Studiendekanat für
Elektrotechnik, Informatik und Mathematik
Operating System Group

TUHH
Technische Universität Hamburg

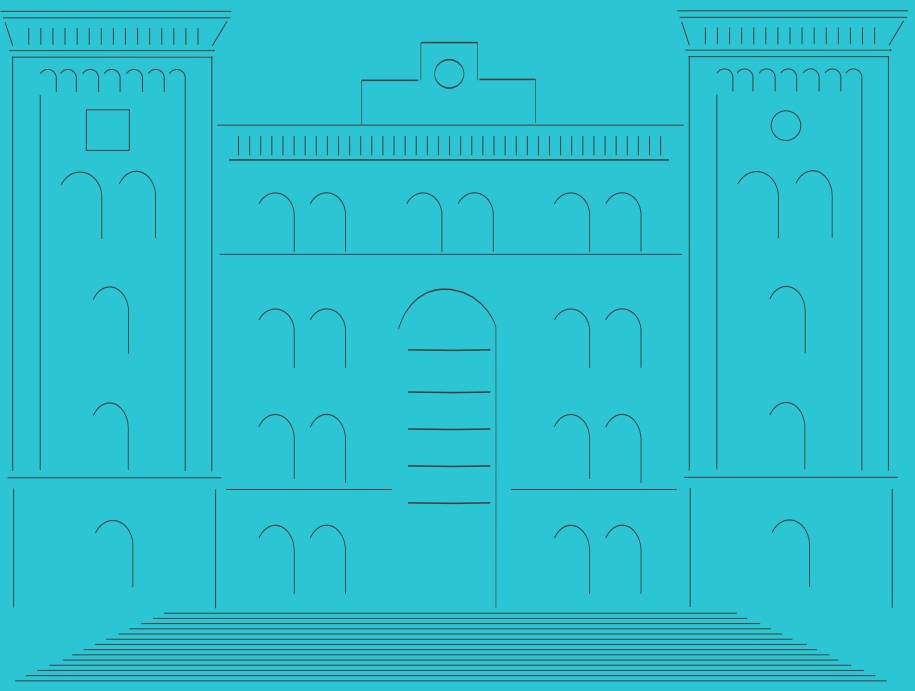
Kelvin Andres Reichenbach

System-call offloading via Linux' io_uring on the Jailhouse partitioning hypervisor

Bachelorarbeit im Fach Informatik

17. November 2021

Please cite as:
Kelvin Andres Reichenbach, "System-call offloading via Linux' io_uring on the Jailhouse partitioning hypervisor" Bachelor's Thesis, Technische Universität Hamburg, Operating System Group, November 2021.



Technische Universität Hamburg
Studiendekanat für Elektrotechnik, Informatik und Mathematik
Operating System Group
Am Schwarzenberg-Campus 3 (E) · 21073 Hamburg · Germany

System-call offloading via Linux' io_uring on the Jailhouse partitioning hypervisor

Bachelorarbeit im Fach Informatik

vorgelegt von

Kelvin Andres Reichenbach

geb. am 16. Januar 1997
in Hamburg, Deutschland

angefertigt am

Operating System Group

Studiendekanat für Elektrotechnik, Informatik und Mathematik
Technische Universität Hamburg

Erstprüfer:	**Prof. Dr.-Ing. Christian Dietrich**
Zweitprüfer:	**Prof. Dr. Heiko Falk**
Betreuer:	**Prof. Dr.-Ing. Christian Dietrich**

Beginn der Arbeit:	**15. September 2021**
Abgabe der Arbeit:	**17. November 2021**

Erklärung

Ich versichere an Eides Statt, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare in lieu of an oath that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Kelvin Andres Reichenbach)
Hamburg, 17. November 2021

ABSTRACT

In the last decade, the trend emerged to equip processors with multiple cores. This is also the case in the embedded systems domain. The *Jailhouse* hypervisor takes advantage of the presence of multiple cores to virtually partition the processor and additional hardware of a single machine. So-called *cells* are assigned to at least one CPU core and some memory to which they have unrestricted access, and that is isolated from other *cells*. Ideally, multiple guests (*inmates*) run in parallel on the same machine without maliciously interfering with each other. Jailhouse was designed for *mixed-criticality systems*, that is, systems that run multiple programs in parallel, each with different priorities. These programs should not affect each other; otherwise, they could lead to fatal consequences. However, it is possible to establish a communication channel between multiple *cells*.

The Jailhouse hypervisor is based on the Linux kernel and requires it for operation and managing other guests. With the enabling of the hypervisor, a so-called *root-cell* is active and houses the Linux kernel. Other *cells* could also house General Purpose Operating System (GPOS), but *bare-metal* applications, running on the assigned hardware, as well.

This Linux kernel comes with a relatively new interface for asynchronous Input/Output (I/O) operations: io_uring. The idea is that an *inmate* can access an io_uring instance that is located in the shared memory by an application in the *root-cell*. This way, an *inmate* can utilize the powerful Linux kernel that already exists on the same machine, and that can be used for system calls that *inmate* can not execute.

The problem is that an io_uring instance is usually allocated in the kernel space that is not accessible by other guests. Additionally, the shared memory is the only possibility to transfer data between multiple guests.

This thesis will implement an approach that allocates an io_uring instance inside the shared memory of two guests. Consequently giving an *inmate* the ability to use this instance, although each *cell* is isolated. Additionally, a library *liburing* is adapted and partly ported into the *inmate* to facilitate the usage of io_uring.

KURZFASSUNG

Im letzten Jahrzehnt kam der Trend auf, Prozessoren mit immer mehr Kernen zu bestücken, was selbst die Branche der eingebetteten System betrifft. Der *Jailhouse* Hypervisor macht sich diese Mehrzahl an Kernen zu nutze und partitioniert virtuell diese und weitere Hardware einer einzelnen Maschine. Sogenannten Zellen (*cells*) werden jeweils mindestens ein Prozessorkern und Speicher zugewiesen, auf die sie uneingeschränkt und isoliert zugreifen können. Im Idealfall laufen mehrere Gäste (*inmates*) parallel auf derselben Maschine, ohne sich gegenseitig schadhaft angreifen zu können. Jailhouse wurde für *mixed-criticality systems* konzipiert, also Systeme, auf denen Programme mit jeweils unterschiedlichen Prioritäten parallel ausgeführt werden. Diese Programme dürfen sich nicht gegenseitig beeinflussen, weil es sonst fatale Folgen haben könnte. Es kann ein gemeinsamer Speicher zwischen mehreren Zellen aufgebaut werden, der als Kommunikationskanal dient.

Der Jailhouse Hypervisor baut auf dem Linux Kernel für seine Operation und das Managen der Gäste auf. Mit der Aktivierung des Hypervisors bleibt die sogenannte *root-cell* stets aktiv und beinhaltet den Linux Kernel. Die anderen Zellen können ebenfalls ganze Betriebssysteme behausen, aber auch *bare-metal* Anwendungen, die direkt auf der zugeteilten Hardware laufen.

Dieser Linux Kernel kommt mit einer relativ neuen Schnittstelle für asynchrone Input/Output (I/O) Operationen: io_uring. Die Idee ist nun, dass ein *inmate* aus seiner Isolation heraus auf eine io_uring-Instanz im gemeinsamen Speicher zugreifen kann, die eine Anwendung in der *root-cell* dort platziert hat. So kann der *inmate* sich die Anwesenheit des mächtigen Linux Kernels auf derselben Maschine zunutze machen und Systemaufrufe an diesen übermitteln, die er selbst gar nicht ausführen kann.

Das Problem ist, dass eine io_uring-Instanz im Speicher des Kernels liegt und ein anderer Gast auf diesen nicht zugreifen kann. Zudem ist der gemeinsame Speicher die einzige Möglichkeit, um Daten zwischen mehreren Gästen austauschen zu können.

In dieser Arbeit wird eine Implementation entwickelt, die das Allokieren einer io_uring-Instanz im gemeinsamen Speicher zweier Gäste zulässt. Infolgedessen wird einem *inmate* die Möglichkeit gegeben, trotz Isolation, auf diese Instanz zuzugreifen und diese zu benutzen. Zusätzlich wird die Bibliothek *liburing* in Teilen in den *inmate* portiert, um die Handhabung des io_urings zu erleichtern.

CONTENTS

1 INTRODUCTION

Although CPUs are becoming more powerful each year, handling more tasks simultaneously due to an increase in core and thread count, it is still essential to keep the CPU utilization as high as possible, indicating that the processor is used most efficiently. To achieve this, we should avoid the state of idling, therefore, mitigating running out of scheduling tasks for the processor. Input/Output (I/O) operations can be done in a synchronous or "blocking" manner, i.e., a process will request an I/O operation through a system call to the kernel and immediately gets blocked until the I/O operation is successfully completed or fails. After the system call is returned, the process will be scheduled again to continue executing subsequent instructions. The problem arises if there are other instructions after the I/O operation that could have been computed while waiting for the system call to return because their execution is independent of the I/O operation. This is where asynchronous I/O (AIO) operations come to play, as processes can request an asynchronous system call to, for example, read from a file and shortly after continuing the execution of code once the I/O operation is queued and handled in the background. The application could actively wait for the result by polling the completion status; however, then this would not make a difference to the aforementioned synchronous I/O operation. Besides that, the kernel can also signal the application when the I/O operation returned successfully or not. During that time, the application can compute other instructions that are independent of this file read or even request more asynchronous I/O operations, therefore keeping the CPU busy by executing code that would have been needed to be run later some time nevertheless.

In 2002, the work on an AIO interface for Linux ("*aio*") began. Still, it never gained serious attraction, as the API was complicated to use, and *actual* asynchronous I/O was restricted to prerequisites. Even though it should be asynchronous, there were cases in which AIO could still end up blocking [Jon]. After several attempts to patch the *aio* interface, a new API coined *io_uring* was built from the ground up, solving all problems that users had with *aio* and was merged into the Linux kernel 5.1 in 2019. It is based on two separate ring buffers for submitting I/O operations and retrieving the return value, respectively, that are shared between the kernel space and user space. I will go into more detail throughout this thesis because *io_uring* is one of the two main implementations that I will build on and needs to be modified.

On the other hand, there is Siemens' *Jailhouse*, a *Linux-based partitioning hypervisor* [1]. The intended use of this particular hypervisor is simplicity and in the embedded system domain, e.g., cars and airplanes, rather than being a full-fledged hypervisor like *KVM* virtualizing multiple General Purpose Operating Systems (GPOSs) simultaneously. Jailhouse partitions its guests into dedicated CPU cores and memory regions and handles malicious accesses between them, which otherwise would result in an exit of the misbehaving guest. The hypervisor is said to be *Linux-based* because it needs an already booted Linux for enabling, but afterward, Linux will run as a guest itself and is

[1] https://github.com/siemensss/jailhouse

used to manage the creation and destruction of other guests or disable the hypervisor in general. A real-world example is a multi-copter running an autopilot software system, handling critical and real-time flight control, while the Linux partition works on uncritical tasks, for example, camera tracking [Ram+17]. This is called a *mixed-criticality system*, made up of different applications running on the same hardware but with varying levels of criticality. Going back to the example, losing the ability of camera tracking is uncritical, whereas a failure of the flight control system can be catastrophic.

It could be assumed that we can take advantage of the Linux guest for operations and drivers that otherwise, we would have to (re-)write on our own and bring into our guest. Suppose there is already a full-blown and versatile OS in the system. Why not just utilize it instead of running another Linux in parallel, potentially wasting hardware resources and losing real-time capabilities. The problem is that Jailhouse is designed to partition guests strictly in a way that they cannot interact with each other, except for a rudimentary guest-to-guest communication through shared memory and interrupts.

This thesis will exploit this rudimentary communication and develop an approach to establish a communication channel between the Linux guest and an isolated bare-metal application, such that the latter can leverage the presence of the Linux guest through issuing system calls and getting the returned values back. Chapter 2 explains the Jailhouse hypervisor and *io_uring* interface in more detail. Chapter 3 describes the modifications that have to be made inside the kernel and the preparation of the guest application to use this interface. Chapter 4 evaluates the architecture concerning measurements of speed and the introduced overhead through the guest-to-guest communication. Finally, in Chapter 5 a conclusion is drawn from the results.

2 FUNDAMENTALS

This chapter will explain the two main components that the thesis builds on. First, a hypervisor is introduced that strictly isolates its guests, including a Linux guest that is an inherent part of the hypervisor. Especially the Inter-VM Shared Memory (IVSHMEM) is elaborated furthermore because it provides the guest-to-guest communication that this thesis uses. Following that, the *io_uring* interface is explained as it is a part of Linux and necessary for asynchronous I/O.

2.1 Jailhouse

The *Jailhouse* hypervisor, which was initially developed and still is being maintained by Jan Kiszka at Siemens AG and was presented to the public in 2013[2]. As already summarized in the introduction, this hypervisor is focused on simplicity and security on embedded systems, although it can also be used elsewhere. We could assume it is reasonable and necessary to separate critical and uncritical applications into different hardware components independent of each other. For example, the airbags system in a car should not run on the same processor that is in charge of the entertainment system because an error in the latter one could affect the airbags and potentially cause the death of the occupants. However, having dedicated hardware for each system is expensive, needs more physical space, and communication between devices gets more complex, as each group of peers has to be interconnected. Think of a car radio with its own processor, and then two other processors handling button presses just on the steering wheel and center console, respectively – that are three units that could be merged into one. Following this idea, there is an architectural trend to do precisely this because the hardware is more powerful than ever before, and CPUs are equipped with multiple cores [Bro06]. Jailhouse is a hypervisor that virtually partitions the hardware through software into independent guests that will not interfere with each other. Therefore, continuing the example of the car radio, it is possible to virtually split up a single multi-core CPU and run different applications, each having its dedicated core(s) and memory regions, while still being unaffected if another guest fails. For example, with Jailhouse, we could partition a quad-core CPU into four independent parts: one core for the radio and entertainment system, one core for real-time critical button presses on the steering wheel, e.g., the horn, turn signals, and windshield wipers, one core for uncritical buttons on the steering wheel controlling the volume of the radio and the fourth core for buttons in the center console. In this scenario, we have multiple car components consolidated into a single system to save costs while ensuring no involuntary interference between the guests.

[2] https://github.com/siemens/jailhouse/commit/c690fb976081ac4b1f7f57fc2b64a757f963723b

2.1.1 Setup

Jailhouse leverages an adapted Linux codebase, as it already comes with a lot of features and drivers, is widespread, and may be already used or needed in a system nevertheless. The Figure 2.1 gives an overview of the construction and structure of a Jailhouse that will be explained furthermore. Linux will boot as usual, and upon activation of the hypervisor, which can happen directly at boot or at some point later in time, a kernel module will be loaded and executed, placing Jailhouse between the hardware and Linux itself, therefore, running the OS as a guest now. From now on, Jailhouse is managing and assigning the hardware and controls resource accesses. This makes Jailhouse a hybrid between a Type-1 hypervisor, meaning that it runs bare-metal with guest OS's on top of it, and a Type-2 hypervisor, because it needs to run on an OS (Linux) to initialize itself in the first place [Ram+17]. Even though Linux is not running *directly* on the hardware anymore, it will be assigned all hardware resources and is still used for managing other guests and configuration of Jailhouse itself, like creating new guests or disabling the hypervisor. In the context of Jailhouse, the term "guest" consolidates two things: When starting a new *guest*, the hypervisor will load a so-called *cell configuration* that we will take a closer look at in the following subsection. This cell configuration meticulously states how many CPU cores and which memory regions to use, among other resources, that will be split off from the Linux guest and build a *cell*. After this *cell* owns all the needed resources, the actual inner workings, i.e., the application, will be executed and is named *inmate*, finalizing our understanding of a *guest* in the Jailhouse hypervisor. Applying these definitions, we can divide the Linux *guest* into a cell that only in this case is called the *root cell* and an *inmate*, that is, Linux itself. While non-root cells are arbitrarily created and destroyed, the root cell lives as long as Jailhouse is enabled. Inmates can be bare-metal applications or even other OSs like yet another Linux instance or an OS optimized for real-time use cases, e.g., FreeRTOS, although these have to be adapted to Jailhouse in the first place and do not run straight out-of-the-box[3].

As a *partitioning* hypervisor, Jailhouse focuses on isolating actual hardware rather than virtualization and overcommitting resources. Therefore, every cell has to occupy at least one physical CPU core that can not be shared with other cells, and the count of cores determines the maximum count of cells.

2.1.2 Cell configuration

Each cell, later containing an inmate, has to be precisely described beforehand, and its configuration cannot be changed at runtime. As of now, Jailhouse uses a single `.c`-file populating only one

[3] https://github.com/siemens/jailhouse/blob/master/Documentation/non-root-linux.txt

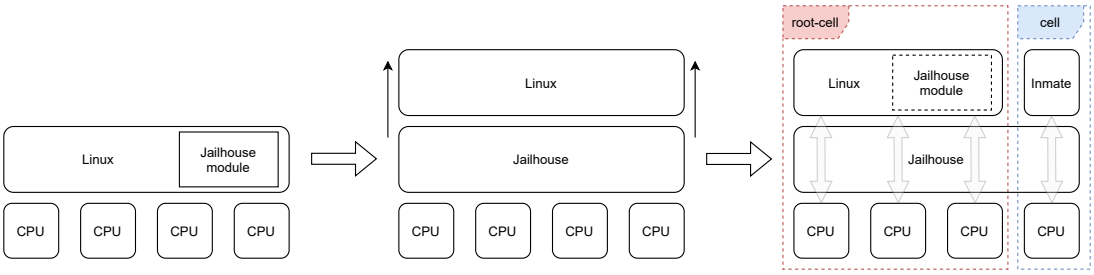


Figure 2.1 – Initialization of the Jailhouse hypervisor and running an inmate on a partitioned CPU

`struct` defining the name of the cell, how many CPU cores should be assigned, and which memory regions to use that will be compiled into a binary. It is not very human-friendly to write such a cell configuration [Val15], but Jailhouse comes with a tool that makes the generation of this file for the x86 architecture a bit easier. For the ARM architecture is has to be written because a generator for it does not exist yet[4]. However, developers still need profound knowledge of the underlying hardware. For example, it has to be specified which physical addresses for different memory regions are wanted to be used in an inmate. Additionally, the size of the regions, and flags determining how each one is used, e.g., if an inmate can only read from a region or also write data into it. The cell configuration also depicts which PCI devices should be accessible from within a cell, like Ethernet, audio, and other peripherals. Jailhouse comes with a tool (`jailhouse config check`) that checks cell configurations for possible errors, e.g., overlapping memory regions[5].

2.1.3 IVSHMEM

Despite being focused on isolation and encapsulating all inmates, it could be desirable to let inmates communicate with each other by sharing data and event signaling through interrupts. Jailhouse introduces an Inter-VM Shared Memory (IVSHMEM), which is essentially a shared memory between participating peers that allows a rudimentary form of guest-to-guest communication between inmates but still with safety in mind. IVSHMEM is implemented as virtual PCI devices that have to be defined in each cell configuration of all participating cells as well as memory regions that will be exclusively used for this inter-cell communication.

The memory regions have to be contiguous because only the base address of the shared memory has to be given in each cell configuration. As shown in Figure 2.2, IVSHMEM consists of the following segments, starting from the shared memory base address:

[4] https://github.com/siemens/jailhouse/blob/aee017cd0c3abfdc94951037435078cf57fbcb53/README.md#configuration
[5] https://github.com/siemens/jailhouse/blob/master/tools/jailhouse-config-check

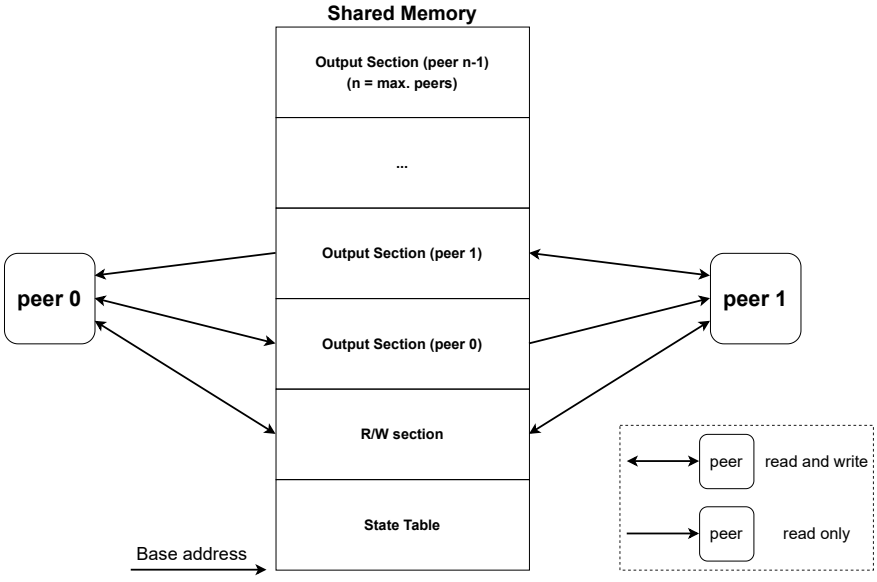


Figure 2.2 – Structure of the shared memory region that is required for IVSHMEM (Inter-VM Shared Memory)

- State Table: Holds state values of all peers and is read-only for them. Peers can write values to their Local State register representing their current state that is propagated into the State Table. If a peer disconnects or resets, their entry in the State Table will be set to zero. Peers inform each other about their current state through the State Table.

- Read/Write section: As the name suggests, this section can be used for reading and writing values from/into the memory. All connected peers have permission to do so. There is no memory protection such that peers can overwrite the values of other peers.

- Output sections: Each peer gets its own section on which it is the only being able to write values into it, while other peers only have the permission to read from it.

For example, while peer 0 in Figure 2.2 can read from **and** write to the *read/write section* and *output section (peer 0)*, it can only read from *output section (peer 1)*. Jailhouse moderates memory accesses and will immediately stop an inmate that tries to write into a section it has no permission to do so. Figure 2.2 also shows the minimal configuration of IVSHMEM with just two peers, therefore needing four shared memory regions. However, it is possible to connect up to 65536 peers – provided that each peer gets its own *output section*.

IVSHMEM also supports event signaling via interrupts. In the *Configuration Space Header* of the virtual PCI device for IVSHMEM, the location in memory of the *Register Region* is specified in *Base Address Register (BAR) 0* [6]. Said region holds the following registers that will be needed for communication through interrupts:

- ID: Is the ID of the device/peer that was specified in the cell configuration with `.shmem_dev_id` and is unique among all peers. It is read-only and can not be changed at runtime.

- Maximum Peers: Gives the maximum number of possible peers and is configured in `.shmem_peers`. Read-only as well and stays the same at runtime. As already hinted above can range from 2 to 65536.

- Interrupt Control: As of now, this register only uses the first bit to indicate whether interrupts should be generated on a State change or if the peer writes into the Doorbell register.

- Doorbell: 32-Bits long register in which the first 16 bits correspond to the *interrupt vector* to be triggered and the last 16 Bits specify the ID of the target device.

- State: This is the previously mentioned *Local State Register* that the owning peer can read from and write to. If *Interrupt Control* is set to 1 the value in *State* will be copied into the *State Table*, making it readable for all other connected peers.

Jailhouse ships with a demo user space application and inmate that showcase a minimal example of how IVSHMEM is set up and used. It should be noted that with Jailhouse version `0.12` the *IVSHMEM Device Specification* is still work-in-progress and not stable[7]. This thesis uses IVSHMEM and its ability of guest-to-guest communication to allocate io_uring inside the shared memory, such that the Linux inmate and a bare-metal application can interact with the same io_uring.

[6] https://github.com/siemens/jailhouse/blob/master/Documentation/ivshmem-v2-specification.md
[7] https://github.com/siemens/jailhouse/blob/master/Documentation/ivshmem-v2-specification.md

2.1.4 Related work

Despite Jailhouse's pioneering approach to *hijack* a General Purpose Operating System (GPOS) like Linux and using its already existing features for managing the partitioning hypervisor itself and other cells, resulting in a relatively small codebase, this still comes with downsides. From a security and real-time perspective, you have to verify and prove that your code is correct and does not yield any errors at some point later in time. This verification gets easier with fewer source lines of code (SLOC) that have to be checked. With almost 30k SLOC in 2017, made up of 3.4k SLOC for the hypervisor core and roughly 5.4k to 7.4k SLOC for different architectures [Ram+17], Jailhouse seems to be manageable in terms of verifying the codebase. However, this does not include the Linux kernel that is mandatory for Jailhouse to work and that in version 5.10.31 consists of a staggering 20M SLOC[8]. Of course, you don not have to include every module and driver that is available for Linux. However, in the end, it is still a large codebase that has to be verified to use Jailhouse as a partitioning hypervisor in a mixed-criticality system.

From this problem emerged a new static partitioning hypervisor called *Bao*[9] that Jailhouse inspired [Mar+20]. It comes with the same features, i.e., static assignment of hardware resources to specific guests, guest-to-guest communication through IVSHMEM, and cell configurations. However, it does not rely on Linux for booting and initialization [SMP21]. Besides that, it can still run Linux and other GPOSs as guests; the only dependency is an underlying firmware for hardware initialization, which significantly improves boot-time for critical applications and reduces the trusted computing base (TCB). This makes Bao a minimal hypervisor that, as of now, is targeted for embedded systems only and supports the ARMv8 architecture and soon RISC-V as well, while Jailhouse can additionally run on x86.

A more sophisticated hypervisor, *Xen* [Bar+03], recently went the same route and implemented a method to partition hardware without a GPOS managing guests. Usually, Xen boots the *dom0* guest, which can be compared with Jailhouse's *root cell*, which typically runs Linux and manages the creation and destruction of *non-root cells*, called *domU*. In a *dom0-less* system configuration, Xen will still boot *dom0* but in parallel *domU* guests as well that are independent of *dom0* [Ste19]. Taking it one step further, Xen can only boot *domU* guests, which is called *true dom0-less* system configuration. (True) dom0-less systems are configured via a *Device Tree* that, similarly to Jailhouse, defines properties for each guest, e.g., how many CPU cores and memory space is assigned [Ste19]. It is possible to allocate memory to each guest statically by specifying the memory start address and size[10].

2.2 io_uring

In the following section, the other ingredient of this thesis is further explained. The io_uring interface will be used to handle system calls between the user space application and inmate. After a short introduction of io_uring, the differences between synchronous and asynchronous I/O are elaborated. I am continuing with a deeper explanation of the inner workings of io_uring and how to use it.

At the time of writing, this *io_uring* is a relatively new asynchronous I/O interface merged into the Linux Kernel 5.1 in 2019. It was created and is mainly maintained by Jens Axboe, who was unsatisfied with the already present *Linux Native AIO* interface, its performance in respect to modern hardware capable of very low latencies and limited use cases.

[8] Counted with `sloccount` (https://dwheeler.com/sloccount/). Kernel version 5.10.13 is the latest adapted Linux kernel that Jailhouse supports (https://github.com/siemens/linux/commit/eb6927f7eea77f823b96c0c22ad9d4a2d7ffdfce)

[9] http://www.bao-project.org/

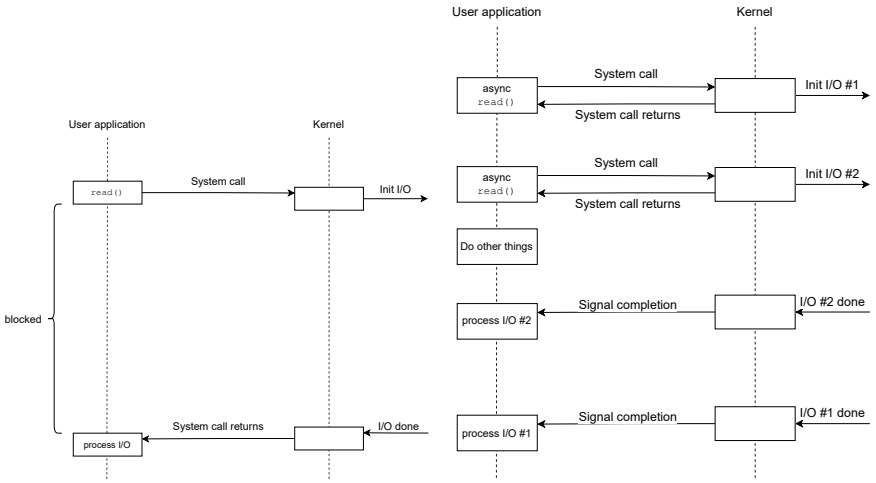
[10] https://xenbits.xen.org/docs/unstable/misc/arm/device-tree/booting.txt

2.2.1 Difference between synchronous and asynchronous I/O

Before talking about *asynchronous* I/O, we have to know the difference to *synchronous* I/O. As an example, we have an application that wants to read from a file. Programs in the user space are not privileged enough to *directly* perform these kinds of operations and therefore have to ask the kernel to do it for them through a *system call*. A system call, specifying an I/O operation, is sent to the kernel and the kernel will initiate the according operation. The application will halt until the system call returned. Now, there are two different mechanisms of system calls and their corresponding I/O operations:

- Synchronous I/O: After the application sent a system call to the kernel, its execution is blocked. The kernel initiates the I/O operation and waits for its completion. After the operation completed, either successfully or not, the kernel sends a signal back to the application that was still in a halting state during that time and did not continue executing more instructions. When the response of the kernel is received, the execution continues and the application is notified about the return value of the operation, compare Figure 2.3a. This is practical if we only use a few system calls at different places in our code and immediately work on the file that was read after the application continued running. However, suppose we have multiple system calls shortly after each other. In that case, it will decrease the efficiency of the application and hardware because the CPU has to change between *user mode* and *kernel mode* every time a system call is used. Additionally, the application can issue only one system call at a time because it has to wait for the completion of the I/O operation. If the CPU is capable of executing multiple threads, it will also waste precious CPU time that could have been used for parts in the application that may not depend on the reading of the file, and that could have been executed while waiting for completion of the I/O operation.

- Asynchronous I/O: The problems that come with synchronous I/O will be fixed with an asynchronous behavior of the I/O operations. Instead of waiting until the file is read and can be used in the application, we can issue that *read* operation as an asynchronous I/O operation through a different system call. With asynchronous system calls, the application still waits for the system call to return, but with asynchronous I/O it will return immediately after the I/O operation was issued successfully by the kernel, as shown in Figure 2.3b. After the response of the kernel the application continues with its execution of further instructions that do not depend on the I/O operation or even issue new I/O operations. The I/O operation signals the kernel when it is done and the kernel will notify the application that the operation was successfully or not. Now, the application can process the I/O data. This significantly improves hardware utilization, shortens the execution time of the application, and even allows for multiple system calls in parallel. However, asynchronous I/O comes with the caveat that the programmer has to be prepared for its asynchronous nature and that system calls can return in a different order than they were issued, see Figure 2.3b. We also have to keep in mind that one system call may depend on another one and that it should wait until it is done, otherwise resulting in a wrong and unexpected behavior.



(a) synchronous I/O **(b)** asynchronous I/O

Figure 2.3 – Subfigures visualize the execution of synchronous and asynchronous I/O

2.2.2 Inner workings of io_uring

Fundamentally, io_uring consists of two separate ring buffers that will be used for a submission queue (SQ) and completion queue (CQ), respectively. *SQ* is filled by the application and holds the I/O operations that should be executed. These operations are stored as submission queue entrys (SQEs) inside the *SQ*, while the *CQ* is filled by the kernel and contains completion queue entrys (CQEs) that each represent the return values of the *SQEs*. A high-level overview is presented in Figure 2.4 and compared with the following real-world example using io_uring's terminology.

1. *Worker A* has a car part that needs processing in form of spraying it with paint that he is not allowed to do by himself, making him the equivalent of a user space application with restricted

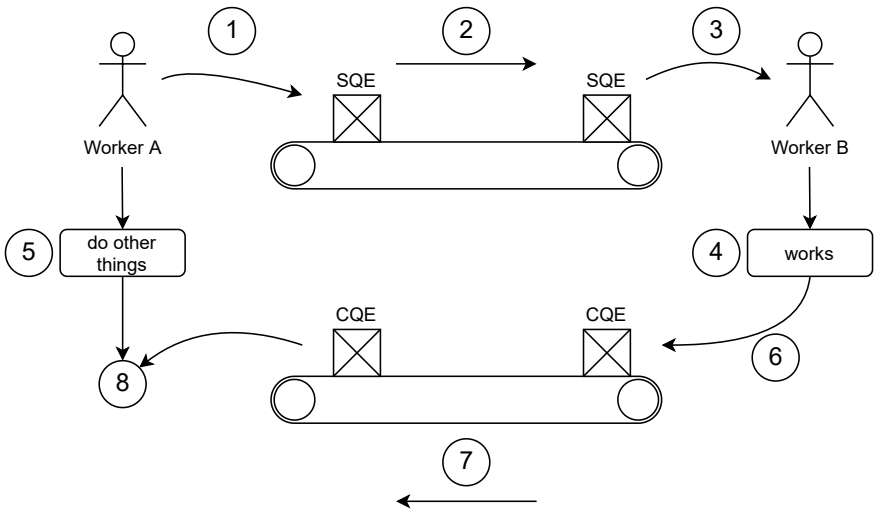


Figure 2.4 – The mechanism of io_uring as a real-world example

permissions. Therefore, *Worker A* puts this part inside a box (SQE) and places it on a conveyor belt.

2. The conveyor belt is the SQ and can only hold a certain amount of boxes at once. If the boxes are not taken away at the end of the belt they pile up and *Worker A* cannot put any more boxes onto the belt.

3. *Worker B* is on the other side of the belt (SQ) and privileged enough to paint car parts, what can be compared to him being the kernel. He sees the unprocessed box (SQE) on the belt and takes it with him.

4. *Worker B* (kernel) opens the box and spray-paints the car part.

5. Instead of waiting for that one car part to be painted, *Worker A* can do other work that does not depend on this particular part that is either still sitting on the belt (SQ) or is currently processed by *Worker B*.

6. When *Worker B* is done he puts the painted part inside a new box (CQE) and onto another conveyor belt (CQ).

7. Similar to the conveyor belt before (SQ), boxes (CQEs) can pile up if the worker on the other side does not take them off the belt.

8. *Worker A* takes the box off the belt, opens it, and can now use the painted car part for further manufacturing.

Both *rings* (in the example, conveyor belts) are shared between the kernel space and user space, therefore reducing unnecessary copies between the spaces because both parties can access the same shared *rings*.

After a high-level example of io_uring's mechanism and the introduction of its components, we will take a deeper look at io_uring itself. During setup, the kernel will allocate the io_uring in its own kernel space that is not accessible by the user space and afterward returns a file descriptor. This file descriptor uniquely identifies this instance of the io_uring that was just set up. Now, to make this instance accessible for the user space application as well, the application will take this file descriptor and uses the system call `mmap(2)` to create new mappings of the SQ, CQ and an array for SQEs in its own virtual address space. The io_uring is now shared between the kernel space and user space and both can access it.

Figure 2.5 visualizes the rings in the memory and we will continue to explain the workflow inside of io_uring. io_uring comes with a low-level interface that requires the developer to do most of the steps we are going to see now. The rings are already set up, mapped into the user space and io_uring is ready to receive SQEs. The size of the rings, i.e., how many `ENTRIES` each can hold, is specified during initialization and should be a power of two in the range from 1 to 4096.

1. Each ring has two pointers (*head* and *tail*) pointing to single `ENTRIES`. If both point onto the same entry, the ring is empty and if *tail* is just one entry behind *head* the ring is full cannot take any more SQEs. Initially, the ring is empty and both point onto the same entry.

2. The user space application wants to insert SQE #1 into the SQ. It checks the position of *tail* and if the next entry is empty, the user places SQE #1 into the entry *tail* is pointing at and afterward sets *tail* to the next position clock-wise. The user continues placing SQE #2 and #3 into the SQ, moving *tail* every time a step further away from *head*, which does not move for now.

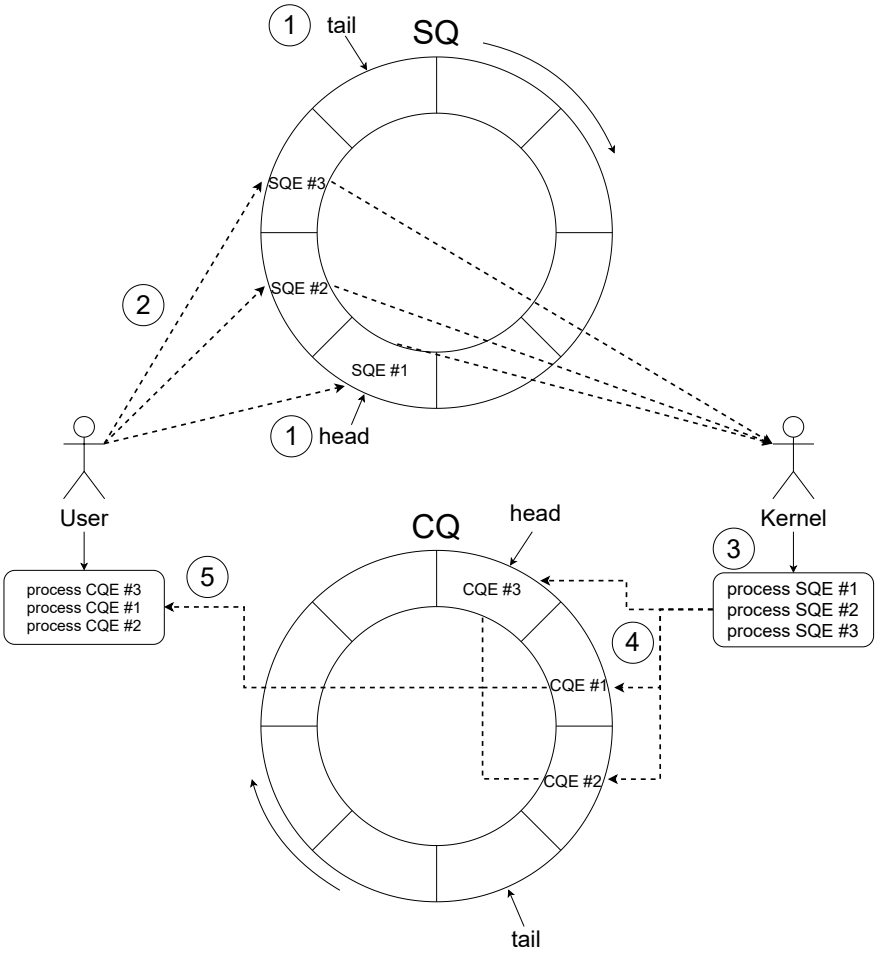


Figure 2.5 – io_urings concept of two separate ring buffers. SQ is the Submission Queue and CQ is the Completion Queue

3. The application signals the kernel to *enter* the io_uring, telling the kernel that unprocessed SQEs are queued up. The kernel reads from the SQ and initiates asynchronous I/O operations that are stated in the SQEs. These operations can be, for example, *reads/writes* on files or *sendmsg/recvmsg* on sockets. After processing a SQE, the kernel increments the position of *head* inside the SQ. This entry is now free to use for another SQE in the future.

4. After the completion of I/O operation, the kernel will do the same procedure as the application before, with the only difference that the kernel puts the return values of each operation into CQEs that are placed inside the CQ. *tail* is moved accordingly. Note, that CQEs do not have to be in the same order as they were as SQEs. In this case, completion of SQE #3 was the fastest and therefore placed first inside the CQ.

5. Now, the application looks at the position of *head* and *tail* of the CQ and if they are not equal it implies that the ring is not empty and therefore CQEs are ready to be read. After reading a CQE the application increments *head* by one, making this entry as *seen* and available for a new CQE again.

As already said, the low-level interface requires the developer to do most these steps manually, however, they are boilerplate code and repeated each iteration. Therefore, to reduce the lines of code and not having to deal with the raw io_uring structures, there is also a much easier-to-use wrapper for this interface that is called *liburing*[11]. *liburing* boils down io_uring to just a few functions that will hide the low-level code needed for asynchronous I/O through this interface. The creator of io_uring, Axboe, recommends using *liburing* instead of the raw interface [Jen]; therefore, we will continue to use this library and its functions in this thesis. In Chapter 3, *liburing* will be adapted and brought into the inmate giving it the ability to use io_uring inside an isolated cell. We will continue with a small example on how else io_uring can be used with *liburing*.

2.2.3 How to use io_uring

Now, the goal of Listing 2.1 it to show how convenient it is to use *liburing* instead of the low-level io_uring interface. Setting up an io_uring instance using *liburing* begins with the declaration of a variable that will hold the ring, see line 1, and will be used to identify the instance uniquely. Followed by line 2 to finally initialize it with the number of `ENTRIES` the SQ can hold, the address of our ring variable and optional flags, that will be none in this case. An example for a flag is `IORING_SETUP_IOPOLL` that tells io_uring to perform busy-waiting for an I/O completion, meaning it will constantly poll the state of the operation instead of waiting for an asynchronous Interrupt request (IRQ), reducing latencies but also consuming more CPU resources. Similarly, there is line 3 with the difference that it takes a pointer to an io_uring-specific structure `io_uring_params` that includes flags as well as other parameters that can be changed. Invoking the initialization function automatically creates all necessary structures and mappings.

For submission, we need some kind of envelope that will hold our submission queue entry (SQE) and will be executed by the kernel. This is done by initializing a pointer to such a structure, and a function that will fill it with an empty SQE gathered from the io_uring, see line 5, where `&ring` is our io_uring instance from before. If the submission queue is full, meaning all `ENTRIES` are already taken, `NULL` will be returned.

Next, there are several submission helpers, each representing a single type of I/O operation supported by io_uring, e.g., reading/writing from a file descriptor or sending/receiving from a socket, that will be used to prepare the yet empty but already reserved SQEs. For simplicity, we will prepare a no-op system call for this specific SQE in line 6 and additionally set some user data in line 7 to distinguish this SQE from other ones. `struct io_uring_sqe` has a field `user_data` that is copied over from the SQE to the CQE later on and stays unchanged. At this point, we can repeat the process to request `ENTRIES`-1 more SQEs for system calls that should be executed asynchronously.

Now comes the part that will tell io_uring to *submit* all open SQEs to the kernel for consumption. This is achieved in line 9. As this interface is asynchronous, we may continue with the execution of our application until hitting a point where we need the return value of the system call. To get this return value, first, we have to declare a CQE, the counterpart to an SQE, with line 11 that will hold the value. Then we have to wait for the completion in line 12 that will block until any CQE is ready and afterward point our pointer to it. Suppose we have set any `user_data` in the SQE. In that case, it will be available through line 12 and can be used to distinguish between different SQEs that may have been submitted before line 9 because a CQE does not contain any information about the I/O operation it holds the return value of. After we have processed the CQE and therefore do not need it anymore, we have to call the function in line 14 such that this CQE in the CQ is marked as read and can be used for another CQE in the future. This prevents the kernel from overwriting a CQE in the

[11] https://github.com/axboe/liburing

ring buffer that has not been processed by the application yet, and it will move the CQ ring head to the next *unseen* CQE; otherwise, line 12 will always return the same CQE.

When we don not need the io_uring instance anymore, we simply tear it down in the last line 16, which will unmap all shared ring buffers.

2.2.4 Summary

To summarize this section, we have seen the Jailhouse hypervisor that virtually partitions the underlying hardware and assigns it to so-called *cells* individually. Cells are populated by *inmates* that can be General Purpose Operating Systems (GPOSs) or bare-metal applications. This thesis will use Jailhouse in Chapter 3 for providing a statically partitioned system with a Linux *root-cell* and a bare-metal application. Furthermore, we have looked at the inner workings of the asynchronous io_uring interface, the difference to synchronous I/O, and how the interface is used with the library *liburing*, that simplifies working with io_uring.

```
1  struct io_uring ring;
2  io_uring_queue_init(ENTRIES, &ring, 0);
3  // Optionally: io_uring_queue_init_params();
4
5  struct io_uring_sqe *sqe = io_uring_get_sqe(&ring);
6  io_uring_prep_nop(sqe);
7  io_uring_sqe_set_data(sqe, (void *)0x1000);
8
9  io_uring_submit(&ring);
10
11 struct io_uring_cqe *cqe;
12 io_uring_wait_cqe(&ring, &cqe);
13
14 io_uring_cqe_seen(&ring, cqe);
15
16 io_uring_queue_exit(&ring);
```

Listing 2.1 – Sample application using liburing

3 ARCHITECTURE

First, we will outline the goal, the problem, and the proposed communication flow between the Linux root-cell and the inmate. There is a user space application in the Linux guest, and there is an inmate in an isolated cell that wants to do asynchronous I/O operations. The inmate is a simple bare-metal application; therefore, its code base is relatively small and does not come with the feature-richness of a General Purpose Operating System (GPOS). The goal is that the inmate can use *liburing* to access io_uring of Linux instead, which resides in the root-cell; therefore, *offloading* system calls as the title of this thesis suggests. The problem is that Jailhouse was designed to keep its guests isolated and only allows minimal communication through shared memory and interrupts. We will begin with an explanation of the communication flow.

3.1 Communication flow

The user space application acts as a man-in-the-middle (MITM) between the Linux kernel and the inmate, as shown in Figure 3.1. The application has to initialize the io_uring inside the Inter-VM Shared Memory (IVSHMEM) such that the inmate can access the memory location of the io_uring. From now on, the application waits for incoming Interrupt requests (IRQs) from the inmate. In this thesis, the inmate should assume that the io_uring is already initialized before its execution. Therefore, the inmate does not have to wait and can immediately access the rings inside the shared memory. The inmate can now prepare SQEs, as shown before in Listing 2.1 (lines 5-7). When all SQEs are prepared, the inmate sends an IRQ to the application in the root-cell, including the count of SQEs in the SQ. The application, already awaiting the IRQ, then invokes the system call `io_uring_enter(2)` that is responsible for getting io_uring to complete all pending SQEs. The inmate waits for a CQE to be ready, processes it, and marks it as *seen*, how it is shown in Listing 2.1 (lines 11-14). As suggested in Figure 3.1, if the inmate issued more than one SQE, the inmate waits for the next CQE and repeats the process; otherwise, it goes back to preparing new SQEs or continues with the program.

In summary, there is only one IRQ from the inmate to the application required, and the latter is just forwarding the inmate's request through a system call to the kernel. Preparing SQEs and reading CQEs does not need any communication between the inmate and application because of io_uring's technique of shared ring buffers that both parties have access to.

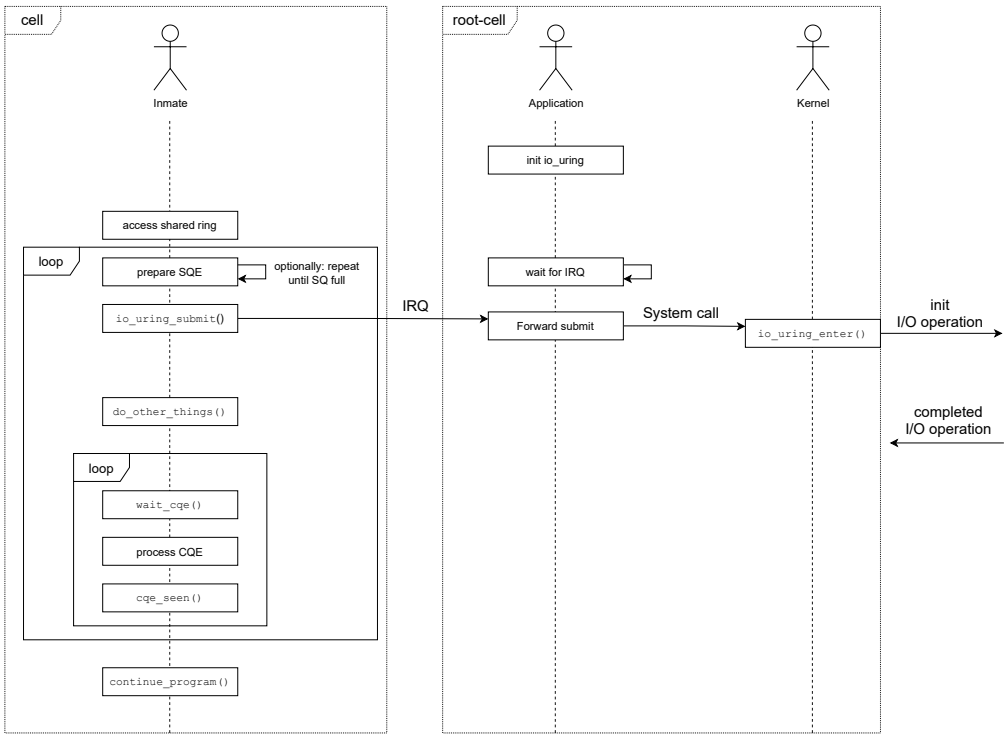


Figure 3.1 – The communication flow between inmate, application, and kernel

3.2 Approach

After an explanation of the communication flow, a general approach is presented. The thesis will go into more detail about the implementation in Section 3.3.

As explained in Section 3.1, the user space application begins with the initialization of the io_uring in the shared memory because otherwise, the inmate would not have access to the rings in the kernel space. Therefore, setting up an io_uring, a custom flag must be specified to tell the kernel to locate io_uring with its context and SQ- and CQ-rings at a user-defined memory address. This custom flag is named `IORING_SETUP_PARAMS_MEMORY` and is set in the structure of the parameters. Additionally, we include parameters that let the application point to the desired memory location where the instance should be located. Usually, io_uring would use `mmap(2)` to map the rings from the kernel space into the user space individually and get pointers to each of them. The application holds a structure of the io_uring and pointers to its members that will be filled from the returned pointer of `mmap(2)` with the corresponding offsets in the parameter structure. These offsets are provided by io_uring during the setup. In our case, we do not want a mapping because Jailhouse already mapped the IVSHMEM into the inmate that will be used. Locating an io_uring outside of the kernel space at a user-defined memory address is elaborated in Section 3.3.1. This raises the problem that io_uring is not in charge anymore of allocating the instance and does not fill in the necessary offsets that are crucial for the application to access the instance. To overcome this problem, we need an *anchor* inside of the io_uring structure that the inmate will use in Section 3.3.2 to calculate the

offset between its mapped shared memory and the actual memory address the application specified. Each pointer of io_uring has to be *redirected* such that the inmate can access the interface.

Finally, after placing the io_uring at a memory address in the shared memory and moving the pointer to the correct location, *liburing* needs to be modified to send IRQs from the inmate to the application instead of a system call that is not available inside the inmate. From now on, the inmate can use the *liburing* interface to issue asynchronous I/O operations. This concludes the general approach that will be further explained in Section 3.3.

3.3 Implementation

After a rough overview and familiarizing ourselves with the implementation steps that have to be made, we begin with a more detailed view on locating an io_uring at a custom memory address. Then we will examine how the pointers work and should be relocated and subsequently explain how *liburing* is migrated into the inmate.

3.3.1 Locating io_uring outside of the kernel space

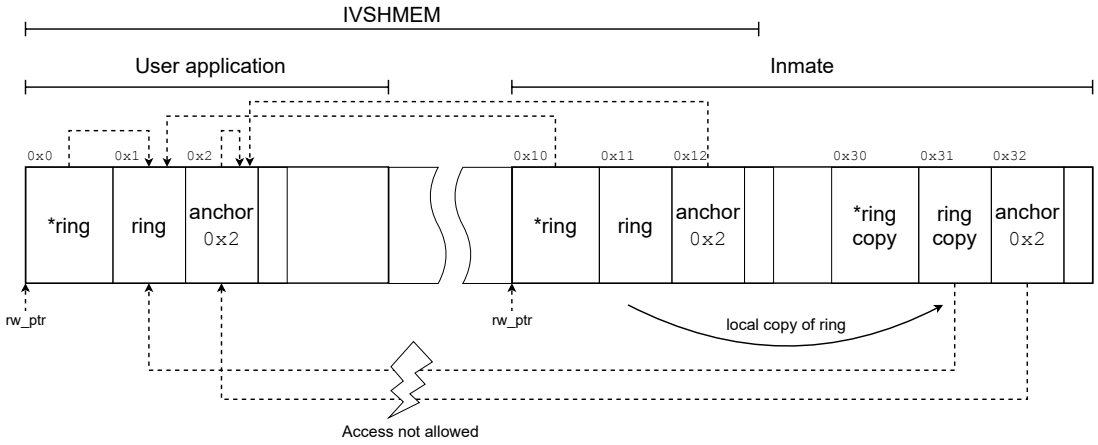
As briefly explained in Section 3.2 we do not want the kernel to allocate the io_uring structure and rings inside the kernel space; otherwise, the inmate would not be able to access it. During initialization of the Inter-VM Shared Memory (IVSHMEM), the application maps the memory regions from Figure 2.2 into its own memory space, from which the *read/write* section will be utilized to place the io_uring inside. The *ring* pointer is allocated at the beginning of the *r/w* section and the custom flag `IORING_SETUP_PARAMS_MEMORY`, including two pages of the *r/w* section for each ring (SQ and CQ), are set. Afterward, io_uring initializes and will use these pointers for the members of *ring*, instead of the pointers that would have been returned by each `mmap(2)`.

On the other side is the inmate that initializes IVSHMEM to access the *r/w* section as well. Similarly, the inmate creates a pointer that points to the beginning of the *r/w* section. At this address resides the *ring* pointer of the application, and its content is copied into the inmates' memory space. Next, a *ring* pointer is created that points to the *ring* in the *r/w* section to reference it. At this point, we have a copy of the *ring* in our inmate and a pointer that points to the *original ring* in the shared *r/w* section. Remember, although it is a shared memory, both parties do not use the same memory address for *ring* because it is a mapping. Therefore, this *ring* copy inside of the inmate has its pointers pointed to memory addresses residing in the application's memory space, as shown in Figure 3.2a. If the inmate wants to use the io_uring, Jailhouse would intercept this malicious memory access and trigger a *VM Exit* for the inmate, thereby stopping it. In the next section, we will explain how to mitigate this behavior.

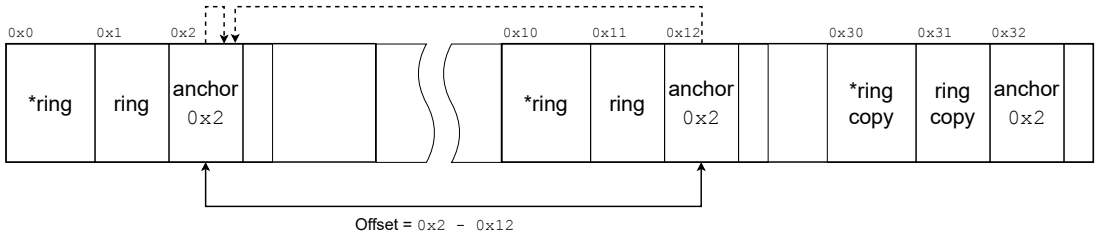
3.3.2 Relocating pointers in the inmate

Continuing with Figure 3.2a, we have to *redirect* the pointers inside the *ring* copy to the *ring* pointer inside of the inmate. This is achieved with a new member inside of io_uring's structure, called *anchor*, which purpose is to calculate the offset between *ring* in the application and inmate. During io_uring's setup, this *anchor* will be assigned its own memory address at which it is allocated.

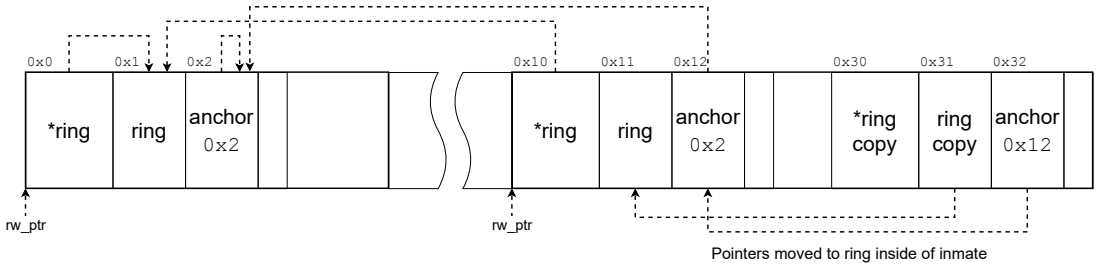
Remember, the inmate has a local *ring* copy and a pointer pointing to *ring*. Now, the inmate accesses its own *ring* and calculates the offset from the address that is stored in *anchor* and the actual memory address where it is located. The result gives the *distance*, i.e., offset, between both *ring* pointers that are mapped into two different memory spaces but are still shared between the inmate



(a) Overview of mapped *r/w* section and pointers inside *ring* structures



(b) Calculating the offset between the *ring*s inside the *r/w* section



(c) Redirecting pointers of local *ring* copy to mapped *ring* inside the inmate

Figure 3.2 – Relocating pointers inside of *ring* copy to point to the mapped *ring* structure inside the inmates shared memory

and application, see Figure 3.2b. For now, both the *ring* inside the inmate and the *ring* copy are still pointing to *ring* inside of the application. The goal is to point the pointers of the *ring* copy to the *ring* that is inside of the inmate and therefore accessible. For each pointer in the io_uring structure, we subtract the calculated offset from the address of each member and assign this new address to the equivalent pointers in the *ring* copy. As shown in Figure 3.2c, *ring* copy is now pointing to *ring* that resides in the same memory space of the inmate. Finally, we can manually insert SQEs into the *ring* copy that will be propagated into the other *ring* pointers, therefore, making it available for the application.

However, instead of working *raw* with the *ring* structure, we will adapt *liburing* and migrate its functionality into the inmate. This is accomplished in the next Section 3.3.3.

3.3.3 Migration of liburing

To work with the io_uring more comfortably and to provide the user with a familiar interface, we migrate *liburing* into the inmate. Because io_uring heavily relies on pointers, it is relatively easy to import fundamental functions of *liburing* that we need for operation in the inmate. For example, the function to retrieve an empty SQE (Section 2.2.3, line 12) can be copied into the inmate as it is. In this thesis, the operation *no-op* was ported into the inmate to be used for measurements in Chapter 4. No-op does not perform any I/O operation. Nevertheless, this SQE will traverse a complete cycle through the SQ and CQ as explained in Section 2.2.2.

However, as addressed in the introduction of Chapter 3, the inmate is a bare-metal application that does not have a kernel executing system calls. Therefore, *liburing* inside the inmate needs to be modified such that it uses IRQs to the user space application instead of system calls. Jailhouse provides the inmate with a function `send_irq()` that is received by the application and is alternatively used for a system call.

3.4 Summary

We have started this chapter with a description of the communication flow between the user space application, kernel, and inmate. This led us to understand that the application is fundamentally placed between the kernel and inmate to act as a bridge between them.

Following that, an implementation approach was elaborated, which was further explained in the last section.

4 ANALYSIS

After proposing the architecture to offload system calls from a bare-metal application to a Linux guest that resides on the same machine and explaining the implementation, this thesis evaluates its performance in this chapter. We begin with an introduction of QEMU that emulates Jailhouse with its guests. Afterward, the underlying hardware of the machine that was used for evaluation is stated. Next, measurements are presented and compared between the user space and inmate to give insights on the latency.

4.1 Evaluation environment

It is favorable to have a dedicated evaluation environment that does not change throughout the measuring process for meaningful results. In this thesis, QEMU is used for development and measurements; however, we will discuss problems with QEMU as a platform for analysis in Section 4.2.3.

4.1.1 QEMU

QEMU is *a generic and open source machine emulator and virtualizer*[12]. As an emulator, QEMU can run operating systems and applications on different hardware than initially designed for, e.g., it is possible to run the Raspberry Pi OS, which is based on ARM, on a machine with the x86 architecture. This is achieved by dynamically translating the *targets* (the OS or program that is emulated) CPU instructions into host (the architecture QEMU is running on) instructions [Bel05]. The benefits are emulations between various architectures with reasonably good performance. Besides that, QEMU is also a virtualizer, meaning it runs operating systems already compatible with the underlying architecture and, therefore not having to translate between target and host code. For vastly better performance than with emulating, the hardware is virtually split up, and portions are assigned to the target OS, such that it can interact with the hardware directly.

Another advantage of having a virtual machine (VM) is the (virtual) isolation from the host machine. A guest sits in a sandbox controlled by QEMU and protects the host from malicious behavior of the guest. Additionally, a guest can crash or reboot without affecting the host machine, which is beneficial when dealing with low-level OS code that potentially can crash the VM.

QEMU was chosen as a development and virtualization platform because of its minimalism and ability to run OSs that were not adapted for QEMU beforehand. On top of that, Jailhouse

[12] https://www.qemu.org/

already came with a ready-to-use image for QEMU[13], therefore without having to deal with cell configurations and their creation as mentioned in Section 2.1.2.

4.1.2 Hardware

Although performance and measurements vary between hardware and virtualization, the specifications of the used machine are presented in Table 4.1. Two cores were assigned to QEMU, one for the root-cell and inmate each, and 1GB of RAM.

4.2 Measurements

Measurements are essential to verify that an implementation is functional and operates as intended. This chapter will present the measurements of different tests that evaluate the latency and overhead introduced by offloading system calls through io_uring.

4.2.1 Method

While time is a reasonable measurement unit because a developer would like to know how long it will take from issuing an submission queue entry (SQE) to receiving the corresponding completion queue entry (CQE), this thesis utilizes another method.

Relatively modern processors of the x86 architecture come with a time stamp counter (TSC)[14]. This counter increments a register on every CPU cycle since it was reset. As more multi-core processors emerged, the TSC lost its accuracy because programs are usually not fixed to a single core and run parallel with other programs. The current value of the counter is read with the instruction `rdtsc`.

Even though the TSC is not accurate any more in most situations, it can be used with Jailhouse because this hypervisor can split off single cores and assign them to bare-metal applications, such that they do not have to share CPU time with other guests. In this thesis, a modified inmate is used that contains the proposed implementation of io_uring and liburing and three tests for measuring the elapsed CPU cycles. First, the overhead has to be measured that will tell how many cycles elapse between calling `rdtsc` right after each other. The current TSC is saved in a variable `start`, and in the following line, the counter is retrieved again and saved in the variable `end`. The difference between these two values gives us the number of cycles elapsed. Now, we can insert any functions or instructions between these two `rdtsc`s calls to know how many cycles were needed for the execution of the encapsulated operations. How io_uring performed inside the inmate is presented next.

[13] https://github.com/siemens/jailhouse-images#quickstart-for-virtual-targets
[14] https://man.netbsd.org/x86/rdtsc.9

Property	Specification
Processor	Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz
CPUs	8
RAM	7841MB
OS	Ubuntu 18.04.6 LTS
cat /proc/version	Linux version 5.4.0-84-generic

Table 4.1 – Specifications of the machine for development and measurements

4.2.2 Results

The following three measurements were recorded:

- Overhead of rdtsc()
- io_uring issues a `NOP` without setting data on the SQEs
- io_uring issues a `NOP` with setting data on the SQEs

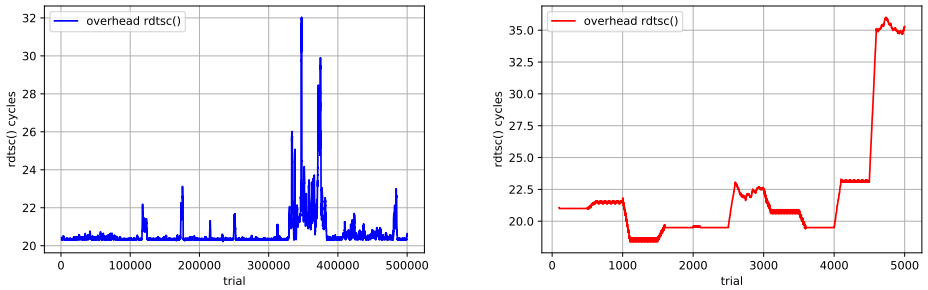
These measurements were once performed inside the user space application without the implementation presented in this thesis because the application has access to the kernel and io_uring and once inside the inmate, utilizing the implementation that builds on IVSHMEM. Due to limitations, there are only 5,000 records for each measurement inside the inmate, whereas 500,000 records for the user space application.

All measurements are cleaned from outliers, i.e., any value that is more than three standard deviations from the mean. A moving average was applied because the measurements fluctuated vastly and were not correctly representable in the graphs.

In Figure 4.1 the overhead of `rdtsc()` is measured and compared between the user space application and the inmate. The application has its mean at around 20 CPU cycles for the overhead of `rdtsc()`, whereas the inmate has its mean at 21 cycles. There is just a tiny difference noticeable, which makes sense because this measurement does not rely on $io_uringandthesharedmemory$.

Now, in Figure 4.2 we compare the most minimal cycle for fully traversing through the io_uring interface. First, only a NOP is issued, meaning there is no hardware we have to wait for, and no data inside the SQEs is set. In this comparison, we begin to see a significant difference between the application and inmate. The application averages at 4,137 CPU cycles, while the inmate needs 340,567 cycles on average. This is since the inmates io_uring has to pass through much more layers than the application. As seen in Figure 3.2, the inmate has a *ring* copy that points to another *ring* inside its own virtual memory space. However, this other instance is just a pointer as well that is also mapped from the root-cell into its own cell. The inmate has to dereference a lot more pointers than the application. Next, Jailhouse strictly manages the hardware and memory accesses, which must be checked for malicious accesses each time a cell wants to get data from memory.

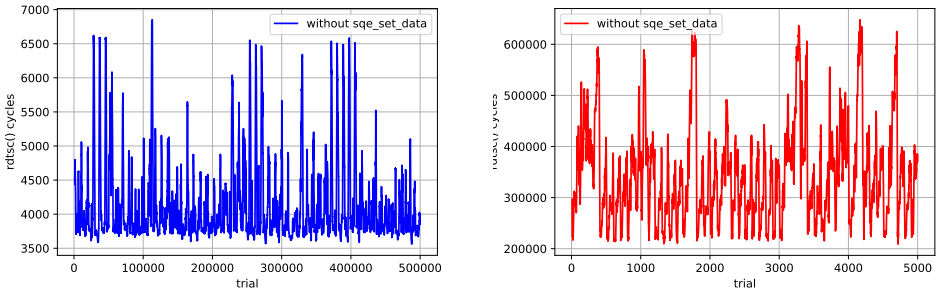
However, in the real world, it is desirable to set data on each SQE; otherwise, it is impossible to distinguish different SQEs from each other, except a maximum of one SQE is issued at all times or the ring size is equal to 1. Therefore, a comparison with setting data on an SQE is given in Figure 4.3.



(a) Running `rdtsc()` only in user space

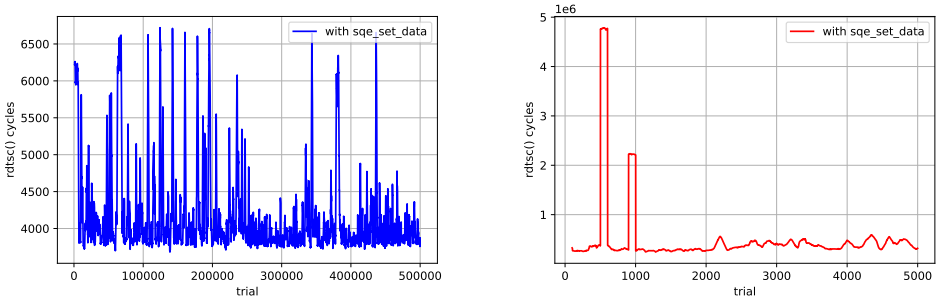
(b) Running `rdtsc()` only in inmate

Figure 4.1 – Comparison of `rdtsc()` overhead between user space and inmate



(a) without `io_uring_sqe_set_data()` in user space **(b)** without `io_uring_sqe_set_data()` in inmate

Figure 4.2 – Comparison of issuing SQEs without `io_uring_sqe_set_data()` between user space and inmate



(a) with `io_uring_sqe_set_data()` in user space **(b)** with `io_uring_sqe_set_data()` in inmate

Figure 4.3 – Comparison of issuing SQEs with `io_uring_sqe_set_data()` between user space and inmate

With a mean of 4,170 CPU cycles in the user application, it takes slightly more cycles than without setting data in the user space (Figure 4.2a). Due to another memory access and more dereferencing, the inmate takes 487,894 cycles on average – nearly 150,000 cycles more than before.

Anyway, these measurements have to be taken with a grain of salt elaborated in the next section.

4.2.3 Measuring in VMs

These measurements were performed inside of Jailhouse that was virtualized with QEMU, while QEMU does not run bare-metal and needs to run on top of a General Purpose Operating System (GPOS). These are a lot of layers that the user space application and inmate have to pass through, which has an impact on the count of cycles necessary to execute instructions.

A more appropriate measurement has to be done on bare-metal hardware, which Jailhouse is intended for but was not in the scope of this thesis.

4.3 Summary

In this chapter, we have introduced and explained the environment that was used for taking measurements. The performance of io_uring inside the user application (root-cell) and inside the inmate (cell) was measured and evaluated for the most minimal usage of the io_uring interface. We were able to see a vast increase of CPU cycles, counted with the time stamp counter (TSC), in the inmate due to the fact of using many pointers that are needed to overcome Jailhouse's partitioning of guests and thereby using the interface. It was also suggested that these measurements are not precise and should be taken with caution.

5 CONCLUSION

This thesis shows the implementation of a mechanism to offload system calls from virtually isolated guests in the *Jailhouse* hypervisor. The hypervisor statically partitions the hardware and assigns those resources to *cells*. A *root-cell* is always present in the Jailhouse and houses the Linux kernel as its guest. The other part of this thesis is the relatively new *io_uring* interface that allows applications to issue asynchronous I/O calls, e.g., reading from files.

This thesis aims to exploit the shared memory that Jailhouse can establish between two cells for communication. A user space application in Linux initialized io_uring inside the memory, such that the inmate can access it. However, modifications to io_uring had to be made because the rings are usually allocated in the kernel space of Linux that is not accessible by other guests. The *liburing* library was adapted in small portions to the inmate, so developers can use the already known functions to interact with the io_uring.

The performance between the usage of io_uring inside the Linux kernel and inmate is drastically different, but since the inmate has to pass through many layers of memory and the user space that acts as a broker between the inmate and kernel. However, measurements were inaccurate because they were recorded in QEMU.

In conclusion, this thesis implemented a proof-of-concept that leverages the partitioning hypervisor Jailhouse and io_uring to provide non-root-cells with asynchronous I/O. However, the implementation includes the bare minimum of functionality and does not have a useful purpose because only a `NOP` operation of io_uring was implemented and used for measurements.

In the future, more asynchronous I/O of io_uring can be implemented, showcasing inmates doing actual I/O operation through the interface and more comparisons to operations in the application. The implementation must also be ported to real hardware, running Jailhouse bare-metal, to get better readings of the CPU cycles used.

LIST OF ACRONYMS

VM	virtual machine
IRQ	Interrupt request
GPOS	General Purpose Operating System
I/O	Input/Output
IVSHMEM	Inter-VM Shared Memory
BAR	Base Address Register
SLOC	source lines of code
TCB	trusted computing base
SQE	submission queue entry
CQE	completion queue entry
MITM	man-in-the-middle
TSC	time stamp counter

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[Bar+03] Paul Barham et al. "Xen and the art of virtualization." In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 164–177.

[Bel05] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA. 2005, p. 46.

[Bro06] Manfred Broy. "Challenges in automotive software engineering." In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 33–42.

[Jen] Jens Axboe. *Efficient IO with io_uring*. URL: https://kernel.dk/io_uring.pdf.

[Jon] Jonathan Corbet. *Fixing asynchronous I/O, again*. URL: https://lwn.net/Articles/671649/.

[Mar+20] José Martins et al. "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems." In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.

[Ram+17] Ralf Ramsauer et al. "Look Mum, no VM Exits! (Almost)." In: *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '17)* (Dubrovnik, Croatia). 2017. URL: http://arxiv.org/abs/1705.06932.

[SMP21] Bruno Sá, José Martins, and Sandro Pinto. "A First Look at RISC-V Virtualization from an Embedded Systems Perspective." In: *arXiv preprint arXiv:2103.14951* (2021).

[Ste19] Stefano Stabellini. *Xen Dom0-less*. Xilinx. Apr. 2019. URL: https://static.linaro.org/connect/bkk19/presentations/bkk19-512.pdf (visited on 11/04/2021).

[Val15] Valentine Sinitsyn. *Jailhouse*. Linux Journal. June 8, 2015. URL: https://www.linuxjournal.com/content/jailhouse (visited on 11/12/2021).