Master of Science in Informatics at Grenoble
Master Informatique
Specialization Distributed Infrastructures

Bridging the Performance Gap Between Converged RDMA Dataplane and Kernel-Bypass

Ilya Meignan--Masson

September 4^{th}, 2023

Research project performed at Barkhausen Institut

Under the supervision of:
Michael Roitzsch and Maksym Planeta

Defended before a jury composed of:
Bruno Raffin
Martin Heusse
Olivier Richard
Thomas Ropars

September 2023

Abstract

While kernel-bypass is often considered an essential feature of high performance networks, it breaks the traditional OS architecture and prevents a fine-grain control of the kernel over the communications. Converged RDMA Dataplane (CoRD) is an attempt at solving this problem by giving back the control over dataplane operations to the kernel effectively enabling a better control and utilisation of the network resource. However, CoRD suffers from a massive overhead compared to its kernel-bypass counter-part. CoRD was implemented by porting the user library code inside the Infiniband driver and thus their performance should be equivalent but our measurements show that the overhead exceeds the delay imposed by a system call. In this work, we identified the causes of this overhead and improved the driver performance by up to 65% to reach a sub-microsecond overhead compared to kernel-bypass.

Acknowledgements

I would like to thank all the people that contributed directly or indirectly to this work :

- Maksym Planeta, for his invaluable and seemingly limitless supply of advice and directions both during the research and for the writing of this thesis.
- Michael Roitzsch, Olivier Richard and the other members of the jury.
- the members of the TUD-OS team and especially Tianhao Wang for the nice and interesting discussions.
- my friends and family for the long distance support.
- and a very special candle that brightens my life even across continents.

Résumé

Bien que le contournement du noyau (kernel-bypass) soit souvent considéré comme une caractéristique essentielle des réseaux à haute performance, ce mecanisme ne rentre pas dans l'architecture traditionnelle d'un système d'exploitation et empêche le noyau de contrôler avec précision les communications. Converged RDMA Dataplane (CoRD) tente de résoudre ce problème en redonnant au noyau le contrôle des opérations du dataplane, ce qui permet un meilleur contrôle et une meilleure utilisation des ressources du réseau. Cependant, CoRD souffre d'un surcoût important par rapport à la version implementant le contournement du noyau. CoRD ayant été implémenté en portant le code de la bibliothèque utilisateur à l'intérieur du pilote Infiniband, leurs performances devraient être équivalentes. Cependant nos expérimentations montrent que le surcoût dépasse le délai normalement imposé par un appel système. Dans ce travail, nous avons identifié les causes de cet overhead et amélioré jusqu'à 65% les performances du pilote pour atteindre un surcoût inférieur à la microseconde par rapport au kernel-bypass.

Contents

— 1 —
Introduction

Infiniband is a networking standard widely used in clusters nowadays for its very low latency and high throughput, in the order of micro-seconds for the latency and in the hundreds of giga bits per second for the bandwidth. The current user-level library relies on kernel-bypass to reach this level of performance. However, as High Performance Computing and Cloud architectures and especially network are getting closer, enabling efficient OS-level control over networking dataplane operations seems mandatory to reach this convergence. To do so, Converged RDMA Dataplane (CoRD) [22] has been proposed to put the kernel back in the control path of dataplane operations through a system call and a kernel-level driver.
As Miemietz et al. showed, a normal system call takes around 300 ns. Since CoRD's implementation simply ports the user-level pipeline into the kernel, the only performance overhead should come from the system call. However, we measured that CoRD has an overhead of up to 2 micro-seconds compared to the library with kernel-bypass. In this work, we try to explain the difference between the overhead of a system call and the overhead of CoRD and to eliminate what causes the extra overhead.
To do so, we analyse the performance of micro-benchmarks looking for a source of overhead. Once identified, we remove it and we restart the analysis until we find all the sources. Using this methodology, we identified and removed 4 parts of the pipeline that were causing a performance degradation resulting in an improvement of up to 65% of the overhead. The first part includes some vulnerabilities mitigations, which, when disabled, remove a significant overhead on the system call performance of affected systems. The second part covers the copying of user-level memory into the kernel that can be avoided using a shared memory region. The third part simplifies the dispatching of functions inside the driver as its architecture, which hasn't been designed for this, was inducing a performance overhead. The final parts improves the mechanism of passing one of the parameter by keeping the kernel pointer to the object in the user-level library.
With those optimisations, the overhead of CoRD is reduced to 600-800 ns which means that there is still an extra overhead compared to a system call. We believe that removing this extra overhead would require to re-design the kernel Infiniband driver for this purpose. Another option could be to try to reduce the overhead of the system call itself but this would also require a re-architecting of the system call interface.
This thesis is organised as follow. First, we provide some background on the Linux kernel and on the Infiniband standard in chapter 2. Then we present the problem and detail the related work in chapter 3. Chapter 4 details our exploration of the kernel driver performance while chapter 5 presents our evaluation and the future work. Finally we conclude in chapter 6.

— 2 —
Background

2.1 Background

2.1.1 The Linux kernel

One of the role of the kernel in an Operating System (OS) is to isolate application from each others with virtual memory. Since the kernel has access to the address space of all the applications, this also requires a strict separation between user and kernel execution context and virtual memory. This is achieved by leveraging CPU protection rings for code execution and with the support of the Memory Management Unit (MMU) for memory protection. The kernel exposes its interface to applications via system calls (syscall). The CPU changes its current execution ring during a syscall to be able to execute kernel code.

Another role of the kernel in an OS is to abstract the hardware resources and to provide an interface for the applications to interact with them. The Linux kernel supports a very large range of hardware devices through a evenly large collection of drivers. To avoid having to load into kernel memory the code of all the drivers, the mechanism of modules was created. Modules can be compiled separately from the rest of the kernel sources and loaded dynamically at runtime which aims at keeping the amount of useless code loaded in memory as little as possible. In the traditional OS architecture, access to device is only possible by the usage of syscalls. Commonly used operations have led to the creation of dedicated syscalls like `read` or `write` while the general-purpose syscall `ioctl` is meant for device-specific operations.

The switch between user execution context and kernel execution context usually called mode switch has a significant overhead. On top of the direct cost of saving the context, switching the protection ring and computing the address of the syscall function, the number of TLB and cache misses increases due to the break in memory locality. In the end, the cost of a syscall can be estimated to a few hundreds of nanoseconds ([25]).

2.1.2 Infiniband

Infiniband is a high performance network communication standard massively used in High Performance Computing (HPC) systems (241 systems of the TOP500 in June 2023 [6]) due to its guarantees on high throughput and low latency. Nvidia produces Infiniband network card called Host Channel Adapter (HCA) that connects to other HCAs via Infiniband switches. Linux supports Infiniband cards with a driver since 2005. The main userspace library available

	Connection	Datagram
Reliable	RC	RD
Unreliable	UC	UD

Table 2.1: Types of possible exchanges

for applications is the `libibverbs` library ([3]). Operations on the network card are divided between control-plane, *e.g.* connection creation/closing, and data-plane operations, *e.g.* send, receive, read, write. Data-plane operations exchange messages over the network. On top of the normal `send()` and `recv()` primitives, Infiniband cards support Remote Direct Memory Access (RDMA) operations. RDMA allows an application to first register a memory region in the HCA. This region can be accessed directly by the card without the CPU involved. Infiniband supports more than 4 types of exchanges some of which are presented in table 2.1 depending on the level of service and the error-recovery mechanisms required by the application.

Internally, the Infiniband HCA uses queues to handle the messages. During dataplane operations, three queues are directly involved, the send queue (SQ), the receive queue (RQ) and the the completion queue (CQ). The first two are grouped in a queue pair (QP). Dataplane operations are composed of three primitives : `post_send()`, `post_recv()` and `poll_cq()` that interact with those queues. Fig 2.1 shows the sequence of operations during a Send operation. This figure assumes that the necessary control-plane operations have been executed and they are not presented. The Send operation begins when the sender, respectively the reciever, calls `post_send()`, resp. `post_recv()`. Those function will place a work request (WR) either in the SQ or in the RQ depending on the function. The HCA will handle the transfer of the data. After calling `post_send()`/`post_recv()`, both the reciever and the sender will continuously call `poll_cq()`. When the reciever's HCA recieves the message, it will place the data at the memory address registered during the call to `post_recv()` and will place a Completion Queue Event (CQE) in the CQ. Similarly, the reciever's HCA will send an acknowledgement to the sender's HCA which will also create a CQE for the `post_send()` operation. The two applications are notified of the completion when `poll_cq()` returns an element.

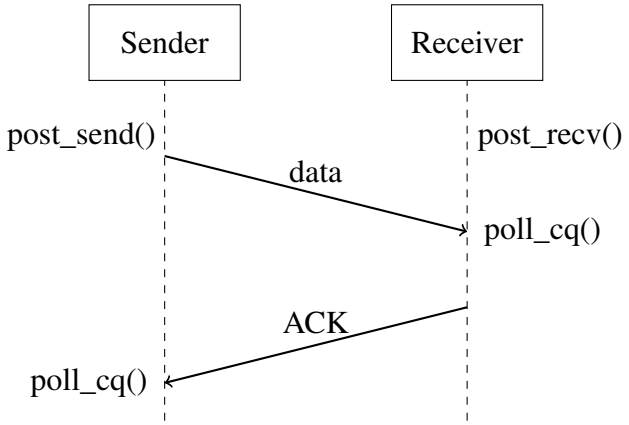

Figure 2.1: Sequence of operations for a Send request

The `libibverbs` library uses 3 mechanisms during data-plane operations to achieve high

throughput and low latency : zero-copy, polling and kernel-bypass.

- Zero-copy, or rather no unnecessary copy, means that the message data is only copied once from the application memory to the HCA memory and not once more inside the driver.
- Polling means that the application should poll for the completion of a request instead of relying on interruptions.
- Kernel-bypass means that the kernel gives the control over the HCA to the application thus removing the need for a syscall.

Those three mechanisms are mandatory to reach the highest level of performance. However kernel-bypass, as [22] shows, breaks the traditional OS architecture while being the least impactful on performance. Kernel-bypass prevents a fine-grain of the OS over the network operations.
Due to their relative proximity, the idea to move HPC and Cloud system architectures closer to one another have been proposed in recent years. HPC systems could benefit from increased elasticity and resource utilisation while Cloud architectures already have those features but require to be re-fitted to handle HPC workloads.
However, Cloud features like elasticity or resource sharing require the control of the OS over the system including the network. Kernel-bypass prevents mechanisms like OS-level scheduling, firewalling or live-migration. Converged RDMA Dataplane (CoRD) is an attempt to solve this issue.

— 3 —
Problem statement

3.1 Problem statement

We conducted a study on the latency of sending a message one-way on the L system described in section 5.1 using CoRD compared to using the baseline with multiple message size. Figure 3.1 present the result of this study. We can see that CoRD has a latency overhead of around 2 μs for the Reliable Connection (RC) communication and around 1.7 μs for the Unreliable Datagram (UD) communication.

CoRD was designed by porting the user-library implementation of the *dataplane* operations into the kernel driver. This means that the kernel driver pipeline performance should be equivalent to the performance of the library plus the system call overhead. However, this study [25] showed that the overhead of an empty system call on a similar system as L is around 300 ns. This figure is significantly higher to the 2 μs that we measured.
This work thus tries to explore the reasons behind this difference in performance. The goal is to find parts of the pipeline that are causing an overhead and to remove them if possible.

3.2 Related work

The use of kernel-bypass networking in the Cloud environement, *i.e.* through the use of framework like DPDK [1], has been shown to prevent a number of features necessary to Cloud platforms. Among those features are the virtualisation of NICs ([15], [24]), a fine-grain control of the resources ([19]) and the monitoring of applications ([7]).
On HPC systems, a kernel controlled network enables a more efficient co-location of applications through a better scheduling of oversubscribed MPI ranks ([9], [11]) or application-oblivious memory pooling over the network [27] that can reduce memory under-utilisation. In general, it allows for more diverse workloads and improved resource utilisation.

One way of bringing this level of control to dataplane network operations is to offload the logic to the network card. In this case, it requires either a SmartNIC ([10], [29], [30]) or hardware modifications ([21], [28]). However, these changes are costly which motivates the choice to solve the problem in software. In this case, the problem with user-level networking is that it is usually application specific and lacks kernel control whereas the problem with kernel-level networking is that it is usually less efficient. One solution is to use a different OS architecture

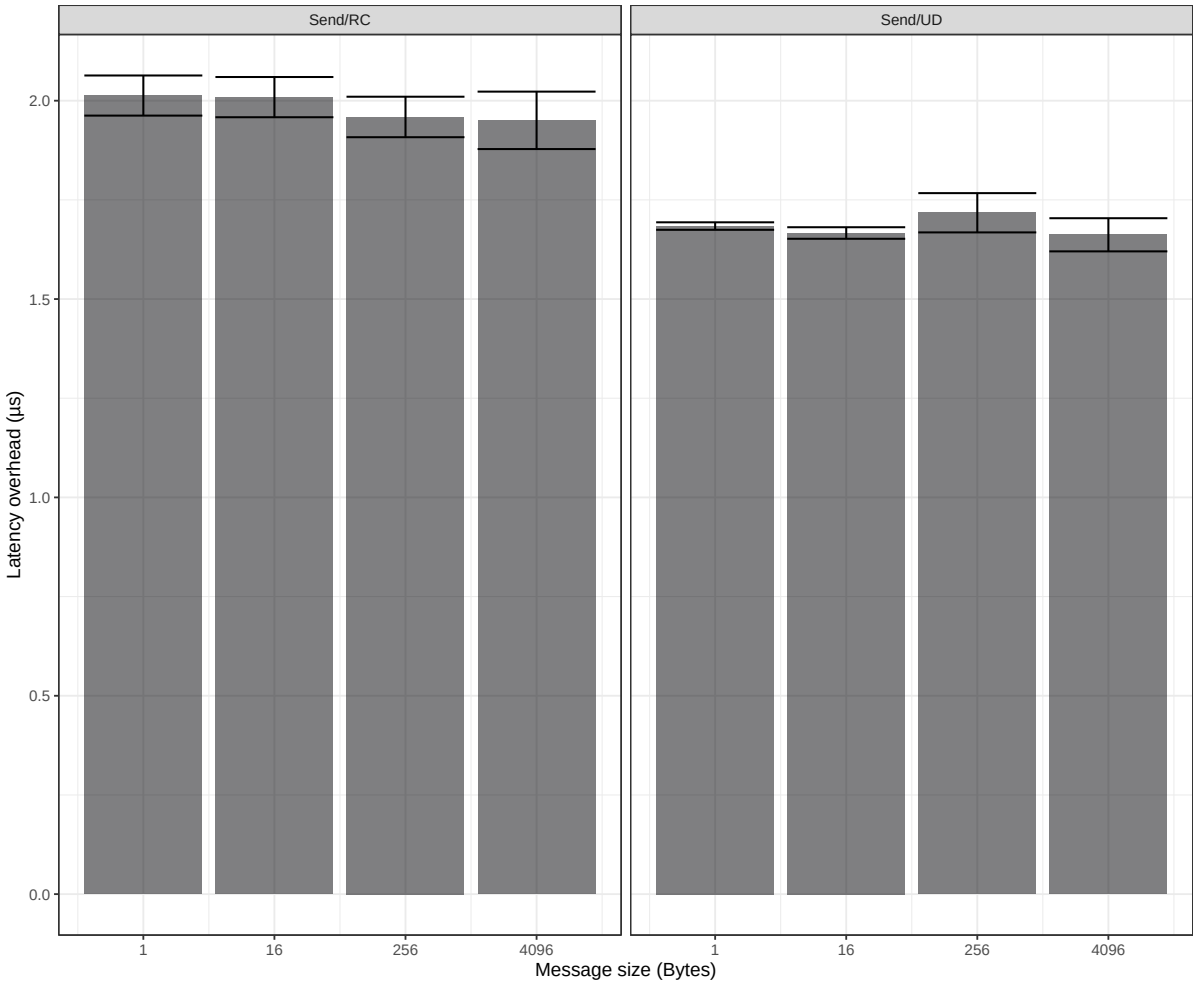

Figure 3.1: Comparing the overhead of the latency of a one way communication using the CoRD version on the L system

to solve one of those issues. On one hand, the micro-kernel architecture includes user-level networking provided to the applications and controlled by the OS ([23]). On the other hand, multi-kernel architectures uses a specific lightweight kernel for performance sensitive operations while keeping a full weight kernel (usually Linux) to provide the applications with the abstractions and the APIs that they expect ([13], [14]). Micro-kernels based architectures are becoming more common on Cloud platforms since they provide other features like online reconfiguration but it is usually less efficient for HPC workloads. On the other hand, multi-kernel architectures have also failed to become popular in HPC systems perhaps due to their complexity and for some of them their lack of compatibility with existing application. In the end, the most common OS architecture is the monolithical kernel based OS most of the time represented by a Linux distribution. This is why we decided to focus our study on CoRD [22] which tries to give back the control of dataplane Infiniband operations to the Linux kernel.

To our knowledge, this work is the first one to explore the performance of data plane operations inside the Infiniband driver of the Linux kernel. However, in the recent years, the problem of improving the performance of kernel-level I/O or network operations has been extensively

studied. Works focusing on storage and I/O in general are also relevant here because they are very similar both in the problems and in the order of magnitude of the latency considered.

Most works focus on improving the kernel/user interface for any high performance I/O operation. In Linux, there are only two ways to communicate between user space and kernel space, syscalls which can be batched ([12]) or intrisincally improved in software ([25]) or in hardware ([20]). The second option is memory-mapped I/O operations ([26]).

Offloading some parts of the user-level library in the kernel is another option. This can be done either by leveraging Linux mechanisms like eBPF ([32]) or XDP ([17]), with a custom automatic offloading mechanism ([33], [25], [31]).

However, to our knowledge, only [16] explored the option of improving the driver itself in this case by re-architecting the storage stack.

— 4 —

Towards a more efficient Infiniband Linux driver

To improve the performance, we identify parts that are causing an overhead by studying the performance of the CoRD pipeline using timing analysis or other statistical tools like flamegraphs. Table 4.1 presents the percentages of sample taken in each of the main function of the driver. Using this flamegraph we could identify 4 groups of functions that were adding an overhead. We also re-generated flamegraphs as we removed those parts to see how the performance had evolved. As presented in Table 4.2, the security mitigations implemented on syscall return are the biggest performance overhead so this is the first optimisations. The second target is the copying of the user-level memory to the kernel represented by the calls to `copy_from_user()` and the calls to `kmalloc()`. The third target is the overhead imposed by the dispatching of the *dataplane* functions. The fourth and final target is the design choice of keeping an index to the queues on the user-level library and not directly the kernel pointer causing some processing to recover this pointer.

Function(s)	% of CPU time
entry point of the hardware driver (`mlx5_post_send`, actual work)	8.82%
exit point of the core driver (`ib_uverbs_post_send`)	36.48%
entry point of the core driver (`ib_uverbs_ioctl`)	57.8%
entry point of the kernel	88.85%

Table 4.1: Functions inside the kernel with their corresponding % of CPU time during the `write_bw` benchmark summarized from the flamegraph generated using [2]. Each function percentage is included in the percentage of the function below it.

For each section, we start by detailing the target and the idea of the optimisation. Then we also describe how we implement it.

4.1 Turning off security vulnerability mitigations

Opening the side-channel Pandora box.

Function(s)	% of CPU time
exit to user-space	18.87%
copying and allocating related functions	15.76%
dispatching of the requested operation	13.56%
Functions to recover the queues pointer	9.38%
post_send function on the mlx5 driver (actual work)	8.82%

Table 4.2: Groups of functions that we extracted from the flamegraph summarized in Table 4.1. The last line contains the driver function that actually does the operation on the HCA.

Idea

The kernel of an OS is important target for security attacks. Any attack that manages to get the control of the execution while in supervisor mode has access to the whole system whereas a compromised user-level application can be contained or handled by the kernel.
In recent years, the linux kernel has been hardened against a series of micro-architectural side channel vulnerabilities . These attacks include Meltdown, Spectre and Microarchitectural Data Sampling (MDS) but also other vulnerabilities like Retbleed or Special Register Buffer Data Sampling (SRBDS) which are based respectively on Spectre and MDS. We note here that we are only describing here the mitigations that induce an overhead to a mode switch.
Meltdown, which allows a process to access memory belonging to the kernel or to other processes bypassing the usual security checks, was mitigated for the kernel memory by the implementation of Kernel Page Table Isolation (KPTI). The isolation is enforced by only keeping a minimal amount of kernel pages in a user-level process page table, except the pages necessary to execute syscalls and other user/kernel communication. This mitigates the vulnerability as the user-level process is completely unaware of most of the kernel pages. In turn, this means that the page table has to be changed during a mode switch thus inducing most of the cost of a regular context switch; TLB-flushing and page-table switching.
Spectre is an entire class of side channel attacks exploiting speculative execution and branch prediction features of most modern CPUs. We focus here on the bound check bypassing version, called in Linux version 1. Some CPUs speculate the execution of some branches even without checking if the accessed memory is in a valid range. Those speculative memory accesses induces the loading of memory in the CPU without privilege or validity checks and created potential side channels with which an attacker can access otherwise protected memory. The mitigation for this attack in the Linux kernel is done by adding an LFENCE barrier in the `copy_from_user` function which serializes all the load instructions that were issued before the barrier. This induces an overhead on the aforementioned function which is heavily used in the module code.
MDS is similar to Spectre as it exploits speculative access to memory. In the case of MDS, the target are buffers used internally by the CPU for speculative execution. It is mitigated by explicitly flushing those buffers during a mode switch, hence adding an overhead on the `enter_from_user_mode()` and `exit_to_user_mode()` functions.

The vulnerabilities affecting our two experimentation systems are presented in Table4.3. Saturn is vulnerable to all of the vulnerabilities and is running the mitigations whereas Oracle is only affected by Spectre(v1).

System	MDS	Meltdown	Spectre (v1)
L	Vulnerable: Clear CPU buffers attempted, no microcode; SMT disabled	Mitigation: PTI	Mitigation: usercopy/swapgs barriers and __user pointer sanitization
O	Not affected	Not affected	Mitigation: usercopy/swapgs barriers and __user pointer sanitization

Table 4.3: Security vulnerabilities for the Saturn and the Oracle profile as presented by `sysfs` in the `/sys/devices/system/cpu/vulnerabilities` directory

Implementation

This optimisation is the easiest to implement. We simply modified the kernel build configuration and add `CONFIG_CMDLINE="mitigations=off"` to turn off MDS, Meltdown and Spectre mitigations. This kernel command line parameter turns off more mitigations than the ones stated but only the one stated have a significant impact on the performance of the functions that we target.

4.2 Using shared memory

Spoiling the copying and malloc-ing party.

After removing the mitigations, we identified that a significant portion of the system call is spent in memory allocation and copy operations. These operations come from the syscall interface and are not necessary in the kernel-bypass pipeline. In this section, we try to remove those operations. However, removing those operations implies to lower the security of the OS, we propose a design based on a shared memory region that removes the need for those operations without introducing security flaws.

Idea

As we described in the previous section, the interface between the kernel and a user-space process is an especially critical point in terms of security for the kernel. For memory exchanges, Linux kernel developers must use the functions defined in `uaccess.h` (*e.g.* `copy_from_user`, `copy_to_user`, etc...) when accessing user-level memory.
User-level processes could try to attack the kernel by giving it an address in the kernel section of the address space, potentially tricking the kernel into rewriting its own memory. Another attack could be to provide a faulty address, *e.g.* a `NULL` pointer. In this case, without proper checking, the use of such address could lead to a failure which, due to the monolithic architecture of the kernel, would result in a complete kernel crash. It is worth noting that some recent CPU implement two hardware features, Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP). These features provides hardware support to prevent kernel access (or execution) of user-level memory. They are implemented using flags in a Control Register. They are turned on at boot time, thus require an explicit instruction to turn them off, preventing more unwanted user-memory access or execution.

The `uaccess` functions contain checks preventing these attacks as well as mechanisms to disable temporarily SMAP if necessary. This obviously adds an overhead compared to accessing directly user-level memory. Other induced cost can be the extra memory needed by two allocations, on the user heap and on the kernel heap as well as the potential cache miss caused by the change in the address of the same value.
The idea of this optimisation is thus to access as much as possible the user-level library memory directly. To do this efficiently while preventing security issues, a memory region must be shared between the kernel and the application.

Implementation

To understand our implementation of the optimisation, we have to describe the architecture of the driver that we modified.
When calling the driver using `ioctl`, the execution flow can be split in two phases. A first phase where the module dispatches the requested operation and a second one dedicated to deserializing the request and calling the hardware driver function that actually executes the request. Figure 4.1 presents this architecture with our modifications. The functions of the first phase are shared across any kind of request (even control plane operations) while each request has a different function for the second phase. Therefore, we need a *fast* branch in the first phase along the original one while, for the second phase, we can directly modify the three functions of the critical path (`ib_uverbs_post_send`, `ib_uverbs_post_recv` and `ib_uverbs_poll_cq`).

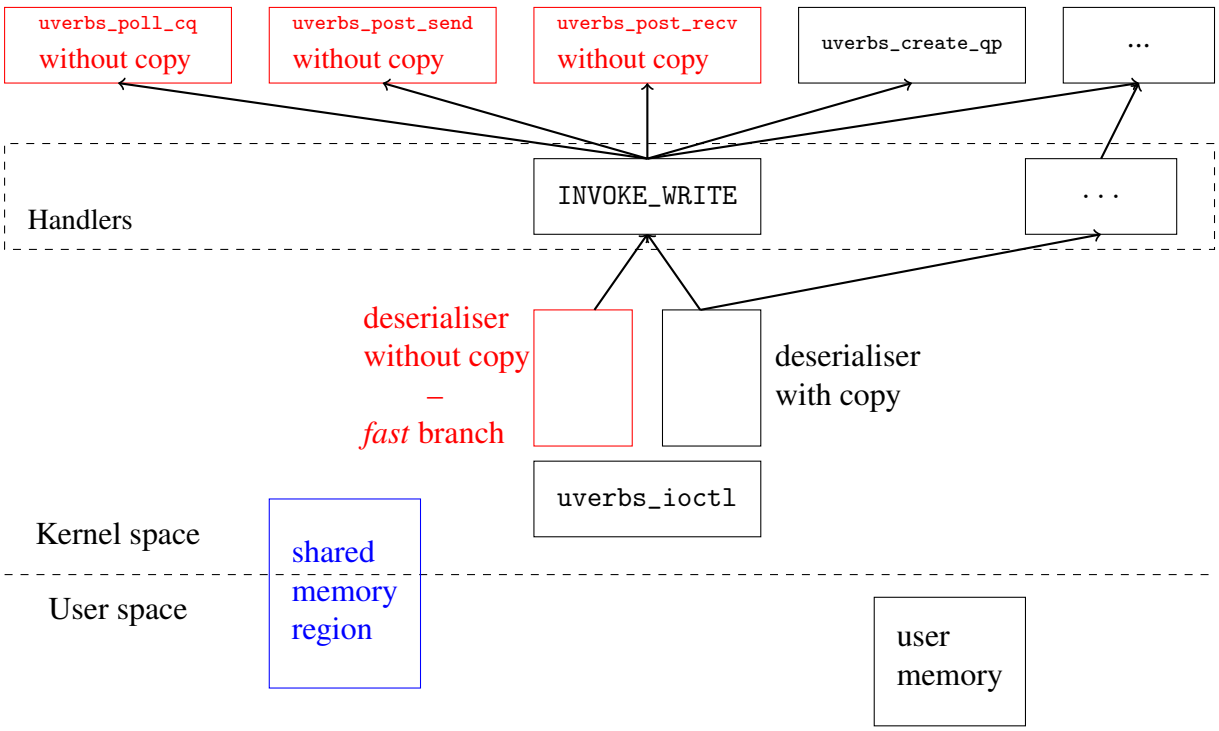

Figure 4.1: Architecture of the driver

```
struct ib_uverbs_ioctl_hdr {
  __u16 length;
  __u16 object_id;
  __u16 method_id;
  __u16 num_attrs;
  __aligned_u64 reserved1;
  __u32 driver_id;
  __u32 reserved2;
  struct ib_uverbs_attr  attrs[];
};
```

Excerpt 4.1: Detail of the original ib_uverbs_ioctl_hdr structure

When an application calls one of the library function, the library serializes the requests arguments. It then transfers the control to the module using the `ioctl` syscall. `ioctl` takes two arguments, one for a file descriptor and one for a pointer. This second arguments is casted to a pointer to a structure of type `ib_uverbs_ioctl_hdr`. The fields of this structure can be found in Listing 4.1.

To implement the branching that we described at the beginning of this section, we added a boolean field called `use_fastcall` to the structure presented in Listing 4.1. The entry point checks this field and branches accordingly in the *fast* path or not. In the *fast* path, the request is not copied from user memory but simply passed to the next function. To do so, we need to disable temporarily SMAP by using the `stac()` function which, if the CPU supports SMAP will set the Alignment Check (AC) flag of the EFLAGS, stored in the Control register 4 (CR4) of the CPU. This flag is then cleared upon return to user mode with the `clac()` function.
For the function of the second phase, we modified them so that they would use pointers to the relevant attributes. As an example, Listing 4.2 show the relevant parts of the `ib_uverbs_poll_cq` function before and after the modifications. We can see that the copying of the request and of the response are replaced by the use of a pointer respectively to the `inbuf` and the `outbuf` field of the `ucore` attribute. The exact same modification is done on the `ib_uverbs_post_send()` and the `ib_uverbs_post_recv()` functions.

```
// Before
struct ib_uverbs_poll_cq       req;
struct ib_uverbs_poll_cq_resp  resp;
copy_from_user(&req, attrs->ucore.inbuf, min(attrs->ucore.inlen, req_len));
copy_to_user(attrs->ucore.outbuf, &resp, sizeof resp);
// After
struct ib_uverbs_poll_cq       *req = (struct ib_uverbs_poll_cq*)attrs->ucore.inbuf;
struct ib_uverbs_poll_cq_resp  *resp = (struct ib_uverbs_poll_cq_resp*)attrs->ucore.outbuf;
```

Excerpt 4.2: Detail of the ib_uverbs_poll_cq function before and after the optimisation

Contrary to `poll_cq` which does not need to call `kmalloc` during its execution, the `post_send` and `post_recv` allocate objects of type `ib_send_wr`, respectively `ib_recv_wr` in order to fit the data provided by the library. We cannot simply use the pointer in this case because the type of the work request depends on the type of message that is being processed, *e.g.* rdma, atomic, etc. and a different structure must be used in each case. Therefore, we decided to allocate on the stack one object of each type and use it first to avoid the allocation. If multiple work request are being processed in the same call to `post_send` or `post_recv`, then the work request after the first one are allocated as in the original version.

Another important point of the implementation is the shared memory region. As we currently describe it, the driver simply exposes the kernel to the kind of attacks that we described in the previous section (invalid memory access, *etc.*). In order to protect the kernel while keeping the performance improvement that results from accessing user-memory directly, a per-thread shared memory region may be used. For our implementation, we simply allocated a large region of user memory (8KiB because the largest payload is 4KiB and for alignement purposes). The requests are then passed to the driver on this memory region.

4.3 Calling the driver functions directly

Geodesic considerations.

Idea

We described in Section 4.2 that the execution flow in the module is split into two phases. To be more precise, in the original version, the header object passed to the module does not contain the id of the function that is requested. This identifier is rather passed as one of the attributes stored in the flex array at the end of the header. In order to reach the requested function and the second phase, the first phase ends by calling one of the several handlers. There is one for each broad type of function, *e.g.* `INVOKE_WRITE`, `INFO_HANDLES`, `QUERY_PORT`, etc., which does some processing depending on the type and then calls the requested function. The address of those handlers is stored in a `radix_tree`. Figure 4.2 presents the new architecure with our optimisation.

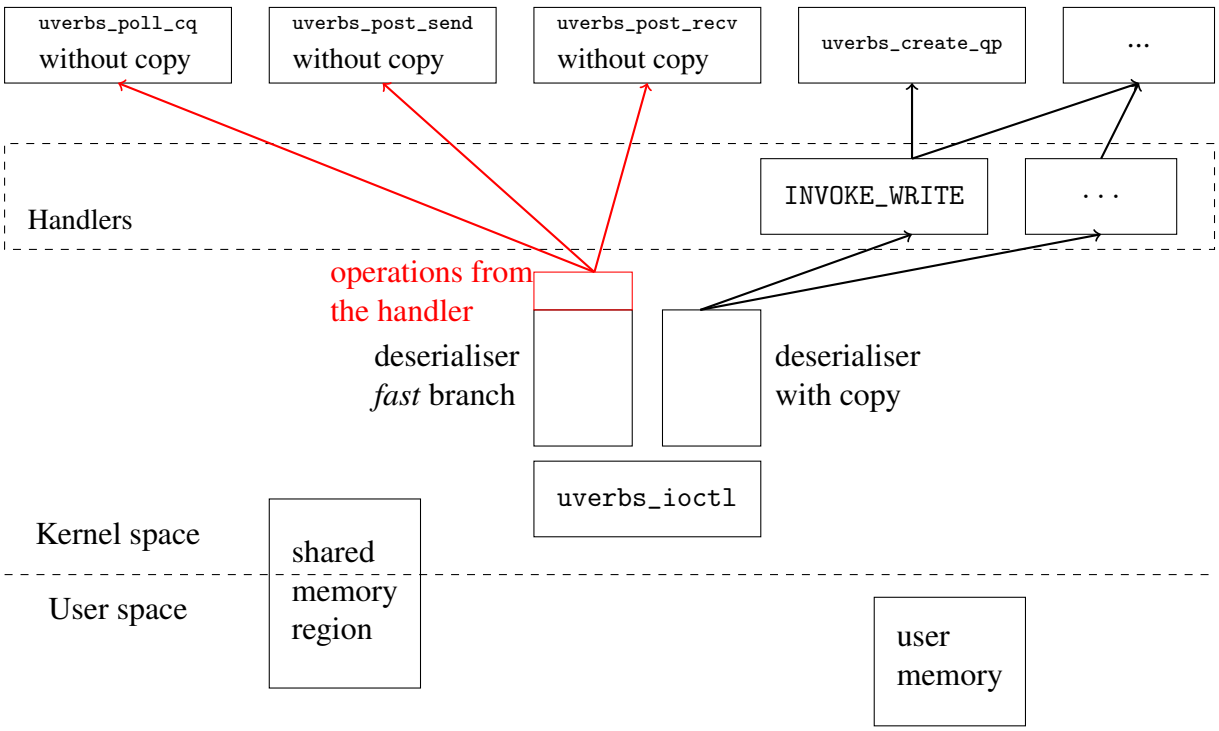

Figure 4.2: Architecture of the driver

This architecture is necessary to provide backward compatibility with the invocation of the module with the `write` syscall. When calling with `write`, the execution enters the module directly at the second phase in the correct function. The idea of this optimisation is then to include the id of the requested function in the header passed to `ioctl`.

Implementation

The implementation of this optimisation is straight forward. We add a field in the `ib_uverbs_ioctl_hdr` structure. Since we already added a field on this structure for the previous optimisation, the final version can be found in Code 4.3.

```
1  struct ib_uverbs_ioctl_hdr {
2    __u16 length;
3    __u16 object_id;
4    __u16 method_id;
5    __u16 num_attrs;
6    __aligned_u64 reserved1;
7    __u32 driver_id;
8    __u32 reserved2;
9    ib_uverbs_cmd_verbs __u8 use_fastcall; // from previous optimisation
10   ib_uverbs_cmd_verbs __u8 fastcall_method; // added
11   struct ib_uverbs_attr  attrs[];
12 };
```

Excerpt 4.3: Detail of the ib_uverbs_ioctl_hdr structure after our modifications

The entry point of the driver then reads this field and dispatches to the function of the second phase directly. The handlers are not only dispatching, they also do some processing that we had to move to the functions of the first phase. However, this move was easy to do since we already had the *fast* path created at the previous optimisation.

4.4 Passing the queues pointers from userspace

The beginning of wisdom is to call things by their proper name

The final target of optimisation is the overhead caused by the passing the message queues by handle which in turn imposes some processing to recover the pointer to the queue. Exposing the kernel pointer to userspace introduces a security vulnerability.

Idea

For isolation purposes, each application needs to create its own queues and then pass them as arguments to `post_send()`, `post_recv()` and `poll_cq()` during a communication. However, in the original version of the module, the driver only returns a handle, an identifier of the queue, to the library when creating the queues. While this provide security because it hides the true address, it requires some processing in the module to recover the address from the handle. Another aspect of this processing that the reference to all the Infiniband objects are counted and kept up to date. This adds an overhead but is necessary in order to prevent the driver from leaving unused memory in the kernel heap. Because the idea of this work is to explore the parts that are degrading performance and not to design a secure system, we do not implement

the reference counting on the kernel address. The idea of this optimisation is then simply that the user-space applications gets the pointer to the queues and passes this pointer to the communication functions.

Implementation

The implementation of this optimisation is two fold. First, the library must receive the pointer to a queue when it is created and thus the module should return it. Then the library should pass the pointer and the module correctly use it.

To return the queue kernel pointer to userspace, we used the same mechanism that returns the handle. The library calls the driver function with a syscall and passes a pointer to user-memory where the driver should write the requested value. In the case of the kernel pointer, the library stores the value as a 64 bits integer. When calling dataplane operations, the queue pointer replaces the handle. Once inside the driver, only the functions of the second phase are changed to simply use the value as a pointer instead of calling `rdma_lookup_get_uobject` and `rdma_lookup_put_uobject` which were used previously.

— 5 —
Evaluation and Discussion

5.1 Experimental setup

We use two systems for our evaluation, a local system L and a remote system O hosted on the Oracle Cloud.
L involves two servers with a Intel(R) Core(TM) i5-4590 CPU with a base frequency of 3.3GHz, 16 GB of DDR3 memory and equiped with an Infiniband 100Gbps network card. They are connected directly with a special Infiniband cable. O is deployed using BM.Optimized3.36 bare-metal compute shapes availble for the Oracle Cloud ([4]). Those shapes are provisioned with an Intel Xeon 6354 with a base frequency of 3GHz, 512 GB of DDR4 memory and with an Infiniband 100Gbps network card. Turbo mode and Hyperthreading are disabled for the experiments.

On the sofware side, we use two different environment, the baseline with kernel-bypass and CoRD. Kernel-bypass, or baseline in the remaining of this thesis uses an unmodified Linux kernel at revision 6.2-rc7 as well as the libibverbs library [3]. CoRD is the version used in ([22]) with minor bug fixes. It bypasses the libibverbs library by triggering an ioctl syscall for data-plane operations. On the kernel side, those operations have been ported from user-space making so that all the operations go through the kernel-driver.
For the evaluation, we use the `perftest` benchmark suite [5]. Those benchmarks are meant to evaluate the performance of Infiniband operations either in terms of latency or in terms of bandwidth. Other related metrics like the message rate are also measured. The benchmarks exchange a message filled with random data back and forth between a client and a server. The size of the message and the Infiniband operation (*e.g.* Send, RDMA Write, RDMA Read, etc..) are some of the parameters.
During the experimentations, we include in the output various labels like commit hashes or hardware configuration allowing to repeat the experiment in the same conditions.

5.2 Evaluation

On top of the measurements that we realised during the implementation part and that guided our exploration, we also conducted an experiment that present the result of the successive optimisations compared to the original CoRD. We measured the latency with the latency class of perftest benchmarks [5]. We realised the experiment with 4 sizes of messages and with all 6

version, the kernel-bypass baseline, CoRD and our 4 optimisations. Each optimised version contains the previous optimisations which means that `"Direct pointer"` represents the version with all our optimisations. We repeated 5 times each configuration and we plotted the latency overhead compared to the baseline. Figure 5.1 and 5.2 present those results respectively for system L and for system O.
We can see that the successive optimisations always improve the performance on all configurations. The overhead has been reduced by up to 1.36 μs between CoRD and the version with all our optimisations. The smallest improvement is on system O at 4KiB message size with only 0.511 μs of reduced overhead. Another interesting point is that we see the difference between system L and O in term of security vulnerabilities in the results. The improvement yielded by the first optimisation is significant on system L, it is even the largest improvement but it is not even significant for many configurations on system O. The improvement of the second optimisation seems significantly equal on both system around 0.4 μs. Optimisations 3 and 4 give smaller improvements on both systems. On L, up to 200 ns and 100 ns respectively. On O, the difference between `"Shared region"` and `"Fast dispatch"` on one side and `"Fast dispacth"` and `"Direct pointer"` on the other side is not even significant for most configurations. However, `"Direct pointer"` is always significantly better than `"Shared region"`. Overall, the overhead has been reduced by up to 65% on L, from 2μs to 0.7 μs and by up to 45% on O, from 1.6 μs to 0.9 μs.

We also realised an experiment to measure the maximum bandwidth when using the different versions. Once again, we measured 4 different sizes of message and we plotted the relative overhead with 1 being the baseline. Figure 5.3 and 5.4 present the results respectively for system L and for system O. Confidence intervals are ommited because they are not big enough.
First, we can see that the two system do not have the same scaling with the message size. L has a similar relative performance for all the message sizes up to 256B. The relative bandwidth is small, between 15% of the baseline for CoRD to 30% of the baseline for the version with all our optimisations. For 4KiB, the performance of the baseline decreases significantly and thus the relative performance increases sharply. With all our optimisations, the bandwidth reaches around 90% of the baseline performance. On the other hand, on O, the performance does not differ significantly across the different message sizes. CoRD is at around 35% of the baseline while the version with all our optimisations is between 50 and 60% with the only outlier being with RC at 4KiB at 70% of the baseline.

We also decided to generate a flamegraph in the same conditions as the one that we presented at the start of our study but with the version with all our optimisations. The results are presented in Table 5.3 with a recall of the original percentages with CoRD. The main conclusion that can be made from this comparison is that we see here the effect of our optimisations. The entry (and exit) point of the kernel has decreased by around 20% which corresponds to the removal of the mitigations. `ib_uverbs_post_send` has also decreased by around 15% which is due to the use of the shared memory region which removes all the copy. However, we can see that the hardware driver function percentage of CPU time has not increased this much, only 4 %. When looking at the full flamegraph, we can see that the depth of the call graph has decreased due to our optimisations. However, the numbers presented in Table 5.3 show that we have not decreased the overhead of the core driver enough so that the hardware driver represent the majority of the time (like in the kernel-bypass version).

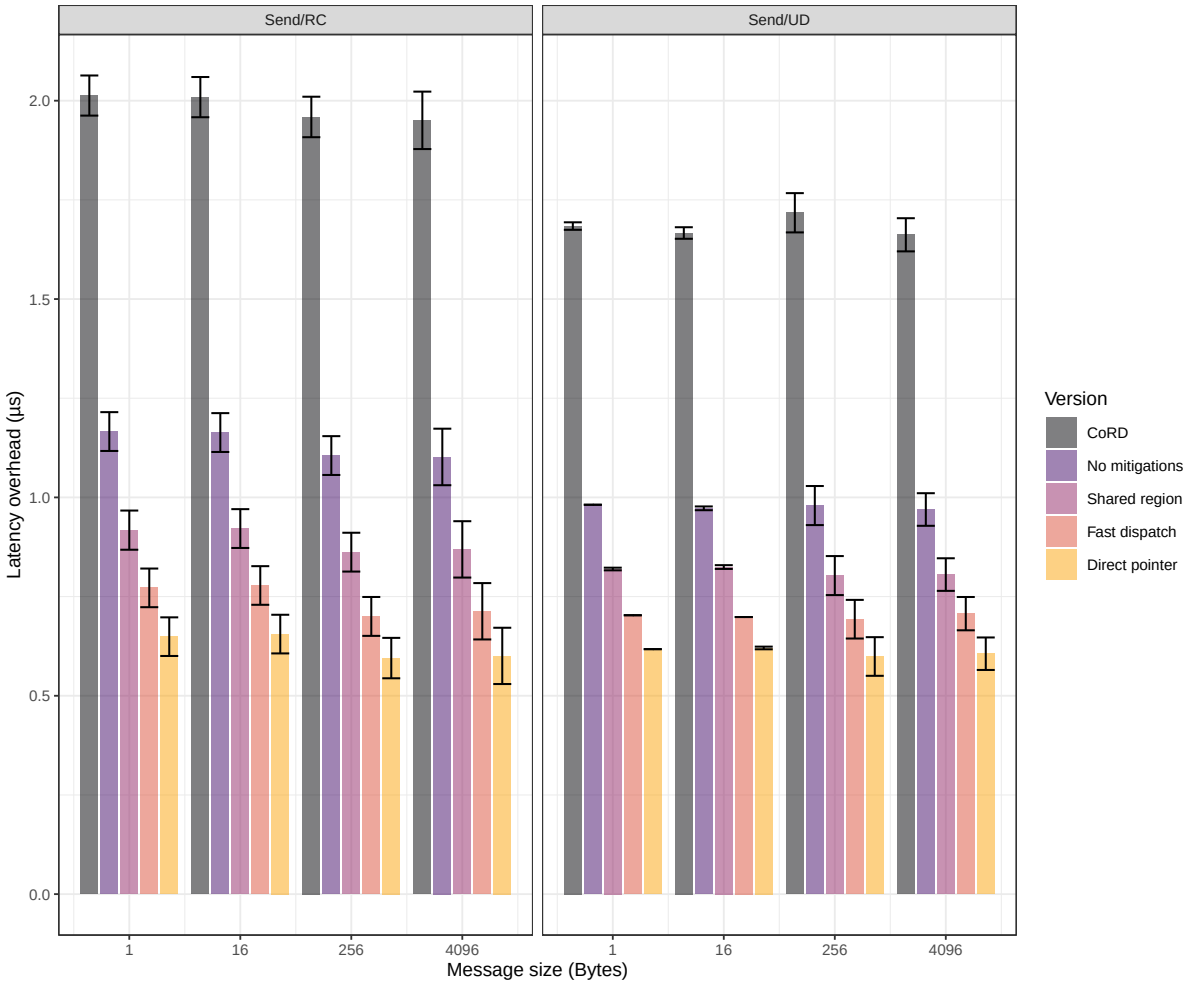

Figure 5.1: Comparing the overhead of the latency of a round trip using the original CoRD and the successive combinations of optimisations on system L.

According to the Infiniband specification ([18]), the size of the header of a normal RC packet, containing neither a RDMA nor an atomic operation, is 78 bytes. For a 4 bytes message, this produces a total packet size of 656 bits. Considering a 100 Gbps Infiniband network card, this means that the maximum number of packets of this size that can be handled by the card is $100 * 2^{30}/656 \approx 163\,680\,156$ pps (packet per second). This leaves only $1/163\,680\,156 \approx 6$ ns to process each packet. For a 4096 bytes message (*i.e.* the maximum payload size for a UD communication), the CPU should not spend more than 300 ns per packet in order to reach the maximum bandwidth of the network card.
We decided to compare those theoretical numbers with our experimental results. The bandwidth benchmarks include a message rate metric. From this metric, we can compute the time per packet as we did in the previous paragraph. We computed the time per packet using the same results presented in the previous paragraph. The results are presented in Table 5.1 and 5.2 respectively for system L and O. Compared to the latency measurements that we conducted previously, those numbers are interesting as they represent the performance of the driver at maximum CPU utilisation. From those results, we can clearly that the original CoRD version takes too long to reach the maximum of the bandwidth. The baseline only nearly reaches the

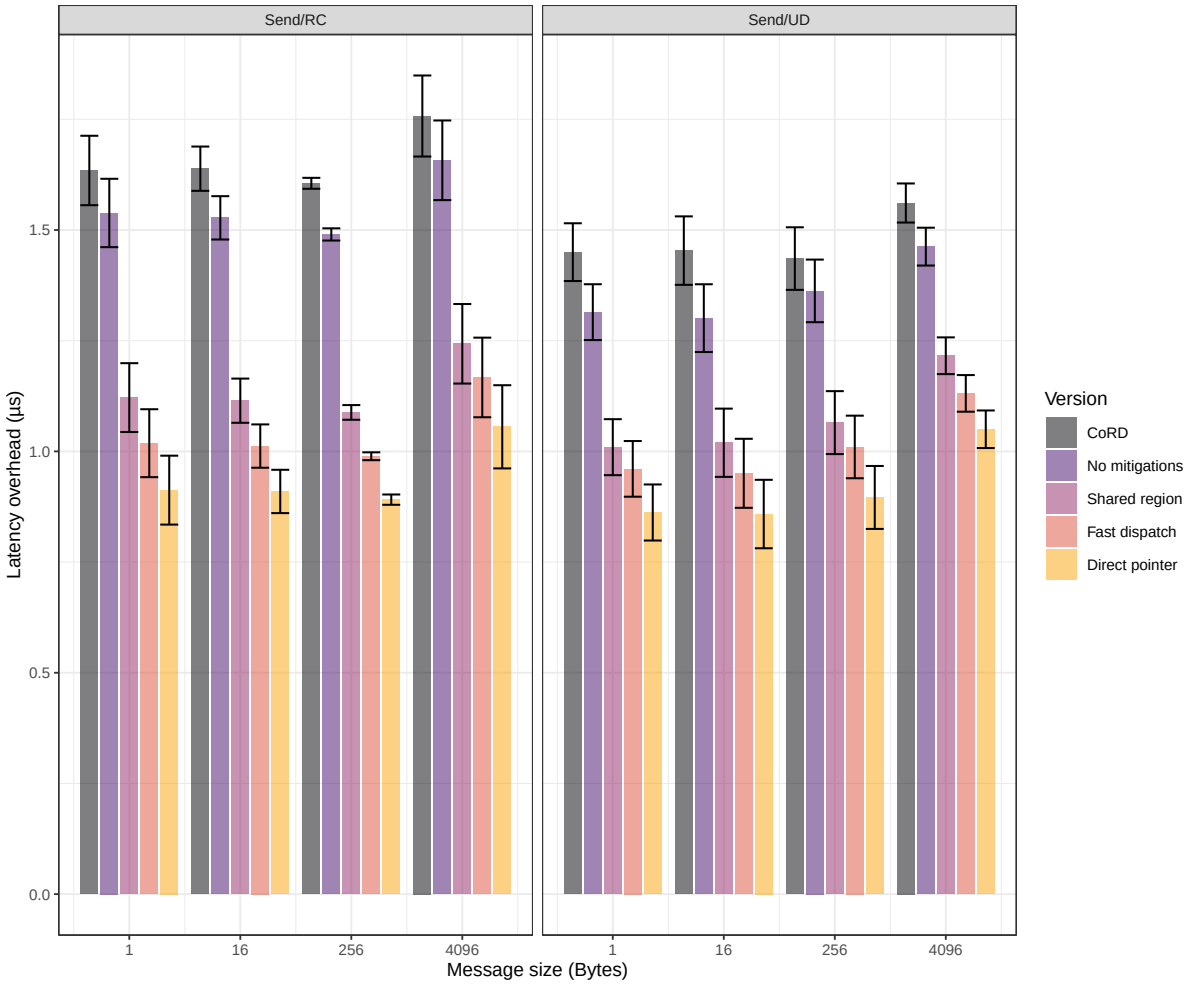

Figure 5.2: Comparing the overhead of the latency of a round trip using the original CoRD and the successive combinations of optimisations on system O.

Version/message size	1	16	256	4096
Theoretical max bw	6ns	7ns	25ns	311ns
baseline	99ns	99ns	100ns	352ns
CoRD	1096ns	1094ns	1104ns	1089ns
Optimised	317ns	317ns	320ns	373ns

Table 5.1: Time per packet (in ns) at different message size and with the kernel-bypass version, CoRD and our optimised version on system L. The first line contains the maximum time per packet to reach the maximum bandwidth.

maximum bandwidth time per packet for packets of 4KiB. Our optimised version also nearly reaches it for packets of 4KiB. We can also see that some configurations have a nearly constant time per packet like the baseline on O and the CoRD version and our optmised version on both system. This means that the size of the message has no impact on the time per message and that the processing time is thus dominated by a part that does not depend on the message size.

To conclude, our optimisations had a significant impact on the performance of the driver. With them, we managed to reduce the latency of up to 65% on system L and of up to 45% on

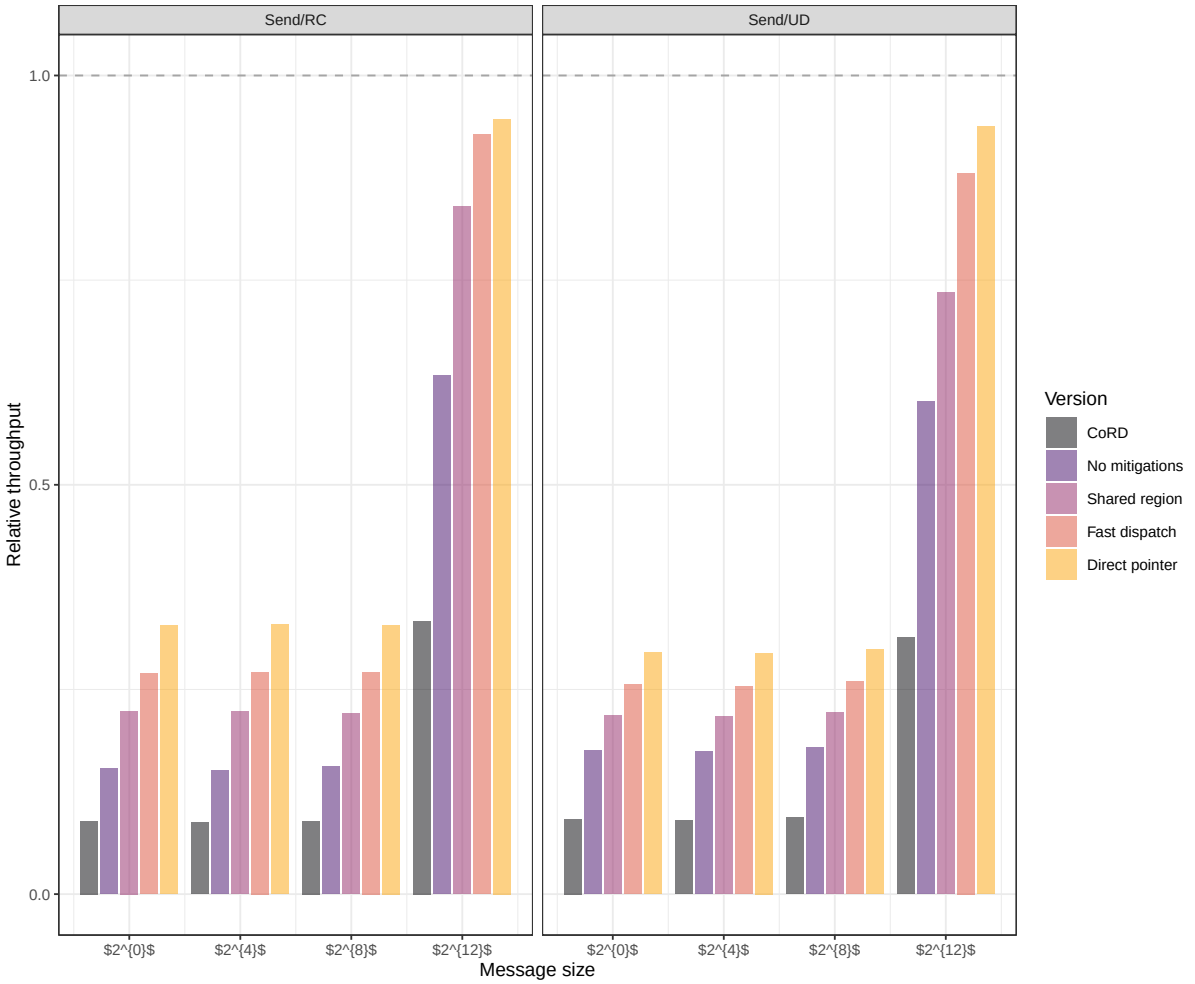

Figure 5.3: Comparing the relative maximum bandwidth using the original CoRD and the successive combinations of optimisations on system L. 1 is the maximum bandwidth reached by the baseline

system O.
To try to explain the difference in performance improvement between system L and system O, we believe that system L should be considered as an older system where the CPU is slower overall than on system O. This is highlighted by the fact that the CPU of system L is more vulnerable than the one of system L and thus has to execute more mitigations in software.

Version/message size	1	16	256	4096
Theoretical max bw	6ns	7ns	25ns	311ns
baseline	366ns	367ns	368ns	351ns
CoRD	1057ns	1059ns	1060ns	1058ns
Optimised	664ns	661ns	662ns	586ns

Table 5.2: Time per packet (in ns) at different message size and with the kernel-bypass version, CoRD and our optimised version on system O. The first line contains the maximum time per packet to reach the maximum bandwidth.

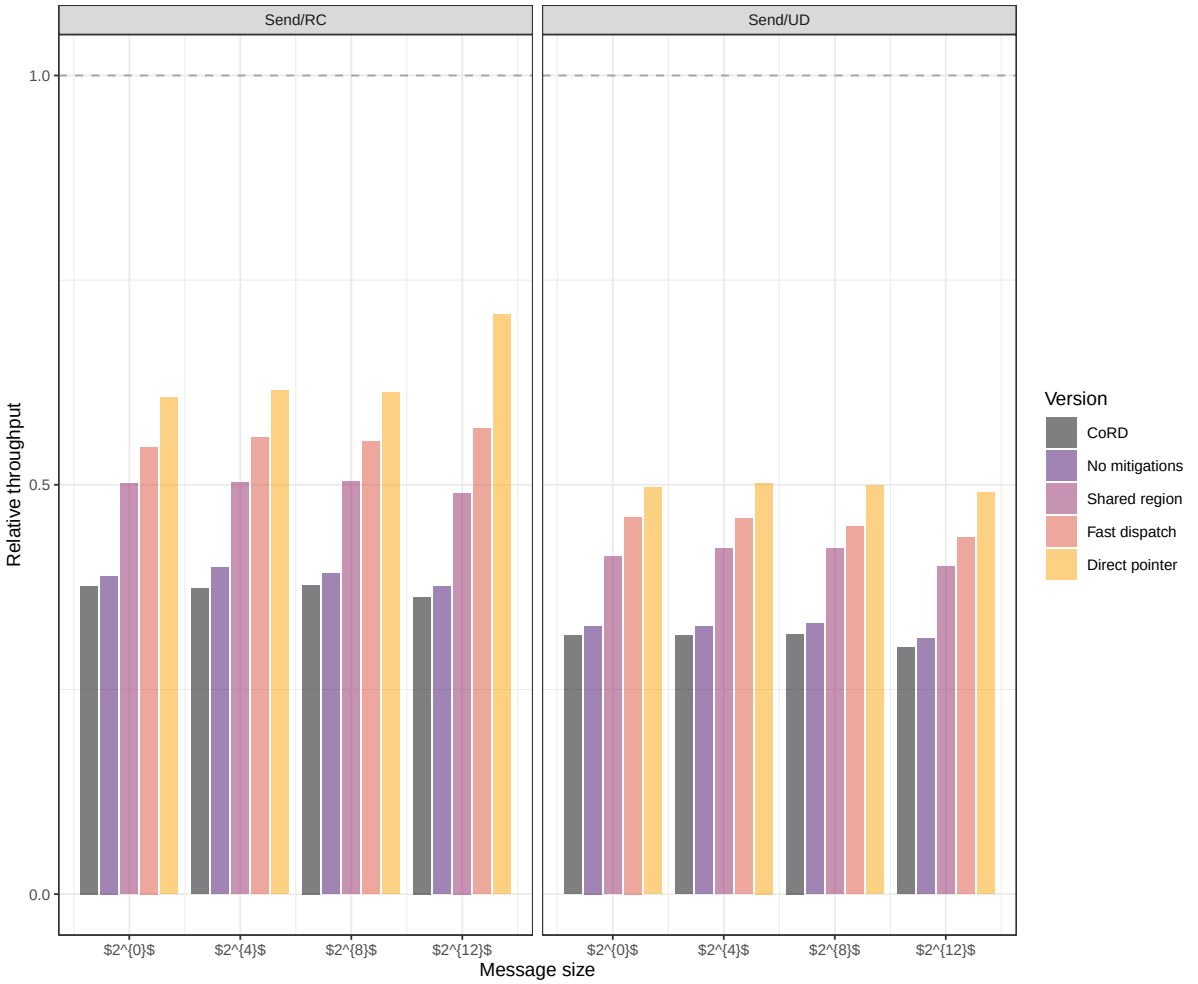

Figure 5.4: Comparing the relative maximum bandwidth using the original CoRD and the successive combinations of optimisations on system O. 1 is the maximum bandwidth reached by the baseline

Function(s)	`opti_4`	CoRD
entry point of the hardware driver (`mlx5_post_send`)	12.65%	8.82%
exit point of the core driver (`ib_uverbs_post_send`)	19.01%	36.48%
entry point of the core driver (`ib_uverbs_ioctl`)	45.97%	57.8%
entry point of the kernel	68.43%	88.85%

Table 5.3: Functions inside the kernel with their corresponding % of CPU time compared between using the optimised `opti_4` driver and the original CoRD on system L.

Since we consider performance improvement, simply removing instructions that require CPU time plays a significant role in the improvement. For the absence of significant scaling on system 0, an argument could be that system L involves two directly connected nodes whereas system O nodes are connected via a switch. This means that the overall latency of a message will be longer on O. Thus, even the improvements yielded equivalent results on the local nodes, it would play a smaller role in the overall bandwidth of system O because of the switch.

5.3 Discussion and Future work

While the previous section has showed that our optisations managed to reduce the overhead of CoRD, there is still some parts of the overhead that are unexplained.
First and foremost, we have not reached our goal of reducing the overhead of a driver call to the overhead of a syscall. The lowest overhead that the driver yields is 700 ns which is still 400 ns more than what [25] has measured as the overhead of a syscall. This means that there is still some parts that are degrading the performance that we have not found yet as we showed with the flamegraph of `opti_4` in Table 5.3.
We believe that a deeper re-architecting and study of the driver is necessary to this end.

Even if we improved significantly the driver performance, it is still significantly lower than the kernel-bypass baseline. In order to reach a level of performance equivalent to the baseline, the overhead of the system call must be removed. However, this requires either to use a different interface than a syscall or to re-design the syscall interface for this purpose. In both cases, this would imply solving the problem of designing an efficient yet portable user-kernel interface in Linux which, as Atlidakis et al. show, is still an open question.

In the future, we also plan on conducting a more complete evaluation of our improved version of CoRD, especially using more realistic applications than micro-benchmarks.

— 6 —
Conclusion

In this work, we explored some of the reasons behind the overhead of CoRD. We conducted our study by measuring the areas of the Infiniband that were inducing an overhead compared to the kernel-bypass version that calls directly the hardware driver function. We found 4 parts on the kernel side that were hindering the performance. The first one is the mitigations of the Meltdown and MDS security vulnerabilities which add some processing during the return from kernel to user space. The second one is the copying of memory from user to kernel memory that we replaced with a shared memory region. The third one is the mechanism of dispatching inside the driver which is necessary for the compatibility with multiple hardware drivers but is in turn degrading the performance. The fourth one is the mechanism of protecting the kernel memory by returning only a handle for the message queues to the user-space library.
With these modifications to the driver, we managed to reduce the overhead of a communication by up to 65%. However, the overhead is still between 600 and 800ns which is greater than the overhead of an empty system call. This means that there is still some parts which are degrading the performance of the driver. Our methodology is not sufficient to find those parts and we believe that a re-architecting of the driver for this purpose would be necessary. Moreover, in the goal of bringing the performance of CoRD on par with the performance of kernel-bypass, the overhead of the syscall has to be removed too. For this, we believe that the current syscall interface is not suited for this high performance networking purposes. To reach kernel-bypass level of performance, we believe that a re-architecting of the syscall interface is necessary.

Bibliography

[1] Dpdk website. https://www.dpdk.org/. [Online; accessed 23/08/2023].

[2] Original blog post presenting the flamegraph. https://www.brendangregg.com/flamegraphs.html. [Online; accessed 16/08/2023].

[3] Github repository containing the libibverbs library. https://github.com/linux-rdma/rdma-core. [Online; accessed 16/08/2023].

[4] Detail from the Oracle documentation detailing the shapes available. https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm#bm-hpc-optimized. [Online; accessed 16/08/2023].

[5] Github repository of the perftest benchmark suite. https://github.com/linux-rdma/perftest. [Online; accessed 16/08/2023].

[6] TOP500 statistics subpage. https://www.top500.org/statistics/list/. [Online; accessed 16/08/2023].

[7] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 28–34, November 2021. doi: 10.1109/NFV-SDN53031.2021.9665095.

[8] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: the old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1–17, New York, NY, USA, April 2016. Association for Computing Machinery. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901350. URL https://dl.acm.org/doi/10.1145/2901318.2901350.

[9] Jan Bierbaum, Maksym Planeta, and Hermann Härtig. Towards Efficient Oversubscription: On the Cost and Benefit of Event-Based Communication in MPI. In *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 1–10, November 2022. doi: 10.1109/ROSS56639.2022.00007.

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman

Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. pages 51–66, 2018. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/firestone.

[11] 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 (IPDPS)*, pages 43–53, May 2019. doi: 10.1109/IPDPS.2019.00016. ISSN: 1530-2075.

[12] Luis Gerhorst, Benedict Herzog, Stefan Reif, Wolfgang Schröder-Preikschat, and Timo Hönig. AnyCall: Fast and Flexible System-Call Aggregation. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, pages 1–8, October 2021. doi: 10.1145/3477113.3487267. URL http://arxiv.org/abs/2201.13160. arXiv:2201.13160 [cs].

[13] 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)*, pages 1041–1050, May 2016. doi: 10.1109/IPDPS.2016.80. ISSN: 1530-2075.

[14] Balazs Gerofi, Rolf Riesen, Masamichi Takagi, Taisuke Boku, Kengo Nakajima, Yutaka Ishikawa, and Robert W. Wisniewski. Performance and Scalability of Lightweight Multi-kernel Based Operating Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 116–125, May 2018. doi: 10.1109/IPDPS.2018.00022. ISSN: 1530-2075.

[15] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 1–14, New York, NY, USA, July 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7. doi: 10.1145/3387514.3405849. URL https://doi.org/10.1145/3387514.3405849.

[16] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. pages 113–128, 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/hwang.

[17] Toke HÃ¸iland-JÃ¸rgensen, 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, pages 54–66, New York, NY, USA, December 2018. Association for Computing Machinery.

ISBN 978-1-4503-6080-7. doi: 10.1145/3281411.3281443. URL https://dl.acm.org/doi/10.1145/3281411.3281443.

[18] InfiniBand Trade Association. *InfiniBand Architecture Specification*, volume 1. InfiniBand Trade Association, 1.3 edition, March 2015. URL https://cw.infinibandta.org/document/dl/8567.

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

[20] Dmitry Kuznetsov and Adam Morrison. Privbox: Faster System Calls Through Sandboxed Privileged Execution. 2022. URL https://www.usenix.org/conference/atc22/presentation/kuznetsov.

[21] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. *ACM SIGARCH Computer Architecture News*, 45(1):449–466, April 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037710. URL https://doi.org/10.1145/3093337.3037710.

[22] Planeta Maksym, Jan Bierbaum, Michael Roitzsch, and Hermann Härtig. CoRD: Converged RDMA Dataplane for High-Performance Clouds. In *WORDS 2023*.

[23] Michael Marty, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Marc de Kruijf, Paul Turner, Valas Valancius, Xi Wang, Amin Vahdat, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, and William C. Evans. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles - SOSP '19*, pages 399–413. ACM Press, 2019. ISBN 978-1-4503-6873-5. doi: 10/ggcdnx. event-place: Huntsville, Ontario, Canada.

[24] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A Framework for eBPF-Based Network Functions in an Era of Microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, March 2021. ISSN 1932-4537. doi: 10.1109/TNSM.2021.3055676. Conference Name: IEEE Transactions on Network and Service Management.

[25] Till Miemietz, Maksym Planeta, and Viktor Laurin Reusch. New Mechanism for Fast System Calls, December 2021. URL http://arxiv.org/abs/2112.10106. arXiv:2112.10106 [cs].

[26] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing Memory-mapped I/O for Fast Storage Devices. pages 813–827, 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/papagiannis.

[27] Ivy Peng, Roger Pearce, and Maya Gokhale. On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 183–190, September 2020. doi: 10.1109/SBAC-PAD49847.2020.00034. ISSN: 2643-3001.

[28] 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*, pages 47–63, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/planeta.

[29] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 152–158, Ann Arbor Michigan, June 2021. ACM. ISBN 978-1-4503-8438-4. doi: 10.1145/3458336.3465281. URL https://dl.acm.org/doi/10.1145/3458336.3465281.

[30] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 352–367, New York, NY, USA, March 2022. Association for Computing Machinery. ISBN 978-1-4503-9162-7. doi: 10.1145/3492321.3519569. URL https://doi.org/10.1145/3492321.3519569.

[31] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 195–211, New York, NY, USA, October 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483569. URL https://doi.org/10.1145/3477132.3483569.

[32] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. {XRP}: {In-Kernel} Storage Functions with {eBPF}. pages 375–393, 2022. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zhong.

[33] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. Userspace Bypass: Accelerating Syscall-intensive Applications. pages 33–49, 2023. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/zhou-zhe.