The Optimization of IVSHMEM Based on Jailhouse

Jiaming Zhang, Fengyun Li, Liu Yang, Yucong Chen, Hubin Yang, Qingguo Zhou, Yan Li(✉), and Rui Zhou(✉)

School of Information Science and Engineering, Lanzhou University, Lanzhou, Gansu, China
{jmzhang2020,220220941880,220220942471,chenyc18, yanghb2019,zhouqg,ynali,zr}@lzu.edu.cn

Abstract. The hypervisor, with its resource isolation, security guarantees, and ability to meet high real-time requirements, offers significant advantages in real-time scenarios. Furthermore, its communication capabilities play a crucial role in enabling collaborative computation tasks across different virtual machines. The Jailhouse hypervisor, known for its real-time capabilities and secure embedded platform, demonstrates outstanding performance in real-time scenarios. However, the inter-virtual machine (inter-VM) communication protocol based on Jailhouse is not yet mature, necessitating optimization to enhance its suitability for real-time communication scenarios. Firstly, the existing communication mechanism underwent reconstruction, involving the disabling of the one-shot interrupt mode and expanding the shared memory area. Secondly, an experimental platform was established on the Raspberry Pi-4B, configuring the real-time system and adopting the io_uring methods. Finally, experimental evaluations were conducted to assess the differences in communication delay, throughput, and data transmission delay before and after the communication protocol reconstruction. Additionally, the mitigating effect of the new communication mechanism on VMexit behavior was also evaluated. The experimental results demonstrate that the enhanced communication mechanism significantly reduces both the system call overhead and the number of VMexit compared to the native communication protocol (Inter-VM Shared Memory, IVSHMEM). Moreover, the throughput exhibits a notable improvement of approximately 200 MB/s.

Keywords: Virtualization · Jailhouse · IVSHMEM · communication mechanism · RTOS

1 Introduction

Amidst the escalating software complexity and the prevailing shift towards heterogeneous computing platforms, consolidating multiple functionalities within

J. Zhang and F. Li—These authors contributed equally to this work.

C. Li et al. (Eds.): APPT 2023, LNCS 14103, pp. 54–75, 2024.
https://doi.org/10.1007/978-981-99-7872-4_4

a single hardware platform is considered an optimal approach to tackle issues related to space, weight, power consumption, and economic considerations [1]. One effective strategy for resource consolidation is the utilization of virtualization technology. By employing a hypervisor, it becomes possible to operate multiple cells with different operating systems on a single platform, thereby enhancing the integration capabilities of the hardware platform. Consequently, the concept of distributed systems becomes highly significant, as it allows for the deployment and execution of applications across multiple physical or virtual machines, further enhancing system flexibility and efficiency. This combination of virtualization and distributed processing has stimulated the development of Mixed Critical Systems (MCS), where tasks with varying levels of criticality are executed by corresponding operating systems [22]. Virtualization technology plays a central role in resource partitioning and task execution within MCS.

Solutions for implementing mixed critical systems within a single system can be classified into two categories [3]: dual-kernel solutions and resource partitioning solutions [30]. The former category comprises RTLinux, RTAI, and Xenomai, while the latter includes Bao [15], ACRN [12], and Jailhouse [2]. This article specifically focuses on the resource partitioning solution-Jailhouse.

Jailhouse [18,23,24] represents a typical static partitioning hypervisor that emphasizes static resource allocation. Its compact codebase stems from its lack of a scheduler or virtualization services. This characteristic not only facilitates the certification of security features but also contributes to the vibrant development of the Jailhouse community [7]. When constructing MCS, Jailhouse exhibits two primary advantages [13,21]. Firstly, in comparison to virtualization managers such as KVM and Xen, which are more tailored to server environments, Jailhouse better fulfills the real-time requirements of embedded platforms. Secondly, Jailhouse conforms to the demands of safety-critical design due to its small footprint, aligning well with certification requirements. In contrast, other hypervisors similar to Jailhouse, such as SafeG [20] and Quest-V [27,28], have limited application scenarios or lower adaptability in practical usage.

Inter-VM communication functionality serves as a crucial component of hypervisors. To optimize Inter-Process Communication (IPC) efficiency, shared memory technology has been explored [5,6], aiming to reduce data replication and transmission overhead. While Jailhouse provides a socket interface based on the TCP/IP protocol [17] for inter-VM communication, which establishes communication channels between multiple physical machines, its suitability for latency-sensitive tasks is restricted due to protocol stack requirements, frequent data copying, and multiple context switches [12]. Jailhouse offers the IVSHMEM (Inter-VM Shared Memory) protocol at the lower level, which enables efficient data communication with minimal overhead and is particularly well-suited for local environment implementation. However, the absence of an extensive user-level API and concerns about the maturity of the shared memory protocol impose limitations on IVSHMEM, preventing it from becoming the prevailing inter-VM communication method in Jailhouse.

Based on this premise, the primary focus of this research is to enhance the inter-VM shared memory protocol, IVSHMEM, in Jailhouse, specifically for real-time communication scenarios. The objective is to facilitate communication between two virtual machines, meeting the demands of predictability and high throughput required by real-time systems, while abstaining from the utilization of virtual networks for data exchange. The main contributions of this article are as follows:

1. Investigating the inter-cell[1] communication protocol based on shared memory in Jailhouse, with a particular emphasis on the performance overhead and optimization directions for real-time communication.
2. Reshaping the current communication mechanism by optimizing the communication process, expanding the shared memory region, addressing issues related to excessive VMexit behavior and small shared memory, and designing memory barriers and synchronization mechanisms to ensure synchronized data access.
3. Establishing a Linux and RTOS experimental environment based on Jailhouse on the Raspberry Pi 4B platform, configuring real-time systems, and exploring the use of the asynchronous I/O mechanism, io_uring, to enhance I/O performance.
4. Conducting experimental evaluations to assess the changes in communication latency, throughput, and data transfer latency before and after the reconstruction. The aim is to validate the applicability, feasibility, and advantages of the new communication mechanism in real-time communication scenarios between two cells.

2 Related Works

Communication plays a crucial role in building MCS. Data exchange is required between Host-Guest and Guest-Guest through communication. However, communication between Guest-Guest cells may incur higher costs compared to Guest-Host communication. Hence, optimizing communication mechanisms is essential to ensure the efficiency and reliability of MCS. The design and optimization of inter-cell communication mechanism is a key research area in Jailhouse. This section explores the communication process between cells, provides an overview of the IVSHMEM communication protocol and the background of the io_uring technology, and reviews the relevant research progress.

2.1 IVSHMEM Communication Protocol

Figure 1 provides an example using a Linux cell to illustrate the communication flow between the root cell (cell 0) and the Linux cell (cell 1). Inter-cell communication is facilitated through the utilization of shared memory. During this

[1] It is worth noting that in Jailhouse, a virtual machine is commonly referred to as a "cell". Therefore, the terms "inter-VM" and "inter-cell" are equivalent in meaning.

process, the application program in the root cell copies user space data from the kernel space and writes it into the shared memory using a PCI device. The Linux cell, on the other hand, reads the data from the shared memory and transfers it through various layers until it reaches the application program within the Linux cell. To facilitate these operations, the IVSHMEM PCI device is managed by the UIO (Userspace I/O) device driver. Through the override of the mmap system call, the driver enables efficient read and write access to diverse memory regions within the device and the shared memory.

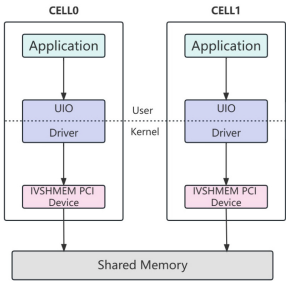

Fig. 1. Inter-cell communication process

The IVSHMEM communication protocol, based on Jailhouse, establishes specifications for inter-cell communication, enabling seamless communication between cells through shared memory and interrupt signaling mechanisms. The IVSHMEM PCI device acts as an interface between the host's shared memory interface (POSIX) and the applications running within the cells. It utilizes Linux event file descriptors to facilitate the transmission of interrupt signals between virtual machines [14]. The functionality and configuration of the IVSHMEM PCI device are primarily determined by three components: the device configuration register set, the device register region, and the shared memory region. The configuration section, visible to the operating system, contains relevant information such as the vendor ID and device ID, which enables the operating system to load the appropriate driver.

The latest version of the IVSHMEM protocol, version 2.0, provides a range of functionalities and features [8]. These functionalities encompass support for up to 65536 interconnected communication nodes, multiple types of shared memory regions, interrupt-based signaling for node communication, support for different shared memory protocols, and memory mapping implementation for the device register region. Initially completed in Jailhouse version 0.9 in 2018, the protocol has undergone improvements in the latest version of Jailhouse (0.12, released in 2020), which address specific communication protocol issues and add support for the Raspberry Pi-4B platform [9].

In 2017, Masaki Miyagawa et al. conducted a comparative study and testing of the IVSHMEM shared memory protocol and the traditional TCP/IP communication protocol. The study focused on different stages of the Jailhouse boot

process and performed in-depth analysis of the memory regions in the cell configuration files, using QEMU (v2.8.1) and Jailhouse (v0.6) on an industrial platform [16]. In 2019, the official Jailhouse community announced the release of IVSHMEM 2.0 version [11]. Ramos et al. investigated the process of inter-partition communication using the IVSHMEM protocol on the BananaPi-M1 platform managed by Jailhouse. The study specifically focused on the transmission of messages between two partitions [17]. In another study, Schade et al. developed an intelligent industrial controller for a computer numerical control (CNC) machine on the same BananaPi-M1 platform. The controller implemented tool overload detection and predictive maintenance, using Jailhouse to coordinate concurrent execution and resource access between Linux and FreeRTOS. Communication between the real-time operating system (RTOS) and Linux was achieved through the simplified RPMsg protocol (RPMsg-Lite), while data exchange and transmission utilized the IVSHMEM interface.

2.2 IO_uring

Traditional I/O operations are typically synchronous, also known as blocking I/O. In this approach, each operation is initiated by the application, which then pauses and waits for the operation to complete. Read/write functions are used to perform read and write operations on the underlying files. However, compared to asynchronous I/O, blocking I/O's performance is limited by the file type and device capabilities, resulting in potential program blocking. Clearly, blocking I/O is inadequate for meeting the demands of high real-time scenarios. Asynchronous I/O allows the application to execute other tasks during the waiting period, significantly reducing the frequency and overhead of system calls and improving efficiency. As shown in Fig. 2, Linux I/O operations involve data reading and writing, and a typical I/O operation goes through two stages: data preparation and data copying. After a user-level process initiates an asynchronous I/O request, it promptly receives status information returned by the kernel. The process can continue its execution rather than being in a blocked state. The kernel awaits data completion, performs data copying to the user's memory, and eventually sends a signal to the user process, notifying it that the I/O operation has been completed.

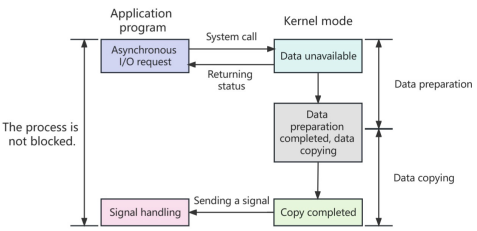

Fig. 2. Asynchronous I/O operation flow

Io_uring is an asynchronous I/O implementation provided by the Linux kernel starting from version 5.1. Its main feature is the ability to reduce the overhead of system calls and alleviate the cost of data copying. The reason for considering the integration of io_uring technology into Jailhouse primarily lies in its capability to achieve zero-copy transfers by constructing a shared ring buffer between the kernel and user space. This eliminates the need for data copying involved in traditional data transfers between the kernel and user space. Although the overhead of a single system call is minimal, frequent system calls can become a performance bottleneck in high-performance applications. Optimizing I/O performance is an important direction for improving real-time systems and contributes to enhancing system predictability [29].

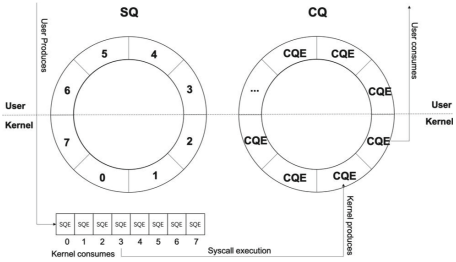

Fig. 3. io_uring Framework

In Fig. 3, io_uring establishes a shared memory region between the user and the kernel using mmap. It constructs two lockless ring queues, namely the submission queue (SQ) and the completion queue (CQ), based on memory barriers. The SQ queue is used for the user program to submit I/O tasks to the kernel, and the completed tasks are placed in the CQ queue, from which the user program retrieves the results. During the submission of tasks and the return of task results, the user program and the kernel share the data in the ring queues. I/O requests and completion events no longer need to be passed through system calls, completely avoiding the overhead of copy_to_user/copy_from_user operations.

In Linux, when using synchronous or asynchronous programming interfaces, each I/O request typically requires at least one system call. However, in io_uring, multiple requests can be submitted at once, with each Submission Queue Entry (SQE) describing an I/O operation. This is achieved through a single system call, further reducing the overhead of system calls. Additionally, the polling mode of io_uring further reduces system calls and interrupt notifications.

Related studies have explored the implementation of asynchronous I/O using the ivshmem shared memory protocol with io_uring, as demonstrated by Reichenbach et al. [19].

3 Design and Implementation

According to the actual application case (ivshmem demo) source code of IVSHMEM officially provided by Jailhouse, it is analyzed that the communication process of IVSHMEM has the following drawbacks and performance overhead.

1. Low-frequency interrupt signal sending: The use of the alarm command limits the sending of interrupt signals to once per second, resulting in a low frequency.
2. Frequent VMexit: The communication protocol requires writing to the interrupt enable register and device status register in the device register area through VMexit. This behavior occurs frequently during interrupt handling and status updates, leading to performance overhead.
3. Waste of shared memory space: In the current practical application of the communication protocol, the same data is written separately to the rw and out regions corresponding to the current cell. This approach results in wasted shared memory space.

These factors can affect the real-time communication, throughput, and overall system performance. To address the performance overhead of the current IVSHMEM protocol in practical applications, this paper proposes optimizations in the following four aspects.

3.1 Reconstruction and Mapping of IVSHMEM Shared Memory Regions

The current IVSHMEM protocol divides the shared memory region into three regions with different read and write permissions (a Read-Only State Table region used to define and describe the status and attributes of cells, a Read/Write Region used for data sharing and a In Region used to read data from other cells). When communicating between two or three cells, a significant amount of unused memory space is present. Therefore, in this paper, we have reconstructed the shared memory region of the IVSHMEM device by modifying the Jailhouse and Linux device driver code. The original 36 KB Read-Write Region has been modified to 64 KB, providing two new designs for shared memory types to support structured data and maximize the utilization of the shared memory region, thus improving communication efficiency. Additionally, the practice of identifying the current cell's status by writing to the device status register has been eliminated. This approach consumed a significant portion of memory space while offering minimal status information.

The two shared memory structures are shown in Fig. 4. Shared Memory Type I divides the shared memory into 16 fixed-length frames, with each frame being 4KB. It supports concurrent usage of shared memory by multiple processes, where different frames are used by different applications to avoid data synchronization overhead. Shared Memory Type II treats the 64KB shared memory region as a unified whole and dynamically allocates memory using the malloc

tool. Based on the Sender ID and Receiver ID in the protocol header, this type of memory primarily serves the data communication between two cells' processes. The Semaphore field indicates data availability, and the Semaphore and other shared memory region data are promptly updated using device memory barrier primitives. The communication protocol fields also encompass current data frame length, total number of data frames, current frame number, and data pointer, among others. For the two shared memory types, mapping is performed separately. The mapping process uses a character pointer array to index different types of shared memory regions to their respective memory locations. By designing the two shared memory types, structured data can be used in shared memory, supporting concurrent access by multiple processes, meeting dynamic shared memory requirements, and expanding the shared memory region of the IVSHMEM PCI device.

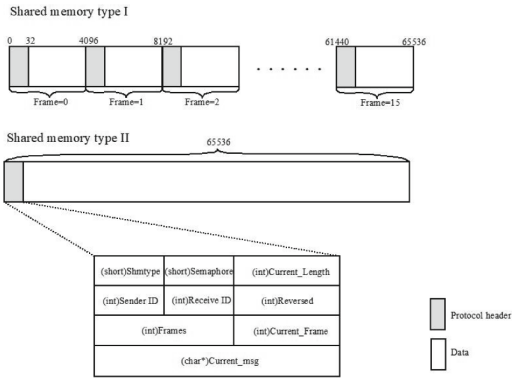

Fig. 4. Two types of shared memory

In the user-space program, the prot field in the mmap system call is modified to set the read-write attributes of the memory mapping region. Moreover, modifications are necessary in the cell configuration file, Jailhouse code, device driver code, and user-space code. Subsequently, the kernel needs to be recompiled and installed, and Jailhouse must be enabled.

3.2 Device Memory Barriers and Synchronization Mechanisms

Mechanism Design. Considering the selected platform architecture, the design also takes into account the memory barrier and synchronization mechanisms when operating on device memory regions. A memory barrier for device memory is a method of controlling the order of execution of CPU or other device instructions, aimed at maintaining data consistency in multi-threaded or multi-process environments. Synchronization mechanisms, such as mutex locks and semaphores, are common programming techniques used to regulate the access

order of shared resources among multiple processes or threads to prevent data race conditions. Proper application of device memory barriers and synchronization mechanisms can ensure timely synchronization of device memory regions between two cells, improving operational stability, efficiency, and preventing data conflicts.

In the ARMv8 architecture, the device memory barrier mechanism adjusts the order of memory operations by differentiating between instruction and data caches and employing three different memory barrier instructions: instruction synchronization barrier (ISB), data memory barrier (DMB), and data synchronization barrier (DSB). By setting the cache attribute of the shared memory region to MAP_CACHED and using memory barrier primitives, the timeliness of the data is ensured.

Although memory barriers ensure consistency in CPU memory access order, prevent data races and out-of-order execution, the order of shared memory operations also needs to be consistent between communicating parties. To achieve the desired consistency, an interrupt-based approach is employed, effectively guaranteeing the order of shared memory operations. The interrupt-based approach listens for interrupt events on the device file object to ensure the order of operations. Once an interrupt occurs, the object associated with the interrupt event saves a semaphore, which is used to determine whether a specific I/O event behavior is satisfied. If the condition is met, the system considers that an interrupt signal has been received from the other communicating object, and then the interrupt handler function is executed, thus ensuring the order of operations.

Modules Design. In order to fully utilize the reconstructed shared memory regions, we propose a design approach for data initialization and the send/receive modules, combining knowledge of kernel barriers and synchronization. We have redesigned the data preparation and communication processes, dividing them into the communication initialization module and the data communication module. The former is responsible for two types of memory mapping and io_uring initialization, while the latter handles the initialization, sending, and receiving of data for the two types of shared memory.

Taking shared memory type II as an example (as shown in Fig. 5), the communication initialization module is divided into two parts: mapping and reconstructing the shared memory, and io_uring initialization. The specific steps are as follows:

1. Map the discrete IVSHMEM PCI device's shared memory to virtual memory and consolidate it into a unified block divided into 16 data frames of 4 KB each.
2. After memory mapping, format the communication protocol data structure in the memory header. For shared memory type I, the header of each data frame needs to be individually initialized.
3. Utilize io_uring as a replacement for the read operation to perform I/O operations for reading data from external storage devices. The read data is seg-

mented according to the frame length and the data pointers are filled into the corresponding fields of the shared memory protocol area.

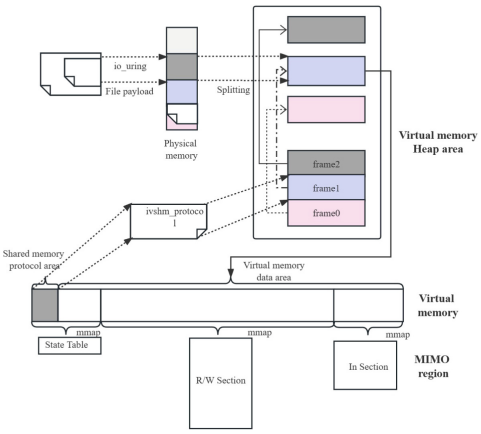

Fig. 5. Initialization Module (Communication Memory Type II)

After initialization, data frames are sent in parallel for shared memory types I and II, with type I sending frames concurrently and type II sending frames sequentially. Memory barriers and semaphores are used to ensure data consistency and synchronization. This process represents the data sending process for both shared memory types, and the data receiving process is the reverse of the data sending process. Taking shared memory type II as an example (Fig. 6), after initialization, each frame is sequentially sent. The first data frame is written into the shared memory data area, then an interrupt vector value is written to the doorbell register in the register area to send an interrupt signal to the communication object. Finally, the critical section is entered, where memory barriers ensure data consistency. Within the critical section, the process waits for the semaphore indicating that the data has been read and receives the interrupt signal sent by the communication object before exiting the critical section. Once the current frame data is sent, the next frame data sending process begins.

3.3 Disabling the One-Shot Interrupt Mode

Disabling the one-shot interrupt mode can improve system predictability and reliability, as well as enhance system performance in real-time communication scenarios. Instead of using periodic alarm-based interrupts, direct manipulation of the mmio region's registers is employed to increase the frequency of interrupt signal transmission.

Enabling the one-shot interrupt mode involves setting the Privilege Control Register in the device's feature extension register group to 1. In this mode, the

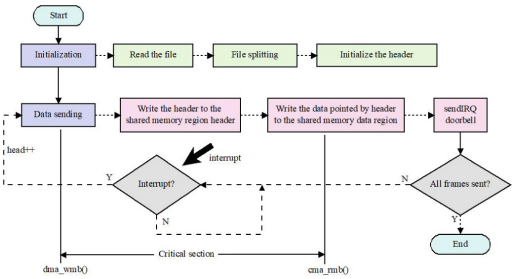

Fig. 6. Data Transmission Module (Communication Memory Type II)

Interrupt Control Register, responsible for enabling interrupts, is automatically reset to 0 after each interrupt delivery. To allow for subsequent interrupts to occur, the interrupt enable register needs to be reconfigured to 1 within the application's interrupt handler. However, enabling the one-shot interrupt mode may introduce certain issues, such as unnecessary interrupts and additional processing overhead (i.e., VMexit behavior).

To tackle this issue, the approach of disabling the one-shot interrupt mode is adopted. Within the IVSHMEM device driver, the pertinent statements responsible for writing to the device's memory are commented out, thereby maintaining the interrupt enable register at 1 and preventing unnecessary operational overhead. By deactivating the one-shot interrupt mode, there is no longer a requirement to perform the re-enable interrupt operation in the interrupt handler, which is commonly used to ensure proper reception of the subsequent interrupt.

3.4 Applications of IO_uring

To further enhance performance, the traditional read method is replaced with the asynchronous I/O approach of io_uring, resulting in lower system call overhead and improved latency benefits.

The liburing tool is utilized to implement asynchronous I/O operations. Liburing is an open-source library designed to simplify and manage the io_uring interface, allowing developers to more conveniently implement asynchronous I/O operations. It provides a higher-level interface that eliminates the need to directly interact with the native io_uring interface, thereby streamlining the process. Consequently, the invocation interface of the liburing tool differs from the native io_uring interface, as shown in Fig. 7.

The specific workflow for using io_uring is as follows:

1. Initialize the circular buffer by calling the io_uring_queue_init() function. This function accepts two parameters: the Queue Depth (QD) and a ring object (struct io_uring). The QD value is shared between the Submission Queue Entries (SQEs) and Completion Queue Entries (CQEs), with the number of CQEs being twice the number of SQEs.

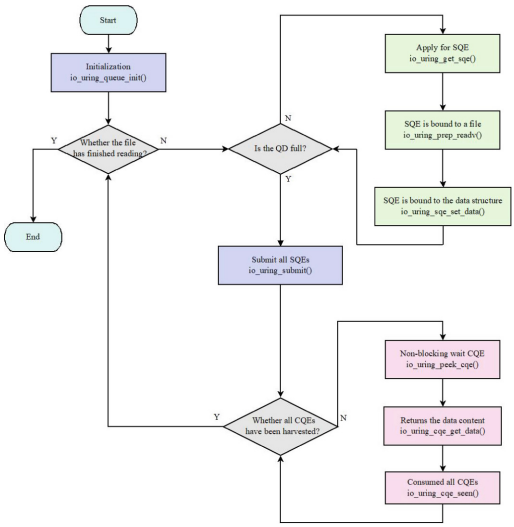

Fig. 7. io_uring data reading process

2. Perform the necessary preparations before submitting the requests, ensuring that all QDs are utilized. This involves allocating SQEs, associating them with file objects and readv operations, reading data, and binding the data with the SQEs. Each read operation reads BS bytes (approximately the size of the shared memory region) and is repeated QD times.
3. Submit the I/O requests. Once all the QDs are fully utilized, the requests are collectively submitted using the io_uring_submit(ring) system call. The io_uring method significantly reduces the number of system calls. In traditional approaches, each request requires at least one system call. However, with io_uring, multiple requests (each represented by a distinct SQE, corresponding to an I/O operation) can be added at once, and the submission is completed with a single system call, io_uring_submit(). Moreover, by employing polling, the kernel can process the SQEs without relying on io_uring_submit(), thereby reducing system performance overhead.
4. The kernel processes the submitted requests and appends the completion events (CQEs) to the end of the completion buffer. Each SQE corresponds to a CQE and contains the status of the respective request.

4 Evaluation and Analysis

The experimental platform used in this study is Raspberry Pi-4B, with a Linux and RTOS system built on Jailhouse. The Linux kernel version used is 5.4.16, and during the kernel compilation process, the kernel configuration file needs to be selected. The Jailhouse version used in this setup is 0.12.

Based on the specific requirements of the Raspberry Pi-4B platform and considering the prototype validation process of Jailhouse, Ralf Ramsauer et al. chose

to run a slim Linux operating system with the PREEMPT-RT real-time extension in the secure critical cell [10,26]. In addition to using a Micro-Kernel as the RTOS kernel in a dual-kernel configuration, another approach for achieving real-time capabilities within the Linux kernel itself is Linux (PREEMPT-RT) [25]. This approach involves modifying the existing Linux kernel, including but not limited to modifications to components such as the general timer, interrupt handling structures, and mutex locks, to support real-time capabilities. This work was successfully merged into the mainline Linux kernel in 2004 [4]. Therefore, in this study, we also choose the Linux (PREEMPT-RT) approach as the target RTOS cell for designing the inter-cell communication mechanism.Regarding the configuration of RTOS, it is approached from both the kernel and cell configurations. Specifically, this includes enabling the dynamic tick and tickless options in the kernel, disabling interrupt balancing optimization and the RCU (read-copy-update) callback mechanism, enabling kernel preemption, and disabling processor frequency scaling and idle-state management features.

4.1 Performance Comparison Between Two I/O Methods

To compare the system call counts and latency differences between the two I/O methods of read and io_uring when reading data from a file, this article utilizes a system call count testing script and a latency testing script for the purpose of conducting the tests.

Listing 1.1. System Call Count Evaluation Script

```
sudo strace -o strace.log -c tools/ivshmem-demo ${DATA_file} ${QD}
cat strace.log | grep "total" | awk {print $3}
```

Listing 1.2. Latency Testing Script

```
static unsigned long emul_division(u64 val, u64 div)
{
   unsigned long cnt = 0;
   while (val > div) {
       val -= div;
       cnt++;
   }
    return cnt;
}

u64 timer_ticks_to_ns(u64 ticks)
{
    return emul_division(ticks * 1000, timer_get_frequency() / 1000 / 1000);
}
```

Comparison of System Call Counts Between Read and io_uring. By using the testing script (Listing 1.1), the data file was consecutively read 10 times, and the results are shown in Fig. 8. The QD value represents the maximum number of concurrent tasks that an application can handle. The appropriate selection of the QD value depends on the system hardware performance and the requirements of the application. To fully utilize system resources, enhance system throughput, and improve performance, it is essential to determine the

suitable QD value for the current target platform through experimentation. To optimize system call overhead on the experimental platform used in this paper, we recommend selecting a QD value higher than 64, such as 128.

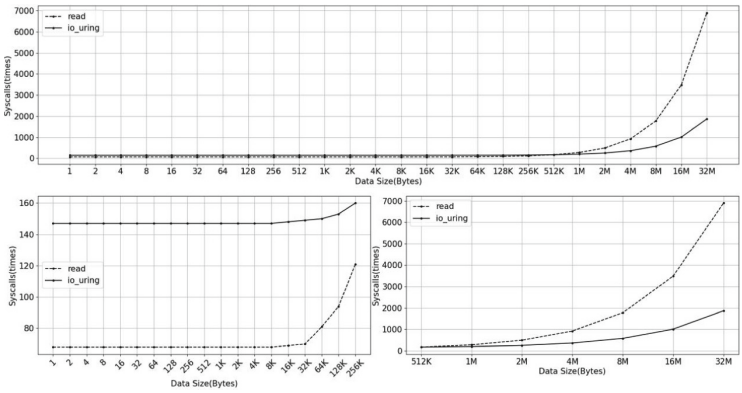

Fig. 8. Comparison of system call frequencies between io_uring and read

The results demonstrate that the read method incurs lower system call overhead when the data size is below the 1 MB threshold. However, as the data size exceeds 1 MB, the system call count of the read method gradually increases, highlighting the disparity between the read method and the io_uring method. At a data size of 32 KB, the system call count of the read method sharply rises. Starting from a data size of 512 KB, the difference in system call counts between the two methods becomes less significant, but the read method experiences a much higher increment compared to the io_uring method. When the data size reaches 32 MB, the read method falls behind the io_uring method by approximately 5000 system call counts. This difference may be attributed to the cache sizes of the experimental platform (L1-Dcache: 32 KB, L2-cache: 1 MB). In contrast to the rapid increase in system call counts of the read method, the io_uring method exhibits a slower increase, making it more suitable for scenarios with larger data sizes.

Comparison of Latency in Data Retrieval Between Read and io_uring. With the help of the latency testing script (Listing 1.2), the latency differences between the two methods were tested when reading text content of different data sizes. A suitable QD value was selected as the latency result for the io_uring method, and the experiments indicate that a QD value of 8 has a positive impact on improving data read latency.

As shown in Fig. 9, considering the latency differences between the two methods across all data sizes, the io_uring method consistently exhibits better latency performance than the read method. Below the threshold of 1 MB data size, the read method demonstrates more noticeable latency jitter compared to io_uring.

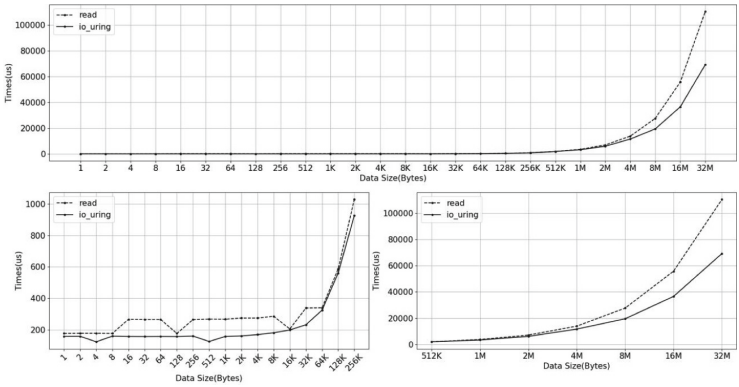

Fig. 9. Comparison of data retrieval latency between io_uring and read

As the data size exceeds 1 MB, the latency difference between the two methods gradually increases with the increasing data size. At a data size of 32 MB, the maximum difference is observed, with the io_uring method showing a latency advantage of approximately 40 ms over the read method. This difference may be attributed to the cache size of the experimental platform. When the cache is filled with data, cache flushing operations occur, resulting in a significant increase in latency.

4.2 Comparison Between Two Types of Shared Memory

In this article, we have constructed two distinct types of shared memory, as depicted in Fig. 4. We designate the first type as new-IVSHMEM (I) and the second type as new-IVSHMEM (II). The former supports parallel transmission and has 16 segments, with each segment capable of accommodating 4 KB of data. The latter supports sending a larger amount of data (64 KB) in one go. The data submission methods for new-IVSHMEM(I) and new-IVSHMEM(II) differ: new-IVSHMEM(I) submits the data once after filling all 16 segments, while new-IVSHMEM(II) submits the data after filling each segment, requiring 16 consecutive submissions.

Comparison of Transmission Latency. As shown in Fig. 10, as the data size increases, both new-IVSHMEM(I) and new-IVSHMEM(II) experience an increase in transmission latency. However, the segmented design of type I, which allows for parallel transmission of 16 frames of data, results in better latency performance compared to the non-segmented design of type II. The experimental results demonstrate a latency difference of at least 15 ms between the two types.

Comparison of VMexit. The number of VMexit is an important metric for evaluating a hypervisor. The design goal of Jailhouse is to minimize interference

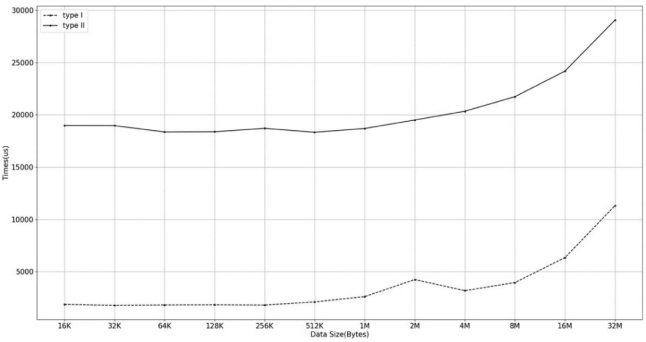

Fig. 10. Transmission latency

with the transactions running inside a cell. The VMexit count indirectly reflects the level of involvement of Jailhouse in the transactions running within the cell.

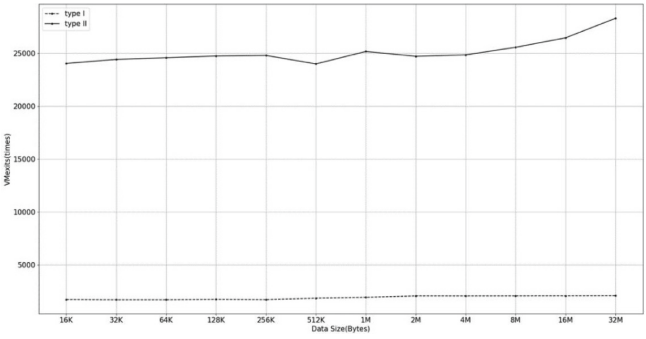

Fig. 11. VMexit counts

As shown in Fig. 11, the evaluation results of the VMexit count during data transmission for new-IVSHMEM(I) and new-IVSHMEM(II) align with the latency results, with type I outperforming type II. Higher latency is typically accompanied by a higher number of VMexit, showing a positive correlation between the two. The experiments indicate a minimum difference of 20,000 VMexit per transmission. The designed communication mechanism aims to mitigate VMexit behavior, aligning with Jailhouse's design philosophy of providing partitioning functionality without excessive interference in the internal transactions of the cell. Additionally, this approach contributes to system stability and enhances reliability in real-time communication scenarios.

4.3 Comparison Between Different IVSHMEM Protocols

This section primarily compares the differences in communication latency, data throughput, and data transmission latency between the pre-reconstruction and post-reconstruction IVSHMEM shared memory protocols.

Performance Improvement After Reconstruction. In this study, the IVSHMEM shared memory protocol underwent four aspects of refactoring, with this section focusing on the disabling of the single interrupt mode. Disabling the single interrupt mode in the IVSHMEM protocol saves one mmio operation on the device register area in each interrupt handler, thus alleviating VMexit behavior.

To evaluate the benefits in terms of VMexit and time brought by disabling the single interrupt mode, we conducted evaluations in the root cell using a Linux-based cell. Through simulation and statistical measurement of actual time costs using the latency testing script (Listing 1.2), we measured the time cost for each operation (a total of 100,000 times) and calculated the mean, resulting in a time cost saving of 1.076497 microseconds by disabling the single interrupt mode, accompanied by a reduction of one VMexit. This observation highlights the effectiveness of the new communication mechanism in mitigating VMexit behavior.

Communication Latency Testing. Through reconstructing specific aspects of the IVSHMEM protocol, the processes of sending and receiving interrupts were optimized. The sender transmits a single byte of data and receives an acknowledgment signal from the receiver upon successful data reception. Both new-IVSHMEM (I) and new-IVSHMEM (II) types exhibit consistent performance in terms of communication latency. Taking new-IVSHMEM (I) as an example, Fig. 12 illustrates a comparison of communication latency between the original IVSHMEM protocol (IVSHMEM-demo) and the communication mechanism proposed in this study (new-IVSHMEM (I)).

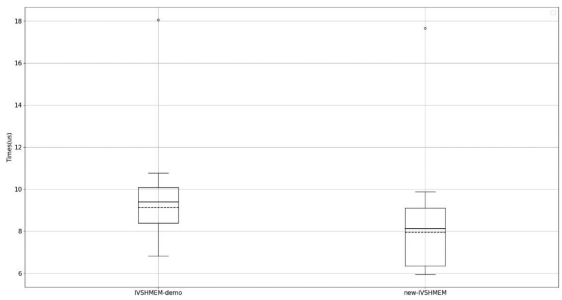

Fig. 12. Comparison of Communication Latency

By comparing the average and median values, it is observed that the new-IVSHMEM method exhibits a difference of approximately 1–2 ms in communication latency compared to the IVSHMEM-demo method. The latter outperforms the former in terms of latency performance, thanks to the disabling of the single interrupt mode, which reduces the VMexit overhead during each communication.

Throughput Testing. To compare the latency and estimate the difference in throughput between IVSHMEM-demo based on the IVSHMEM protocol and new-IVSHMEM based on the communication mechanism proposed in this paper when transmitting 100MB data, the test was conducted 10 times. The specific test results are shown in Table 1, and the differences between the two are illustrated in Fig. 13.

Table 1. Latency Data (Throughput Test)

IVSHMEM-demo/us	new-IVSHMEM/us
84804	74006
83960	70346
84197	71931
87615	71551
86727	75218
88841	70918
82429	70657
83287	71340
84919	71313
87905	74218

Through the comparison, it was found that the throughput of IVSHMEM-demo based on the IVSHMEM protocol is 1170.02 MB/s, while the throughput of new-IVSHMEM designed in this study reaches 1386.01 MB/s, achieving an approximately 200 MB/s throughput improvement. This difference is mainly attributed to the redesigned shared memory region. As new-IVSHMEM can utilize a larger shared memory region, its data throughput is superior to that of IVSHMEM-demo.

Data Transmission Latency Testing. The difference in data transmission latency (median) between new-IVSHMEM and IVSHMEM-demo methods is compared in Fig. 14. For different data sizes, new-IVSHMEM outperforms IVSHMEM-demo, and the latency gap gradually increases with the increase in data volume. When the data size is 32 MB, new-IVSHMEM brings a latency benefit of approximately 5–6 ms. This is attributed to the reconstructed shared memory region and optimization of the native communication protocol described in

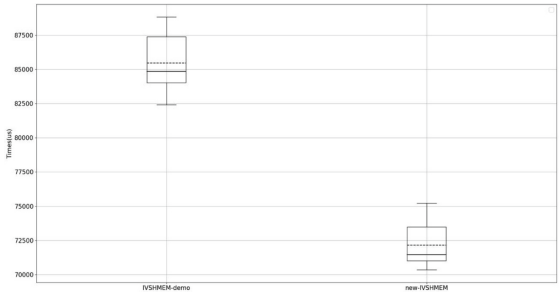

Fig. 13. Data transmission latency difference (Throughput Test)

this paper. When the data volume exceeds 32 KB, the latency difference between the two becomes more significant, which may be related to the cache size of the experimental platform. Additionally, the new communication mechanism has a larger shared memory region, reducing the number of data submissions and lowering latency. This advantage is particularly prominent when handling large volumes of data.

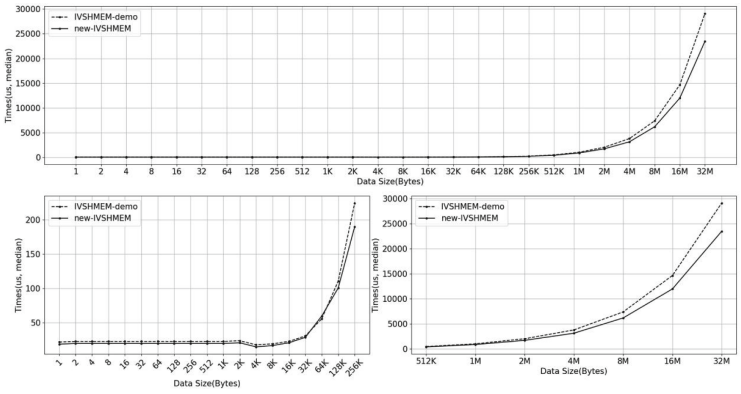

Fig. 14. Comparison of data transmission latency between new-IVSHMEM and IVSHMEM-demo

Based on the comprehensive analysis of the experimental data, the results demonstrate that the communication mechanism designed in this study exhibits lower system call overhead and reduced VMexit compared to the native communication protocol (IVSHMEM). These findings highlight the advantages of the proposed mechanism in enhancing system predictability in real-time communication scenarios.

5 Conclusion

Given the increasing complexity of software, the increasing demand for deploying different systems on the same platform has contributed to the development of mixed critical systems. In this context, the hypervisor assumes a pivotal role in resource partitioning and task execution. Jailhouse, as a lightweight and safety critical design hypervisor, has broad practical application prospects in fields that require real-time virtualization such as intelligent driving and industrial automation. This article aims to provide practical support and solutions for communication issues in future vehicle systems and automation systems by optimizing the Jailhouse IVSHMEM communication protocol.

This study centers on the IVSHMEM communication protocol in Jailhouse and presents a redesign of the shared memory region, proposing two novel design schemes that optimize the communication process and reduce redundant performance overhead. By introducing io_uring to replace traditional read methods, the efficiency of I/O reading is improved. Comparative experiments conducted on both a Linux and RTOS experimental platform based on Jailhouse illustrate that the newly devised mechanism offers advantages in terms of throughput and communication latency.

The IVSHMEM communication protocol remains an ongoing area of development. While the enhanced communication protocol presented in this paper outperforms the performance of the official native communication protocol, it is essential to acknowledge that there are lingering unresolved issues, including:

1. System predictability offers potential for improvement.Currently, the memory-mapped portion of device registers lacks caching, while the shared memory region uses caching, which could impact the system's predictability owing to cache behavior.
2. To facilitate inter-cell communication in Jailhouse, it is conceivable to provide better encapsulation of the current communication protocol's behavior or apply mature alternative communication protocols to Jailhouse. Particularly in scenarios involving communication among multiple cells, a new mechanism may be required to achieve synchronization of operations, further enhancing the work presented in this paper.
3. To minimize Jailhouse's interference in inter-cell communication activities, exploration can be done on how to further avoid VMexit behavior, thereby improving communication performance and aligning with Jailhouse's design philosophy.

Acknowledgements. This work was partially supported by Gansu Province Key Research and Development Plan - Industrial Project under Grant No. 22YF7GA004, Gansu Province Science and Technology Major Project - Industrial Project under Grant No. 22ZD6GA048, the Fundamental Research Funds for the Central Universities under Grant No. lzujbky-2022-kb12, lzujbky-2021-sp43, lzujbky-2020-sp02, lzujbky-2019-kb51 and lzujbky-2018-k12, National Natural Science Foundation of China under Grant Nos. U22A20261 and 61402210. Science and Technology Plan of Qinghai

Province under Grant No.2020-GX-164, and Supercomputing Center of Lanzhou University. We appreciate co-author Mr. Jiaming Zhang's hard work during his postgraduate for the contribution of this paper is inspired by his master thesis [30].

References

1. Biondi, A., Marinoni, M., Buttazzo, G., Scordino, C., Gai, P.: Challenges in virtualizing safety-critical cyber-physical systems. In: Proceedings of Embedded World Conference 2018, pp. 1–5 (2018)
2. Cao, H.: Research on Communication Between Virtual Machines Based on Soft-RoCE in Jailhouse. Master's thesis, Lanzhou University (2022)
3. Corbet, J.: Linux in mixed-criticality systems. https://lwn.net/Articles/774217/. Accessed 7 June 2023
4. Corbet, J.: Safety-critical realtime with linux. https://lwn.net/Articles/734694/. Accessed 7 June 2023
5. Druschel, P.: A high-bandwidth cross-domain transfer facility. In: Proceedings of the 14th ACM Symposium on Operating Systems Principles 1993 (1993)
6. Gamsa, B., Krieger, O., Stumm, M.: Optimizing IPC performance for shared-memory multiprocessors. In: 1994 International Conference on Parallel Processing, vol. 1, pp. 208–211. IEEE (1994)
7. Hernandez, C., et al.: Selene: self-monitored dependable platform for high-performance safety-critical systems. In: 2020 23rd Euromicro Conference on Digital System Design (DSD), pp. 370–377. IEEE (2020)
8. Kiszka, J.: ivshmem-v2-specification. https://github.com/siemens/jailhouse/blob/master/Documentation/ivshmem-v2-specification.md. Accessed 3 June 2023
9. Kiszka, J.: Jailhouse 0.12 released. https://lwn.net/Articles/811509/. Accessed 3 June 2023
10. Kiszka, J.: Jailhouse: a linux-based partitioning hypervisor. https://lwn.net/Articles/574273/. Accessed 7 June 2023
11. Kiszka, J.: Reworking the inter-vm-shared-memory devices. https://static.sched.com/hosted_files/kvmforum2019/4b/KVM-Forum19_ivshmem2.pdf. Accessed 7 June 2023
12. Li, H., Xu, X., Ren, J., Dong, Y.: ACRN: a big little hypervisor for IoT development. In: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 31–44 (2019)
13. Lu, D.: Research on the Impact of Jailhouse on Dynamic Execution Paths in Linux. Master's thesis, Lanzhou University (2021)
14. Macdonell, A.C.: Shared-memory optimizations for virtual machines. Ph.D. thesis, University of Alberta (2011)
15. Martins, J., Tavares, A., Solieri, M., Bertogna, M., Pinto, S.: 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)
16. Miyagawa, M.: Applying jailhouse to the civil infrastructure system. https://elinux.org/images/f/ff/JapanTechnicalJamboree61_JailhouseR1_eng.pdf. Accessed 7 June 2023
17. Ramos, D.: Exploring IVSHMEM in the Jailhouse Hypervisor. Ph.D. thesis, Instituto Superior de Engenharia do Porto (2019)

18. Ramsauer, R., Kiszka, J., Mauerer, W.: Building mixed criticality linux systems with the jailhouse hypervisor. In: Embedded Lunux Conference + OpenIoT-Summit, Portland, OR, 21–23 February 2017 (2017). https://www.youtube.com/watch?v=pvs0fv-gnvw
19. Reichenbach, K.A.: System-call offloading via linux' io_uring on the jailhouse partitioning hypervisor (2021). https://osg.tuhh.de/Theses/2021/2021_ba_kelvin_reichenbach.pdf
20. Sangorrin, D., Honda, S., Takada, H.: Integrated scheduling in a real-time embedded hypervisor. IPSJ SIG Technical Reports 2010–18(2) (2010)
21. Shen, Y., Wang, L., Liang, Y., Li, S., Jiang, B.: Shyper: an embedded hypervisor applying hierarchical resource isolation strategies for mixed-criticality systems. In: 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1287–1292. IEEE (2022)
22. Sinha, S.: Scheduling policies and system software architectures for mixed-criticality computing. Department of Computer Science, Boston University, Technical report (2018)
23. Sinitsyn, V.: Understanding the jailhouse hypervisor, part 1. https://lwn.net/Articles/578295/. Accessed 5 June 2023
24. Sinitsyn, V.: Understanding the jailhouse hypervisor, part 2. https://lwn.net/Articles/578852/. Accessed 5 June 2023
25. torvalds: linux-jailhouse-enabling. https://github.com/siemens/linux/. Accessed 7 June 2023
26. Wang, C., Yang, F., Wang, H., Guo, P., Hou, J.: Improving real time performance of linux system using rt-linux. In: Journal of Physics: Conference Series, vol. 1237, p. 052017. IOP Publishing (2019)
27. West, R., Li, Y., Missimer, E.: Quest-v: a virtualized multikernel for safety-critical real-time systems. arXiv preprint arXiv:1310.6349 (2013)
28. West, R., Li, Y., Missimer, E.: Quest-v: a virtualized multikernel for safety-critical real-time systems. arXiv e-prints arXiv:1310.6349 (2013)
29. Yugang, M., Shiyou, J.: Research on predictability of distributed real-time systems. Comput. Res. Dev. **37**(6), 661–667 (2000)
30. Zhang, J.: Optimal Design of Communication Mechanism between Linux and RTOS based on Jailhouse. Master's thesis, Lanzhou University (2023)