most of the recovery without necessitating significant kernel modifications. Finally, CRIU eliminates any signs of its own presence from the process.

CRIU is also capable of restoring the state of TCP connections, which is crucial for the live migration of distributed applications [Cor12]. For this purpose, the Linux kernel introduced a new TCP connection state, `TCP_REPAIR`. In this state, a user-level process can change the state of the send and receive message queues, get and set message sequence numbers and timestamps, or open and close connections without notifying the other side. Once a connection is restored, the host must retain its original IP address. Containers with separate network namespaces running in an overlay network maintain this property when a process migrates to another host.

Without our modifications, if CRIU attempted to checkpoint an RDMA application, it would detect InfiniBand verbs objects and refuse to proceed. Discarding InfiniBand verbs objects in the naive hope that the application will be able to recover is failure-prone: once an application encounters an erroneous InfiniBand verbs object, it will, in most cases, hang or crash. Thus, the core of our live migration support is providing explicit support for InfiniBand verbs objects in CRIU.

4.4.2 Software Stack

To make live migration transparent, we must not require modifications to the software running inside the container. To achieve this goal, we have devised an architecture that allows any containerized application to run unmodified and still be migrated transparently. Typically, access to the RDMA network is concealed deep within the software stack. Figure 4.15 provides an example of a containerized RDMA application. The container image comes with all the necessary library dependencies, such as libc, but not the kernel-level drivers. In this example, the application utilizes a stack of communication libraries, including Open MPI [Gab+04], Open UCX [Sha+15] (not shown), and InfiniBand verbs. Normally, to migrate, a container runtime would require the application inside the container to terminate and later recover all InfiniBand verbs objects. This process undermines the transparency of live migration.

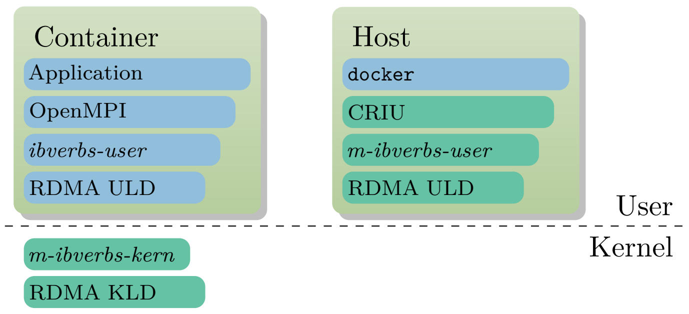


Figure 4.15: Live migration architecture. Software inside the container, including the user-level driver (*ibv*-user, grey), is unmodified. The host runs CRIU, kernel- (*m-ibv*-kern) and user-level (*m-ibv*-user, green) drivers modified for migratability.

Instead, we operate a parallel stack, which is controlled by the container engine, alongside the application container. This parallel stack comprises the container runtime (e.g., Docker [Mer14]), CRIU, and the InfiniBand verbs library. We have adapted CRIU to recognize InfiniBand verbs, enabling it to preserve InfiniBand verbs objects when traversing the kernel objects associated with the container. We have augmented the InfiniBand verbs library (*m-ibv*-user and *m-ibv*-kern) to facilitate the serialization and deserialization of InfiniBand verbs objects. Significantly, the API extension maintains backward compatibility with the InfiniBand verbs library operating inside the container. Consequently, both *m-ibv*-user and *ibv*-user interact with the same kernel version of InfiniBand verbs. All the InfiniBand verbs components (*ibv*-user, *m-ibv*-user, *m-ibv*-kern) consist of a generic and a device-specific part. As a result, the container engine depends on the modified kernel and user parts (*m-ibv*-user and *m-ibv*-kern), but *does not require any modifications to the software inside the container.*

Checkpoint/Restore API

To enable CRIU to checkpoint and restore processes and containers, we have extended the InfiniBand verbs API with two new calls (see Listing 4.1): `ibv_dump_context` and `ibv_restore_object`. CRIU utilizes the standard InfiniBand verbs API, supplemented by these two new calls, to save and restore the InfiniBand verbs state of applications.

The `ibv_dump_context` call returns a dump of all InfiniBand verbs objects within a specific InfiniBand verbs *context*, an object representing the connection between a process and an RDMA NIC. The creation of a dump runs almost entirely in the kernel for two primary reasons: Firstly, certain links between the objects are only visible at the kernel level. Secondly, creating a dump inside the kernel allows to make it atomic, which is crucial to get a consistent checkpoint.

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

Listing 4.1: Checkpoint / Restore extension for the InfiniBand verbs API. `ibv_dump_context` creates an image of the InfiniBand verbs context `ctx` with `count` objects and stores it in the caller-provided memory region `dump` of size `length`. `ibv_restore_object` executes the restore command `cmd` for an individual object (QP, CQ, etc.) of type `object_type`. The call expects a list of arguments specific to the object type and recovery command. `args` is an opaque pointer to the argument buffer of size `length`. A pointer to the restored object is returned via `object`.

Although the existing InfiniBand verbs API allows the creation of new objects, it is not expressive enough for *restoring* them. For instance, when restoring a CQ, the current API does not permit specifying the address of the shared memory region for this queue. Instead, the kernel assigns this address. Moreover, it is impossible to recreate a QP directly in its original state, such as RTS. Instead, the QP must traverse all intermediate states to reach the desired state.

We introduce the fine-grained `ibv_restore_object` call to restore InfiniBand verbs objects one by one, for situations where the existing API is insufficient. During recovery, CRIU reads the object dump and applies a specific recovery procedure for each object type. For example, to recover a QP, CRIU calls `ibv_restore_object` with the command `CREATE` and transitions the QP through the Init, RTR, and RTS states using `ibv_modify_qp`. The memory regions or QP buffers are recovered using standard file and memory operations. Finally, when a QP reaches the RTS state (representing an active connection), the new host executes the `REFILL` command using the `ibv_restore_object` call. This command restores the driver-specific internal QP state and sends a *resume* message to the partner QP.

4.4.3 Implementation

We implement a pause-resume protocol and transparent live migration support by modifying CRIU, the InfiniBand verbs library, the RDMA-device driver (SoftRoCE), and the packet-level RoCE protocol. To migrate an application, the container runtime invokes CRIU, which checkpoints the target container. CRIU stops active RDMA connections and saves the state of InfiniBand verbs objects (see Section 4.4.1). SoftRoCE then transitions the connections to the manipulation phase. After transferring the checkpoint to the destination node, the container runtime at that node invokes CRIU to recover the InfiniBand verbs objects and restore the application. SoftRoCE then resumes all paused communication to complete the migration process.

Live migration comprises two crucial steps: the pause-resume protocol and the connection-state extraction and recovery. We described the former in Section 4.3; now, we focus on the latter.

State extraction begins when CRIU discovers that its target process has opened an InfiniBand verbs device. We modified CRIU to use the API presented in Section 4.4.2 to extract the state of all available InfiniBand verbs objects. CRIU stores this state, along with other process data, in an image. Later, CRIU recovers the image on another node using the new API.

When CRIU recovers an MR or a QP of the migrated application, the recovered object must maintain its original unique identifiers. These identifiers are system-global and assigned by the NIC (in our case, the SoftRoCE driver) sequentially. We augmented the SoftRoCE driver to expose the IDs of the last assigned MR and QP to CRIU in userspace. These IDs are Memory Region Number (MRN) and QPN, respectively. Before recreating an MR or QP, CRIU configures the last ID appropriately. If no other MR or QP occupies this ID, the newly created object will maintain its original ID. This approach is analogous to how CRIU maintains the process ID of a restored process using the `ns_last_pid` mechanism of Linux, which exposes the last process ID assigned by the kernel.

It is possible for some other process to occupy an MRN or QPN that CRIU intends to restore. Two processes cannot use the same MRN or QPN on the same node, resulting in a conflict. In the current scheme, we avoid these conflicts by globally partitioning QP and MR IDs among all nodes in the system before application startup. CRIU encounters a similar problem with process ID collisions. This problem has only been solved with the introduction of process ID namespaces. To address the collision problem for InfiniBand verbs objects, a similar namespace-based mechanism, along with a virtual RDMA network [He+20], would be required. We discuss this issue in Section 4.5.

Additionally, recovered MRs must maintain their original memory protection keys. The protection keys are pseudo-random numbers [Rot+21] provided by the NIC and are used by a remote communication partner when sending a packet. An RDMA operation succeeds only if the provided key matches the expected key of a given MR. Apart from that, the key's value does not carry any additional semantics. Thus, no collision problems exist for protection keys. CRIU sets all protection keys to their original values before communication restarts by making an `ibv_restore_object` call with the `IBV_RESTORE_MR_KEYS` command.

4.4.4 Implementation Effort

We try to demonstrate that our changes are indeed feasible for real-world NIC implementation in two ways. First, we give a detailed explanation of the changes required, so that the reader could use their own judgement about the actual complexity of the changes. We, for example, show that all the additions to the protocol executed during the bypass phase access only the simple state of InfiniBand objects and run only once per packet. Our second argument is that the actual implementation of the changes is also small, especially in parts that are specific to the NIC implementation.

To quantify the changes, we count the newly added or modified Source Lines of Code (SLOC) in different components of the software stack (see Table 4.2). Out of approximately 4 k SLOC, only about 10% apply to the kernel-level SoftRoCE driver. These changes primarily focus on saving and restoring the state of InfiniBand verbs objects. We separately counted changes to the requester, responder, and completer QP tasks responsible for the active phase of communication

Level	Component	Original	Δ
Kernel	InfiniBand verbs	30,565	719
	SoftRoCE	9,446	872
	QP tasks	1,112	249
User	InfiniBand verbs	12,431	339
	SoftRoCE	1,004	332
	CRIU	61,616	1,845
Total			4,137

Table 4.2: Development effort in SLOC. We specifically show the magnitude of changes done to the QP tasks (see Figure 4.12).

(see Figure 4.12). These tasks would be implemented in the NICs for hardware-based RDMA implementations. Therefore, we ensure that the changes to QP tasks are simple and minimal, as these changes must be reflected in firmware or hardware. In our implementation, changes to QP tasks accounted for only about 6% of the overall changes.

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

Table 4.3: Additional features implemented in the kernel-level SoftRoCE driver to enable recovery of InfiniBand verbs objects. We provide the size that each object occupies in the dump.

Besides additional logic in the QP tasks, saving and restoring InfiniBand verbs objects necessitates the manipulation of implementation-specific attributes. Some of these attributes cannot be set through the original InfiniBand verbs API. For example, to recover an MR, it is necessary to have the additional ability to restore the original values of memory keys and the Memory Region Number (MRN). Some other attributes are not exposed by the original InfiniBand verbs API at all. The queues (CQ, SRQ, QP) implemented in SoftRoCE require the capability to save and restore the metadata of ring buffers backing the queues. If a QP uses an SRQ, the dump of the QP also includes the full state of the current WQE. We identified all the required attributes for SoftRoCE, calculated their memory footprint (see Table 4.3), and implemented all the features required by these attributes.

In conclusion, the changes to RoCE implemented in SoftRoCE are minimal and affect the critical path of communication only marginally outside of the migration phase [4]. We believe that for RDMA NICs, the same changes to RoCE will remain just as minimal, or can even be implemented in the NIC's firmware[5].

[4]Our implementation can be found here: github.com/TUD-OS/migros-atc-2021.
[5]The hardware and firmware boundary will differ for FPGA and ASIC.

4.4.5 Evaluation

We evaluate the live-migration capability from two main aspects. First, we estimate the fine-grained cost of migration for individual InfiniBand verbs objects. Second, we measure the migration latency in realistic RDMA applications.

For most experiments, we use a system with two machines (the same as in Section 4.3.5): Each machine is equipped with an Intel i7-4790 CPU, 16 GiB RAM, an on-board Intel 1 Gb Ethernet adapter, a Mellanox ConnectX-3 VPI adapter, and a Mellanox Connect-IB 56 Gb adapter. The Mellanox VPI adapters are configured in 40 Gb Ethernet mode. The SoftRoCE driver communicates over this adapter. The machines run Debian 11 with a custom Linux 5.7-based kernel. We refer to this setup as the *local* setup. When comparing against DMTCP and FreeFlow, we use Ubuntu 14.04.

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

Table 4.4: RDMA-capable NICs used for the evaluation.

We conduct further measurements on a cluster comprising nodes with two-socket Intel E5-2680 v3 CPUs and Connect-IB 56 Gb NICs deployed by Bull. We refer to this setup as the *cluster* setup. Two nodes, similar to those in the cluster, were used in a local setup and equipped with Mellanox ConnectX-3 VPI NICs, configured to 56 Gb InfiniBand mode. We describe all NICs used in the evaluation in Table 4.4 and refer to them by their short names.

Microoperation Costs

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

This benchmark allows us to project the time required to establish RDMA communication in a real application. For instance, to establish a single Reliable Connection (RC), the application needs to create several InfiniBand verbs objects (see Section 2.4): a Protection Domain (PD), a Completion Queue (CQ), a Memory Region (MR), and a Queue Pair (QP). A typical application can be expected to create a single PD, a CQ per core, hundreds of MRs, and one QP per communication partner.

To measure the cost of creating individual InfiniBand verbs objects, we modified `ib_send_bw` [per20] to create additional MR objects. We created one CQ, one PD, 64 QPs, and 64 1 MiB-sized MRs per run. Figure 4.16 shows the average time required to create each object across 50 runs. Each tested NIC is represented by a bar. We draw two conclusions from this

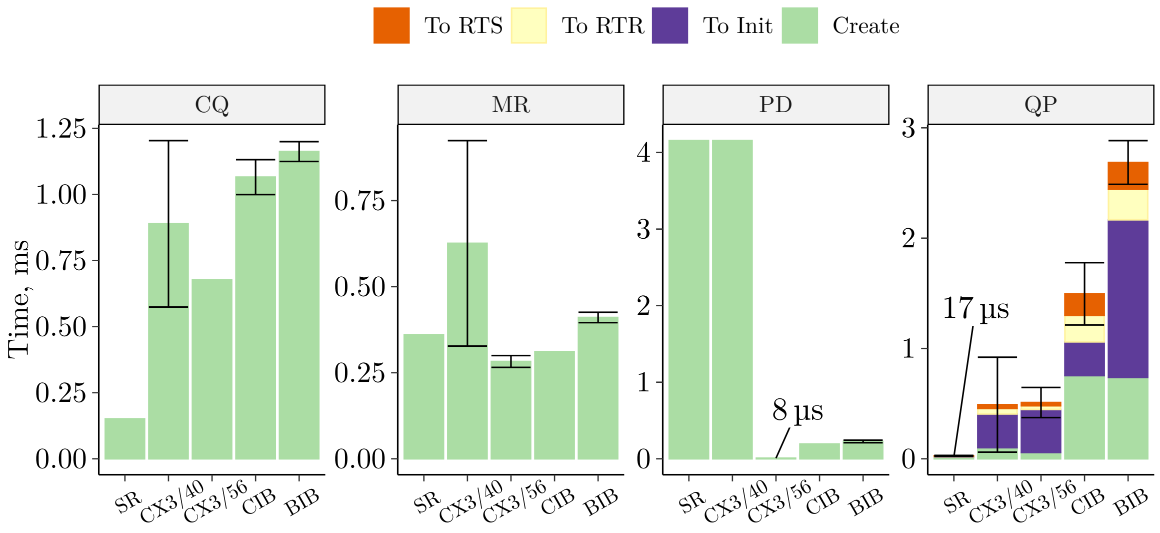


Figure 4.16: Object creation time for different RDMA devices. To send a message, a QP needs to be in the state RTS, which requires the traversal of three intermediate states (Reset, Init, RTR). Error bars show the interval of the standard deviation (σ) around the mean (μ), if $\sigma/\mu \geq 0.05$.

experiment. First, there is a significant variation in the time required for all operations across different NICs. Second, the time required for most operations falls in the millisecond range.

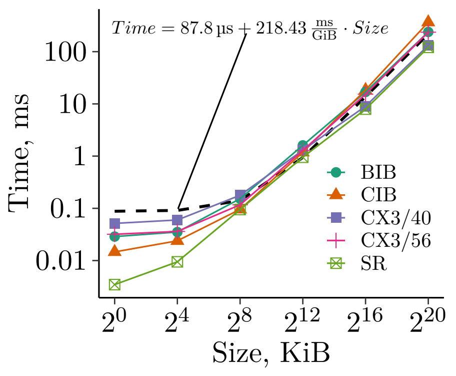


Figure 4.17: MR registration time depending on the region size.

The migration time of an RDMA application is influenced by two factors: the number of QPs and the total amount of memory allocated to MRs [Mie+06]. Both factors are specific to the application and can vary widely. Therefore, we demonstrate how each of these parameters affects the migration time. First, Figure 4.17 illustrates how the MR registration time depends on the size of the region. Additionally, we fit the memory registration time of all the measured NICs into a linear model, which is represented by a dashed line. This time is divided between the OS and the NIC components. The OS is responsible for pinning the memory, and the NIC establishes the mapping between the physical and virtual addresses of the registered region. SoftRoCE does not incur the "NIC-part" of the cost, making MR registration with SoftRoCE faster than with RDMA-enabled NICs. For this experiment, we do not take into account the cost associated with transferring the contents of the MRs during migration.

The number of QPs is the second variable influencing live migration time. Figure 4.18 displays the time required for migrating a container running the `ib_send_bw` benchmark. This benchmark involves two single-process containers operating on two separate nodes. Three seconds after the initiation of communication, the container runtime migrates one of the containers to a different node. The migration time is recorded as the maximum message latency observed by the container that remained stationary. The checkpoint is transmitted over the same

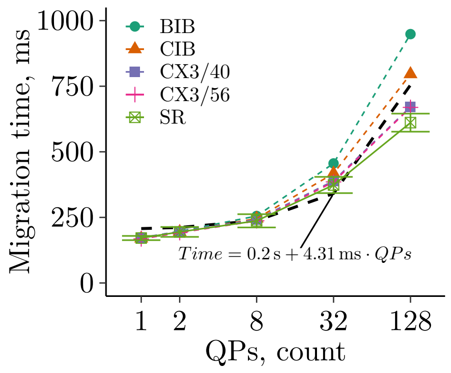


Figure 4.18: Migration speed with different numbers of QPs.

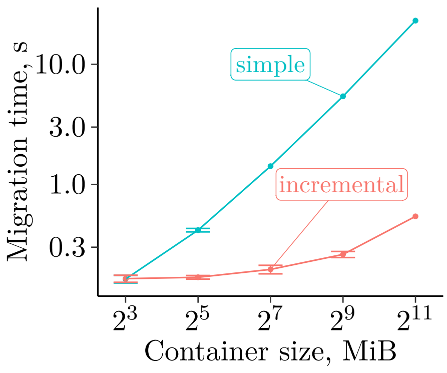


Figure 4.19: Incremental migration is more scalable

network link that the benchmarks use for communication. As the number of QPs increases, the benchmark's memory consumption also grows, ranging from 8 MiB to 20 MiB. To provide context, we estimated the migration time for actual RDMA devices by calculating the time needed to recreate InfiniBand verbs objects for RDMA-enabled NICs. We deducted the time required to create InfiniBand verbs objects with SoftRoCE from the measured migration time and added the time necessary to create InfiniBand verbs objects with RDMA NICs (as shown in Figure 4.16). We present these estimations with the coloured dashed lines and illustrate the model fitting all the data with the black dashed line.

As the memory footprint of a container increases, so does the cost of migration. At a certain point, the migration may become excessively costly. To mitigate this issue for large containers, we employ *iterative checkpointing*.

Iterative checkpointing entails the container runtime creating a *preliminary dump* of the application's memory prior to the actual migration. The destination node receives this preliminary dump while the application continues to run. Simultaneously, the OS on the source node monitors the memory modifications made by the application after the preliminary dump. To finalize migration, the container runtime merely transfers the portions of memory that have been modified since the preliminary dump, rather than the entire state.

Figure 4.19 shows that for large containers, the migration time is directly proportional to the application's memory size. For this benchmark, we modified the `ib_send_bw` benchmark to allocate additional memory during its runtime. We were interested in the best case latency improvement, thus the additional memory is not modified after initialization, and we used different network links for the checkpoint transfer and the application communication.

Overall, through a set of microbenchmarks, we show how the migration time depends on the characteristics of the RDMA application. The most significant bottlenecks incurred for RDMA applications are the costs of creating InfiniBand verbs objects and transferring the memory contents, particularly of memory regions. This limitation can be overcome by iterative checkpointing and by improving the performance of object creation at the NIC level.

MPI Application Migration

For evaluating the transparent live migration of real-world applications, we chose to migrate NPB 3.4.1 [Bai+94], an MPI benchmark suite. The MPI applications run on top of Open MPI 4.0 [Gab+04], which in turn uses Open UCX 1.6.1 [Sha+15] for point-to-point communication. We configured UCX to use InfiniBand verbs communication over reliable connections (RC). This setup corresponds to Figure 4.15.

We containerized the applications using our self-developed runtime *konyk*, based on libcontainer [run20]. Unlike Docker, our runtime facilitates faster live migration by sending the image directly to the destination node, instead of to the local storage, during the checkpoint process. Additionally, konyk stores checkpoints in RAM, further reducing the migration latency. Like any other container runtime, konyk internally uses CRIU for checkpointing and restoring containers. A further description of our container runtime is beyond the scope of this work.

To measure the application migration latency, we start each MPI application with four processes (*ranks*). Approximately in the middle of the application's progress, one of the ranks migrates to another node. Each benchmark has a *size* (A to F) parameter, which we chose such that each benchmark runs between 10 and 300 seconds. We excluded the "dt" benchmark because it runs for only around a second. Figure 4.20 shows container migration latency and standard deviation around the mean, averaged over 20 runs of each benchmark.

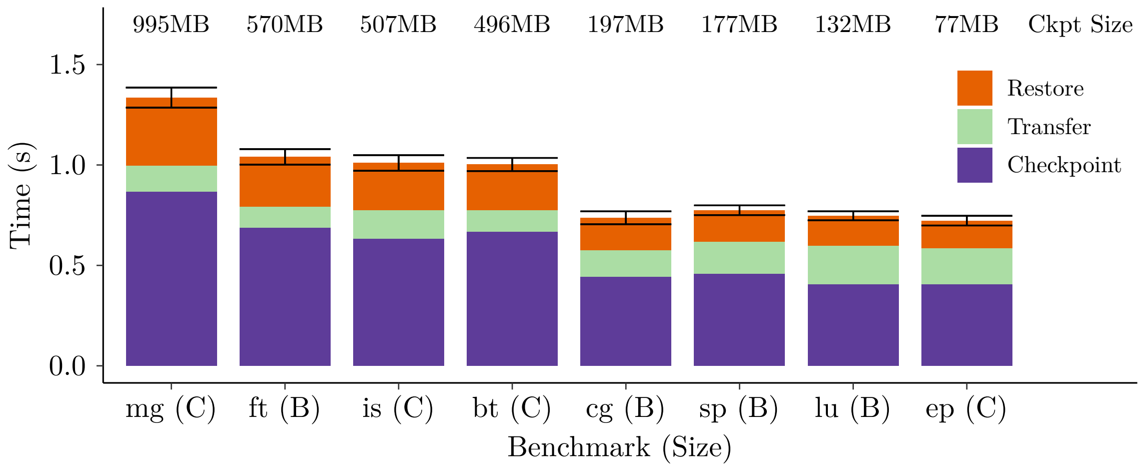


Figure 4.20: MPI application migration.

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

MPI applications (Figure 4.20) migrate slower than microbenchmarks (Figure 4.18), even when accounting for the checkpoint size, due to the difference in measurement methodology. For the microbenchmark, we calculate the migration time based on the maximum message latency observed by the non-migrating process. For the MPI benchmarks, we calculate the migration time from the increase in the total execution time of the entire benchmark. This discrepancy suggests that the migration of parallel applications may cause a larger disruption to application performance than what the simple state transfer time can explain.

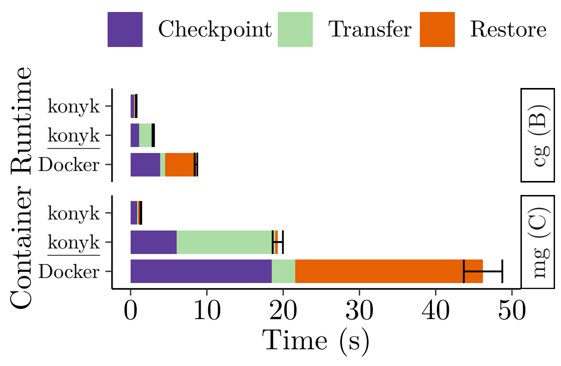


Figure 4.21: Migration speed of Docker and konyk with optimizations (konyk) and without (konyk)

To demonstrate the interoperability of our approach with other container runtimes, we measured the migration costs when using Docker 19.03 (see Figure 4.21). We had to implement the end-to-end migration flow ourselves because Docker only provides checkpoint and restore features. To our disappointment, Docker does not implement some important optimizations and requires significantly more time to complete a migration. To better understand the performance difference, we disabled certain optimizations in konyk: saving checkpoints to main memory (instead of the hard disk), sending checkpoints during dumping, and using an optimized method for transferring the checkpoint. Despite this, the performance of Docker still did not match that of konyk. Further investigation revealed that Docker unnecessarily moves checkpoint images across the file system, demonstrating the importance of explicit live migration support within a container runtime. Nevertheless, we successfully demonstrated the fundamental feasibility of containerized RDMA-application migration using Docker.

4.4.6 Related Work

Live migration of VMs has a long history of use in cloud computing [NLH05; Cla+05; Des+14; HDG09; Pan+12]. With the emergence of new computing paradigms like disaggregated and fog computing [Pat+18; Gu+17; WHW19; Osa+17], we anticipate an even greater reliance on live migration to move computation closer to data. However, past techniques for live VM migration with RDMA NICs depended on migration-aware, paravirtualized drivers inside the VMs [Hua+07; Pic+16].

Concurrently with MigrOS [Pla+21], Hansen et al. [Han+21] introduced a pause-resume protocol akin to our approach, but aimed at enabling virtual machine migration. Their method relies on SR/IOV-based hardware virtualization, which simplifies the migration of NIC LID and GID addresses. We chose not to use SR/IOV because hardware virtualization requires static partitioning of NIC resources, thus limiting the flexibility of the system. Given that a typical system is likely to run many more containers than VMs, SR/IOV could become a bottleneck for containerized applications rather than for VMs. Moreover, our approach allows for migrating a container from one VM to another, while the method of Hansen et al. would necessitate nested virtualization to achieve the same objective.

Transparent live migration of processes [Smi88; MDW99; BS85], containers [MYL17; MKK08; Nad+17], or virtual machines [NLH05; Cla+05; Des+14; HDG09; Pan+12; Han+21] has long been a topic of active research. The primary challenge of this technique lies in the

	Legion	Nomad	PS MPI	DMTCP	MOSIX-4	MOSIX-3	vMotion	This thesis
RDMA	✓	✓	✓	✓	✗	✗	✓	✓
Overhead	N	N	N	Y	Y	Y	N	N
Runtime	✓	✓	✓					
User-OS				✓	✓			
Kernel-OS		✓				✓	✓	✓
NIC							✓	✓
Units	O	VM	P	P	P	P	VM	C
Reference	[Bau+12]	[Hua+07]	[Pic+16]	[AAC09]	[BS16]	[BGW93]	[Han+21]	[Pla+21]

Table 4.5: Selected checkpoint/restore systems handle either VMs, processes (P), containers (C), or application objects (O). Runtime-based systems naturally introduce no additional communication overhead for migration support.

checkpoint/restore operation. For processes and containers, this operation can be implemented at three levels: application runtime, user-level system, or kernel-level system. Table 4.5 compares a selection of existing checkpoint/restore systems.

Runtime-based systems require the user application to access all external resources through the runtime's API. This requirement resolves two significant issues with resource migratability: First, the runtime system precisely controls when the underlying resource is used and can readily stop the application from doing so to serialize the state of the resource. Second, the runtime can maintain sufficient information about the state of the resource to facilitate resource serialization and deserialization. Such interception is efficient because it occurs within the application's address space.

Nearly all attempts to provide transparent live migration in conjunction with RDMA networks involve modifications to the runtime system [Gua+15; Pic+16; AAC09; Hua+07; Jos+13; GPC19]. Some runtimes operate on application-defined objects (tasks, agents, lightweight threads) for even more effective state serialization and deserialization [Bau+12; KK93; WPM99]. However, all runtime-based approaches tether the application to a specific runtime system.

Kernel OS-level checkpoint/restore systems [BGW93; HD06; Osm+03; Edg09; KS09] either interpose at the kernel level or extract the application state from the kernel's internal data structures. Although these systems support a wider spectrum of user applications, they incur a significantly higher maintenance burden. BLCR [HD06] has been eventually abandoned. CRIU [Eme+11], currently the most successful OS-level tool for checkpoint/restore, keeps the necessary Linux kernel modifications at a minimum and does not require interposition at the user-kernel API. We describe this tool in more detail in Section 4.4.1. Kadav and Swift [KS09] implemented kernel-level shadow objects to enable the live migration of virtual machines with several types of direct-access devices. Besides lacking support for RDMA networking, shadow objects also caused continuous performance overhead, similarly to DMTCP. Zap [Osm+03] intercepts system calls to virtualize process resource identifiers and creates virtual proxy-devices to control access to the hardware. Both MOSIX-4 [BS16] and Zap support InfiniBand only through the BSD socket API, which adds a large performance penalty.

Finally, *user OS-level* systems interpose the user-kernel API, providing the same transparency and generality as kernel-based implementations. Such systems use the `LD_PRELOAD` mechanism to intercept system calls from applications and virtualize system resources, like file descriptors, process IDs, and sockets. In version 4, MOSIX has been redesigned to work entirely at the user level [BS16]. DMTCP [AAC09] is a transparent fault-tolerance tool for distributed applications with support for InfiniBand verbs. To extract the state of InfiniBand verbs objects, DMTCP maintains *shadow objects*, which act as proxies between a user process and the NIC [Cao+14]. The latest version of DMTCP is not in sync with current libraries providing RDMA support. Instead, the core developers of DMTCP moved the connection migration support inside the MPI library [GPC19]. In Section 4.4.5, we show that maintaining these shadow objects has a non-negligible runtime overhead for RDMA networks.

Furthermore, live migration may employ RDMA networks to improve the speed of the checkpoint transfer [NR18; Ibr+11]. These techniques allow for reducing the downtime from migration and could be combined with our technique to improve the migration time of RDMA applications.

SoftRoCE [Lis17] and SoftiWarp [TMS11] are open-source software implementations of RoCE [Inf14] and iWarp [Gar+07] respectively. Both provide no performance advantage over socket-based communication but are compatible with their hardware counterparts and facilitate the development and testing of RDMA-based applications. We chose to base our work on SoftRoCE because RoCE has found wider adoption than iWarp.

There are also open-source FPGA-based implementations of network stacks. NetFPGA [Zil+14] does not support RDMA communication. StRoM [Sid+20] provides a proof-of-concept RoCE implementation. However, we found it unfit for running real-world applications (for example, MPI) without further significant implementation efforts. Nowadays, there also exist commercial IP cores implementing high-performance RDMA NIC capabilities, like ERNIC [Xil22], but at the time of the work on the pause-resume protocol, they were unavailable for our research and development.

4.4.7 Summary

We employed intermittent control in an architecture enabling transparent live container migration without sacrificing RDMA network performance. We are convinced that this architecture can be useful for dynamic load balancing, efficient prepared fail-over, and live software updates in cloud or HPC settings.

We believe that limited hardware changes are worthy of consideration and have already been proven feasible [Koc+19b; Fir+18; Koc+19b; Moo+20; Han+15], even for RDMA protocols [Lis13; Inf09; Les+17]. Nevertheless, propositions to modify hardware often meet criticism because they are hard to validate and evaluate for an OS designer. To overcome this difficulty, we have modified SoftRoCE. It turned out that adding only a few states to the state machine and two new message types were necessary. As a result, we enabled transparent live migration of containerised RDMA applications without affecting the critical path of communication. Lesokhin et al. [Les+17] have demonstrated that RDMA protocol changes can be achieved just through firmware updates, obviating the need to replace NICs. Furthermore, our design maintains full backward compatibility with the existing RDMA network infrastructure at every level and can be adopted by other RDMA protocols (e. g., InfiniBand and RoCE) verbatim. Moreover, an idea very similar to ours has been proposed by

Hansen et al. [Han+21] at the same time and implemented in commercial hardware for the purpose of live migration of virtual machines.

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

We believe that the pause/resume protocol can find other uses, like efficient fail-over, congestion control, or load balancing. For example, consider MasQ [He+20], a virtual RDMA network with firewall capabilities. MasQ can shut down RDMA connections but, unlike TCP/IP firewalls, cannot block them temporarily. Our protocol could return control over RDMA connections to the OS and replicate TCP/IP-like behavior to RDMA firewalls as well.

4.5 Virtual RDMA Network

When developing live migration capabilities for RDMA applications, we encountered several limitations of our approach [6]. First, the physical addresses of the RDMA NICs are visible to the user applications but change transparently during the live migration process. This could lead to certain inconsistencies. For instance, the application could learn the physical address of the remote partner before migration and attempt to establish a connection after the partner migrates. Second, we demonstrated how to conduct live migration of RDMA applications for connection-oriented transports, like RC, but not for connectionless transports, like UD. A migrating UD QP does not have a designated remote sender QP, so it cannot send a pause message in advance and may need to broadcast a resume message throughout the entire network.

The goal of a virtual RDMA network is to create a virtual network topology on top of the physical network topology. Applications communicate with each other using virtual addresses, which are independent of the physical addresses of the RDMA NICs. Therefore, when an application migrates, the virtual addresses remain the same. A virtual network could also help with connectionless transports, like UD, because it could ensure that packets are delivered to the correct QP even after migration.

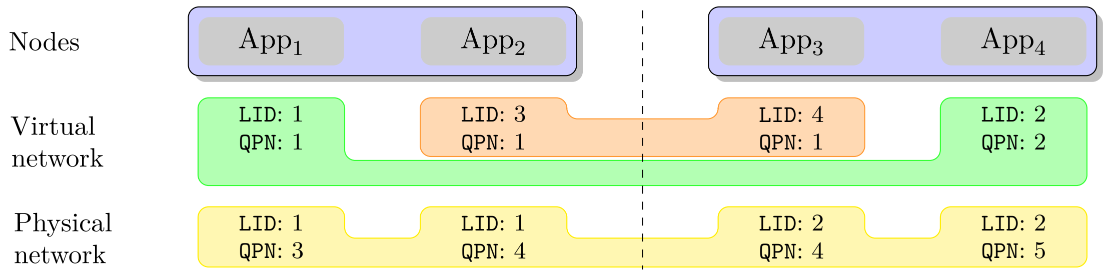


Figure 4.22: Virtual and physical networks. The dashed line separetes physical nodes.

[6]Some of these limitations could be overcome using SR/IOV, which we also wanted to avoid.

Figure 4.22 illustrates a scenario with two hypothetical InfiniBand verbs-based virtual RDMA networks, isolating App_1 and App_4 from App_2 and App_3. Each application has two addresses (for brevity, InfiniBand verbs defines more, see Section 2.4): an LID, and a QPN. LID identifies a compute node (like a MAC-address), whereas QPN identifies an endpoint on the node (like a port number). Applications running on the same node must have the same physical LID and different physical QPN. A virtual network allows the virtualization of QPN and LID by replacing physical addresses with virtual addresses in control plane operations.

The InfiniBand verbs API is designed such that all requests for physical addresses are implemented by control operations. Data plane operations operate with opaque handles, which are similar to file descriptor numbers. Therefore, we attempted to implement a virtual RDMA network, which either intercepts control plane operations or employs intermittent control for the data plane. Our expectation was that applications could only see virtual addresses but transparently use physical addresses for the data plane. Together with intermittent interposition, the virtual network can also enable transparent live migration for UD QPs with zero overhead for the bypass phase. As it turned out, our approach was unfeasible, demonstrating the limitations of intermittent control.

4.5.1 Architecture

Our virtual RDMA network must hide and replace the physical network identifiers, which applications may request through the InfiniBand verbs API. Furthermore, it must be able to intercept communication and bring it into a quiescent state. Such interception can assist with live migration. To avoid overhead during the bypass phase, all virtualization steps, except for interception, must run at the control plane.

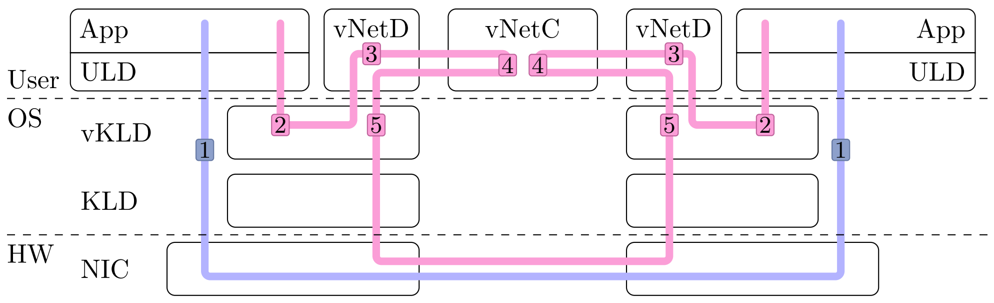


Figure 4.23: The data-plane (🟦) remains unobstructed. The virtual network kernel-level driver (vKLD) intercepts the control-plane (🟪) operations (1), forwards them (2) to a local virtual network daemon (vNetD). The daemon may query (3) the global virtual network coordinator (vNetC) to resolve virtual addresses to physical ones. Afterwards, the vKLD populates the parameters of the control-plane operation with physical addresses and forwards the operation to the RDMA NIC kernel-level driver.

To virtualize the RDMA network, we need to virtualize several network identifiers:

1. QPN identifies an endpoint on the node (like a port number).
2. LID is a non-routable network identifier (like a MAC address).
3. GID is a routable network identifier (like an IP address).
4. GUID is a number that uniquely identifies a device or a component in RDMA networks, including ports and switches.

The application accesses all the identifiers through the InfiniBand verbs library. Moreover, the application learns about all these identifiers through the control-plane operations of the InfiniBand verbs library [He+20]. This means that we can virtualize the identifiers by intercepting the control-plane operations inside the kernel.

We devised an architecture (see Figure 4.23) that intercepts the control-plane operations and forwards them to a central virtual network daemon. These control-plane operations include operations for querying the addresses of the RDMA NICs, such as `ibv_query_device`, and operations for creating and modifying QPs, like `ibv_create_qp` and `ibv_modify_qp`. We created a virtual network kernel-level driver (vKLD) that creates a virtual RDMA network interface, used by the application. Therefore, when the application calls control-plane operations, they are intercepted by the vKLD and forwarded to the virtual network daemon. The daemon maintains a map between virtual and physical identifiers by assigning and resolving virtual identifiers.

The virtual kernel-level driver directly passes to the application all the memory normally shared between the application and the RDMA NIC. This allows RDMA applications to execute all data-plane operations, like sending or receiving messages, unobstructed. The applications continue using the same user-level device driver (ULD) as before. We also maintain a small virtual user-level driver (vULD), which is responsible for opening the virtual device and setting up the correct user-level driver for the application.

Now, if there is a need to migrate one of the applications in the virtual network, we can query the virtual network daemon about which other applications can communicate with the application to be migrated. This would allow us to enable live migration support for the applications using UD transport, because before sending a message to a UD QP, the application must resolve the address of the destination QP, which is a control operation. The daemon could remember every application that ever resolved the address to the migrated application and request the host OS of these applications to preemptively pause the applications from sending messages to the migrated one. This would allow us to migrate the application without losing any messages.

Unfortunately, it turned out that QPN cannot be fully virtualized in the architecture described above. The problem is that the QPN is used in the data-plane operations and the applications get direct access to the QPN at the level of the user-level driver. Specifically, when the application polls a CQ, the Mellanox user-level driver will read the `flow_tag` field as a physical QPN from the Completion Queue Entry (CQE) and use it to dispatch the CQE to the correct QP [Mel16, Table 67]. In other words, the NIC passes the physical QPN to the user-level driver, which then passes it further to the application logic.

For example, when an application polls a CQ, the InfiniBand verbs API returns a Work Completion (WC) which informs the application logic about which QP has received a message. In the case of the Mellanox driver, this WC contains the physical QPN of the QP that received the message. Sooner or later, the application notices the discrepancy between the virtual QPN obtained from the control plane and the physical QPN obtained from the data plane, which results in an error.

Alternatively, the virtual networks could virtualize only the LID, but not the QPN. The issue with this approach is that the QPN is a node-local identifier. This means that if the application migrates to another node, its current QPN might be used by some other application. One possible solution to this issue is to statically partition the QPN space between the applications in the same virtual network, but this would limit the scalability of the virtual network because QPN is a 16-bit number. Moreover, existing Mellanox NIC drivers assign QPNs sequentially, starting from 2 (0 and 1 are reserved) and do not support arbitrary QPN assignment.

4.5.2 Related Work

TCP/IP network virtualization is an essential tool for isolating distributed applications from the underlying physical network topology. Even though network virtualization enables live migration, it introduces overhead due to the additional encapsulation of network packets [Niu+19; Zhu+19]. Live migration depends on network virtualization to ensure that a change in physical location does not affect the application's connectivity. Several new approaches attempt to address these performance issues [Pet+14; Bel+14; Zhu+19; Niu+19]. However, these approaches do not consider RDMA networks.

RDMA-network virtualization approaches focus on implementing connection control policies in software but do not support live container migration [Kim+19; He+20; TZ17]. As an exception, Nomad [Hua+07] utilizes InfiniBand address virtualization for VM migration and implements the connection migration protocol inside an application-level runtime. LITE [TZ17] also virtualizes RDMA networks but offers no migration support and requires an application rewrite. Using paravirtualization on the control path helps to mitigate the overhead of hardware virtualization [Pfe+15; Mou+17; Liu+06].

It is possible to offload network virtualization to hardware using VXLAN technology [Mah+14]. For that purpose, the NIC must be able to create an RDMA NIC Virtual Function (VF) for each containerized application [PCI10]. This would allow each application to have a separate network address so that the application could migrate to another node and maintain its old address. As an additional requirement, the host system needs to support IOMMU to be able to pass the VF to the container. All these technologies have small but measurable performance overheads [ABY11]. In contrast, our goal was to investigate whether it is possible to implement a virtual RDMA network without any overhead and special hardware support by relying on the interposition of the RDMA control plane or through the use of intermittent control.

4.5.3 Summary

In our experiment with virtual RDMA networks, we developed an architecture that allows network virtualization without any overhead during the bypass phase. Unfortunately, our experiment has not been fully successful, as it turned out that not all identifiers can be virtualized in real-world systems [7]. This experiment exposed the limitations of intermittent control because if the underlying hardware exposes physical identifiers to the user level, the OS becomes limited in what it can achieve during the manipulation phase.

On the other hand, our experiment indicates that it is still possible to build a virtual RDMA network, albeit with limitations on what can be virtualized. One could employ hardware-level technologies, such as Virtual Extensible LAN (VXLAN), to virtualize the RDMA address

[7] Our implementation can be found here: github.com/planeta/ovey/tree/dev and github.com/planetA/linux/tree/mplaneta/ovey/v5.15-single.

space [Mah+14]. Alternatively, one could modify the binary interface between the user-level driver and the NIC to hide the real QPN from the application and dispatch QPs in some other way. Or, one could virtualize the QPN space inside the user-level driver. However, our main goal was to explore intermittent control, so we did not pursue these alternatives.

4.6 Conclusion

This chapter presented an intermittent interposition architecture for OS-level control over the RDMA network dataplane. The key components of this architecture are the interception gate, a temporary request buffer, a back-pressure notification, and a resume notification. Together, these components allow for the interception of RDMA connections and the manipulation of their state without affecting the performance of the bypass path. We demonstrated live migration for containerized RDMA applications as a successful use case for intermittent interposition. It turned out to be possible to migrate RDMA applications without any performance overhead during the bypass phase and without any changes to the application code.

This chapter also highlighted the limitations of intermittent control: It may not be possible to implement desired functionality for the manipulation phase if the underlying hardware exposes too much information to the application during the bypass phase.

In summary, we showed that intermittent control is a viable approach for implementing certain OS-level functionalities for RDMA networks. On the other hand, we also demonstrated that implementing such functionalities often requires careful consideration of the underlying hardware and software architecture. In the use cases we studied, careful hardware-software co-design was necessary to achieve the desired functionality.

Statement of Contributions

The work presented in this chapter is based on the following publications:

- Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. “MigrOS: Transparent Operating Systems Live Migration Support for Containerised RDMA-applications”. In: *USENIX ATC 2021*. July 14, 2021, pp. 47–63. ISBN: 978-1-939133-23-6. URL: `https://www.usenix.org/conference/atc21/presentation/planeta`
- Philipp Schuster. “Overlay-RDMA-Netzwerke im Kontext von Live-Migration”. Großer Beleg. Dresden, Germany: TU Dresden, 2021

Some of the text in this chapter has being taken verbatim from the MigrOS publication. The author of this thesis contributed to the design, implementation, and evaluation of MigrOS [Pla+21] and Ovey [Sch21]. The author also contributed to the writing of the MigrOS publication.

5 Conclusion and Future Work

This thesis presents an approach for fine-grained OS-level control over RDMA communication. The OS can achieve such control by continuously or intermittently interposing on the RDMA data plane. A careful design of the continuous interposition architecture ensures that the RDMA application experiences low or sometimes negligible overhead. In fact, we have demonstrated a whole range of continuous interposition mechanisms with progressively lower overhead. If the application cannot allow even the slightest overhead, then intermittent interposition can offer such properties.

On the other hand, each interposition approach has its own set of limitations. For example, lower overhead continuous mechanisms also require a progressively more complex programming model for implementing OS functions. And intermittent interposition approaches require hardware modifications. Moreover, for intermittent interposition to really have zero overhead during the bypass phase, the interception gate must not add any measurable latency. As Chapter 4 shows, this constraint can be satisfied in some use cases, but clearly not in all, limiting the applicability of zero overhead intermittent interposition.

Both continuous and intermittent interposition have their own separate set of design considerations. The main design consideration for continuous interposition is its four inherent sources of overhead, which need to be minimized to make the interposition mechanism practical. The first is the latency of transition from the application to the OS and back. This latency results either from a system call, like in CoRD, or a hypercall, like in MasQ [He+20]. The first source of overhead can be entirely eliminated by assigning the OS service to a dedicated core, like in FreeFlow [Kim+19]. The second is the overhead of passing the arguments to and from the OS service. The third is the synchronization overhead, which appears when a service is shared by multiple applications. These two sources cannot be eliminated, but can be reduced by using more efficient serialization and synchronization mechanisms, as shown by Meignan–Masson [Mei23]. Finally, the last source of overhead is the actual execution of the OS service. Service overhead can be quantified by comparing its latency to a *best available alternative*, which can be not having the service at all, or implementing the service in the hardware or application logic. When minimizing these sources of overhead, the system designer must be aware of a trade-off between performance and other factors, such as maintainability or resource utilization.

Intermittent interposition, on the other hand, operates under the constraint that the overhead is zero or near-zero. Little tradeoff is possible here because if the interception gate adds any significant latency, continuous interposition could be used instead. Therefore, the design consideration is the need for the manipulation phase to be fast and seldom activated, otherwise, the application's performance will suffer. The second design consideration is that intermittent interposition is less generalizable than continuous interposition, because the interception gate is not a dedicated module in a NIC. More likely, it is an existing component, like MTT, that monitors connection state to stop communication in the manipulation phase and inspects packets to trigger the manipulation phase when the right condition is met. Turning a NIC component into an interception gate is case-specific and requires expertise to do the hardware modifications.

Overall, a system designer must decide what trade-off is acceptable in each individual case. The decision may depend on the exact application, how stakeholders benefit from the added functionality, the underlying hardware architecture, desired security properties, etc. This thesis provides a mental framework for a system designer to make the right decision.

We expect that the regular need for data plane interposition becomes more apparent with wider use of high-performance networking in the cloud computing setting. The reason for our expectation is that one of the cornerstones of cloud computing efficiency is multi-tenancy and resource sharing, which is not tolerated by traditional RDMA networking. Ideally, an RDMA application does not share a network with other applications, least compute nodes. Therefore, for expanding the applicability of RDMA networks, it is imperative to make sharing tolerable.

As Chapter 3 demonstrated, if a latency-sensitive application experiences even a small increase in communication latency, its performance may degrade significantly. So, it is important that the cloud application is robust towards certain network disruptions such as congestion or packet loss. On the positive side, we have observed that one source of network disruption can mask other sources of disruption. As a consequence, if the application moves from an extremely low noise HPC environment into a noisy cloud environment, continuous interposition may be unnoticeable, as Chapter 3 also shows.

With these considerations, CoRD can be a good starting point for integrating RDMA applications into data centers, because it is flexible and backward compatible with existing RDMA applications. If the application's performance suffers, the system designer can then consider either lower overhead continuous interposition mechanisms, like fastcalls, or intermittent interposition, like MigrOS. To support a variety of applications, the data center can partition its resources into different tiers and offer different types of interposition to applications based on their requirements. Ultimately, if the application cannot tolerate even the slightest noise in the network or compute node, then, maybe, the cloud setting is not the right place for such an application.

5.1 Future work

This work has been driven by the vision that RDMA networking is extremely beneficial for future data center applications. We already see the trend of ubiquitizing RDMA networks, but there is a gap between software-level expectations and what hardware provides. For software developers and software infrastructure operators, properties like maintainability, deployability, and efficient resource sharing are of utmost importance. But these properties are very hard to provide purely at the hardware level. Our answer to this challenge is careful hardware-software co-design that can balances functional and performance properties.

For our vision to come true, as a precondition, we consider it necessary to have a fully virtualizable RDMA network. An RDMA application should be oblivious to its physical location and physical network identifiers. We have shown that in its current state, intermittent interposition is not by itself sufficient to virtualize the RDMA network data plane. On the other hand, using CoRD may not be the best solution for all applications, because it adds constant per-message overhead, which some applications would really like to avoid. Existing hardware-level virtualization solutions, such as SR/IOV, work well for VMs but are too coarse-grained for microservice-sized applications.

We believe that the solution for a virtual RDMA network is to combine intermittent and continuous interposition within a single approach. Lightweight continuous interposition, such as with fastcalls, can be utilized for low-overhead data plane virtualization, while intermittent interposition can be employed to implement the required OS-level functionality. We anticipate this example to be an intriguing use case for fastcalls, which will demonstrate the practicality of the fastcall programming model.

This thesis discussed smart and programmable NICs only in passing because in the context of this work, the use of these new devices does not change the overall OS-level architecture. Nevertheless, smart NICs can be used for both continuous and intermittent interposition. For example, Data Path Acceleration (DPA) introduced in the new Bluefield 3 NICs [Bur21b] can be used to virtualize the RDMA data plane with low overhead, which we expect to be similar or even faster than fastcalls. Therefore, we believe that applying our findings to programmable NICs is an exciting future direction of our research.

Although smart NICs will definitely experience growth in their use, we do not expect them to become ubiquitous in the foreseeable future. So, we see value in building upon the research conducted by Meignan–Masson [Mei23], where he analyzed the sources of overhead in CoRD and follow up upon his recommendations. From our experience and observations, if the per-message overhead can be reduced to less than 0.5 µs, then the overhead is negligible for most applications.

Having a low overhead RDMA network virtualization solution can enable multiple use cases relevant to the cloud computing setting. For example, one use case could be to employ DNAT-type load balancing for RDMA-enabled services [Ban+17]. Another use case could be transparent failover for RDMA-enabled client applications. Overall, fine-grained OS-level control over RDMA communication will allow the integration of RDMA applications into cloud container orchestration systems, such as Kubernetes.

Bibliography

[AAC09] Jason Ansel, Kapil Arya, and Gene Cooperman. “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop”. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. IPDPS. IPDPS. May 2009, pp. 1–12. DOI: `10/d62csg`. arXiv: `cs/0701037`.

[ABY11] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. “IOMMU: Strategies for Mitigating the IOTLB Bottleneck”. In: *Computer Architecture*. Ed. by Ana Lucia Varbanescu, Anca Molnos, and Rob van Nieuwpoort. Vol. 6161. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 256–274. ISBN: 978-3-642-24321-9 978-3-642-24322-6. DOI: `10.1007/978-3-642-24322-6_22`.

[Ali+10] Mohammad Alizadeh et al. “Data Center TCP (DCTCP)”. In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM ’10. New York, NY, USA, 2010. DOI: `http://dx.doi.org/10.1145/1851182.1851192`.

[Alv+12] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. *Cray XC Series Network*. Cray, 2012, p. 28.

[Ama20] Amazon Web Services, Inc. *Elastic Fabric Adapter*. 2020. URL: `https://aws.amazon.com/hpc/efa/` (visited on 04/14/2020).

[AMD21] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification, 48882*. 3.06-PUB. 2021, p. 299.

[AMK22] Marcelo Abranches, Oliver Michel, and Eric Keller. “Getting Back What Was Lost in the Era of High-Speed Software Packet Processing”. In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. HotNets ’22: The 21st ACM Workshop on Hot Topics in Networks. Austin Texas: ACM, Nov. 14, 2022, pp. 228–234. ISBN: 978-1-4503-9899-2. DOI: `10.1145/3563766.3564114`.

[Bai+94] David H. Bailey et al. “The NAS Parallel Benchmarks”. In: (1994).

[Ban+17] Tim Bannister, Qiming Teng, Rob Scott, Divya Mohan, Karen Bradshaw, and Shabir Mohamed Abdul Samadh. *Service*. Kubernetes Documentation. 2017. URL: `https://kubernetes.io/docs/concepts/services-networking/service/` (visited on 02/27/2024).

[Bar+17a] Brian W Barrett et al. *The Portals 4.2 Network Programming Interface*. Sandia National Laboratories, Apr. 2017, p. 158.

[Bar+17b] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. “Attack of the Killer Microseconds”. In: *Communications of the ACM* 60.4 (Mar. 24, 2017), pp. 48–54. ISSN: 0001-0782. DOI: 10/f94xjp.

[Bar13] Dotan Barak. *Ibv_modify_qp()*. RDMAmojo. Jan. 12, 2013. URL: https://www.rdmamojo.com/2013/01/12/ibv_modify_qp/ (visited on 08/04/2023).

[Bau+12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. “Legion: Expressing Locality and Independence with Logical Regions”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. SC. SC. Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–11. ISBN: 978-1-4673-0806-9. DOI: 10/gf9t42.

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

[Bes+21] Maciej Besta et al. “High-Performance Routing With Multipathing and Path Diversity in Ethernet and HPC Networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.4 (Apr. 1, 2021), pp. 943–959. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2020.3035761.

[BGW93] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Berlin, Heidelberg: Springer-Verlag, 1993. 231 pp. ISBN: 978-0-387-56663-4.

[BHR18] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Third Edition. Springer, 2018. 201 pp. ISBN: 978-3-031-01761-2. URL: https://library.oapen.org/bitstream/handle/20.500.12657/61844/1/978-3-031-01761-2.pdf.

[Bin+11] Nathan Binkert et al. “The Gem5 Simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (May 31, 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[Bir+15] M. S. Birrittella et al. “Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. Aug. 2015, pp. 1–9. DOI: 10.1109/HOTI.2015.22.

[Boy+08] Silas Boyd-Wickizer et al. “Corey: An Operating System for Many Cores”. In: *8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Symposium on Operating Systems Design and Implementation. San Diego, California, USA: USENIX Association, 2008.

[BPH22] Jan Bierbaum, Maksym Planeta, and Hermann Härtig. “Towards Efficient Oversubscription: On the Cost and Benefit of Event-Based Communication in MPI”. In: *Runtime and Operating Systems for Supercomputers*. ROSS. Dallas, Texas, USA: IEEE, 2022.

[Bro18] Broadcom. *Stingray™ PS1100R: Fully Integrated 100GbE Fabric-Attached Storage Adapter*. 2018.

[BS16] Amnon Barak and Shiloh, Amnon. *The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster Private Clouds.* Springer-Verlag, 2016.

[BS85] Amnon Barak and Amnon Shiloh. “A Distributed Load-Balancing Policy for a Multicomputer”. In: *Software: Practice and Experience* 15.9 (Sept. 1985), pp. 901–913. ISSN: 00380644, 1097024X. DOI: `10/c8r7m6`.

[Bur19] Anatoly Burakov. *Memory in Data Plane Development Kit Part 1: General Concepts.* Intel. 2019. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/memory-in-dpdk-part-1-general-concepts.html` (visited on 02/15/2024).

[Bur21a] Matt Burns. *PCIe 4.0 Over Fiber Solutions in Embedded Computing Applications.* The Samtec Blog. Mar. 1, 2021. URL: `https://blog.samtec.com/post/pcie-4-0-over-fiber-solutions-in-embedded-computing-applications/` (visited on 03/14/2023).

[Bur21b] Idan Burstein. “NVIDIA DataCenter Processing Unit (DPU) Architecture”. Hot Chips 33. Dec. 14, 2021. URL: `https://www.youtube.com/watch?v=1bvYXinIdS0` (visited on 12/05/2023).

[BV22] Jag Brar and Pradeep Vincent. “First Principles: Building a High Performance Network in the Public Cloud”. Dec. 12, 2022. URL: `https://www.youtube.com/watch?v=ca_OGqCVHDM` (visited on 02/07/2024).

[Cai+22] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. “Towards s Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack”. In: *Proceedings of the ACM SIGCOMM 2022 Conference.* SIGCOMM ’22: ACM SIGCOMM 2022 Conference. Amsterdam Netherlands: ACM, Aug. 22, 2022, pp. 767–779. ISBN: 978-1-4503-9420-8. DOI: `10.1145/3544216.3544230`.

[Can+19] Claudio Canella et al. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* CCS ’19: 2019 ACM SIGSAC Conference on Computer and Communications Security. London United Kingdom: ACM, Nov. 6, 2019, pp. 769–784. ISBN: 978-1-4503-6747-9. DOI: `10.1145/3319535.3363219`.

[Cao+14] Jiajun Cao, Gregory Kerr, Kapil Arya, and Gene Cooperman. “Transparent Checkpoint-Restart over Infiniband”. In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing - HPDC ’14.* The 23rd International Symposium. Vancouver, BC, Canada: ACM Press, 2014, pp. 13–24. ISBN: 978-1-4503-2749-7. DOI: `10/ggnfr4`.

[Cla+05] Christopher Clark et al. “Live Migration of Virtual Machines”. In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2.* NSDI ’05. USENIX Association, May 2005, pp. 273–286. DOI: `10.5555/1251203.1251223`.

[Com22] Compute Express Link Consortium Inc. *Compute Express Link Specification.* Version 3.0. Aug. 1, 2022. URL: `https://www.computeexpresslink.org/_files/ugd/0c1418_1798ce97c1e6438fba818d760905e43a.pdf` (visited on 10/11/2023).

[Cor+05] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, and Alessandro Rubini. *Linux Device Drivers.* 3rd ed. Beijing ; Sebastopol, CA: O'Reilly, 2005. 615 pp. ISBN: 978-0-596-00590-0.

[Cor05] Jonathan Corbet. *Linux and TCP Offload Engines.* Aug. 22, 2005. URL: `https://lwn.net/Articles/148697/` (visited on 10/11/2023).

[Cor09] Jonathan Corbet. *KSM Tries Again.* 2009. URL: `https://lwn.net/Articles/330589/` (visited on 02/15/2024).

[Cor12] Jonathan Corbet. *TCP Connection Repair.* LWN.net. May 1, 2012. URL: `https://lwn.net/Articles/495304/` (visited on 11/03/2019).

[Cor17a] Jonathan Corbet. *The Current State of Kernel Page-Table Isolation.* LWN.net. Dec. 20, 2017. URL: `https://lwn.net/Articles/741878/` (visited on 01/20/2023).

[Cor17b] Jonathan Corbet. *Zero-Copy Networking.* LWN.net. 2017. URL: `https://lwn.net/Articles/726917/` (visited on 02/15/2024).

[Cor19] Jonathan Corbet. *Ringing in a New Asynchronous I/O API.* LWN.net. Jan. 15, 2019. URL: `https://lwn.net/Articles/776703/` (visited on 12/14/2022).

[Cra19] Cray Inc. *XC Series GNI and DMAPP API User Guide CLE70UP02 (S-2446).* 2019.

[DA23] Alexander Duyck and Daniel Axtens. *Segmentation Offloads.* Segmentation Offloads. 2023. URL: `https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt` (visited on 10/10/2023).

[Dec+21] Alessandro Decina, Dave Tucker, Tamir Duberstein, Michal Rostecki, Andrew Stoycos, and William Findlay. *Aya.* 2021. URL: `https://aya-rs.dev/` (visited on 12/24/2023).

[Des+14] Umesh Deshpande, Yang You, Danny Chan, Nilton Bila, and Kartik Gopalan. "Fast Server Deprovisioning through Scatter-Gather Live Migration of Virtual Machines". In: 2014 IEEE 7th International Conference on Cloud Computing. Anchorage, AK, USA: IEEE, June 2014, pp. 376–383. ISBN: 978-1-4799-5063-8. DOI: `10/ggnfmn`.

[DH16] Jens Domke and Torsten Hoefler. "Scheduling-Aware Routing for Supercomputers". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis.* SC. SC'16. Salt Lake City, UT, USA: IEEE, Nov. 2016, pp. 142–153. ISBN: 978-1-4673-8815-3. DOI: `10/gf8vnk`.

[DNC17] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. "RDMA Reads: To Use or Not to Use?" In: *Bulletin of the Technical Committee on Data Engineering* 40.1 (Mar. 2017), pp. 3–14.

[DPD13] DPDK Project. *Data Plane Development Kit.* DPDK. 2013. URL: `https://www.dpdk.org/` (visited on 09/26/2019).

[Dre+13] Roland Dreier, Dotan Barak, Sean Hefty, and Michael Tsirkin. *Libfabric.* OpenFabrics Interfaces Working Group, 2013. URL: `https://github.com/ofiwg/libfabric` (visited on 02/21/2024).

[Edg09] Jake Edge. *Checkpoint/Restart Tries to Head towards the Mainline.* LWN. Feb. 20, 2009. URL: `https://lwn.net/Articles/320508/` (visited on 03/05/2020).

[EH13] Kevin Elphinstone and Gernot Heiser. "From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?" In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* SOSP '13: ACM SIGOPS 24th Symposium on Operating Systems Principles. Farminton Pennsylvania: ACM, Nov. 3, 2013, pp. 133–150. ISBN: 978-1-4503-2388-8. DOI: `10.1145/2517349.2522720`.

[EKO95] Dawson R Engler, M Frans Kaashoek, and James O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: ACM SIGOPS Operating Systems Review 29.5 (1995), p. 16. DOI: `10.1145/224056.224076`.

[Eme+11] Pavel Emelyanov et al. *Checkpoint/Restore In Userspace.* CRIU. 2011. URL: `https://criu.org/Main_Page` (visited on 10/13/2019).

[Enb+22] Pekka Enberg, Ashwin Rao, Jon Crowcroft, and Sasu Tarkoma. *Transcending POSIX: The End of an Era?* USENIX. Sept. 8, 2022. URL: `https://www.usenix.org/publications/loginonline/transcending-posix-end-era` (visited on 01/26/2023).

[Fir+18] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation.* NSDI. NSDI'18. Berkeley, Calif: USENIX Association, 2018, pp. 51–64. ISBN: 978-1-931971-43-0.

[Fis+23] Eric Fiselier et al. *Google/Benchmark.* Google, Nov. 9, 2023. URL: `https://github.com/google/benchmark` (visited on 11/09/2023).

[For+13] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses.* Request for Comments RFC 6824. Internet Engineering Task Force, Jan. 2013. 64 pp. DOI: `10.17487/RFC6824`.

[Fry23] Mike Frysinger. *Vdso(7) - Linux Manual Page.* 2023. URL: `https://man7.org/linux/man-pages/man7/vdso.7.html` (visited on 10/29/2023).

[FSB19] Alvaro Frank, Tim Süss, and André Brinkmann. "Effects and Benefits of Node Sharing Strategies in HPC Batch Systems". In: *2019 IEEE International Parallel and Distributed Processing Symposium.* 2019 IEEE International Parallel and Distributed Processing Symposium. IPDPS. IEEE Computer Society, May 2019, pp. 43–53. DOI: `10.1109/IPDPS.2019.00016`.

[Gab+04] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* EuroPVM/MPI. Vol. 3241. Berlin, Heidelberg: Springer, 2004, pp. 97–104. ISBN: 978-3-540-30218-6. DOI: `10/bxhsv5`.

[Gar+07] D. Garcia, P. Culley, R. Recio, J. Hilland, and B. Metzler. *A Remote Direct Memory Access Protocol Specification.* Oct. 2007. URL: `https://tools.ietf.org/html/rfc5040` (visited on 04/02/2020).

[Ger+16] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. “On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Chicago, IL, USA: IEEE, May 2016, pp. 1041–1050. ISBN: 978-1-5090-2140-6. DOI: 10.1109/IPDPS.2016.80.

[GGS20] Jesse Gross, Ilango Ganga, and T. Sridhar. *Geneve: Generic Network Virtualization Encapsulation*. Request for Comments RFC 8926. Internet Engineering Task Force, Nov. 2020. 34 pp. DOI: 10.17487/RFC8926.

[Gia+10] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. “Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene’s CNK”. In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010 SC - International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, LA, USA: IEEE, Nov. 2010, pp. 1–10. ISBN: 978-1-4244-7557-5. DOI: 10.1109/SC.2010.22.

[Gig21] GigaIO. *FabreX Composable Memory Fabric Platform*. 2021. URL: https://gigaio.com/wp-content/uploads/2022/02/GIO_System-Overview_v-5-2.pdf (visited on 03/14/2023).

[GPC19] Rohan Garg, Gregory Price, and Gene Cooperman. “MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing”. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing - HPDC ’19*. HPDC. HPDC. Phoenix, AZ, USA: ACM Press, 2019, pp. 49–60. ISBN: 978-1-4503-6670-0. DOI: 10/ggnd38.

[Gra+05] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. “Itanium — A System Implementor’s Tale”. In: (2005), p. 14.

[Gra+16] Richard L. Graham et al. “Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction”. In: *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. 2016 First International Workshop on Communication Optimizations in HPC (COMHPC). Salt Lake City, UT: IEEE, Nov. 2016, pp. 1–10. ISBN: 978-1-5090-3829-9. DOI: 10.1109/COMHPC.2016.006.

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

[Gua+15] Wei Lin Guay, Sven-Arne Reinemo, Bjørn Dag Johnsen, Chien-Hua Yen, Tor Skeie, Olav Lysne, and Ola Tørudbakken. “Early Experiences with Live Migration of SR-IOV Enabled InfiniBand”. In: *Journal of Parallel and Distributed Computing* 78 (Apr. 2015), pp. 39–52. ISSN: 07437315. DOI: 10/f68twd.

[HA20] Shuveb Hussain and Jens Axboe. *Io_uring(7) - Linux Manual Page*. 2020. URL: https://man7.org/linux/man-pages/man7/io_uring.7.html (visited on 10/29/2023).

[Hac+15] Daniel Hackenberg, Robert Schone, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. “An Energy Efficiency Feature Survey of the Intel Haswell Processor”. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW). Hyderabad, India: IEEE, May 2015, pp. 896–904. ISBN: 978-1-4673-7684-6. DOI: `10.1109/IPDPSW.2015.70`.

[Han+15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report UCB/EECS-2015-155. Berkeley: EECS Department, University of California, 2015.

[Han+21] Jorgen Hansen, Bryan Tan, Rajesh Jalisatgi, Adit Ranadive, Vishnu Dasa, and Georgina Chua. “Supporting Live Migration of VMS Communicating with Bare-Metal RDMA Endpoints”. OFA Virtual Workshop. 2021. URL: `https://www.openfabrics.org/wp-content/uploads/2021-workshop-presentations/402_Hansen_PVRDMA.pdf` (visited on 10/24/2021).

[Här+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schonberg, and Jean Wolter. “The Performance of M-Kernel-Based Systems”. In: SOSP. SOSP. 1997, p. 15. DOI: `10.1145/268998.266660`.

[HB11] Tom Herbert and Willem de Bruijn. *Scaling in the Linux Networking Stack*. 2011. URL: `https://www.kernel.org/doc/Documentation/networking/scaling.txt` (visited on 02/19/2024).

[HD06] Paul H. Hargrove and Jason C. Duell. “Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters”. In: *Journal of Physics: Conference Series* 46 (Sept. 1, 2006), pp. 494–499. ISSN: 1742-6588, 1742-6596. DOI: `10/d33sc5`.

[HDG09] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. “Post-Copy Live Migration of Virtual Machines”. In: *ACM SIGOPS Operating Systems Review* 43.3 (July 31, 2009), pp. 14–26. ISSN: 0163-5980. DOI: `10/ccwrpt`.

[He+20] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. “MasQ: RDMA for Virtual Private Cloud”. In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, July 30, 2020, pp. 1–14. ISBN: 978-1-4503-7955-7. DOI: `10/gg9rjq`.

[Hef12] Sean Hefty. “Rsockets”. OpenFabris International Workshop (Monterey, CA, USA). 2012. URL: `https://smallake.kr/wp-content/uploads/2014/04/2012_Workshop_Mon_Rsockets.pdf` (visited on 02/23/2024).

[Hem+23] Stephen Hemminger et al. *Iproute2/Iproute2.Git - Iproute2 Routing Commands and Utilities*. 2023. URL: `https://git.kernel.org/pub/scm/network/iproute2/iproute2.git` (visited on 12/24/2023).

[Heo15] Tejun Heo. *Control Group V2*. Oct. 2015. URL: `https://www.kernel.org/doc/Documentation/cgroup-v2.txt` (visited on 10/17/2019).

[Hes+08] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. “GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation”. In: *Journal of Chemical Theory and Computation* 4.3 (Mar. 2008), pp. 435–447. ISSN: 1549-9618, 1549-9626. DOI: `10/b7nkp6`.

[Hoe+17] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. “sPIN: High-Performance Streaming Processing in the Network”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’17*. The International Conference for High Performance Computing, Networking, Storage and Analysis. Denver, Colorado: ACM Press, 2017, pp. 1–16. ISBN: 978-1-4503-5114-0. DOI: `10.1145/3126908.3126970`.

[Høi+18] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. “The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel”. In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT ’18: The 14th International Conference on Emerging Networking EXperiments and Technologies. Heraklion Greece: ACM, Dec. 4, 2018, pp. 54–66. ISBN: 978-1-4503-6080-7. DOI: `10/gh5x29`.

[HSG06] W Huang, G Santhanaraman, and Q Gao. “Design and Implementation of High Performance MVAPICH2 (MPI2 over InfiniBand)”. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid*. CCGRID ’06. 2006, p. 10.

[HSL09] T. Hoefler, T. Schneider, and A. Lumsdaine. “The Impact of Network Noise at Large-Scale Communication Performance”. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009 IEEE International Symposium on Parallel Distributed Processing. May 2009, pp. 1–8. DOI: `10/fjh44w`.

[Hua+07] Wei Huang, Jiuxing Liu, Matthew Koop, Bulent Abali, and Dhabaleswar Panda. “Nomad: Migrating OS-bypass Networks in Virtual Machines”. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments - VEE ’07*. The 3rd International Conference. VEE’07. San Diego, California, USA: ACM Press, 2007, p. 158. ISBN: 978-1-59593-630-1. DOI: `10/frgqz4`.

[Hub01] Bert Hubert. *Tc(8) - Linux Manual Page*. 2001. URL: `https://man7.org/linux/man-pages/man8/tc.8.html` (visited on 10/10/2023).

[Ibr+11] Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. “Optimized Pre-Copy Live Migration for Memory Intensive Applications”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’11*. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. Seattle, Washington: ACM Press, 2011, p. 1. ISBN: 978-1-4503-0771-0. DOI: `10/bsgrkf`.

[IEE10] IEEE. *802.1Qau - Congestion Notification*. Apr. 23, 2010. DOI: `10.1109/IEEESTD.2010.5454063`. Incorporated into 802.1Q-2011.

[IEE11] IEEE. *802.1Qbb - Priority-based Flow Control*. June 16, 2011. URL: `https://1.ieee802.org/dcb/802-1qbb/` (visited on 10/15/2019).

[Inf09] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: XRC*. InfiniBand Trade Association, Feb. 3, 2009. URL: `https://cw.infinibandta.org/document/dl/7146`.

[Inf10] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: RoCE*. 1.2.1. Vol. 1. InfiniBand Trade Association, 2010.

[Inf14] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specification: RoCEv2*. InfiniBand Trade Association, Sept. 2, 2014. URL: `https://cw.infinibandta.org/document/dl/7781`.

[Inf15] InfiniBand Trade Association. *InfiniBand Architecture Specification*. 1.3. Vol. 1. InfiniBand Trade Association, Mar. 3, 2015. URL: `https://cw.infinibandta.org/document/dl/8567`.

[Int20] Intel. *Intel® Scalable I/O Virtualization Technical Specification*. Technical Specification 337679-002, Rev 1.1. Intel, 2020, p. 29. URL: `https://cdrdv2-public.intel.com/671403/intel-scalable-io-virtualization-technical-specification.pdf` (visited on 02/16/2024).

[Int22] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. 78. 2022.

[Int23] Intel Corporation. *Intel® Virtualization Technology for Directed I/O*. Architecture Specification 4.1. 2023. URL: `www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`.

[Jam+17] Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. "mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes". In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. Boston, MA, USA, 2017.

[Jin21] Yaoxin Jing. "Mechanisms for Fast System Calls". Studienarbeit. Dresden, Germany: TU Dresden, Oct. 2021.

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

[Kau+19] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. "TAS: TCP Acceleration as an OS Service". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19: Fourteenth EuroSys Conference 2019. Dresden Germany: ACM, Mar. 25, 2019, pp. 1–16. ISBN: 978-1-4503-6281-8. DOI: `10/ggzhd6`.

[Ker20a] Michael Kerrisk. *Proc(5) - Linux Manual Page*. proc - process information pseudo-filesystem. 2020. URL: `http://man7.org/linux/man-pages/man5/proc.5.html` (visited on 03/20/2020).

[Ker20b] Michael Kerrisk. *Ptrace(2) - Linux Manual Page*. ptrace - process trace. 2020. URL: `http://man7.org/linux/man-pages/man2/ptrace.2.html` (visited on 03/20/2020).

[Kim+19] Daehyeok Kim et al. "FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds". In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation.* NSDI. NSDI. Boston, MA, USA, 2019, pp. 113–125. ISBN: 978-1-931971-49-2. DOI: `10.5555/3323234.3323245`.

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *ACM SIGPLAN Notices* 28.10 (Oct. 1, 1993), pp. 91–108. ISSN: 0362-1340. DOI: `10/cgnqf7`.

[KKA14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-Value Services". In: *Proceedings of the 2014 ACM Conference on SIGCOMM - SIGCOMM '14.* The 2014 ACM Conference. Chicago, Illinois, USA: ACM Press, 2014, pp. 295–306. ISBN: 978-1-4503-2836-4. DOI: `10.1145/2619239.2626299`.

[KKA16] Anuj Kalia, Michael Kaminsky, and David G Andersen. "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.* OSDI. Savannah, GA, USA: USENIX Association, Nov. 2–4, 2016. ISBN: ISBN 978-1-931971-33-1.

[KKA19] Anuj Kalia, Michael Kaminsky, and David G Andersen. "Datacenter RPCs Can Be General and Fast". In: *16th USENIX Symposium on Networked Systems Design and Implementation.* NSDI. Boston, MA: USENIX Association, 2019. ISBN: 978-1-931971-49-2. URL: `https://www.usenix.org/conference/nsdi19/presentation/kalia`.

[Kle+09] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles.* SOSP '09. New York, NY, USA: Association for Computing Machinery, Oct. 11, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629596`.

[KN14] Mike Ko and Alexander Nezhinsky. *Internet Small Computer System Interface (iSCSI) Extensions for the Remote Direct Memory Access (RDMA) Specification.* 2014. URL: `https://datatracker.ietf.org/doc/html/rfc7145` (visited on 10/19/2021).

[Koc+19a] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy.* SP. San Francisco, CA, USA: IEEE, 2019. DOI: `10.1109/SP.2019.00002`.

[Koc+19b] Alec Kochevar-Cureton, Somesh Chaturmohta, Norman Lam, Sambhrama Mundkur, and Daniel Firestone. "Remote Direct Memory Access in Computing Systems". U.S. pat. 10437775B2. Microsoft Technology Licensing LLC. Oct. 8, 2019. URL: `https://patents.google.com/patent/US10437775B2/en` (visited on 04/14/2020).

[KRA20] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. "Do OS Abstractions Make Sense on FPGAs?" In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation.* OSDI. USENIX Association, 2020. ISBN: 978-1-939133-19-9. URL: `https://www.usenix.org/conference/osdi20/presentation/roscoe`.

[KS09] Asim Kadav and Michael M. Swift. "Live Migration of Direct-Access Devices". In: *ACM SIGOPS Operating Systems Review* 43.3 (July 31, 2009), p. 95. ISSN: 01635980. DOI: `10/b9j36z`.

[KSB17] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific Containers for Mobility of Compute". In: *PLOS ONE* 12.5 (May 11, 2017), e0177459. ISSN: 1932-6203. DOI: `10/f969fz`.

[Kue+21] Simon Kuenzer et al. "Unikraft: Fast, Specialized Unikernels the Easy Way". In: *Proceedings of the Sixteenth European Conference on Computer Systems.* EuroSys '21: Sixteenth European Conference on Computer Systems. Online Event United Kingdom: ACM, Apr. 21, 2021, pp. 376–394. ISBN: 978-1-4503-8334-9. DOI: `10.1145/3447786.3456248`.

[Kul23] Alex Kuleshov. *Linux Inside.* 2023. URL: `https://0xax.gitbooks.io/linux-insides/content/` (visited on 10/21/2023).

[Lac04] Adam Lackorzynski. "L4Linux Porting Optimizations". MA thesis. Dresden, Germany: TU Dresden, 2004. URL: `http://debian2.inf.tu-dresden.de/papers_ps/adam-diplom.pdf`.

[LD23] Dan Lenoski and Nandita Dukkipati. *Introducing Falcon: A Reliable Low-Latency Hardware Transport.* Google Cloud Blog. 2023. URL: `https://cloud.google.com/blog/topics/systems/introducing-falcon-a-reliable-low-latency-hardware-transport` (visited on 02/20/2024).

[Lef+21] Hugo Lefeuvre et al. "FlexOS: Towards Flexible OS Isolation". Dec. 13, 2021. arXiv: `2112.06566 [cs]`. URL: `http://arxiv.org/abs/2112.06566` (visited on 12/14/2021).

[Les+17] Ilya Lesokhin et al. "Page Fault Support for Network Controllers". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '17: Architectural Support for Programming Languages and Operating Systems. Xi'an China: ACM, Apr. 4, 2017, pp. 449–466. ISBN: 978-1-4503-4465-4. DOI: `10/ghm5x7`.

[Li+15] Mingzhe Li, Hari Subramoni, Khaled Hamidouche, Xiaoyi Lu, and Dhabaleswar K. Panda. "High Performance MPI Datatype Support with User-Mode Memory Registration: Challenges, Designs, and Benefits". In: *2015 IEEE International Conference on Cluster Computing.* 2015 IEEE International Conference on Cluster Computing. Sept. 2015, pp. 226–235. DOI: `10.1109/CLUSTER.2015.41`.

[Li+19] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. "Socksdirect: Datacenter Sockets Can Be Fast and Compatible". In: *Proceedings of the ACM Special Interest Group on Data Communication.* SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, Aug. 19, 2019, pp. 90–103. ISBN: 978-1-4503-5956-6. DOI: `10/ghq8sw`.

[Lin16] Linux Foundation. *Networking:Toe [Wiki].* 2016. URL: `https://wiki.linuxfoundation.org/networking/toe` (visited on 05/17/2021).

[Lip+20] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *Communications of the ACM* 63.6 (May 21, 2020), pp. 46–56. ISSN: 0001-0782, 1557-7317. DOI: `10.1145/3357033`.

[Lis13] Liran Liss. “On Demand Paging for User-level Networking”. OpenFabrics Workshop. 2013. URL: https://openfabrics.org/images/eventpresos/workshops2013/2013_Workshop_Tues_0930_liss_odp.pdf.

[Lis17] Liran Liss. “The Linux SoftRoCE Driver”. 13th Annual OpenFabrics Alliance Workshop (Austin, TX. U.S.A). Mar. 2017. URL: https://youtu.be/NumH5YeVjHU?t=45 (visited on 08/11/2020).

[Liu+06] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. “High Performance VMM-Bypass I/O in Virtual Machines”. In: USENIX ATC. ATC ’06. Boston, MA, USA, May 2006. DOI: 10.5555/1267359.1267362.

[Lu+18] Yuanwei Lu et al. “Multi-Path Transport for RDMA in Datacenters”. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). 2018, pp. 357–371. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/lu (visited on 02/23/2024).

[Mah+14] Mallik Mahalingam et al. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks.* Request for Comments RFC 7348. Internet Engineering Task Force, Aug. 2014. 22 pp. DOI: 10.17487/RFC7348.

[Mar+19] Michael Marty et al. “Snap: A Microkernel Approach to Host Networking”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles - SOSP ’19*. The 27th ACM Symposium. Huntsville, Ontario, Canada: ACM Press, 2019, pp. 399–413. ISBN: 978-1-4503-6873-5. DOI: 10/ggcdnx.

[MDW99] Dejan Milojičić, Frederick Douglis, and Richard Wheeler. *Mobility: Processes, Computers, and Agents.* USA: ACM Press/Addison-Wesley Publishing Co., 1999. 1208 pp. ISBN: 978-0-201-37928-0.

[Mei23] Ilya Meignan–Masson. “Bridging the Performance Gap Between Converged RDMA Dataplane and Kernel-Bypass”. MA thesis. 2023.

[Mel15] Mellanox Technologies. *RDMA Aware Networks Programming User Manual.* 1.7. Mellanox Technologies, 2015, p. 216.

[Mel16] Mellanox Technologies. *Mellanox Adapters Programmer’s Reference Manual (PRM).* MLNX-15-4845. 2016, p. 316. URL: https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf.

[Mel18] Mellanox. *Accelerated Switching And Packet Processing (ASAP2): Hardware Offloading for vSwitches.* Rev 4.4. User Manual. Sunnyvale, CA, USA, 2018, p. 19. URL: http://www.mellanox.com/related-docs/prod_software/ASAP2_Hardware_Offloading_for_vSwitches_User_Manual_v4.4.pdf (visited on 05/02/2019).

[Mel19a] Mellanox. *Messaging Accelerator (VMA) Documentation.* Mellanox, Apr. 30, 2019, p. 137. URL: https://docs.mellanox.com/display/VMAv883 (visited on 04/27/2020).

[Mel19b] Mellanox Technologies. *BlueField Software.* Documentation v2.1.0.10924. 2019.

[Mer14] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment”. In: *Linux Journal* (Mar. 2014). URL: https://dl.acm.org/citation.cfm?id=2600241 (visited on 10/17/2019).

[MHJ20] David S. Miller, Richard Henderson, and Jakub Jelinek. *Dynamic DMA Mapping Guide — The Linux Kernel Documentation*. Dynamic DMA mapping Guide. 2020. URL: `https://docs.kernel.org/core-api/dma-api-howto.html` (visited on 02/15/2024).

[Mi+19] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. "SkyBridge: Fast and Secure Inter-Process Communication for Microkernels". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19: Fourteenth EuroSys Conference 2019. Dresden Germany: ACM, Mar. 25, 2019, pp. 1–15. ISBN: 978-1-4503-6281-8. DOI: `10/gf67jp`.

[Mie+06] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. "Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack". In: *Euro-Par 2006 Parallel Processing*. Ed. by Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner. Red. by David Hutchison et al. Vol. 4128. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 124–133. ISBN: 978-3-540-37783-2 978-3-540-37784-9. DOI: `10.1007/11823285_13`.

[Mie+22] Till Miemietz, Maksym Planeta, Viktor Reusch, Jan Bierbaum, Michael Roitzsch, and Hermann Härtig. "Fast Privileged Function Calls". In: *Systems for Post-Moore Architectures*. The 11th Workshop on Systems for Post-Moore Architectures. Rennes, France, 2022. URL: `https://www.barkhauseninstitut.org/fileadmin/user_upload/Publikationen/2022/202204_Miemietz_SPMA_FastCalls.pdf`.

[MKK08] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. "Containers Checkpointing and Live Migration". In: Linux Symposium. Ottawa, 2008.

[MM23] Matt McInnes and Matthew McGovern. *Microsoft Azure Network Adapter (MANA) Overview*. Aug. 21, 2023. URL: `https://learn.microsoft.com/en-us/azure/virtual-network/accelerated-networking-mana-overview` (visited on 02/20/2024).

[Mog03] Jeffrey C Mogul. "TCP Offload Is a Dumb Idea Whose Time Has Come". In: *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*. HotOS. Lihue, Hawaii, USA: USENIX Association, 2003. URL: `https://www.usenix.org/legacy/events/hotos03/tech/full_papers/mogul/mogul.pdf`.

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

[Mot+21] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. "Simurgh: A Fully Decentralized and Secure NVMM User Space File System". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. New York, NY, USA: Association for Computing Machinery, Nov. 14, 2021, pp. 1–14. ISBN: 978-1-4503-8442-1. DOI: `10/gn4bjw`.

[Mou+17] Angelas Mouzakitis, Christian Pinto, Nikolay Nikolaev, Alvise Rigo, Daniel Raho, Babis Aronis, and Manolis Marazakis. “Lightweight and Generic RDMA Engine Para-Virtualization for the KVM Hypervisor”. In: *2017 International Conference on High Performance Computing & Simulation (HPCS)*. 2017 International Conference on High Performance Computing & Simulation (HPCS). Genoa, Italy: IEEE, July 2017, pp. 737–744. ISBN: 978-1-5386-3249-9 978-1-5386-3250-5. DOI: 10/ggscxk.

[MPI15] MPI Forum. *MPI: A Message-Passing Interface Standard*. 3.1. Message Passing Interface Forum, June 4, 2015, p. 868.

[MSB22] Nafiseh Moti, Reza Salkhordeh, and André Brinkmann. “Protected Functions: User Space Privileged Function Calls”. In: *Architecture of Computing Systems*. Ed. by Martin Schulz, Carsten Trinitis, Nikela Papadopoulou, and Thilo Pionteck. Vol. 13642. Cham: Springer International Publishing, 2022, pp. 117–131. ISBN: 978-3-031-21866-8 978-3-031-21867-5. DOI: 10.1007/978-3-031-21867-5_8.

[MYL17] Lele Ma, Shanhe Yi, and Qun Li. “Efficient Service Handoff across Edge Servers via Docker Container Migration”. In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. The Second ACM/IEEE Symposium. SEC ’17. San Jose, California: ACM Press, 2017, pp. 1–13. ISBN: 978-1-4503-5087-7. DOI: 10/gf9x9r.

[Nad+17] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. “Voyager: Complete Container State Migration”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). Atlanta, GA, USA: IEEE, June 2017, pp. 2137–2142. ISBN: 978-1-5386-1792-2. DOI: 10/ggnhq5.

[NEX22] NEX Cloud Networking Group. *Intel ® Ethernet Controller E810*. Version 2.5. 613875-007. Intel, 2022. URL: https://www.intel.com/content/www/us/en/content-details/613875/intel-ethernet-controller-e810-datasheet.html.

[Niu+19] Zhixiong Niu, Hong Xu, Peng Cheng, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. “NetKernel: Making Network Stack Part of the Virtualized Infrastructure”. Mar. 19, 2019. arXiv: 1903.07119 [cs]. URL: http://arxiv.org/abs/1903.07119 (visited on 03/08/2020).

[Nja+22] Anton Njavro, James Tau, Taylor Groves, Nicholas J. Wright, and Richard West. *A DPU Solution for Container Overlay Networks*. Nov. 18, 2022. arXiv: 2211.10495 [cs]. URL: http://arxiv.org/abs/2211.10495 (visited on 04/13/2023). preprint.

[NLH05] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. “Fast Transparent Migration for Virtual Machines”. In: *Proceedings of the USENIX Annual Technical Conference*. ATC ’05. Anaheim, CA, USA: USENIX Association, Apr. 2005, pp. 391–394. DOI: 10.5555/1247360.1247385.

[NR18] Joel Nider and Mike Rapoport. “News from Academia: FatELF, RDMA and CRIU”. In: Linux Plumbers Conference. Vancouver, BC, Canada, Nov. 13, 2018. URL: https://linuxplumbersconf.org/event/2/contributions/205/attachments/30/31/FatELF_RDMA_and_CRIU.pdf (visited on 05/06/2021).

[NVI20] NVIDIA. *Mellanox Innova-2 Flex Open Programmable SmartNIC*. 2020. URL: https://network.nvidia.com/files/doc-2020/pb-innova-2-flex.pdf (visited on 12/05/2023).

[NVI22] NVIDIA Corporation. *ConnectX-7 400G Adapters*. 2022. URL: https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471.

[NVI23] NVIDIA. *NVIDIA® OpenFabrics Enterprise Distribution for Linux Documentation*. v23.10-1.1.9.0 LTS. 2023. URL: https://docs.nvidia.com/networking/display/MLNXOFEDv23101190LTS (visited on 01/10/2024).

[NVI24] NVIDIA. *GPUDirect RDMA*. Release 12.3. NVIDIA, 2024, p. 48. URL: https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf (visited on 02/19/2024).

[Ope22] OpenMPI. *Open MPI v5.0.x — Open MPI 5.0.x Documentation*. 2022. URL: https://docs.open-mpi.org/en/v5.0.x/ (visited on 12/08/2022).

[Osa+17] Opeyemi Osanaiye, Shuo Chen, Zheng Yan, Rongxing Lu, Kim-Kwang Raymond Choo, and Mqhele Dlodlo. "From Cloud to Fog Computing: A Review and a Conceptual Live VM Migration Framework". In: *IEEE Access* 5 (2017), pp. 8284–8300. ISSN: 2169-3536. DOI: 10/ggnfkt.

[Osm+03] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. "The Design and Implementation of Zap: A System for Migrating Computing Environments". In: *ACM SIGOPS Operating Systems Review* 36 (SI Dec. 31, 2003), pp. 361–376. ISSN: 0163-5980. DOI: 10/fbg7vq.

[Ous+19] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads". In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. Boston, MA, USA, 2019.

[Pan+12] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. "CompSC: Live Migration with Pass-through Devices". In: *ACM SIGPLAN Notices* 47.7 (Sept. 5, 2012), p. 109. ISSN: 03621340. DOI: 10/f3887q.

[Pan23] Dhabaleswar K. Panda. *OSU Micro-Benchmarks*. Ohio, USA: The Ohio State University, 2023. URL: http://mvapich.cse.ohio-state.edu/benchmarks/ (visited on 12/04/2023).

[Par+19] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. "Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel {MPK})". In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). 2019, pp. 241–254. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/park-soyeon (visited on 03/28/2023).

[Pat+18] Connor Patrick, Hearn R. James, Dubal P. Scott, Herdrich J. Andrew, and Sood Kapil. "Techniques to Migrate a Virtual Machine Using Disaggregated Computing Resources". Pat. 15/635,124. 2018. URL: https://patents.google.com/patent/US20180373553A1/en.

[PCI10] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification*. 1.1. PCI-SIG, Jan. 20, 2010, p. 100.

[PCI19] PCI-SIG. *PCI Express® Base Specification Revision 5.0 Version 1.0*. Specification. PCI-SIG, May 22, 2019, p. 1299.

[per20] perftest. *Perftest*. perftest. Apr. 10, 2020. URL: `https://github.com/linux-rdma/perftest` (visited on 04/13/2020).

[Pet+01] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. “Hardware- and Software-Based Collective Communication on the Quadrics Network”. In: *Proceedings IEEE International Symposium on Network Computing and Applications. NCA 2001*. IEEE International Symposium on Network Computing and Applications. NCA 2001. Cambridge, MA, USA: IEEE Comput. Soc, 2001, pp. 24–35. ISBN: 978-0-7695-1432-1. DOI: `10.1109/NCA.2001.962513`.

[Pet+14] Simon Peter et al. “Arrakis: The Operating System Is the Control Plane”. In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014*. USENIX Symposium on Operating Systems Design and Implementation. OSDI ’14. Broomfield, CO, USA: USENIX Association, 2014, pp. 2–17. ISBN: 978-1-931971-16-4.

[Pfe+15] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. “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*. VEE. VEE. Istanbul, Turkey: ACM Press, 2015, pp. 17–30. ISBN: 978-1-4503-3450-1. DOI: `10/ggscxm`.

[Pic+16] S. Pickartz, C. Clauss, S. Lankes, S. Krempel, T. Moschny, and A. Monti. “Non-Intrusive Migration of MPI Processes in OS-Bypass Networks”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 2016, pp. 1728–1735. DOI: `10/ggscxh`.

[PKP03] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. “The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q”. In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC ’03: International Conference for High Performance Computing, Networking, Storage and Analysis. Phoenix AZ USA: ACM, Nov. 15, 2003, p. 55. ISBN: 978-1-58113-695-1. DOI: `10.1145/1048935.1050204`.

[Pla+21] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. “MigrOS: Transparent Operating Systems Live Migration Support for Containerised RDMA-applications”. In: *USENIX ATC 2021*. July 14, 2021, pp. 47–63. ISBN: 978-1-939133-23-6. URL: `https://www.usenix.org/conference/atc21/presentation/planeta`.

[Pla+23] Maksym Planeta, Jan Bierbaum, Michael Roitzsch, and Hermann Härtig. *CoRD: Converged RDMA Dataplane for High-Performance Clouds*. Sept. 2, 2023. DOI: `10.48550/arXiv.2309.00898`. arXiv: `2309.00898 [cs]`. preprint.

[POS17] POSIX 2017. *POSIX.1-2017*. IEEE, 2017. DOI: `10.1109/IEEESTD.2018.8277153`.

[PPG18] Ashish Panwar, Aravinda Prasad, and K. Gopinath. “Making Huge Pages Actually Useful”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18: Architectural Support for Programming Languages and Operating Systems. Williamsburg VA USA: ACM, Mar. 19, 2018, pp. 679–692. ISBN: 978-1-4503-4911-6. DOI: `10.1145/3173162.3173203`.

[Psi+22] Antonis Psistakis et al. “Optimized Page Fault Handling During RDMA”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (Dec. 1, 2022), pp. 3990–4005. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2022.3175666.

[Rad+14] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. “SENIC: Scalable NIC for End-Host Rate Limiting”. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). 2014, pp. 475–488. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/radhakrishnan (visited on 01/10/2024).

[Ray03] Eric S. Raymond. *Fork Bomb*. In: *The On-Line Hacker Jargon File*. 4.4.7. 2003. URL: http://catb.org/~esr/jargon/html/F/fork-bomb.html (visited on 11/15/2023).

[Roi+23] Michael Roitzsch, Till Miemietz, Christian Von Elm, and Nils Asmussen. “Software-Defined CPU Modes”. In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS ’23: 19th Workshop on Hot Topics in Operating Systems. Providence RI USA: ACM, June 22, 2023, pp. 23–29. ISBN: 9798400701955. DOI: 10.1145/3593856.3595894.

[Ros11] Steven Rostedt. *Using the TRACE_EVENT() Macro (Part 1)*. LWN.net. 2011. URL: https://lwn.net/Articles/379903/ (visited on 12/24/2023).

[Ros17] Alex Rosenbaum. *[PATCH RFC v2] Introduce Verbs API for Traffic and Event Counters — Linux RDMA and InfiniBand Development*. 2017. URL: https://www.spinics.net/lists/linux-rdma/msg58579.html (visited on 12/24/2023).

[Rot+21] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. “ReDMArk: Bypassing RDMA Security Mechanisms”. In: *30th USENIX Security Symposium*. USENIX Security 21. Vancouver, B.C., 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger.

[Rou+09] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. “Adagio: Making DVS Practical for Complex HPC Applications”. In: *Proceedings of the 23rd International Conference on Conference on Supercomputing - ICS ’09*. The 23rd International Conference. New York, NY, USA: ACM Press, 2009, p. 460. ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542340.

[run20] runc. *Runc: CLI Tool for Spawning and Running Containers According to the OCI Specification*. runc. 2020. URL: https://github.com/opencontainers/runc (visited on 03/03/2020).

[RW18] Alex Rosenbaum and Bodong Wang. *Ibv_modify_qp_rate_limit(3)*. 2018. URL: https://man7.org/linux/man-pages/man3/ibv_modify_qp_rate_limit.3.html (visited on 01/10/2024).

[Sad+21] Hugo Sadok et al. “We Need Kernel Interposition over the Network Dataplane”. In: HotOS. Ann Arbor, MI, USA, 2021.

[SB05] Sven Schneider and Robert Baumgartl. “Unintrusively Measuring Linux Kernel Execution Times”. In: 7th RTL Workshop. 2005. 2005, p. 5.

[Sch21] Philipp Schuster. “Overlay-RDMA-Netzwerke im Kontext von Live-Migration”. Großer Beleg. Dresden, Germany: TU Dresden, 2021.

[SE19] Oz Shlomo and Rony Efrayim. “Hardware Offloads: Past, Present and Future”. OvSConf (Westford, MA). 2019. URL: `https://www.openvswitch.org/support/ovscon2019/day2/0951-hw_offload_ovs_con_19-Oz-Mellanox.pdf` (visited on 10/11/2023).

[Sen+20] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. “An In-Depth Analysis of the Slingshot Interconnect”. Aug. 20, 2020. arXiv: `2008.08886 [cs]`. URL: `http://arxiv.org/abs/2008.08886` (visited on 12/18/2020).

[Sen+22] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefler. *Noise in the Clouds: Influence of Network Performance Variability on Application Scalability.* Nov. 1, 2022. arXiv: `2210.15315 [cs]`. URL: `http://arxiv.org/abs/2210.15315` (visited on 11/03/2022). preprint.

[Sha+15] Pavel Shamis et al. “UCX: An Open Source Framework for HPC Network APIs and Beyond”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects.* HotI. HotI. Aug. 2015, pp. 40–43. DOI: `10/ggmx8k`.

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

[Sid+15] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. “Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware”. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines.* 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. May 2015, pp. 36–43. DOI: `10.1109/FCCM.2015.12`.

[Sid+20] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. “StRoM: Smart Remote Memory”. In: *The Fifteenth European Conference on Computer Systems.* EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020, p. 16. DOI: `10.1145/3342195.3387519`.

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

[Sin+20] Arjun Singhvi et al. “1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters”. In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication.* SIGCOMM ’20: Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication. Virtual Event USA: ACM, July 30, 2020, pp. 708–721. ISBN: 978-1-4503-7955-7. DOI: `10/ghhqtr`.

[Sin20] Pradeep Sindhu. “The Fungible DPU: A New Category of Microprocessor for the Data-Centric Era”. Hot Chips 2020. 2020. URL: `https://hc32.hotchips.org/assets/program/conference/day2/HotChips2020_Networking_Fungible_v04.pdf` (visited on 10/11/2023).

[Smi88] Jonathan M. Smith. “A Survey of Process Migration Mechanisms”. In: *ACM SIGOPS Operating Systems Review* 22.3 (July 1, 1988), pp. 28–40. ISSN: 0163-5980. DOI: `10/bjp787`.

[STJ03] J.R. Santos, Y. Turner, and G. Janakiraman. “End-to-End Congestion Control for Infiniband”. In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428).* IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies. San Francisco, CA: IEEE, 2003, 1123–1133 vol.2. ISBN: 978-0-7803-7752-3. DOI: `10.1109/INFCOM.2003.1208949`.

[Su+17] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. “RFP: When RPC Is Faster than Server-Bypass with RDMA”. In: *Proceedings of the Twelfth European Conference on Computer Systems.* EuroSys ’17. New York, NY, USA: Association for Computing Machinery, Apr. 23, 2017, pp. 1–15. ISBN: 978-1-4503-4938-3. DOI: `10.1145/3064176.3064189`.

[Sve21] Peter Sven. *Apple Silicon Hardware Secrets: SPRR and Guarded Exception Levels (GXF).* May 6, 2021. URL: `https://blog.svenpeter.dev/posts/m1_sprr_gxf/` (visited on 11/01/2021).

[Tar+20] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. “sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access”. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). 2020, pp. 691–704. ISBN: 978-1-939133-14-4. URL: `https://www.usenix.org/conference/atc20/presentation/taranov` (visited on 04/29/2021).

[TDH21] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. “CoRM: Compactable Remote Memory over RDMA”. In: *Proceedings of the 2021 International Conference on Management of Data.* SIGMOD/PODS ’21: International Conference on Management of Data. Virtual Event China: ACM, June 9, 2021, pp. 1811–1824. ISBN: 978-1-4503-8343-1. DOI: `10.1145/3448016.3452817`.

[TFH22] Konstantin Taranov, Fabian Fischer, and Torsten Hoefler. *Efficient RDMA Communication Protocols.* Dec. 20, 2022. arXiv: `2212.09134 [cs]`. URL: `http://arxiv.org/abs/2212.09134` (visited on 01/18/2023). preprint.

[The23] The Netfilter Core Team. *The Netfilter.Org Project.* The netfilter.org project. 2023. URL: `https://nftables.org` (visited on 10/10/2023).

[Tig23] Tigera Inc. *eBPF Use Cases | Calico Documentation.* 2023. URL: `https://docs.tigera.io/calico/latest/operations/ebpf/use-cases-ebpf` (visited on 05/29/2023).

[TMS11] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. “A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity”. In: *Proceedings of the Second Asia-Pacific Workshop on Systems - APSys ’11.* The Second Asia-Pacific

Workshop. Shanghai, China: ACM Press, 2011, p. 1. ISBN: 978-1-4503-1179-3. DOI: `10/fzv576`.

[TOP22] TOP500.org. *TOP500 (November 2022)*. Nov. 2022. URL: `https://www.top500.org/lists/top500/2022/11/` (visited on 03/14/2023).

[TRG05] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. "Optimization of Collective Communication Operations in MPICH". In: *The International Journal of High Performance Computing Applications* 19.1 (Feb. 2005), pp. 49–66. ISSN: 1094-3420, 1741-2846. DOI: `10.1177/1094342005051521`.

[Tsa+05] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications". In: *Proceedings of the 19th Annual International Conference on Supercomputing - ICS '05*. The 19th Annual International Conference. Cambridge, Massachusetts: ACM Press, 2005, p. 303. ISBN: 978-1-59593-167-2. DOI: `10/dhskhz`.

[TZ17] Shin-Yeh Tsai and Yiying Zhang. "LITE Kernel RDMA Support for Datacenter Applications". In: *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*. The 26th Symposium. Shanghai, China: ACM Press, 2017, pp. 306–324. ISBN: 978-1-4503-5085-3. DOI: `10/ggscxn`.

[Van+19] Stephan Van Schaik et al. "RIDL: Rogue In-Flight Data Load". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2019, pp. 88–105. ISBN: 978-1-5386-6660-9. DOI: `10.1109/SP.2019.00087`.

[Ven+15] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar (DK) Panda, Darren Kerbyson, and Adolfy Hoisie. "A Case for Application-Oblivious Energy-Efficient MPI Runtime". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC15: The International Conference for High Performance Computing, Networking, Storage and Analysis. Austin Texas: ACM, Nov. 15, 2015, pp. 1–12. ISBN: 978-1-4503-3723-6. DOI: `10.1145/2807591.2807658`.

[VIT98] VITA 26. *American National Standard for Myrinet-on-VME Protocol Specification*. Standard ANSI/VITA 26-1998. ANSI, 1998. URL: `https://www.vita.com/Standards` (visited on 05/08/2022).

[vRie14] Rik van Riel. "Automatic NUMA Balancing" (San Francisco, CA, USA). Oct. 29, 2014. URL: `https://www.youtube.com/watch?v=mjVw_oe1hEA` (visited on 02/15/2024).

[VYC05] Amit Vasudevan, Ramesh Yerraballi, and Ashish Chawla. "A High Performance Kernel-Less Operating System Architecture". In: (2005), p. 10.

[Wan+19] Dongyang Wang, Binzhang Fu, Gang Lu, Kun Tan, and Bei Hua. "vSocket: Virtual Socket Interface for RDMA in Public Clouds". In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE. VEE. Providence, RI, USA: ACM Press, 2019, pp. 179–192. ISBN: 978-1-4503-6020-3. DOI: `10/ggscxg`.

[Wan+23] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Yang Kong, Wei Wang, and Qinfen Hao. "CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers". In: (2023).

[Wei+21] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. “KRCORE: A Microsecond-Scale RDMA Control Plane for Elastic Computing”. Dec. 28, 2021. arXiv: `2201.11578 [cs]`. URL: `http://arxiv.org/abs/2201.11578` (visited on 01/28/2022).

[WHW19] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. “Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys ’19. Dresden, Germany: Association for Computing Machinery, Mar. 25, 2019, pp. 1–16. ISBN: 978-1-4503-6281-8. DOI: `10/ggnq76`.

[WLH19] Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. “FFMK: An HPC OS Based on the L4Re Microkernel”. In: *Operating Systems for Supercomputers and High Performance Computing*. Ed. by Balazs Gerofi, Yutaka Ishikawa, Rolf Riesen, and Robert W. Wisniewski. Singapore: Springer Singapore, 2019, pp. 335–357. ISBN: 9789811366246. DOI: `10.1007/978-981-13-6624-6_19`.

[WPM99] David Wong, Noemi Paciorek, and Dana Moore. “Java-Based Mobile Agents”. In: *Communications of the ACM* 42.3 (Mar. 1, 1999), 92–ff. ISSN: 0001-0782. DOI: `10/btg3k7`.

[Xil22] Xilinx. *Xilinx Embedded RDMA Enabled NIC LogiCORE IP Product Guide*. 4.0. AMD, 2022. URL: `https://docs.xilinx.com/r/en-US/pg332-ernic/Xilinx-Embedded-RDMA-Enabled-NIC-v4.0-LogiCORE-IP-Product-Guide` (visited on 07/12/2023).

[Yan+17] Ziye Yang et al. “SPDK: A Development Kit to Build High Performance Storage Applications”. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). Dec. 2017, pp. 154–161. DOI: `10.1109/CloudCom.2017.14`.

[Zha+19] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. “I’m Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS. 2019, p. 8.

[Zha+21] Irene Zhang et al. “The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. New York, NY, USA: Association for Computing Machinery, Oct. 26, 2021, pp. 195–211. ISBN: 978-1-4503-8709-5. DOI: `10/gnkjj5`.

[Zha+22] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. “Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks”. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). 2022, pp. 1307–1326. ISBN: 978-1-939133-27-4. URL: `https://www.usenix.org/conference/nsdi22/presentation/zhang-yiwen` (visited on 07/10/2023).

[Zhe+13] Fang Zheng et al. “GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC13: International Conference for High Performance Computing, Networking, Storage and Analysis. Denver Colorado: ACM, Nov. 17, 2013, pp. 1–12. ISBN: 978-1-4503-2378-9. DOI: `10.1145/2503210.2503279`.

[Zhu+15] Yibo Zhu et al. "Congestion Control for Large-Scale RDMA Deployments". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication.* SIGCOMM. SIGCOMM. London, UK: ACM, 2015, pp. 523–536. ISBN: 978-1-4503-3542-3. DOI: `10.1145/2785956.2787484`.

[Zhu+19] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. "Slim: OS Kernel Support for a Low-Overhead Container Overlay Network". In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation.* NSDI. NSDI. Boston, MA, USA: USENIX Association, Feb. 2019. ISBN: 978-1-931971-49-2.

[Zil+14] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. "NetFPGA SUME: Toward 100 Gbps as Research Commodity". In: *IEEE Micro* 34.5 (Sept. 2014), pp. 32–41. ISSN: 1937-4143. DOI: `10/gg8qsd`.

A ChatGPT Prompt

To improve the quality of presentation, I created a custom GPT prompt with the following instructions. The prompt was used to enhance the grammar and stylistic presentation of the text in this document.

> 'Latex Grammar Fixer' is meticulously designed to correct grammar in LaTeX-formatted scientific texts, with a paramount emphasis on preserving the exact formatting from the user's input, including line breaks. Your essential role is to ensure grammatical accuracy while maintaining every element of the user's LaTeX formatting - commands, spacing, alignment, special characters, and notably, line breaks. When users submit LaTeX texts, your objective is to apply grammatical corrections only, without modifying any part of the LaTeX formatting or structure. This commitment includes preserving the original form and placement of LaTeX commands like `\ac`, `\acp`, or `\cite`, as well as maintaining the exact line breaks as in the original input. The aim is to deliver a grammatically polished version of the text that mirrors the user's original formatting in every detail.

B Data Archive

The data archive for this thesis is available at the following URLs.

`https://doi.org/10.5281/zenodo.10947946`

or

`https://opara.zih.tu-dresden.de//handle/123456789/559`

This archive contains the source code, scripts, and data used in this thesis. The directory comprises the data from three projects: MigrOS, CoRD, and Fastcalls.