Karlsruhe Institute of Technology

Kernel Bypass for Storage I/O with io_uring

Bachelor's Thesis
submitted by

Floy Schneider

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Yussuf Khalil, M.Sc.

November 20, 2023 – April 20, 2024

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, April 20, 2024

Abstract

In recent years, the access latency of non-volatile storage devices has decreased, with an increase in bandwidth. This has led to the system call and kernel overhead becoming the bottleneck for storage I/O. The Linux kernel recently introduced the io_uring interface for I/O operations to address these and other shortcomings. This interface is based on two ring buffers shared between a userspace process and the kernel. One ring buffer is used to submit commands to the kernel, the other to receive the results.

Alternatively, allowing applications direct access to storage device bypassing the kernel shows significant performance benefits for I/O-heavy applications. Previous approaches such as Intel SPDK rely on a custom interface for kernel bypass which requires explicit application support. Also, they commonly allow an application full access to the storage device with no isolation on a process or file system level.

In this thesis, we present a library for transparently providing kernel bypass for storage I/O based on the io_uring interface. This means our approach is usable with a range of existing applications without modification. We chose an FPGA-based PCIe device equipped with Intel Optane Persistent Memory for our experiments. By utilizing this FPGA device to restrict access via kernel bypass to specific ranges of the persistent memory, we are able to provide similar isolation guarantees as the kernel for access to the storage device.

We show compatibility of our library with existing applications and are able to demonstrate read latency speedups of up to 3.37× compared to kernel-based io_uring when reading files which are not in the page cache.

Contents

Chapter 1

Introduction

Over the last years, especially with the arrival of higher performance flash memory and new storage technologies like Intel Optane [15,21], performance of non-volatile storage has improved both in terms of latency and bandwidth. For traditional system call-based storage I/O the mode switch and kernel overhead have now become the bottleneck for application performance [15, 33].

A new interface called io_uring was recently introduced into the Linux kernel to try to alleviate some of these issues. Commands and results are communicated between the kernel and application via ring buffers in memory shared between application and kernel. This reduces the number of copy operations compared to traditional I/O. Also, an application can place many commands in the submission ring buffer and submit all of them with a single system call [10, 15].

The io_uring interface has been adopted by multiple applications and libraries. Several databases are able to use it, such as the NoSQL wide column data store ScyllaDB, compatible with Apache Cassandra [14, 32] (via the underlying seastar framework [31]), Dragonfly, a drop-in replacement for the popular Redis data store [26] (via the helio framework [16]), and the ClickHouse database [12]. Also, io_uring support is transparently used in the JavaScript runtime NodeJS via the libuv library [25, 29].

In this thesis, we propose adopting io_uring as a means of kernel bypass. By handling storage I/O commands submitted by the application fully in userspace, we can avoid the overhead introduced by the submission system call and kernel I/O stack. By adopting a well-established interface, we can support existing applications without modifications and provide transparent kernel bypass to them. This is in contrast to approaches like Intel SPDK, which utilizes a custom API and thus requires specific support in applications [39].

To gain direct access to the storage device from user space while maintaining desirable properties usually provided by the kernel I/O stack such as access control, we utilize the FPGA based storage device presented by Khalil in 2022 [22, 37].

In this thesis, we will first discuss existing approaches for kernel bypass for storage I/O and present the io_uring interface and the FPGA-based storage device in more detail in Chapter 2. Then, in Chapter 3 we present the design of our library for achieving kernel bypass based on io_uring and consider different design options for its components. In Chapter 4, we present our implementation in detail. Then, we evaluate our approach in Chapter 5 with benchmarks and discuss the results. Finally, we present our conclusion in Chapter 6 and propose possible improvements to our approach and potential future research.

Chapter 2

Background

To give some context to the goals of this thesis, we will discuss some prior work for providing kernel bypass to applications and will highlight their shortcomings which this thesis attempts to address. Then, we give a more thorough introduction to the technologies mentioned in Chapter 1 which this thesis will build upon.

2.1 Related Work

Multiple approaches for offering kernel bypass for storage I/O have been proposed in the past. We will present some of these and highlight their benefits and downsides.

2.1.1 Intel SPDK

Presented by Intel in 2017, the Storage Performance Development Kit (SPDK) project provides a user-space implementation of an NVMe driver, block device abstraction, and basic file system abstraction [39]. It also provides a library for writing high-performance applications on top of these implementations. By exposing the NVMe queues of the storage device directly into userspace, all overhead usually incurred by kernel mode switches and copies between kernel and user space memory can be avoided. Also, SPDK uses asynchronous polling instead of interrupts to detect completion of a request and it utilizes a lockless architecture, overall providing performance benefits from 6× up to 10× more IOPS on an NVMe drive compared to utilizing the kernel driver.

However, the design does not support traditional Linux file system implementations, only offering a simpler non-POSIX-compliant file system. It therefore cannot benefit from services traditionally offered by the kernel such as access control [15,39] and may not be compatible with existing applications. Also, because it grants direct access to NVMe queues, SPDK allows the application unrestricted access to the whole storage device.

In conclusion, SPDK offers full kernel bypass for storage I/O, however it gives up any kernel cooperation for access control. This may be appropriate for use cases where a single application can safely be granted exclusive full access to a storage device, such as cloud storage solutions where the application itself enforces access control [15]. However, in cases where isolation between user space and the storage device is required, this approach falls short. In addition, the benefits of SPDK can only be utilized by applications specifically built with support for its custom interface. For example, the Ceph storage system has specific support to utilize SPDK to access the underlying block devices [11].

2.1.2 rkt-io

Building on top of SPDK, rkt-io, presented in 2021 as a “Direct I/O stack for Shielded Execution,” [33] has its focus on Trusted Execution Environments (TEEs) such as those provided by Intel SGX [8] or AMD SEV [7], providing stronger security guarantees for their storage and network I/O stacks when running on untrusted host systems. They achieve this by moving the file system and block device layers usually provided by the kernel into user space implementations running inside the TEE and use SPDK to access the actual storage device. In doing so, they expose a fully POSIX/Linux ABI-compatible interface to the application, making the kernel bypass transparent to the application [33].

However, like SPDK itself, this approach still provides full exclusive control over the storage device to userspace. Also, the focus of rkt-io on Trusted Execution Environments may make the approach less appealing in scenarios where applications do not require this level of isolation. While offering significantly higher storage throughput compared to other I/O solutions for TEEs, it still offers significantly worse performance (about a quarter of read throughput and a third of write throughput) compared to native system call-based storage I/O [33].

In addition, serious concerns have been raised about the level of confidentiality technologies like Intel SGX purportedly provide. Side-channel attacks which allow retrieval of memory contents from the TEE have been demonstrated [36], allowing to break many of the fundamental guarantees provided by SGX. In practice this means an overreliance on the security promises of SGX may actually weaken the entire system [35].

2.2 Introduction to io_uring

This thesis proposes achieving kernel bypass by utilizing the io_uring interface. To clarify our approach, we first give an introduction to how this interface is used by an application, which is further visualized in Figure 2.1.

The io_uring interface was added to the Linux kernel with version 5.1 in May 2019 [15]. It offers a new interface for asynchronous I/O based on ring buffers in memory shared between userspace and the kernel [10].

Setup Using the `io_uring_setup()` system call, an io_uring instance is set up with a specified number of entries in the ring buffers and additional optional parameters [10]. On success, the kernel will return a file descriptor that is later used to refer to the specific io_uring instance.

Submission To submit requests for processing to the kernel, the application places them into the Submission Queue (SQ) [10]. The Submission Queue is made up of structs referred to as Submission Queue Entries (SQEs), each containing an `opcode` that identifies the requested operation, arguments such as a file descriptor, and additional arguments and flags. The operations and arguments correspond closely to existing system calls such as `read()` or `write()`. Additionally, they contain a 64 bit field for user data which is returned again in the corresponding completion. This is commonly used to hold a pointer into the application heap.

By default, all submissions are processed independently of each other. If sequential processing is required, an SQE can be *linked* to the subsequent one by setting a flag [10]. If a request fails, all subsequent requests linked to it are canceled.

To notify the kernel of the new submissions, the application updates the tail of the Submission Queue and issues an `io_uring_enter()` system call with the file descriptor referring to the io_uring instance [10]. This will cause the kernel to update the head of the ring buffer and start processing the submissions.

Alternatively, the io_uring instance can be set up for kernel-side polling. In this mode, a kernel thread automatically monitors the Submission Queue and picks up submissions without requiring a system call [10, 15]. This mode will not be further explored by our implementation but briefly discussed in Chapter 6.

Completion After processing the submitted request, the kernel will place the results into the Completion Queue (CQ). Each Completion Queue Entry (CQE) contains an integer result, which matches the return value of the system call corresponding to the io_uring operation. For example, for a `READ` operation, it contains the number of bytes read from the file descriptor.

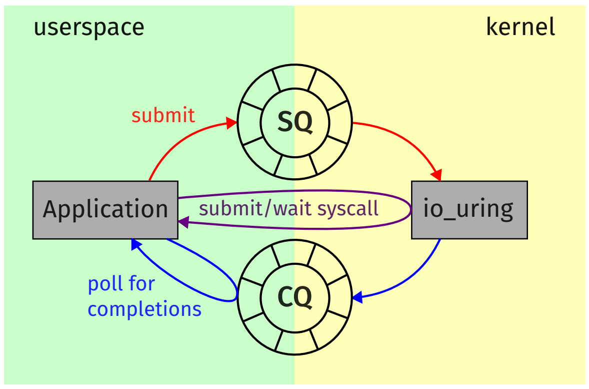

Figure 2.1: Interaction between an application and io_uring. The SQ and CQ are shared between userspace and kernel. The application submits requests to the SQ and issues a system call to notify the kernel. The kernel takes the SQEs, processes them, and places completions into the CQ. The application polls the CQ or waits for completions via system call.

Additionally, the CQE contains a field for flags which appears to currently be unused, and the user data provided in the submission [10]. The user data field allows the application to correlate completions with the earlier submissions. This is necessary, as submissions can be processed out-of-order unless explicitly linked [10].

To retrieve the completions after submission, the application can poll the Completion Queue tail set by the kernel. Alternatively, it can call `io_uring_enter()` again with the number of completions that should be awaited. This system call can signal additional submissions too, potentially further reducing the total number of required system calls [15]. Once the CQEs have been processed by the application, it updates the CQ head accordingly to signal back to the kernel that these CQ slots can safely be overwritten.

liburing Library To simplify the use of the io_uring interface, the liburing library can be used. It provides a simplified API for setting up and interacting with the io_uring interface [10]. We will explain some of the functions it provides later in this thesis when we go through our implementation in Section 3.3.

2.3 Introduction to dcpmm-fpga

Our design heavily relies on the FPGA-based device for accelerating access to non-volatile storage presented by Khalil in 2022 [22]. We will refer to the device as *dcpmm-fpga* in this thesis. The FPGA device contains an Intel Optane persistent memory DIMM [21] and is connected to the computer via PCI Express.

Its main functionality is a `memcpy`-like interface for copying data between system RAM and the persistent memory. This is implemented as two ring buffers for commands, one for each copy direction, from the host to the FPGA device and vice versa. These ring buffers are exposed as Base Address Register (BAR) regions to the operating system. Similar to io_uring, signaling is done via head and tail pointers, with the tail pointer being updated by the CPU and the head pointer by the FPGA. This interaction is demonstrated in Figure 2.2.

There is no separate ring buffer for signaling completion of a request. Instead, all requests are processed in-order and completion is signaled to the application by dcpmm-fpga updating the head pointer [22].

Block Device Driver To allow use with standard OS interfaces, a block device driver was later implemented. It exposes the persistent memory attached to the FPGA as a standard Linux block device. This allows the use of any Linux file system on the device [37].

Virtual Functions In addition to the privileged access utilized by the block device driver, dcpmm-fpga also exposes multiple PCIe Virtual Functions (VFs) via the Single Root I/O Virtualization (SR-IOV) feature. Each VF offers its own set of ring buffers for command submission. Additionally, dcpmm-fpga offers a simple MMU for restricting access per VF to specific address ranges with 2 MiB granularity [37].

The Virtual Functions are managed via the character device provided by the dcpmm-fpga kernel module. By opening the character device, a VF is allocated to the process. Then, the `ioctl` system call can be used to issue different commands, for example preparing pages from user space for DMA access from the FPGA [22].

It is important to note that the isolation mechanism and security guarantees provided by the Virtual Function and MMU rely on the system having an IOMMU which can isolate the I/O address space belonging to a VF from other processes [22].

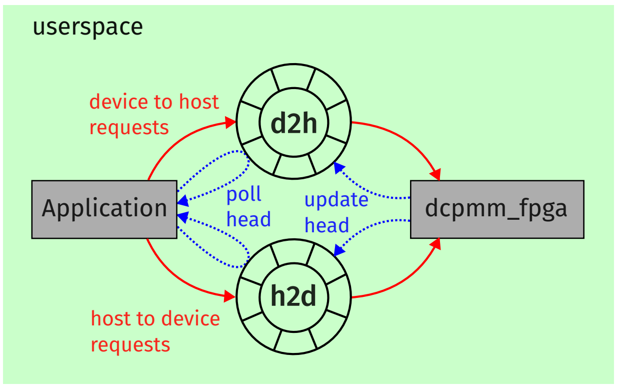

Figure 2.2: Interaction between an application and dcpmm-fpga. The application places requests into the appropriate ring buffer. The FPGA executes them and updates the head index on success. The application polls the head position to wait for completion.

Chapter 3

Design

In this chapter, we present our design for achieving kernel bypass by intercepting the io_uring interface. We discuss different options for the implementation of this approach and their benefits and downsides.

3.1 io_uring Call Interception

io_uring is a system call-based interface offered by the Linux kernel [10], as described in Section 2.2. To achieve kernel bypass, we need to intercept the application before it actually issues the system calls. Multiple approaches for this are conceivable.

Interception Target One option is to intercept all system calls, check the system call number, and either forward the system call as-is or jump into a custom user space handler instead. This can be achieved by overriding the system call wrapper provided by libc. This has two distinct downsides: It introduces some overhead, albeit low, to *all* system calls. Additionally, applications may not necessarily use libc to issue system calls in the first place, such as those written in the go programming language [23]. This means this approach is not fully universal.

Alternatively, a new approach called zpoline, presented by Yasukata et al. in 2023 [40] relies on binary rewriting of the target application. It replaces system call instructions with jump instructions of the same instruction length into a custom handler. Thus, it is more universal than targeting the system call wrappers provided by libc. However, this approach relies on some specifics of the x86-64 CPU architecture and is for example not applicable to ARM CPUs.

Another option is interception at an even higher level: liburing is the recommended wrapper library for interacting with io_uring [10] and is commonly used by applications utilizing the io_uring interface such as ScyllaDB [14] and

ClickHouse [12]. This excludes applications which utilize the system call interface directly, but in turn avoids having to intercept all system calls. Due to the more high-level nature of the liburing interface, this method is possibly simpler to implement than intercepting system calls directly.

Interception Method To intercept calls to libc or liburing, there are still different methods of how *exactly* to achieve interception. Depending on the needs of the target application, two options are available:

If the application is *statically* linked against the library, some symbols provided by the library can instead be wrapped at compile time using the `-wrap` option of the linker and overridden by a custom library at link time [6].

Alternatively, if the application is *dynamically* linked against the library, a simpler approach can be taken: A custom library containing the symbols to be overridden can be specified in the `LD_PRELOAD` environment variable to instruct the dynamic linker to load a custom library first [4], which can override some symbols otherwise provided by the target library. Arguably, this is the more elegant approach as it does not require recompilation of the application and allows quick experimentation.

3.2 Access Control

One important reason for isolating devices from applications using an operating system is for enforcing constraints on how applications can access them. For storage devices, this includes enforcing what kind of access certain users (and their processes) have to specific files [9]. When implementing any sort of kernel bypass, these constraints have to either be waived or weakened, or implemented in some other way.

The dcpmm-fpga device as discussed in Section 2.3 offers a basic implementation of an MMU which is configurable from its kernel module and can be used to grant specific Virtual Functions access to ranges of addresses on the attached persistent memory. This can be used to limit what areas of the storage device the application can access via the kernel bypass.

To determine which ranges the application should have access to when trying to access a file, the permissions have to be checked and the sectors on the storage device belonging to the file have to be determined by the kernel. We will further describe how we achieve this in Section 4.1.

3.3 Design of libuseruring

This thesis presents a library we call *libuseruring* which provides a user space implementation of the liburing interface. It overrides the symbols provided by liburing for the setup of an io_uring instance, submission of commands, and retrieval of results. By conforming to the interface provided by liburing (and the kernel's implementation of io_uring), an application can seamlessly use this library instead and transparently benefit from kernel bypass.

Because only some of the operations offered by io_uring refer to pure storage I/O (namely `read[v]()` and `write[v]()`) and can thus be handled completely in user space, kernel support is still needed for the rest of the operations, such as the socket-related operations `ACCEPT` or `SEND`. Theoretically, these could all be implemented using traditional system calls if completely avoiding kernel-based io_uring is desired, in turn losing most of the benefits of the interface. The approach taken in this thesis is to instead utilize a regular kernel-based io_uring instance and forward all operations that cannot be handled in user space to the kernel. This kernel backend is shown in the top right of Figure 3.1.

In addition to the kernel backend, a dcpmm-fpga-based backend is introduced. Incoming commands can then be dispatched to the appropriate backend, and results gathered from all backends and provided back to the application. The dcpmm-fpga backend is shown in the bottom right of Figure 3.1.

Linking An additional challenge arises from the possibility to *link* multiple operations, as described in Section 2.2: By setting a flag on a Submission Queue Entry, the following SQE is linked to it, meaning its execution will not start before the previous SQE completed successfully. If the first operation results in an error, the linked SQEs are not executed [10]. When dispatching all linked SQEs to the kernel, this causes no issues as the ordering can then be enforced by the kernel. Also, if all linked requests are dispatched to the dcpmm-fpga, ordering is preserved as the command queue is always handled in-order [22].

However, if one request is to be dispatched to one backend and the second to another, linking has to be ensured some other way. Perhaps the simplest solution is to always dispatch linked requests to the kernel, sidestepping the issue at the cost of not using kernel bypass even when available. Alternatively, information about the linked request can be attached to the first request, and only upon completion, the second request is dispatched.

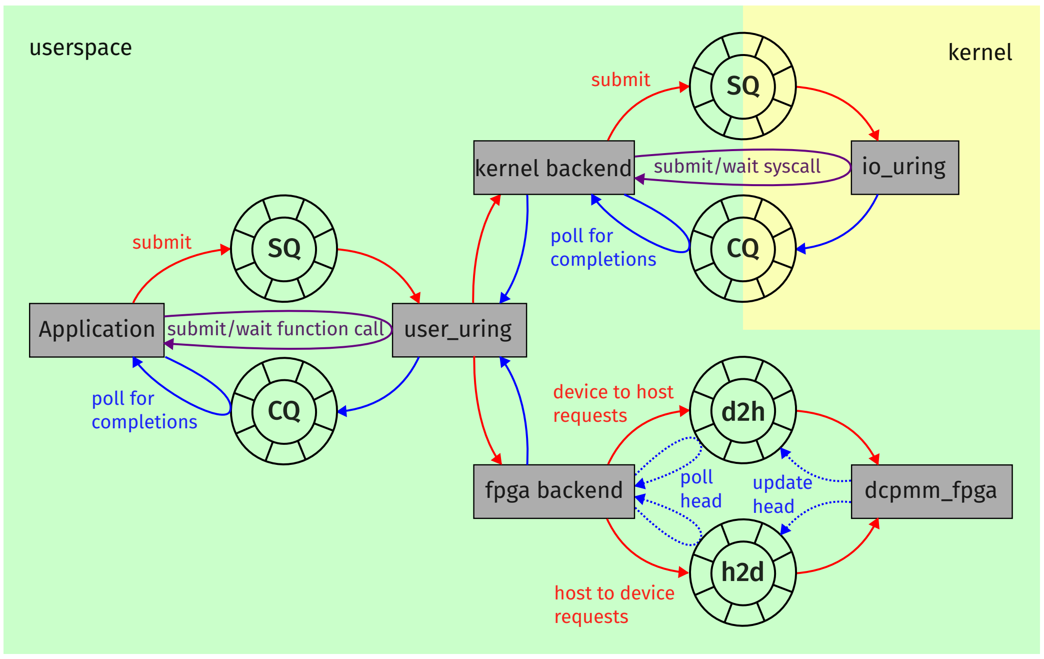

Figure 3.1: Interaction between an application and the libuseruring library. The application has the same ring buffer-based interface as in Figure 2.1. On the other side, libuseruring dispatches commands to either the kernel-based io_uring instance or to the command queues of dcpmm-fpga.

Chapter 4

Implementation

After discussing the general approach and design considerations for implementing kernel bypass using the io_uring interface in the previous chapter, we will now present our implementation. We will start by examining the most low-level additions required to the Linux kernel itself, then present necessary changes to the kernel driver for dcpmm-fpga and its accompanying library, and finally present the implementation of libuseruring itself.

4.1 Kernel Patches

As mentioned in Section 3.2, the dcpmm-fpga supports access control for VFs via a block-based MMU. We need to configure this MMU accordingly to only grant the userspace process access to specific files.

To achieve this, the physical extents of the file on the storage device have to be determined. How files are laid out and managed on the block device differs by file system. Some modern file systems such as ZFS offer advanced features such as RAID support or encryption [13, 38]. The interface offered by dcpmm-fpga currently does not support these advanced features directly. Instead, we can only support file systems where file contents directly correspond to sectors on the block device.

We choose the exFAT file system for our demonstration, as it is comparatively simple in design, while still offering the most basic features expected of modern file systems, such as support for large files and unicode [28]. It uses a simple file allocation table and stores file contents in clusters, whose size is determined based on the size of the entire file system [27]. Due to this simplicity, the required changes are minimal and easy to integrate into the Linux kernel.

To retrieve the extents of a file, we add and export one function from the Linux kernel exFAT module: `exfat_get_extents_for_inode()`. It receives a kernel inode structure[1], a pointer to a target buffer for the file extents, and a pointer to the length of the buffer. Then, it determines the physical offset of the file system partition on the block device and the exFAT cluster size of the specific file system. It calls the internal exFAT function `exfat_map_cluster()`[2] repeatedly with incrementing cluster index to determine the sector offsets of all the clusters of the given inode. These offsets are converted to physical addresses and written back into the target buffer as pairs of start and end addresses for each cluster or group of contiguous clusters. The specific cluster size of the file system is used for this, together with the actual size of the file. This ensures we truncate the last extent to the actual file length instead of also including the rest of the cluster.

The function returns an error if the given buffer is not long enough to store all extents of the file. On success, the provided length is updated to signal to the caller how many addresses were written to the buffer.

4.2 dcpmm-fpga Kernel Module and Library Changes

To expose the described functionality of determining file extents to applications and to configure the MMU to allow access to the file, changes to the dcpmm-fpga kernel module are required. Additionally, to make this feature conveniently available to applications, we make some additions to the dcpmm-fpga library.

4.2.1 Kernel Module Additions

The kernel module for the dcpmm-fpga provides a character device for communication with userspace processes [22], as previously described in Section 2.3. The process can then use the `ioctl` system call to issue requests.

We introduce a new ioctl request `IOCTL_REQUEST_FILE`. It takes a pointer to a `dcpmm_fpga_request_file_data` struct which contains the file descriptor number for the requested file, a pointer to a buffer for the file extents in the same format as described in Section 4.1, and the buffer length.

The kernel module then gets the in-kernel representation of the file descriptor using `fdget()`[3], retrieves the contained inode and checks that the file actually resides on an exFAT file system. It then calls `exfat_get_extents_for_inode()` as defined in Section 4.1 to retrieve the file extents from the file system. Next, it iterates over the extents to add appropriate mappings to the MMU for the current virtual function. The MMU currently only supports a block size of 2 MiB [37], which is larger than the default cluster size even up to a 256 TB drive [27].

[1]`struct inode` in Linux 6.3 source, `include/linux/fs.h:641`
[2]In Linux 6.3 source, `fs/exfat/inode.c:111`
[3]In Linux 6.3 source, `include/linux/file.h:61`

To still ensure full access to the file, we round up to a full block if an extent does not fully cover an MMU block. This however breaks the principle of only giving access to the actual specific files and may expose other parts of the file system to the application. This is undesirable and could be overcome in the future by tuning the block size of the MMU to that of the desired cluster size.

After retrieving the file extents and configuring the MMU accordingly, the extent buffer and length are returned to userspace by updating the provided struct.

4.2.2 dcpmm-fpga Library Changes

To give convenient access to the newly introduced ioctl command described in Section 4.2.1, we add a thin wrapper `request_file_dcpmm()` to the library.

For initiating an asynchronous transfer between main memory and the dcpmm-fpga-attached persistent memory, the two functions `memcpy_from_dcpmm()` and `memcpy_to_dcpmm()` are provided by the library [22]. Additionally, `*_wait()` variants of the functions are provided which busy-loop until the completion pointer of the dcpmm-fpga has reached the submitted request.

However, to allow fully asynchronous operation, we need the ability to submit multiple requests and wait for them at a later point in time. To achieve this, we modify the `memcpy_*` functions to return the index after the submitted request. We add the `dcpmm_fpga_get_[h2d|d2h]_completed_pointer()` functions to make it possible to wait for the completion of a specific request at a later point in time.

4.3 libuseruring Implementation

To intercept the calls to io_uring, we choose the `LD_PRELOAD` approach, as described in Section 3.1. Our library libuseruring provides a shared object that can be preloaded which overrides a handful of symbols usually provided by liburing. These symbols internally resolve to implementations with the `user_uring*` prefix as opposed to `io_uring*` used by liburing [19].

Since our library also needs to interface with an actual kernel-based io_uring instance, we also want to use some symbols provided by liburing. Usually, the dynamic linker automatically resolves symbols on application startup. However, we are ourselves overriding symbols also provided by liburing. By calling `dlsym()`, a symbol can be resolved at runtime [3]. By additionally passing the `RTLD_NEXT` handle, instead of returning the first occurrence of the symbol in the search order (which would the implementation provided by our library), the next occurrence after the current object is returned, which will be the symbol provided by liburing.

4.3.1 Initialization

When an io_uring instance is requested by the application via the intercepted `io_uring_queue_init[_params]()` functions, we take multiple steps:

A kernel-based io_uring instance is initialized in the kernel backend by calling the real init function provided by liburing. The same parameters as provided by the application are passed.

Also, the dcpmm-fpga backend is initialized by calling `dcpmm_fpga_init()` provided by the dcpmm-fpga library. This allocates a virtual function for the process and sets up the command queues for communication to the FPGA.

Finally, regular heap buffers are allocated for the submission and completion queues and the `io_uring` struct is set to point to these buffers instead of the buffers shared with the kernel and is then returned to the application. Also, instead of passing the file descriptor referring to the real kernel-based io_uring instance, an invalid file descriptor is returned. This way, we notice if the application tries to directly issue `io_uring_enter()` system calls instead of utilizing our library. This is useful for catching potential bugs or other liburing functions that also need to be overridden.

We will refer to the "fake" io_uring instance returned to the application as the *user uring*.

4.3.2 Dispatching Submissions

To submit new requests, the application adds SQEs to the Submission Queue and then calls `io_uring_submit()`. We intercept this function and retrieve new submissions from the SQ based on the head and tail pointers of the queue and update the head pointer accordingly to signal completed dispatch of the submissions. The full dispatch flow is shown in Figure 4.1.

File Descriptor Hashmap To keep track of which file descriptors can be handled by the dcpmm-fpga backend, we keep a hashmap of all encountered file descriptors. We utilize the *uthash* library [17] for managing the hashmap. The file descriptor number is used as the key, the values of the hashmap are `user_uring_file` structs. They contain the file descriptor number, a pointer to a buffer for the physical extents of the file as determined by the dcpmm-fpga kernel module, and the length of the buffer.

When a file descriptor is first encountered, its extents are requested and an MMU mapping established by calling `request_file_dcpmm()`. If an error is returned, the file cannot be handled by the dcpmm-fpga backend. To signal this, the buffer pointer in the struct is set to the `NULL` pointer. Either way, the resulting `user_uring_file` struct is added to the hashmap for later lookups.

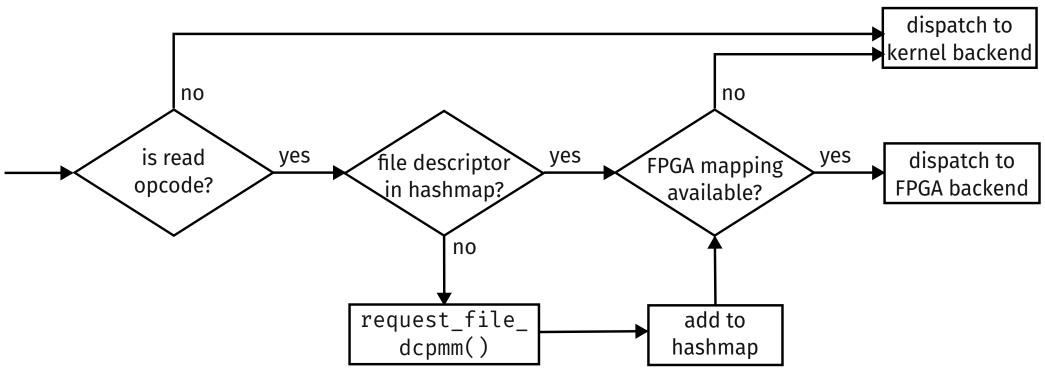

Figure 4.1: Flow for dispatching submissions in libuseruring. The opcode and file descriptor of the operation are checked. Depending on the results, the submission is dispatched to the kernel or FPGA backend.

Dispatch When new SQEs are submitted to the library, they have to be dispatched to the appropriate backend. First, the opcode of the SQE is checked. Currently, we have only implemented support for `READ` and `READV` requests in the dcpmm-fpga backend, so all other requests are immediately dispatched to the kernel backend instead. In the future, write support can be added.

If a read request is dispatched, the file descriptor hashmap is checked to retrieve (or add, if not found) the corresponding `user_uring_file` struct. If extents for the file are available, the request is dispatched to the dcpmm-fpga backend, otherwise the kernel backend is used.

Each time an SQE is dispatched to a particular backend, a corresponding `inflight` counter is incremented. This counter is needed when waiting for completions to determine how many completions can be provided by each backend.

Completion liburing offers multiple functions for retrieving and optionally waiting for completions. They internally all refer to `__io_uring_get_cqe()`, which is overridden by our library.

The application can pass a number of new SQEs to this function that should be submitted before getting the completions. In this case, the same dispatch logic as for `io_uring_submit()` is utilized.

Additionally, the `wait_nr` argument specifies the number of completions that should be waited for. Based on the `inflight` counters mentioned above, we can determine how many completions we can expect from each backend at most. However, if we have more requests spread across backends in flight than the requested number of completions, different strategies for distributing the wait requests are possible.

The strategy we choose is to always retrieve completions from the FPGA backend first and prefer waiting for completions from the FPGA backend. If the FPGA requests have already concluded, this results in no waiting, otherwise we will waste some CPU time spinning until the operations have concluded. This way however, we give the kernel some more time to complete the already submitted requests in the kernel-based io_uring instance. If in the kernel backend, the number of already available completions is greater than the `wait_nr` requested, we save having to issue an `io_uring_enter()` system call to wait for the required completions.

Independent of whether a specific `wait_nr` was specified or not, both backends are asked to return the awaited and all additional available completions. To return the completions, the user uring instance is passed to the backends and they place their CQEs into the user uring Completion Queue and update the tail accordingly.

The number of completions received from each backend is then subtracted from the corresponding `inflight` counter.

4.3.3 Kernel Backend

The kernel backend of the library is fairly straightforward, as it primarily copies entries between queues with no manipulation.

For submission, it obtains a Submission Queue Entry from the kernel io_uring instance via `io_uring_get_sqe()` and copies the contents of the SQE provided by the calling application. After all submissions are dispatched from a call to `user_uring_submit()`, the kernel backend calls the *real* `io_uring_submit()` function to issue a system call to instruct the kernel to start processing [10, 19].

Completions are requested via the *real* `__io_uring_get_cqe()`. All returned Completion Queue Entries are then copied to the user uring CQ and marked to the kernel as seen by calling `io_uring_cqe_seen()` to update the head pointer in the kernel-side Completion Queue accordingly.

4.3.4 FPGA Backend

The dcpmm-fpga also utilizes ring buffers for submission of commands as further described in Section 2.3, but some additional work is required to translate between its interface and the one specified by io_uring.

Dispatch

Due to the nature of the dcpmm-fpga interface, multiple steps need to be taken to generate the appropriate commands for the FPGA from an io_uring Submission Queue Entry, and some additional work is needed for being able to generate matching Completion Queue Entries later on.

Submitting read operations Currently, we only support the `READ` and `READV` opcodes in the dcpmm-fpga backend. These correspond fairly directly to the system calls `pread()` and `preadv2()` [1]. A `READ` io_uring submission contains a file descriptor, an offset from the start of the file, the length of the read, and a pointer to a destination buffer. A `READV` submission also contains a file descriptor and offset, and an array of `iovec` structs each containing a destination pointer and length.

To service either of these submission types, the file extents contained in the `user_uring_file` struct as described in Section 4.3.4 are consulted. The internal function `copy_file_from_fpga()` receives this struct, along with an offset, read length, and pointer to the target buffer. It will then iterate through the file extents until the requested offset is reached and issue one or multiple calls to `memcpy_from_dcpmm()` with the appropriate physical address on the block device and offset into the target buffer. If the requested length spans multiple extents of the file, these are copied bit by bit.

Additionally, the dcpmm-fpga currently only supports copying into pages which are contiguous in I/O virtual address space [22]. This is not necessarily the case with buffers provided by applications. To alleviate this somewhat, we further break down the `memcpy` requests to only cover one page size at a time. This works if the buffers are page-aligned, but will still fail otherwise.

Breaking up the copy operation into single pages will likely degrade the performance compared to a single copy operation. This will be further discussed in Section 4.4.1.

The completion index returned by the last call to `memcpy_from_dcpmm()` as modified in Section 4.2.2 is returned by `copy_file_from_fpga()`. Since the FPGA processes requests in sequence, if this last completion index is reached, all previous requests have also concluded [22]. Additionally, the number of bytes for which a copy request was actually issued is returned (which may be lower than the requested length if the file ends before the requested length is reached).

Save Data Needed for CQE To be able to generate an appropriate CQE when requested later, some information about the issued copy requests needs to be retained. For this, we use a ring buffer with the same length as the FPGA queue. It contains `fpga_cqe` structs which store a completion index, integer result and pointer for user data.

After dispatching a read, a new entry is appended to the ring buffer, containing the last completion index of the read, the result (which is the number of bytes for read requests), and the user data pointer provided in the SQE.

4.3.5 Generating Completions

When completions are requested from the FPGA backend, the first entry from the `fpga_cqe` ring buffer is retrieved. It is checked (and potentially waited for) whether the current completion index has reached or passed the one recorded in the struct. If the operation has concluded, a CQE is constructed from the integer result and user data pointer stored in the `fpga_cqe`, and copied into the user uring CQ.

4.4 Limitations

The implementation presented here still has limitations, some of which were already mentioned above and will be repeated here briefly. Some of these are the direct result of limitations of the dcpmm-fpga device, whereas others are introduced by our library.

4.4.1 Limitations Introduced by dcpmm-fpga

MMU Unit Size As mentioned in Section 4.2.1, the unit size of the MMU of the dcpmm-fpga does not match the utilized cluster size. This results in granting user space less restricted access to the block device than desired. This could be rectified by either enforcing a cluster size that matches the MMU block size, or modifying the dcpmm-fpga to allow granting smaller units of memory.

Contiguous Pages As we discussed in Section 4.3.4, on our current development platform, the dcpmm-fpga can only copy from or to buffers which are in contiguous pages in I/O virtual address space. Since our source and target buffers are passed in from the application, we can make no such guarantees. Thus, we have to break up copies into smaller chunks, sacrificing some performance.

Possible solutions could be modifying the allocator used by the application. A more thorough approach might be using Linux kernel DMA scatterlists in the character device driver for the DMA mappings. The most elegant and simplest solution to this problem is likely switching to a platform which supports an IOMMU with Shared Virtual Addressing, as is available from Intel starting with the Sapphire Rapids generation [24]. This would also eliminate the `ioctl` calls necessary to ensure a DMA mapping for a target page [22], which likely helps improve performance as we will show in Section 5.3.

Copy Size Finally, the dcpmm-fpga can only copy data in chunks of 32 bit or 4 bytes. This may not match the requested length of bytes. Problematically, if we round down, we do not copy enough bytes, but if we round up, we may exceed the size of the target buffer and potentially corrupt other heap data.

A possible workaround is to allocate an internal buffer in this case, copy the rounded-up number of bytes into this buffer and then copy the precise number into the application-provided target buffer, at the cost of introducing additional allocations and copies. This has not been implemented yet as we did not encounter this issue in our testing.

4.4.2 Limitations of our Approach

Right now, we only support read access via kernel bypass. Extending the implementation to cover write access is simple at first glance, but some additional questions will be encountered. If a file is written to at its end, the file will need to grow. This requires kernel cooperation from the file system to allocate new clusters to the file.

A related issue occurs with the current approach: The size of an already opened file may change if some process writes to or truncates it. Right now, we have no way of observing these changes and updating the MMU mappings and library representation of extents accordingly. Similarly, a file may be closed by the application, after which the kernel bypass path should also refuse access to the corresponding file descriptor.

Both of these issues may be partially addressed by monitoring writes to files (via both the io_uring instance and other system calls) by the process and requesting updated file extents from the kernel module. This however does not address changes made to the file by another process. Solving this requires further research to guarantee correct behavior, as further discussed in Section 6.1.3.

Chapter 5

Evaluation

In this chapter, we describe our approach for initially developing and validating our library libuseruring. We then present benchmarks to compare native kernel-based io_uring performance with kernel bypass using our approach and assess the overhead introduced to requests forwarded to the kernel.

5.1 System Setup

All benchmarks were executed on a system with an Intel Core i7-12700K CPU on an msi PRO Z690-A mainboard with four 8 GiB DDR5 memory DIMMs and a 1 TB Samsung 980 NVMe SSD. The dcpmm-fpga device is attached via a PCIe 3.0 ×16 link.

The Intel Core i7-12700K is an SMT-capable 12-core (8 performance, 4 efficiency cores) x86-64 processor from the Alder Lake generation [20].

We used Ubuntu 20.04 with a modified Linux 6.3.0 kernel and liburing 2.4.1, backported from Ubuntu 23.10. The kernel as well as our other components were compiled with gcc 11.4.0 with optimization level `-O2`. Our library libuseruring was compiled with optimization level `-O3` and the `-flto` setting to enable link-time optimization.

5.2 Initial Validation

For initial development and validation we used the `cat_liburing` tool from the suite of io_uring examples from the *Lord of the io_uring* guide [18, 19]. This program reads a file via io_uring and prints its contents to `STDOUT`. Hence, it was chosen as an ideal minimal example for demonstrating io_uring interface interception and kernel bypass for read operations. It issues a single `READV` operation for reading the file into buffers, as further described in Section 4.3.4.

5.3 cat_liburing Benchmark

Since `cat_liburing` issues a single read access, it is an ideal target for comparing the performance of a kernel-based read compared to our approach for kernel bypass.

cat_liburing Modifications We modified `cat_liburing` to measure the time between submitting the requests to io_uring and receiving the results to exclude any initialization overhead from our benchmarks. For this, we obtain high precision timestamps using `clock_gettime()` (using the `CLOCK_MONOTONIC` clock) [2] just before calling `io_uring_submit_and_wait()` and right after it returns. We then calculate the difference and print it to `STDERR`. This allows us to measure just the submission and processing latency.

With the default kernel configuration, `cat_liburing` is only able to read files up to 1 MiB, since the maximum number of iovecs supported by `READV` is 1024 [5], and `cat_liburing` uses 1 KiB iovec target buffers. We increased the target buffer size to 4 KiB (the typical page size on x86), and modified `cat_liburing` to split the read into multiple `READV` operations if the iovec limit is reached to allow reading larger files.

Benchmark Setup We used a python script to execute `cat_liburing` for a specified number of iterations, redirecting the `STDOUT` output to `/dev/null` and collecting the timing results from `STDERR` into a file. We use this script to run different configurations of `cat_liburing` on files of increasing sizes with random content. We ran `cat_liburing` with kernel-based *native* io_uring, with libuseruring in *passthrough* mode without kernel bypass, and finally with libuseruring with kernel bypass.

For the *native* and *passthrough* methods, we also added variants where we clear the page cache between executions to force the file contents to be read from the dcpmm-fpga instead of the page cache. To clear the caches, we execute `echo 3 > /proc/sys/vm/drop_caches` before each `cat_liburing` invocation [34].

Initial testing showed significant overhead introduced in the dcpmm-fpga library for ensuring a DMA mapping is present for the target page of a copy. Setting up this mapping requires another system call for each target page [22, 37]. This would not be required on a platform with Shared Virtual Addressing, as mentioned in Section 4.4.1. To quantify this overhead, we added a *-premap* variant of the benchmark. Here, we manually request DMA mappings for all target buffers before submitting the request, removing the overhead from the time we measure.

We then use a second python script to read the files with the timing results of each benchmark run and calculate the mean and standard deviation of the results.

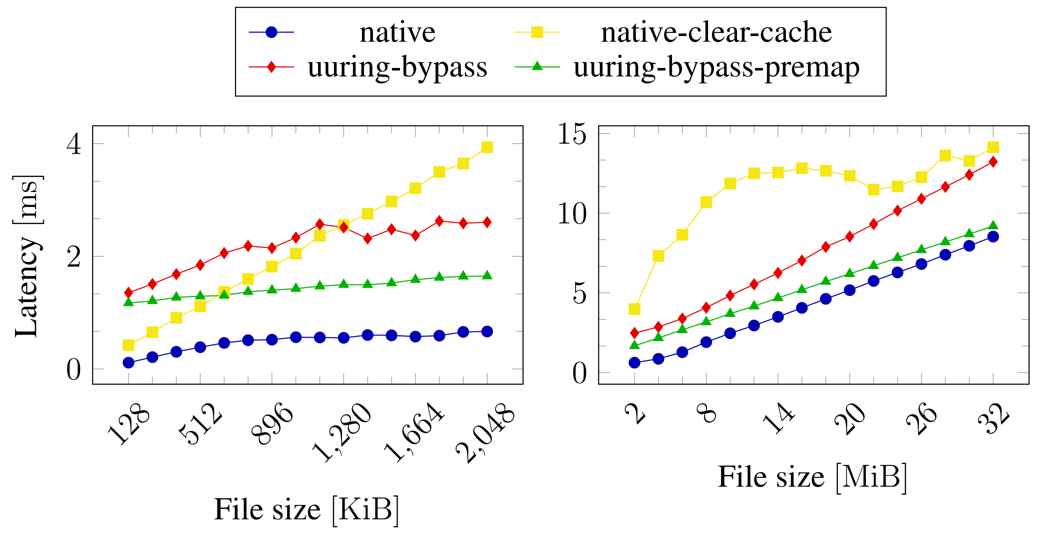

Figure 5.1: Read benchmarks using cat_liburing comparing *native* io_uring (with and without page cache) with our design *uuring* (with and without pre-mapped target pages). The performance of our solution surpasses the kernel without page cache at 1 MiB, whereas the kernel with caching remains the fastest variant across all file sizes.

Benchmark Results For each data point, we collected the timing from 100 invocations of `cat_liburing`. We executed the benchmarks on two adjacent ranges of file size, from 128 KiB to 2048 KiB, and from 2 MiB to 32 MiB.

In the first graph in Figure 5.1, we observe fairly linear behavior of all benchmark variants. For all read sizes, native io_uring has the shortest read times, likely because the target file resides in the page cache after the first invocation, meaning only a copy in main memory is necessary instead of a DMA transfer from the dcpmm-fpga. This is confirmed by comparing native execution to native execution without the file residing in the page cache, which has a significantly higher slope. Our design libuseruring (*uuring*) starts at a slightly higher base latency than the native variants. We are unsure why this is the case. From 640 KiB, our solution with pre-mapped pages is faster than kernel-based io_uring if the file is not in the page cache, from 1 MiB, our solution without pre-mapping also is.

In the second graph in Figure 5.1, we observe some less-than-linear growth in the native io_uring execution without the page cache. We assume some optimization in the read path of the kernel is starting to take effect around this size.[1] All other benchmark variants show close to linear growth, with kernel bypass without

[1]Some experimentation confirmed a similar effect on an NVMe block device, suggesting this effect is not unique to the dcpmm-fpga.

pre-mapping being consistently faster than native io_uring without caching, with a maximum speedup of 2.62× at 8 MiB. In turn, libuseruring with pre-mapped pages is consistently faster than without pre-mapping, with a speedup of up to 3.37× compared to native execution without caching, again at 8 MiB. Native io_uring with the page cache is still the fastest option. However, we can see our approach with pre-mapping closing the gap to native execution with larger file sizes.

For both size ranges, using libuseruring in passthrough mode (forwarding all requests to a kernel io_uring instance) yielded very consistent low overhead for dispatching the requests and collecting results with a maximum additional average latency of 246 µs across all file sizes. Hence, we decided to exclude these series from the graphs in Figure 5.1 to improve clarity.

We initially planned to benchmark a third range of file sizes from 32 MiB to 512 MiB to investigate whether libuseruring with pre-mapping will eventually become faster than native io_uring even when the file is in the page cache. However, we unfortunately encountered issues with establishing the MMU mappings as described in Section 4.2.1. It appeared as if after a certain number of iterations of granting large MMU ranges to a VF and freeing those VFs, read requests to the dcpmm-fpga started to hang, likely because of an MMU fault.

We were unfortunately unable to fix or work around this issue before the end of this thesis. However, initial experimentation suggested continued linear growth of all test series, suggesting libuseruring with pre-mapped pages will indeed be faster than kernel-based io_uring for larger file sizes.

5.4 Demonstration and Benchmark with ScyllaDB

To evaluate our approach on a more realistic workload, we choose ScyllaDB, which also provides the cassandra-stress utility which can be used for benchmarking [30].

We compare native execution of ScyllaDB on kernel-based io_uring to running ScyllaDB with the required `LD_PRELOAD` environment variables to use libuseruring, but with all kernel bypass disabled, so it just uses the kernel backend. As described in the following, we were unable to benchmark ScyllaDB under libuseruring with kernel bypass via the dcpmm-fpga.

Benchmarks were based on a development build of ScyllaDB version 5.4.0 with io_uring support enabled in the libseastar backend. Since we are primarily interested in comparative data and not absolute performance, a development build was used to ease debugging.

dcpmm-fpga Bug Unfortunately, during work on this thesis, a read-write-ordering bug in the dcpmm-fpga firmware was discovered, affecting both use via the privileged block device driver and when using Virtual Functions. We were only able to

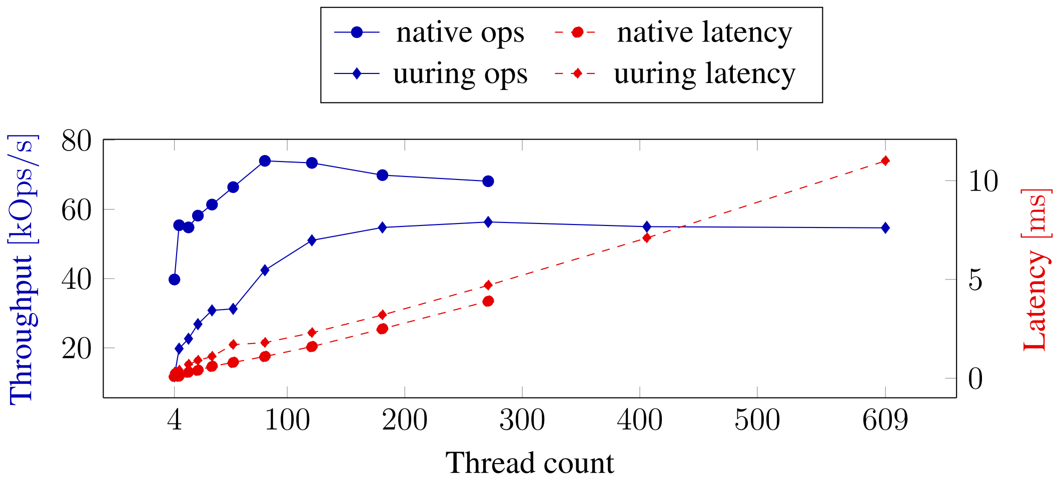

Figure 5.2: Read benchmarks using cassandra-stress on ScyllaDB, comparing throughput in operations per second and request latency of *native* kernel-based io_uring with our design *uuring* in passthrough mode. Native execution consistently shows higher throughput and slightly lower latency than our approach.

reproduce this bug with ScyllaDB and only with large benchmark sizes of about one million write operations. This bug led to partition files of ScyllaDB being corrupted, hence leading to the benchmark being aborted.

The dcpmm-fpga firmware is out of our scope for the matters of this bachelor thesis, and unfortunately we were not able to resolve this bug while ensuring correct operation of the Virtual Functions.

Still, with smaller benchmark sizes, we were able to observe correct operation of both forwarding to the kernel and kernel bypass.

Benchmark Without Kernel Bypass To still collect some insights on the impact our solution has on real-world applications, we choose to run benchmarks on the NVMe SSD of our test system, using libuseruring in passthrough mode. We prepare a test database using `cassandra-stress write`. In Figure 5.2, we show the results of running `cassandra-stress read` executing 1 million read operations with and without kernel bypass. The benchmark utility starts with four concurrent threads for reads and increases the thread count automatically until it identifies the performance peak is surpassed. We observe moderately higher read latency of about 0.8 ms when using libuseruring in passthrough mode. Execution with native io_uring reaches its throughput peak of 74 kOps/s at 81 threads. ScyllaDB running with libuseruring consistently shows lower throughput, reaching its peak of 56 kOps/s only with a much higher thread count of 271.

5.5 Discussion

In this thesis, we set out to provide transparent kernel bypass to applications by intercepting the io_uring interface in order to reduce the overhead introduced to storage I/O operations by the kernel.

In Section 3.1, we discuss different approaches for intercepting calls to io_uring. The `LD_PRELOAD` approach to intercept calls to liburing chosen in this thesis is not able to intercept applications which do not utilize liburing. In practice, this excludes the use of our approach for applications such as NodeJS, as the libuv library does not utilize liburing [25]. Still, we were able to successfully demonstrate interception with the cat_liburing and ScyllaDB applications.

We hoped that the higher-level interface provided by liburing would make it easier to write interception code compared to directly intercepting system calls. However, we discovered many code paths exist in liburing to trigger system calls, so a rather large number of functions needed to be intercepted. We suspect directly intercepting the system calls would lead to less code being required, however this also poses its own downsides discussed in Section 3.1: It introduces additional overhead to *all* system calls, and no truly universal method for achieving this interception exists.

We observe some overhead introduced to io_uring commands which are dispatched to the kernel backend in both benchmarks, with a measured additional latency of 246 µs measured with `cat_liburing` in Section 5.3. However, we measured a higher end-to-end overhead introduced to ScyllaDB of 0.8 ms in Section 5.4. Some overhead is to be expected, as in this case, our library introduces additional checks, function calls, and a copy of the SQE to the kernel Submission Queue. Some of this overhead may be amortized by kernel bypass for read requests. However, as described in Section 3.3, many io_uring operations will still have to be passed to the kernel. This indicates the actual latency overhead experienced by applications depends on their specific usage pattern of the io_uring interface.

For read commands dispatched to the dcpmm-fpga backend, in Section 5.3 we observe speedups of up to 3.37× compared to kernel-based io_uring if the file is not in the page cache. Since our solution is still outperformed by the kernel if the file is already in the page cache, we can conclude the kernel mode change is not the relevant factor here, but indeed our solution is more performant than the kernel I/O read path.

Crucially, our benchmark variant with *pre-mapped* pages should be equivalent to the performance on a platform with Shared Virtual Addressing, as described in Section 4.4.1. The performance of this benchmark variant approaches that of the kernel even if the file is in the page cache. This suggests our solution can outperform the kernel even with cached files for sufficiently large read sizes somewhere above 32 MiB.

Chapter 6

Conclusion

Improvements in the performance of storage devices with novel types of devices such as persistent memory becoming available over the last years have highlighted the kernel can be the bottleneck in I/O-heavy applications. Thus, allowing applications to bypass the kernel is desirable.

Many previous approaches for kernel bypass pose significant downsides, such as requiring use of a custom interface such as with SPDK [39], and granting the application unrestricted access to the storage device, as is the case with SPDK and rkt-io [33] which builds upon it. Thus, applications and the whole system need to be developed with kernel bypass in mind.

In this thesis, we presented an approach for transparently providing kernel bypass to applications by intercepting the io_uring interface. By utilizing the MMU capabilities of the dcpmm-fpga, we also were able to restrict the access of applications to the storage device, although not with the granularity needed for full isolation guarantees.

We showed consistent possible speedups of our approach of up to 3.37× compared to kernel-based io_uring from read sizes above around 640 KiB if the target file is not already in the kernel page cache. We also showed our solution approaches the performance of the kernel even when the file is cached for larger read sizes.

Despite some limitations, we consider our work a successful proof of concept for utilizing the io_uring interface for kernel bypass and were able to demonstrate compatibility with existing applications and possible performance improvements.

6.1 Future Work

We have already discussed several limitations of our current implementation and of the dcpmm-fpga in Section 4.4. We will now present some ideas to further build and improve upon our approach.

6.1.1 Supporting other Backends

The use of the custom dcpmm-fpga device allows us to provide some unique guarantees such as enforcing access control. However, it also limits the universality of our approach.

To make our library more universally usable, a SPDK backend can be added. In conjunction with the simple BlobFS FUSE file system offered by SPDK [39], it would be possible to provide transparent kernel bypass on NVMe drives to applications, bypassing the kernel overhead of the BlobFS FUSE file system. This is only feasible if the application can safely be given exclusive access to the storage device, and if it is compatible with the limited, non-POSIX compliant BlobFS file system.

6.1.2 Submission Queue Polling Support

As mentioned in Section 2.2, in the io_uring setup it is possible to request the kernel to poll the Submission Queue. This further eliminates system calls, as the application no longer needs to actively inform the kernel of new submissions.

By spawning a userspace thread for polling in libuseruring, it is possible to replicate this behavior for the application side Submission Queue. This would enable zero-function-call kernel bypass for the application by eliminating the submission call. By also enabling kernel side polling in the io_uring instance in the kernel backend, system calls from libuseruring can also be further reduced.

6.1.3 Full Compliance with Linux Kernel Behavior

In Section 4.4, we mention some scenarios especially relating to concurrent access to the same file which currently are not handled correctly. As kernel bypass may be implemented for more io_uring opcodes, the complexity of providing behavior consistent with the guarantees of the io_uring interface increases. This is because for correct behavior, both the guarantees of the io_uring interface and the underlying related system call implementations need to be upheld.

In practice, the intersection of the guarantees of the io_uring interface and execution of the operation seems to not be well documented and researched. Further work documenting the guarantees and interactions of these interfaces is needed for correctly re-implementing them.

Bibliography

[1] io_uring_enter(2) — Linux manual page, version liburing-2.4, 2019.

[2] clock_getres(2) — Linux manual page, version 5.10, 2020.

[3] dlsym(3) — Linux manual page, version 5.10, 2020.

[4] ld.so(8) — Linux manual page, version 5.10, 2020.

[5] readv(2) — Linux manual page, version 5.10, 2020.

[6] ld(1) — Linux manual page, version binutils-2.41, 2023.

[7] AMD Corporation. Amd secure encrypted virtualization (SEV). `https://www.amd.com/en/developer/sev.html`.

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing, 2013. `https://www.intel.com/content/dam/develop/external/us/en/documents/hasp-2013-innovative-technology-for-attestation-and-sealing-413939.pdf`.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edition, November 2023.

[10] Jens Axboe. Efficient IO with io_uring, 2019. `https://kernel.dk/io_uring.pdf`.

[11] Ceph authors and contributors. Ceph documentation - BlueStore configuration reference, reef edition. `https://docs.ceph.com/en/reef/rados/configuration/bluestore-config-ref/#spdk-usage`.

[12] ClickHouse, Inc. Fast open-source OLAP DBMS - ClickHouse. `https://clickhouse.com`.

[13] Oracle Corporation. Encrypting ZFS file systems, 2013. https://docs.oracle.com/cd/E26502_01/html/E29007/gkkih.html.

[14] Glauber Costa. How io_uring and eBPF will revolutionize programming in Linux, May 2020.
https://www.scylladb.com/2020/05/05/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux/.

[15] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: A systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, page 120–127, New York, NY, USA, 2022. Association for Computing Machinery.
doi:10.1145/3534056.3534945.

[16] Roman Gershman. Helio - backend development framework in c++ using io_uring and epoll event-loop.
https://github.com/romange/helio/blob/6c18c796ed204e385887b492c619a8f44f9b418b/README.md.

[17] Troy D. Hanson and Arthur O'Dwyer. uthash: a hash table for C structures, 2022. https://troydhanson.github.io/uthash/.

[18] Shuveb Hussain. Example programs from the lord of the io_uring guide.
https://github.com/shuveb/loti-examples.

[19] Shuveb Hussain. Lord of the io_uring.
https://unixism.net/loti/.

[20] Intel Corporation. Intel® core™ i7-12700k processor.
https://www.intel.com/content/www/us/en/products/sku/134594/intel-core-i712700k-processor-25m-cache-up-to-5-00-ghz/specifications.html.

[21] Intel Corporation. Intel® optane™ persistent memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[22] Yussuf Khalil. FPGA-accelerated non-volatile memory access, October 2022.
https://os.itec.kit.edu/97_3878.php.

[23] Sungjin Kim, Byung Joon Kim, and Dong Hoon Lee. Prof-gen: Practical study on system call whitelist generation for container attack surface reduction. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 278–287, 2021. doi:10.1109/CLOUD53861.2021.00041.

[24] Atul Kwatra. Intel labs' contributions to latest Intel® Xeon® scalable processor, October 2023. https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/Intel-Labs-Contributions-to-Latest-Intel-Xeon-Scalable-Processor/post/1441731.

[25] libuv Project. Linux: introduce io_uring support. https://github.com/libuv/libuv/pull/3952.

[26] DragonflyDB Ltd. Dragonfly - the fastest in-memory data store. https://www.dragonflydb.io.

[27] Microsoft Corporation. Default cluster size for NTFS, FAT, and exFAT. https://support.microsoft.com/en-us/topic/default-cluster-size-for-ntfs-fat-and-exfat-9772e6f1-e31a-00d7-e18f-73169155af95.

[28] Microsoft Corporation. File system functionality comparison, 2023. https://learn.microsoft.com/en-us/windows/win32/fileio/filesystem-functionality-comparison.

[29] Node.js Project. Node.js: Maintaining dependencies. https://github.com/nodejs/node/blob/main/doc/contributing/maintaining/maintaining-dependencies.md.

[30] ScyllaDB Project. cassandra-stress. https://github.com/scylladb/scylla-tools-java/tree/master/tools/stress.

[31] Seastar Project. seastar::reactor_options struct reference. https://docs.seastar.io/master/structseastar_1_1reactor__options.html.

[32] N. Suneja. Scylladb optimizes database architecture to maximize hardware performance. *IEEE Software*, 36(04):96–100, July 2019. doi:10.1109/MS.2019.2909854.

[33] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. rkt-io: a direct i/o stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 490–506, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447786.3456255.

[34] Rik van Riel and Peter W. Morreale. Documentation for /proc/sys/vm/, version 2.6.29, 2008. https://docs.kernel.org/admin-guide/sysctl/vm.html.

[35] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice, 2020. https://sgaxeattack.com.

[36] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on Intel CPUs via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354, 2021. doi:10.1109/SP40001.2021.00064.

[37] Lukas Werling, Yussuf Khalil, Peter Maucher, Thorsten Gröninger, and Frank Bellosa. Analyzing and improving CPU and energy efficiency of PM file systems. In *Proccedings of the 1st Workshop on Disruptive Memory Systems*, DIMES '23. Association for Computing Machinery, October 2023. doi:10.1145/3609308.3625265.

[38] Eko D. Widianto, Agung B. Prasetijo, and Ahmad Ghufroni. On the implementation of zfs (zettabyte file system) storage system. In *2016 3rd International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, pages 408–413, 2016. doi:10.1109/ICITACEE.2016.7892481.

[39] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017. doi:10.1109/CloudCom.2017.14.

[40] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 293–300, Boston, MA, July 2023. USENIX Association. https://www.usenix.org/conference/atc23/presentation/yasukata.

Glossary

Base Address Register (BAR) Memory-mapped area provided by PCIe endpoint 9

Completion Queue (CQ) Ring buffer for receiving io_uring completions from the kernel 7, 8, 20, 22, 37

Completion Queue Entry (CQE) Item in CQ 7, 8, 20–22

io_uring Ring buffer-based kernel interface for asynchronous I/O v, 3, 4, 7–9, 11, 13–15, 17, 18, 20, 23, 25–32, 37

libc The C standard library, providing among other things functions and types for interacting with the operating system 11

Single Root I/O Virtualization (SR-IOV) PCIe feature for multiplexing a physical PCIe endpoint device 9

Storage Performance Development Kit (SPDK) User-Space storage stack presented by Intel [39] v, 3, 5, 6, 31

Submission Queue (SQ) Ring buffer for submitting io_uring requests to the kernel 7, 8, 18, 30, 32, 37

Submission Queue Entry (SQE) Item in SQ 7, 8, 13, 18–21, 30

Trusted Execution Environment (TEE) Hardware-assisted mechanism for providing secure memory regions to applications, protected from other privileged system components such as the kernel [33] 6

Virtual Function (VF) Virtual PCIe endpoint offered by a physical endpoint 9, 12, 15, 28, 29